Virtualização – Problemas e desafios

Carlos Eduardo Seo[1] (008278)
IBM Linux Technology Center
Rod. Jornalista Fco. Aguirre Proença
13186-900 – Hortolândia, SP - Brasil
+55 (19) 2132-4339
eduseo@br.ibm.com

RESUMO

Nesse artigo são abordados os atuais problemas e desafios existentes na área de virtualização, em especial aqueles referentes à arquitetura de computadores. O objetivo desse trabalho é, primeiramente, oferecer uma abordagem geral sobre virtualização, conceitos, problemas e desafios. São abordados os temas: consolidação, *deployment*, *debugging*, virtualização em supercomputadores, gerenciamento de memória e gerenciamento de energia. Posteriormente, serão abordados em maior detalhe, dois casos relevantes à arquitetura de computadores: gerenciamento de memória baseado em redundância de dados [9] e o impacto de hierarquias de memória paravirtualizadas em *High Performance Computing* (HPC) [29]. O trabalho conclui que a área de virtualização possui terreno bastante fértil para pesquisa, principalmente nas áreas de sistemas operacionais e arquitetura de computadores.

Palavras-chave

Virtualização, *hypervisor*, desempenho, arquitetura, memória, supercomputadores, *cloud*, consolidação.

1. INTRODUÇÃO

Podemos definir virtualização como uma metodologia ou arcabouço que possibilita a divisão dos recursos de um computador entre múltiplos ambientes de execução [23]. O conceito surgiu nos anos 1960, no *IBM Thomas J. Watson Research Center*, durante o projeto M44/M44X, que visava avaliar os então recentes conceitos de compartilhamento de sistemas. Sua arquitetura era baseada em um conjunto de máquinas virtuais (VMs), uma para cada usuário. A máquina principal era um IBM 7044 (M44) e cada VM era uma réplica (imagem) experimental da 7044 (M44X). O espaço de endereçamento de cada M44X estava contido na hierarquia de memória da M44 e foi implementado através de técnicas de memória virtual e multiprogramação [8,18].

Hoje, em um sistema virtualizado, temos um software *host* sendo executado na máquina física, chamado *Virtual Machine Monitor* (VMM), ou *Hypervisor*. Esse software é responsável pela criação de ambientes simulados de computação, que são as VMs. Sistemas operacionais inteiros podem ser executados nessas máquinas virtuais como se estivessem sendo executados em um hardware real. O termo *Hypervisor* tem origem no sistema IBM VM/370, lançado em 1972, e se referia à interface de paravirtualização. No entanto, o *hypervisor* propriamente dito já existia desde o IBM CP-40, de 1967 [6]. Atualmente, temos dois tipos de *hypervisor*:

- Tipo 1 (*bare-metal*): executam diretamente no hardware do sistema *host*, atuando como controlador de hardware e monitor de sistemas operacionais *guest*. Ex: VMWare ESXi [25], IBM Power VM [12], Xen [28].
- Tipo 2 (*hosted*): executam em um sistema operacional comum e permitem a execução de sistemas operacionais *guest*. Ex: VMWare Workstation [27], Parallels Desktop [19], QEMU [21].

Desde o início da virtualização no final dos anos 1960, empresas como IBM [11], HP [10] e Sun [24] têm desenvolvido e vendido sistemas com suporte a virtualização. No entanto, o foco do mercado não era muito voltado para eles até o começo dos anos 2000. A evolução dos sistemas de hardware – aumento da capacidade de processamento, memória, disco – aliada a necessidade crescente de se fazer mais tarefas computacionais ao mesmo tempo com um custo cada vez menor fez com que a virtualização aparecesse em maior escala nos últimos anos.

Atualmente, não faltam motivações para o uso de virtualização: consolidação de carga de diversos servidores subutilizados em poucos ou apenas um servidor (consolidação de servidores), possibilidade de executar *legacy* software que não funcionam em hardware recente em VMs que simulem hardware compatível, VMs permitem a criação de ambientes seguros e isolados para execução de aplicações não-confiáveis, permite *debugging*/monitoramento de aplicações sem interferir no ambiente de produção, facilita migração de aplicações e servidores, permite simulação de hardware que o usuário não dispõe, entre outras [23].

Tais motivações trazem junto diversos desafios de pesquisa, tais como minimizar *overhead* de controle, gerenciar melhor o uso de memória, otimizar o consumo de energia em *datacenters*, facilitar gerenciamento e *deployment*, etc.

Esse artigo está estruturado da seguinte forma: na Seção 2 serão apresentadas algumas áreas de pesquisa que estão sendo exploradas atualmente no contexto de virtualização. Foram escolhidos alguns trabalhos apresentados em congressos e periódicos recentes para ilustrar cada área. A Seção 3, mostra em maior detalhe dois casos relevantes à arquitetura de computadores. E a Seção 4 apresenta as conclusões desse trabalho.

---

[1] Aluno de doutorado no Instituto de Computação, Universidade Estadual de Campinas, `ceseo@ic.unicamp.br`

2. PROBLEMAS E DESAFIOS

Nessa seção são apresentadas algumas áreas de pesquisa em virtualização, os problemas e desafios existentes. Para cada área, foi escolhido um trabalho publicado em periódicos/congressos recentes (2007-2009) como exemplo de pesquisa.

2.1 Consolidação de servidores

É praticamente impossível não associar virtualização e consolidação de servidores hoje em dia. O uso de máquinas virtuais para consolidação de sistemas em *datacenters* tem crescido rapidamente nos últimos anos devido às facilidades trazidas pela virtualização: facilidade de gerenciamento, disponibilização de máquinas, custo de infra-estrutura e consumo de energia [1]. Como conseqüência, várias empresas hoje oferecem produtos e serviços de virtualização, tais como VMWare, Microsoft [16], IBM and XenSource.

No entanto, consolidação traz uma conseqüência: as cargas das diversas VMs podem interferir em seus desempenhos. Com base nesse cenário, Apparao et al. [1], da Intel Corp., decidiram caracterizar e analisar o desempenho de uma carga representativa de um sistema consolidado.

O trabalho consiste na criação de um *benchmark* que represente uma carga típica de sistemas consolidados (*vConsolidate*). O estudo tem início com uma comparação de desempenho entre as aplicações sendo executadas em um ambiente isolado e depois em um servidor com múltiplas cargas, a fim de medir o impacto da consolidação em cada tipo de aplicação. São tiradas medidas como CPI e L2 cache MPI para tal comparação. É mostrado que qualquer aplicação sofre um impacto considerável em um ambiente consolidado (ver Figura 1). Para aplicações que fazem muito uso de CPU, a maior perda de desempenho é causada por interferências entre os caches e entre os cores. Os estudos mostram que tais aplicações se beneficiam de caches maiores. Já aplicações como *webservers* e *mail servers* se beneficiam mais com maior alocação de banda de rede.

São executados diversos experimentos para verificação de interferência entre cargas distintas, culminando na elaboração de um modelo de desempenho em servidores consolidados para auxiliar no projeto e antecipação do desempenho de cargas virtualizadas em plataformas futuras, levando em conta os aspectos medidos no trabalho.

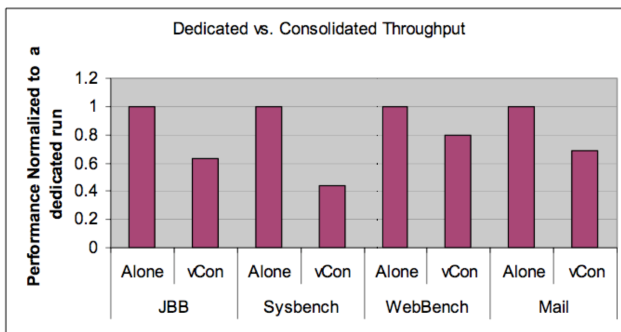


Figura 1: Impacto da consolidação no *throughput* de aplicações [1]

Como trabalhos futuros, os autores citam a necessidade de medir outros *overheads* trazidos pela virtualização: *context switches*, interrupções e *page faults*, visto que o custo (em desempenho) dessas operações muda em um ambiente virtualizado. Além disso, deve ser tratada também a questão da escalabilidade, para o caso da existência de mais de uma *Consolidated Stack Unit* (CSU).

Essa é uma área bastante fértil para a pesquisa dentro de virtualização, pois existem poucos trabalhos e falta muito ainda para que a questão de desempenho em servidores consolidados esteja madura.

2.2. *Deployment*

Em ambientes com várias cargas de trabalho, é interessante o fato de uma máquina virtual possa ser criada e estar pronta para uso o mais rápido possível. Isso se torna mais crítico com a tendência atual de migrar os custos de manutenção e gerenciamento de *datacenters* para terceiros através do *Cloud Computing*. Esse modelo tem a virtualização como um de seus principais vetores, que possibilita a criação (*deployment*) de diversas VMs conforme a demanda dos clientes. No entanto, a maior parte das APIs de *cloud computing* não consegue fazer o deployment das máquinas de forma ágil. Nesse contexto, Lagar-Cavilla et al. [13] propõem uma solução para *deployment* de VMs em menos de 1 segundo chamada *SnowFlock*.

O trabalho se baseia no conceito de *fork* aplicado em VMs. O funcionamento é basicamente igual ao *fork* de processos: uma VM pai faz uma chamada de *fork* que cria um certo número de VMs filhas. Cada VM filha tem um sistema idêntico (inclusive estado) ao da VM pai e também um identificador único (*vmid*). Após o *fork*, o sistema operacional e o disco virtual de cada VM é independente das outras, assim como seus estados. As VMs filhas possuem natureza efêmera, ou seja, uma vez destruídas, sua memória e disco virtual são descartados também. Qualquer alteração que se deseje passar para a VM pai (ou outras VM filhas), deve ser transmitida explicitamente.

O *fork* em tempo abaixo de 1 segundo é conseguido através de técnicas de *lazy state replication*: apenas um pequeno arquivo (*VM descriptor*) é utilizado para criação e inicialização da VM filha. No decorrer da execução da VM filha, um mecanismo copia o estado da memória a partir da VM pai *on-demand*. Algumas modificações são feitas no *kernel* para que as páginas de memória que serão sobrescritas não sejam transferidas durante o *fork*, minimizando o uso de banda de rede. Para se ter uma idéia, o *fork* de um *footprint* de 1GB custa apenas 40MB (ver Tabela 1). Outro mecanismo usado é o *multicast* do estado das VMs, devido a localidade dos acessos à memória pelas VMs. Isso permite o *fork* de diversas VMs a um custo praticamente igual ao do *fork* de uma única VM.

A solução é testada utilizando aplicações típicas de *cloud*: bioinformática, *rendering*, compilação paralela e serviços financeiros. Para cada aplicação, são criadas 128 *threads* de execução: 32 VMs com 4 *cores* SMP distribuídas em 32 *hosts* diferentes. Os resultados mostram que o *SnowFlock* é capaz de instanciar dezenas de VMs em *hosts* diferentes em tempos abaixo de 1 segundo, com um baixo *overhead* de tempo de execução e baixo uso dos recursos de IO do *cloud* (ver Tabela 1), um ganho significativo em relação a solução adotada pelo Xen.

Os autores afirmam que ainda existe espaço para novas pesquisas que explorem o *fork* para VMs. Um exemplo seria o uso dessa técnica juntamente com APIs de paralelismo de dados, como o *MapReduce* [7], ou ainda em situações de migração em massa de VMs entre geografias diferentes.

Tabela 1. *SnowFlock* vs. *Suspend/Resume* (Xen) [13]

Técnica	Tempo[*] (s)	Estado (MB)
SnowFlock	70,63 ± 0,68	41,79 ± 0,7
S/R *multicast*	157,29 ± 0,97	1124
S/R sobre NFS	412,29 ± 11,51	1124

2.3. Gerenciamento de energia

Minimizar os gastos com energia tem sido cada vez mais uma preocupação entre os administradores de *datacenters*. Ao mesmo tempo, a virtualização está cada vez mais sendo adotada nesse ambiente. Dentro desse contexto, Nathuji et al. [17] propõem um novo mecanismo de gerenciamento de energia chamado *VirtualPower Management* (VPM). Trata-se de uma extensão ao Xen que permite um gerenciamento mais eficiente do consumo de energia entre VMs, com impacto mínimo no desempenho das aplicações.

O VPM se baseia basicamente em 2 pontos: (1) oferecer as VMs um conjunto maior de estados possíveis de energia (*soft states*), que são acessíveis as políticas específicas de cada aplicação em cada VM e (2) o uso das mudanças de estado requisitadas pelas VMs como entrada para políticas de gerenciamento de energia em nível de virtualização (mais próximo ao *hardware*). No Xen, toda a inteligência do sistema fica em *Domain0*, o que possibilita a implementação do VPM sem modificações pesadas no *hypervisor*.

Foram testados dois tipos de carga de trabalho: uma transacional e uma de serviços *web*. Em ambos os casos foi verificado um gerenciamento de energia mais ativo, tendo como conseqüência direta o menor consumo de energia, mas sem afetar negativamente o desempenho das aplicações. Em alguns casos, a economia chegou até 34%.

Essa é uma área de pesquisa que, embora tenha muitos trabalhos publicados, sempre tem espaço, visto que nos dias de hoje, melhoras no consumo de energia são muito bem vistas pelo mercado.

2.4. *Debugging*/monitoramento de aplicações

A análise de programas em tempo de execução tem várias aplicações, desde segurança até *debugging*. No entanto, o *overhead* de análise impacta significativamente no desempenho do ambiente de produção. Como uma alternativa viável para solução desse problema, os pesquisadores Chow et al. [5] , da VMWare, formularam uma solução baseada em VMs que permite a realização de análises pesadas em ambiente de produção chamada de *Aftersight*.

[*] Tempo de *deployment* das VMs somado com o tempo de execução do benchmark SHRiMP [22].

Para que isso seja possível, a análise é desacoplada da carga de trabalho – as entradas não-determinísticas da VM de produção são registradas e é feito um *replay* da carga em uma VM diferentes. Com isso, pode-se ter uma análise em tempo real *inline*, *best-effort*, ou ainda, *offline*. Existe ainda a possibilidade de se executar múltiplas análises para uma mesma carga ou incluir novas análises de acordo com a necessidade.

O trabalho é extremamente denso para ser apresentado completamente nesse texto, porém, dos resultados apresentados, dois chamam a atenção. Primeiramente, é mostrado o impacto de uma análise *inline* usando o *Aftersight*. A análise utilizada simula uma execução de antivírus durante uma operação de uso intensivo de disco. Os resultados mostram que o desempenho é cerca de 12% melhor se compararmos com uma análise *inline* tradicional. Outro teste é uma análise pesada sendo executada em paralelo com a carga (*best-effort*). A análise escolhida nesse caso é uma verificação de condição de escrita em um mapa de memória durante uma compilação de *kernel* Linux. Os resultados mostram que o *Aftersight* apresenta desempenho 2x melhor do que o de uma análise *inline* tradicional nesse caso.

Os autores concluem que a análise dinâmica da execução de programas é uma técnica bastante promissora para a solução de diversos problemas, mas que é limitada pelo impacto no desempenho das cargas de trabalho. No entanto, a adoção da virtualização, técnicas como a análise desacoplada mostrada no trabalho se mostram alternativas bastante promissoras.

2.5. Gerenciamento de memória

Um dos grandes desafios em virtualização é o compartilhamento eficiente de memória entre as VMs, que é um dos maiores gargalos na consolidação de sistemas. Pesquisas mostram que é possível melhorar o consumo de memória através do compartilhamento de páginas entre VMs que executam sistemas operacionais e aplicativos similares. Gupta et al. [9] mostram que além disso, duas outras técnicas podem ser utilizadas para se obter melhor uso da memória entre VMs: *patching* de páginas e compressão de páginas. Com base nisso, eles implementam o *Difference Engine*, uma extensão ao VMM do Xen e mostram que os ganhos são bastante significativos (até 90% para cargas similares e até 65% para cargas heterogêneas).

A implementação e análise dessa técnica será descrita em detalhe na seção 3.1.

2.6. Virtualização em supercomputadores

O uso de técnicas de virtualização em supercomputadores ainda é incipiente. Primeiro, porque essas máquinas foram construídas para serem utilizadas em tarefas bastante específicas; segundo, o seu alto custo teoricamente as torna menos competitivas que soluções para sistemas *commodity*. No entanto, existe um projeto em andamento no IBM Research chamado *Project Kittyhawk* [20] que explora a construção e os impactos de um computador de escala global capaz de abrigar a toda a internet como uma aplicação. O trabalho de Appavoo et al. [2] apresenta uma visão geral desse projeto, motivação, descrição do firmware e sistema operacional utilizado e os primeiros resultados da experiência.

Os autores enfatizam que ambos os modelos existentes hoje – *clusters* e sistemas multiprocessados com memória compartilhada – não são adequados para um computador de escala global. É proposto que um híbrido dos dois modelos seria a melhor solução. Um sistema que tenha um processo de fabricação em larga escala já consolidado; empacotamento, alimentação, refrigeração e instalação eficientes; arquitetura de nós simples e minimalista; barramento de interconexão do tipo NUMA escalável; domínios de comunicação configuráveis e impostos pelo *hardware*; e *software* de gerenciamento e controle escalável. Nesse contexto, o Blue Gene/P se mostra uma plataforma ideal para a realização do modelo. A plataforma é descrita a seguir.

Cada nó do Blue Gene/P possui 4 *cores* PowerPC 450 de 850 MHz e 2 GB de RAM. Os nós são agrupados em placas com 32 nós cada e as placas são agrupadas em outro conjunto de 16 placas cada. Cada *rack* do Blue Gene/P tem 2 conjuntos de 16 placas, totalizando 1024 processadores e 2 TB de RAM. Os *racks* podem ser agrupados para constituir uma única instalação, até o limite de 16384 *racks*, o que resulta em um sistema com 67,1 milhões de *cores* e 32 PB de RAM. Apesar desses números monstruosos, é importante notar que cada nó pode ser visto como um computador de uso geral, com processadores, memória e unidades de I/O.

Para explorar as possibilidades da máquina, foi utilizado um *boot loader* (U-Boot) modificado, o L4 Hypervisor [15] e Linux como sistema operacional das VMs. Foram escolhidas diversas aplicações dentro do contexto de web 2.0 para a realização dos experimentos.

Os resultados mostram que é factível o uso do Blue Gene/P como uma plataforma para serviços web. As redes de alta velocidade da plataforma garantem conectividade interna e externa, a alta capacidade de memória permite oferecer *storage* de rede de baixa latência e é possível agrupar diversos processadores em redes colaborativas. Além disso, apresenta rápida escalabilidade (mais rápido do que expandir um *datacenter* comum) e bastante flexibilidade. Além disso, a solução pode ser atrativa em termos de custo também. Servidores *commodity* podem ser baratos individualmente, mas um *cluster* é caro de se comprar e manter (energia e refrigeração). Outro ponto são as conexões de rede: apesar de termos placas muito baratas, a infra-estrutura de *switching* necessária em grandes *clusters* é cara e o preço não escala linearmente com o número de portas. Isso faz com que a solução apresentada seja uma ordem de magnitude mais eficiente (em termos de valores de compra e operação) do que um *cluster* tradicional para uma grande gama de cargas de trabalho *web*.

Tais resultados abrem portas para mais um ramo de pesquisa na área de virtualização, com oportunidades principalmente em nas áreas de sistemas operacionais e arquitetura de computadores.

3. ANÁLISE

3.1. Gerenciamento de memória baseado em redundância de dados

Conforme foi mencionado na seção 2.4, Gupta et al. [9] desenvolveram um mecanismo de gerenciamento de memória no VMM do Xen que possibilita o melhor uso da memória compartilhada entre diversas VMs, através da aplicação de 3 técnicas: compartilhamento de páginas, *patching* de páginas e compressão de páginas.

O compartilhamento foi a primeira técnica utilizada em virtualização. O VMWare ESX Server já fazia o uso de compartilhamento de páginas entre VMs diferentes. No entanto, para que o compartilhamento seja possível, as páginas compartilhadas devem ser idênticas. Porém, é possível tirar proveito de páginas quase idênticas também. Nesse contexto, aparece a técnica de *patching*: páginas quase idênticas podem ter sua parte idêntica compartilhada e *patches* são aplicados para reconstruir a página desejada. Por último, quando as páginas têm probabilidade de não serem usadas num futuro próximo, elas são comprimidas. A Figura 2 mostra um esquemático das 3 técnicas.

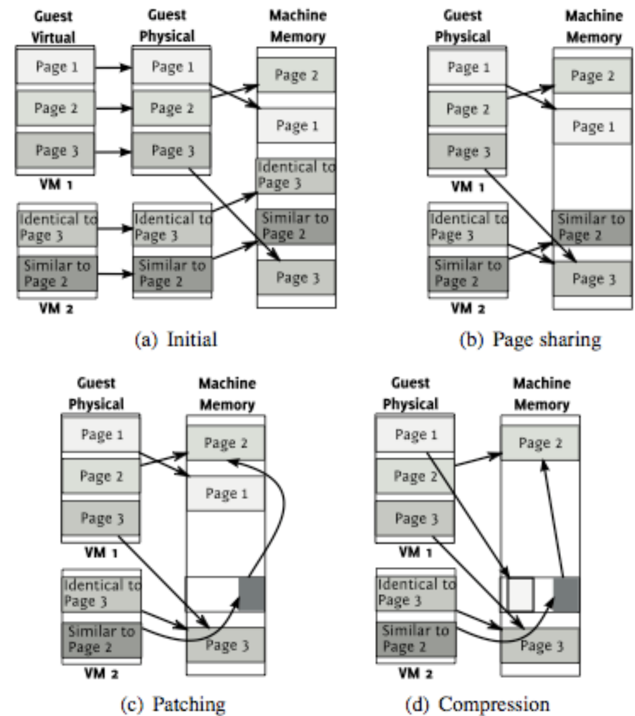


Figura 2: Exemplo mostrando as 3 técnicas utilizadas pelo *Difference Engine*. No final, temos um ganho aproximado de 50% em espaço em relação ao armazenamento original [9].

O compartilhamento é implementado da mesma forma em que trabalhos anteriores: é feita uma varredura na memória e um *hash* de cada página. As páginas são indexadas com base em seu valor de *hash*. Páginas idênticas apresentam o mesmo valor de *hash* e quando uma colisão é detectada, tem-se um potencial candidato ao compartilhamento. É feita então uma varredura *byte* a *byte* para se ter certeza que as páginas são realmente idênticas e, em caso afirmativo, elas são compartilhadas e a memória virtual passa a apontar para uma delas. Páginas compartilhadas são marcadas como somente leitura, de tal forma que, quando VM tenta escrever, uma *page fault* ocorre. O VMM então retorna uma cópia privada da página para a VM que provocou a falha e atualiza os mapas de memória. Se em algum momento nenhuma VM referencia uma página compartilhada, ela é liberada.

Para a operação de *patching*, o *Difference Engine* utiliza um mecanismo de *hash* para encontrar possíveis candidatos. Cada página é indexada através de *hashes* de dois blocos de 64 KB em posições fixas da páginas (a escolha inicial dessas posições é aleatória). Ou seja, cada página tem 2 índices na tabela *hash*. Para que se encontre uma página candidata ao *patching*, o mecanismo

calcula *hashes* para blocos nas mesmas posições da página e procura na tabela a melhor página entre as (no máximo duas) encontradas. Como esse mecanismo possui um *overhead* não desprezível, só é aplicado a páginas que são utilizadas com pouca freqüência.

Já para a compressão, ela só é aplicada quando a taxa de compressão é alta e se as páginas são utilizadas com pouca freqüência, caso contrário, o *overhead* da operação ultrapassa os benefícios trazidos pela técnica. Páginas são identificadas através de um algoritmo que implementa um relógio global para identificar a freqüência de acesso. As páginas comprimidas são invalidadas e movidas para uma área de armazenamento na memória da máquina física. Caso um acesso seja feito, o mecanismo faz a descompressão e retorna a página para a VM requisitante.

Além disso, o *Difference Engine* também identifica páginas candidatas a *swapping* dentre aquelas que foram marcadas para *patching* e compressão. No entanto, a decisão de quanto e quando fazer *swap* para disco fica a cargo da aplicação em *userspace*.

Tudo isso faz com que o gerenciamento de memória seja muito mais eficiente se usamos o Xen + *Difference Engine* do que o VMWare ESX Server, conforme mostram os resultados. Foram utilizados vários cenários, mas para fins de análise, será mostrado aqui apenas um deles, que é bastante significativo pois utiliza uma carga de trabalho do mundo real. São 3 VMs completamente diferentes: Windows XP SP1 executando RUBiS, Debian Linux 3.1 compilando o kernel Linux e Slackware Linux 10.2 compilando o vim-7.0 e executando o *benchmark* lmbench logo em seguida. Se observarmos a Figura 3, veremos que o *Difference Engine* consegue economizar até 45% mais memória que o VMWare ESX Server (que utiliza apenas compartilhamento).

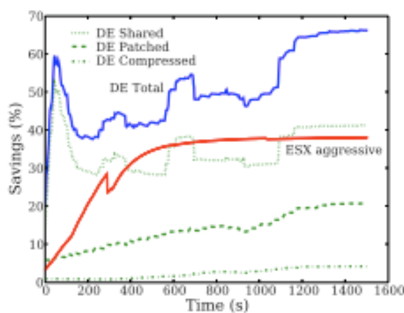


Figura 3: Economia de memória para a carga heterogênea[9].

Nota-se que se usarmos apenas o compartilhamento, o desempenho em geral é similar entre os dois VMMs. No entanto, ao adicionarmos *patching* e compressão, o *Difference Engine* dá um salto grande em desempenho, mostrando que essas técnicas também são importantes no gerenciamento de memória no VMM, pois o *overhead* medido foi de apenas *7%*. Os outros resultados mostram que se obtém uma economia de até 90% em cargas homogêneas e de até 65% em cargas heterogêneas.

Como trabalhos futuros, os autores citam que é possível explorar a utilização do mecanismo do *Difference Engine* para melhorar o desempenho de um sistema operacional convencional isoladamente também.

3.2. Impacto da hierarquia de memória paravirtualizada em HPC

A virtualização foi ignorada por um bom tempo pelos usuários de aplicações de computação intensiva devido ao potencial decremento no desempenho das aplicações. No entanto, para esses usuários, a paravirtualização se mostra uma alternativa interessante. Na paravirtualização, os sistemas operacionais *host* e *guest* são modificados para oferecer o melhor desempenho possível quando é utilizada a virtualização. Vários estudos [29] mostraram que para aplicações de HPC, os sistemas paravirtualizados apresentam desempenho muito próximo do sistema nativo. No entanto, informações a respeito do desempenho em situações nas quais a memória é escassa são raras. Isso é importante porque o desempenho de várias aplicações de álgebra linear depende das características do uso da memória. Nesse contexto, Youseff et al. [30] mostram um estudo detalhado do impacto da hierarquia de memória paravirtualizada em aplicações de HPC. Mais especificamente, deseja-se saber como que a paravirtualização afeta o *autotuning* das aplicações e como ela afeta o desempenho das aplicações. Para isso, será utilizada uma das ferramentas de *autotuning* mais populares que existem, o ATLAS (*Automatically Tuned Linear Algebra Software*) [3], para fazer o *autotuning* de duas bibliotecas muito usadas em álgebra linear: BLAS [4] e LAPACK [14]. Como solução de virtualização, foi utilizado o Xen.

Foram preparados 3 sistemas: um nativo, uma VM com privilégios (*Dom0*, na notação do Xen) e uma VM sem privilégios (*DomU*), todos com Linux 2.6.16. Duas configurações de memória foram adotadas: 256 MB e 756 MB.

Os primeiros experimentos foram feitos visando observar o impacto da paravirtualização no *autotuning*. Os resultados mostram que a paravirtualização não influi na caracterização do sistema (detecção de hardware idêntico para os 3 sistemas), nem impõe *overhead* de desempenho em operações entre registradores de ponto flutuante. No teste de configuração de cache, no qual o ATLAS tenta encontrar o tamanho ótimo do bloco de cache (tanto para L1 quanto para L2) a ser usado em operações de multiplicação entre matrizes, os resultados mostram que o ATLAS não vê diferença entre os sistemas na hora de escolher o tamanho ótimo do bloco para cache L2.

O segundo conjunto de experimentos investiga o impacto da paravirtualização em diferentes níveis da hierarquia de memória durante a execução de aplicações HPC de uso intensivo de memória. Foi utilizado um código de multiplicação entre matrizes com precisão dupla que possui consumo de memória crescente até 350 MB e o desempenho foi medido em MFLOPS.

Os resultados mostram que a paravirtualização tem impacto muito pouco significativo no desempenho desse tipo de aplicação, o que é um boa surpresa, pois a paravirtualização introduz mais um nível de escalonamento de processos e indireção de I/O, o que poderia causar facilmente um aumento no TLB *miss rate*, por exemplo. O que temos é um perfil de hierarquia de memória muito similar entre os 3 sistemas, desde o acesso ao cache, até o *swap* em disco.

Tais resultados mostram que aplicações de HPC podem tirar proveito de estruturas de *clusters* virtuais e *cloud computing*, visto que a virtualização não influencia no desempenho dessas aplicações.

4. CONCLUSÕES

Esse trabalho apresentou uma visão geral da área de virtualização, mostrando quais são as áreas de pesquisa sendo exploradas atualmente e o que ainda existe de oportunidade para novos trabalhos. Para as áreas de sistemas operacionais e arquitetura de computadores, existem ainda vários tópicos de pesquisa que valem a pena ser explorados, principalmente em gerenciamento de memória e o uso de supercomputadores como base de VMs.

5. AGRADECIMENTOS

Agradecimentos ao Prof. Dr. Rodolfo Jardim de Azevedo (IC-UNICAMP) e à Dra. Dilma da Silva (IBM Thomas J. Watson Research Center) pela orientação nesse trabalho de pesquisa.

6. REFERÊNCIAS

[1] Apparao, P.; Iyer, R.; Zhang, X.; Newell, D. & Adelmeyer, T., Characterization & analysis of a server consolidation benchmark, *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on virtual execution environments*. ACM, **2008**, pp. 21-30.

[2] Appavoo, J.; Uhlig, V. & Waterland, A., Project Kittyhawk: building a global scale computer: Blue Gene/P as a generic computing platform, *SIGOPS Oper. Syst. Rev*. ACM, **2008**, Vol. 42 (1), pp. 77-84.

[3] ATLAS
http://math-atlas.sourceforge.net
24/06/2009.

[4] BLAS
http://www.netlib.org/blas
24/06/2009.

[5] Chow, J.; Garfinkel, T. & Chen, P.M., Decoupling dynamic program analysis from execution in virtual environments, *ATC '08: USENIX 2008 Annual Technical Conference*. USENIX Association, **2008**, pp. 1-14.

[6] Creasy, R.J., The origin of the VM/370 time-sharing system, *IBM Journal of Research & Development Vol. 25, No. 5*. IBM, **1981**, pp. 483-490.

[7] Dean, J. & Ghemawat, S., MapReduce: Simplified Data Processing on Large Clusters, *Commun. ACM*. ACM, **2008**, Vol. 51 (1), pp. 107-113.

[8] Denning, P., ACM president's letter: performance analysis: experimental computer science as its best, *Commun. ACM*. ACM, **1981**, Vol. 24 (11), pp. 725-727.

[9] Gupta, D.; Lee, S.; Vrable, M.; Savage, S.; Snoeren, A. C.; Varghese, G.; Voelker, G. M. & Vahdat, A., Difference Engine: Harnessing Memory Redundancy in Virtual Machines, *OSDI'08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation. USENIX Association,* **2008**, pp. 309-322.

[10] HP Co.
http://www.hp.com
12/06/2009.

[11] IBM Corp.
http://www.ibm.com
12/06/2009.

[12] IBM PowerVM
http://www-03.ibm.com/systems/power/software/virtualization
12/06/2009.

[13] Lagar-Cavilla, H.A.; Whitney, J.A.; Scannell, A.M.; Patchin, P.; Rumble, S.M.; de Lara, E.; Brudno, M. & Satyanarayanan, M., SnowFlock: rapid virtual machine cloning for cloud computing, *EuroSys '09: Proceedings of the fourth ACM European conference on computer systems*. ACM, **2009**, pp. 1-12.

[14] LAPACK
http://www.netlib.org/lapack
24/06/2009.

[15] Liedtke, J., On μ-kernel construction, *SOSP '95: Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, **1995**, pp. 237-250.

[16] Microsoft Corp.
http://www.microsoft.com
12/06/2009.

[17] Nathuji, R. & Schwan, K., VirtualPower: coordinated Power management in virtualized enterprise systems, SOSP '07: Proceedings of twenty-first *ACM SIGOPS symposium on operating systems principles*. ACM, **2007**, pp. 265-278.

[18] O'Neill, W., Experience using a time sharing multiprogramming system with dynamic address relocation hardware, *Proc. AFIPS Computer Conference 30*. SJCC, **1967**, pp. 78-117.

[19] Parallels Desktop
http://www.parallels.com/products/desktop
12/06/2009.

[20] Project Kittyhawk
http://domino.research.ibm.com/comm/research_projects.nsf/pages/kittyhawk.index.html
23/06/2009.

[21] QEMU
http://www.nongnu.org/qemu
12/06/2009.

[22] SHRiMP
http://compbio.cs.toronto.edu/shrimp
17/06/2009.

[23] Singh, A., An Introduction To Virtualization.
http://www.kernelthreads.com/publications/virtualization
12/06/2009.

[24] Sun Microsystems Inc.
http://www.sun.com
12/06/2009.

[25] VMWare ESXi
http://www.vmware.com/products/esxi
12/06/2009.

[26] VMWare Inc.
http://www.vmware.com
12/06/2009.

[27] VMWare Workstation
http://www.vmware.com/products/ws
12/06/2009.

[28] Xen.org
http://www.xen.org
12/06/2009.

[29] Youseff, L.; Wolski, R.; Gorda, B. & Krintz, C., Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems, *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*. IEEE, **2006**, pp. 1.

[30] Youseff, L., Seymour, K., You, H., Dongarra, J. & Wolski, R., The impact of paravirtualized memory hierarchy on linear algebra computational kernels and software, *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*. ACM, **2008**, pp. 141-152.

Paralelismo em nível de threads

Tatiana Al-Chueyr
(017396)
CTI Renato Archer
Rodovia Dom Pedro I, km 143
Campinas, SP, Brasil
+55 19 3746-6035
tmartins@cti.gov.br

RESUMO
Neste artigo são apresentadas arquiteturas que empregam paralelismo em nível de threads (TLP). Inicialmente, analisa-se o histórico e as motivações que desencadearam o desenvolvimento desta forma de paralelismo. Então, são apresentados dois paradigmas para implementação de TLP (*Simultaneous Multithreading* e *Chip Multi-processing*). Com base em tais paradigmas, são descritos dois processadores comerciais que oferecem paralelismo a nivel de threads (Hyperthreading e Niagara). Então, discute-se o desempenho da aplicação deste tipo de arquitetura tanto em servidores, quanto em aplicações multimídia e desktop. Por fim, são apresentados alguns desafios que enfrentados no desenvolvimento e na utilização de paralelismo em nível de threads.

Categorias e Descritores de Assunto
C.1.2 [**Arquiteturas de Processadores**]: Arquiteturas de Mútiplos Streams de Dados

Termos Gerais
Modelagem (Design).

Keywords
Paralelismo em nível de threads, multiprocessadores.

1. INTRODUÇÃO
Um dos constantes desafios no desenvolvimento de novas arquiteturas de processadores, ao longo da história, é prover aumento de desempenho associado a baixo custo de produção.

Visando satisfazer tais desafios, nas últimas décadas os processadores passaram por muitas mudanças importantes, dentre as quais destacam-se: (a) a divisão da execução de instruções em estágios; (b) a criação de pipelines, que possibilitam a execução de múltiplas instruções em um mesmo ciclo do processador; (c) o desenvolvimento de processadores superescalares, que além de permitir a execução paralela e simultânea de instruções, também oferecem a sobreposição parcial oferecida por pipelines.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'04, Month 1–2, 2004, City, State, Country.
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

Processadores superescalares exploram o paralelismo em nível de instrução (ILP, sigla para *Instruction Level Parallelism*), podendo utilizar uma série de recursos para isso, tais como: execução fora-de-ordem, hierarquia de memória, previsão de desvios, dentre outros. Entretanto, estas técnicas fazem com que os processadores tornem-se mais complexos e tenham mais transistores dissipando grande quantidade de energia e calor. Exemplos comerciais de processadores superescalares são o Intel Pentium e o Sun UltraSparc.

Uma das estratégias para melhorar o desempenho de processadores superescalares, historicamente, foi aumentar a frequencia de clock, de modo que as instruções pudessem ser executadas cada vez mais rapidamente. Entretanto, para que tal frequencia pudesse ser aproveitada, surgiram dois desafios: o uso eficiente de um elevado número de estágios de pipeline e o aumento de consumo de energia, decorrente de maiores áreas de silício, maior número de transistores e maior dissipação de calor.

Ainda, apesar do ILP disponível em procesadores superescalares ser adequado para muitas aplicações, ele é ineficaz para outras, como programas em que é difícil predizer código. Outras limitações da ILP podem ser lidas na literatura [1].

Em contrapartida ao paralelismo a nível de instruções, observou-se que muitas aplicações apresentam melhor desempenho quando executadas com outra forma de paralelismo: em nível de threads ou TLP (*Thread Level Parallelism*).

O TLP envolve controlar múltiplas threads do processador, permitindo que partes específicas do programa sejam distribuídas entre diferentes processadores e possam executar simultaneamente. A Figura 1 ilustra a transformação de uma execução serial em paralela, na qual uma iteração que demoraria seis ciclos de clock é dividida em três threads e é reduzida a dois ciclos de clock.

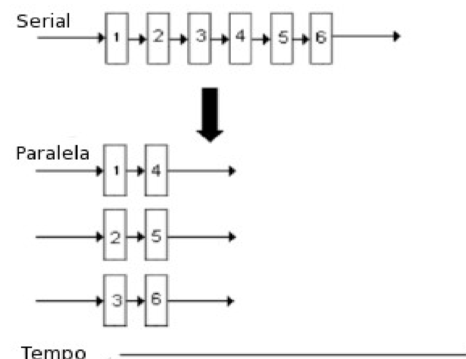


Figura 1. Transformação de código serial para paralelo

Há alguns contextos em que a TLP oferece melhor desempenho que a ILP, outros contextos em que a ILP provê melhor desempenho e, por fim, há situações em que ambas tem desempenho similar [2]. É fundamental que seja analisada a aplicação, para que então seja definida qual forma de paralelismo é a mais adequada.

Um método comum para aumentar o desempenho geral da TLP é o uso de múltiplas CPUs independentes ou multicores. De acordo com a taxonomia de Flynn [3] pode-se classificar multiprocessadores que dão suporte a TLP como MIMD (*Multiple instruction streams, multiple data streams*), onde cada processador controla seu próprio conjunto de instruções e opera seu próprio dado.

A seguir veremos dois paradigmas de processadores que permitem o paralelismo a nivel de threads. Na sequencia, serão apresentadas duas arquiteturas de processadores comerciais: Hyper-Threading (Intel) e Niagara (Sun Microsystem). Então será discutida a aplicação de TLP em contextos distintos, tais como em aplicações multimidia, servidores e aplicativos multimidia. Por fim, apresentamos desafios com os quais tanto arquitetos quanto programadores enfrentam ao lidar com TLP.

2. CLASSIFICAÇÃO

Processadores que implementam TLP são convencionalmente classificados segundo pelo menos um destes paradigmas: *Simultaneous Multithreading* (SMT) e *Chip Multiprocessing (CMP).*

2.1 *Simultaneous Multithreading*

O SMT [4] é um paradigma que tem sido empregado industrialmente [5,6]. Ele permite que instruções de múltiplas threads, simultaneamente, sejam buscadas e executadas no mesmo pipeline, amortizando o custo de mais componentes distribuído entre mais intruções por ciclo.

O *Simultaneous Multithreading* propõe oferecer melhoria no *throughput* através da relação eficiência-área [7].

Uma arquitetura SMT é capaz de atingir alto IPC (instruções por ciclo) devido as seguintes razões:

1. A facilidade de paralelizar tarefas distintas, principalmente quando não há comunicação entre elas;
2. O escalonamento de instruções não-bloqueadas para o despacho pode esconder as altas latências de outras instruções bloqueadas, tais como as de acesso a memória;
3. Através da interpolação de instruções de diferentes tarefas, a execução da SMT otimiza os recursos que poderiam estar ociosos, tais como unidades funcionais, dada a maior variedade de tipos de instruções.

A arquitetura SMT é composta por unidades funcionais compartilhadas e por estruturas e recursos separados, logicamente ou fisicamente.

O Pentium 4 Extreme SMT permite a execução de duas threads simultâneas e proporciona um *speedup* de 1.01 no benchmark SPECint_rate e de 1.07 no SPECfp_rate. Ao executar o processador Pentium 4 para 26 benchmarks SPEC, o speedup varia entre 0.90 e 1.58, com média 1.20 [7].

Já no Power 5, um servidor executa 1.23 mais rápido o benchmark SPECint_rate com SMT, 1.16 mais rápido para SPECfp_rate.

Algumas dificuldades que podem emergir com esta abordagem [1]:

- escolher qual instrução será executada por vez;
- utilizar a estratégia de uma thread preferida faz com que o processador sacrifique parte do *throughput* quando ocorre *stall* de thread;
- criar arquivo de registradores para manter múltiplos contextos (gargalo); e, por fim,
- assegurar que os conflitos gerados pela a cache e a TLP não degradam o desempenho..

2.2 *Chip Multiprocessing*

O CMP é um multiprocessamento simétrico (SMP) implementado em um único chip [8]. Múltiplos núcleos de processador são interconectados e compartilham um dos níveis de cache (convencionalmente o segundo ou terceiro nível). Em geral cada core há b*ranch predictors* e caches de primeiro nível privadas.

O objetivo de uma arquitetura CMP é permitir maior utilização de paralelismo a nível de threads, sem contudo prover suporte a paralelismo a nível de instrução.

Para *workloads* multi-threaded, arquiteturas CMP amortizam o custo do chip entre os múltiplos processadores e permitem comparitlhamento de dados entre caches L2.

Alguns exemplos de arquiteturas comercciais que utilizam CMP: PA-RISC (PA-8800), IBM POWER 4 e SPARC (UltraSPARC IV).

Para que ocorra o uso efetivo de chips CMP é necessário que o sistema operacional empregado dê suporte a multiprocessamento.

2.3 *Comparação*

Em ambos os paradigmas, busca-se o aumento do throughput. A replicação de cores significa que a área e que o *overhead* de energia necessários para o CMP são muito superiores ao SMT. Para um determinado tamanho de chip, um SMT de apenas um core terá suporte a um tamanho maior de L2 do que em um chip multi-core.

Por outro lado, a falta de contenção e execução entre threads tipicamente existente no SMT permite um *throughput* bastante superior para o CMP. Uma das maiores preocupações em adicionar-se múltiplos cores ao chip é o aumento drástico de dissipação de energia, o que agrega a este tipo de processador custos adicionais para resfriamento.

Existem arquiteturas que abordam ambos paradigmas – tanto o CMP quanto o SMT. Um exemplo é a arquitetura Power 5 da Intel, onde há dois núcleos de processadores SMT

A partir da literatura [7], selecionamos dois gráficos que ilustram o desempenho e eficiência no consumo de energia do SMT e do CMP. Tais gráficos podem ser vistos nas Figuras 2 e 3. Para a elaboração destes gráficos foi considerado um processador Power 4.

Como pode ser observado, há situações em que um dos paradigmas apresenta melhor desempenho, e há situações em que o outro é vitorioso. O que reforça que a definição de qual tecnologia utilizar é intimamente relacionada a aplicação final e em que contextos o processador será empregado.

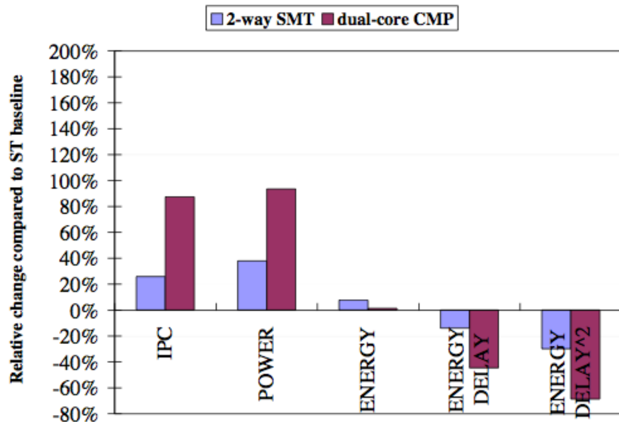


Figura 2. Comparação do desempenho e eficiência do uso de energia no SMT e no CMP para workloads com poucos miss na cache L2.

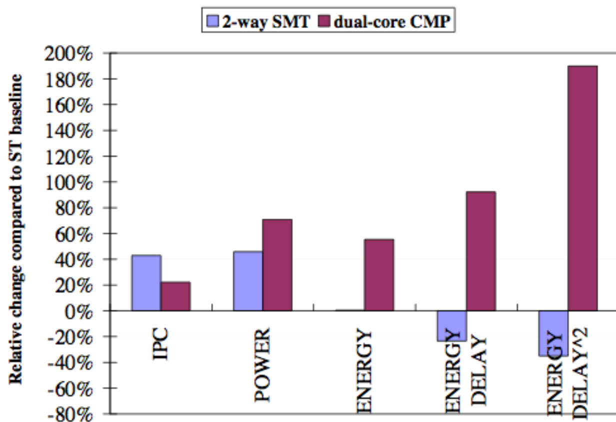


Figura 3. Comparação do desempenho e eficiência do uso de energia no SMT e no CMP para workloads com muitos miss na cache L2.

3. PROCESSADORES COMERCIAIS

3.1 HyperThreading

A tecnologia HyperThreading (ou HT), desenvolvida pela Intel, é a precursora dos processadores de núcleo duplo e múltiplo, tais como o Intel Core 2 Quad. Esta tecnologia se baseia na abordagem SMT.

A HT simula em um único processador físico dois processadores lógicos. Cada processador lógico recebe seu próprio controlador de interrupção programável (APIC) e conjunto de registradores. Os outros recursos do processador físico, tais como, cache de memória, unidade de execução, unidade lógica e aritmética, unidade de ponto flutuante e barramentos, são compartilhados entre os processadores lógicos. Em termos de software, o sistema operacional pode enviar tarefas para os processadores lógicos como se estivesse enviando para processadores físicos distintos, em um sistema de multiprocessamento real.

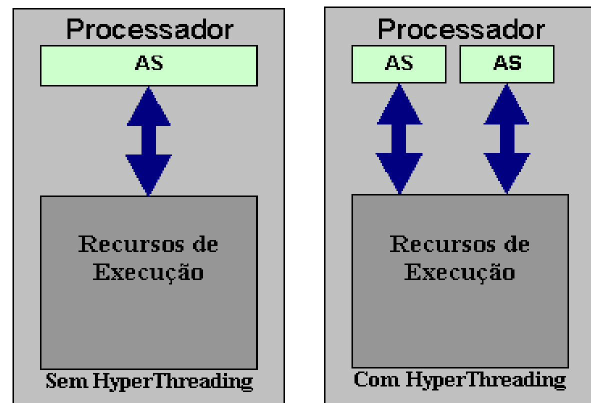


Figura 4. Comparação de um processador com e sem a tecnologia

A Figura 4 ilustra o funcionamento de um processador normal e um processador com a tecnologia HyperThreading. Os registradores e controlador de interrupção foram chamados de "AS". Na área denominada de "recursos de execução" estão todos os recursos que o processador necessita para executar as instruções. No processador com HP ocorre duplicação de registradores, controladores e compartilhado os recursos de execução entre os processadores lógicos, parecendo assim um sistema com dois processadores reais.

Para que se usufrua da tecnologia, tanto o sistema operacional utilizado quando os aplicativos de software têm que dar suporte a HyperThreading.

O primeiro processador da Intel a implementar a tecnologia HyperThreading foi o Intel Xeon. O Intel Xeon utiliza a arquitetura NetBurst e é voltado para o mercado de servidores. Apesar do foco inicial da tecnologia HyperThreading ser processadores para servidores de rede, forem feitos chipsets (Intel 845PE) para os processadores Pentium 4.

Sistemas operacionais como o Windowx XP e algumas distribuições de GNU Linux são SMP (Multiprocessamento Simétrico), ou seja, podem trabalhar com mais de um processador instalado no sistema, dividindo às tarefas entre os mesmos. A tecnologia HyperThreading estende essa idéia de forma que os sistema operacionais e software aplicativos dividam as tarefas entre os processadores lógicos.

As Figuras 5 e 6 ilustram as diferenças entre uma arquitetura multiprocessada (física) e a tecnologia HyperThreading.

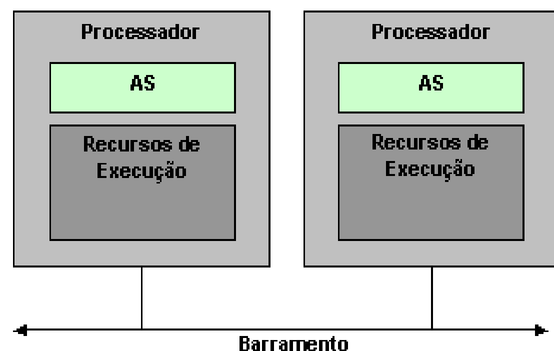


Figura 5. Sistema Multiprocessado sem tecnologia HyperThreading.

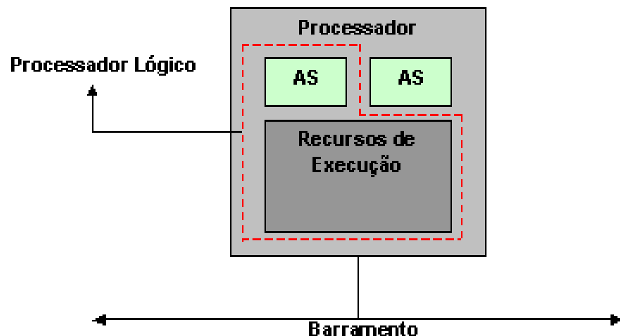


Figura 6. Processador com tecnologia Hyper-Threading.

Nos PCs desktop e workstations simples, a tecnologia HT aproveita da capacidade de multiprocessos, disponível no sistema operacional ou nos aplicativos, dividindo as cargas de trabalho em vários processos, que são designados e enviados independentemente. Num sistema de multiprocessador real, eles são executados em processos diferentes.

Nos servidores e workstations de alto desempenho, a tecnologia HyperThreading habilita a TLP, ao duplicar a arquitetura de cada processador e ao mesmo tempo compartilhar um conjunto de recursos de execuçãod o processador. Ao programar os processos, o sistema operacional trata os dois estados distintos de arquitetura como processadores lógicos ou virtuais separados, o que permite que softwares multiprocessados rodem sem modificações.

Embora a tecnologia HyperThreading não ofereça o nivel de escalonamento de desempenho alcançado ao adicionar um segundo segundo processador, testes de benchmark mostram que alguns aplicativos de servidor tem um desempenho 30% maior. [9].

Os usuários aproveitam do desempenho melhorado, executando múltiplos aplicativos simultaneamente, como, por exemplo, rodar uma análise de vírus ou codificação de vídeo no background e ao mesmo tempo continuar com um jogo. Para os gerentes da TI, a tecnologia HT significa um uso mais eficiente dos recursos do processador, maior saída e desempenho melhorado.

Conforme apresentado nos gráficos disponíveis nas Figuras 7 e 8, obtidas da literatura [9], a tecnologia hyper-threading pode ter ganhos consideráveis em aplicação típicas de servidores (banco de dados, servidores web). Por outro lado, em certas aplicações de servidores e aplicações típicas de desktop, a tecnologia pode apresentar perdas consideráveis de desempenho, conforme mostra a Figura 8.

3.2 Niagara

O Sun Niagara, ou UltraSPARC T1, é um processador que visa atender servidores de alto *workload*, como *webservers* e *datacenters*. Para isso, o Niagara possui 8 cores de processamento (são comercializadas versões de 4 e 6 cores, também), onde cada um executa 4 threads de hardware, totalizando 32 threads [10].

Este processador foi iniciado em um projeto de pesquisa chamado Hydra [11], desenvolvido pelo professor Kunle Olukotun, relacionado a CMP. Com a conclusão do projeto, a empresa Afara Websystems foi criada para comercializar o projeto Hydra. Entretanto, a empresa acabou sendo vendida para a Sun Microsystems, em 2002, onde o projeto foi a base do Niagara.

O Niagara tem como objetivo atingir altos valores de performance em paralelo a consumo adequado de energia. Assim, o processador consome, em média, 72W, chegando a um pico de 79W. O principal fator para isso é a sua arquitetura simples, composta por vários cores de baixa frequência, mas alto throughput. Entretanto, o Niagara possui algumas limitações – sua limitação mais séria é o fato de possuir apenas uma unidade de ponto flutuante (FPU) para 8 cores, e assim é incapaz de processar mais que 1-3% de operações de ponto flutuante.

A tecnologia CMT é utilizada no Sun Niagara. Esta tecnologia consiste na combinação de CMP e VMT (vertical multithreading - apenas uma das threads de cada core pode executar instruções em um ciclo, trocando para outra thread quando a thread ativa precisa buscar dados na memória). No Sun Niagara há 4 threads de hardware por core (com o intuito de maximizar a eficiência de cada core) e 8 cores (de forma a aumentar o poder de processamento).

Cada core possui um pipeline de 6 estágios, um instruction cache L1, um data cache L1 e uma unidade de gerenciamento de memória (MMU), que são compartilhados pelas 4 threads, e todos os cores são ligados ao cache L2, que é compartilhado por todos. Assim, o sistema operacional rodando na máquina enxerga 32 processadores virtuais (ou lógicos).

A motivação por trás desta técnica é o grande atraso causado por acessos a memória – normalmente na ordem de dezenas a centenas de ciclos. Utiliza-se então esta técnica para que instruções de várias threads de software sejam executadas simultaneamente, nas diversas threads de hardware (de forma que a pausa para I/O de uma não afete o throughput geral significativamente).

O Niagara é um processador voltado para aplicações de servidor – cujas cargas de trabalho são caracterizadas por altas taxas de TLP) e baixos níveis de ILP.

O desempenho de um sistema baseado em CMT é altamente relacionada ao sistema operacional utilizado – no caso do Niagara, o Solaris 10 (ou superior), sistema operacional da Sun, é bastante otimizado para isso – alocando corretamente as threads de software para os cores mais eficientes, reduzindo o atraso causado pelo uso de recursos compartilhados (o cache L2 e o bus).

Obtivemos a performance do Sun Niagara em testes realizados contra dois outros servidores: um Dell 2850 Dual 3.2 GHz Xeon com 12GB de RAM, rodando Debian, e um Dell 7250 Itanium (dual 1.5 GHz com 32GB de RAM). O Niagara foi testado no Sun FIRE T2000, uma das máquinas comercializadas pela Sun com ele. Os testes realizados utilizaram o Apache, ou seja, os servidores foram testados como webservers. Foram feitos por Colm MacCarthaigh, do webserver ftp.heanet.ie.

A performance do Niagara é consideravelmente superior a dos outros 2 em volume de transações por segundo (5.700 contra 2.700 do Itanium, o segundo maior – e o valor foi ampliado pra 22.000 em testes posteriores ao benchmark). Além disso, o servidor agüentou quase 150% dos downloads concorrentes do Itanium, e 300% do Xeon (83.000 para o Niagara, 57.000 Itanium, 27.000 Xeon). Além disso, a latência do Niagara, apesar de mais alta em valores mais baixos de downloads, aumenta de forma muito menos acentuada que os outros dois para maiores valores (chegando aos máximos que agüentam com valores perto de um minuto de latência).

Temos também que a energia consumida pelo Sun T2000 é muito mais baixa que a dos dois Dell: em média, aproximadamente 240 W, com picos de 300W, enquanto o Dell possui uma média de 384 W e picos de 480 W, e o Dell 7250 média de 432 W e picos de 538 W. [12]

Os problemas encontrados no Niagara foram, exatamente, em performance individual: quando foram testados I/O de apenas uma thread, ou apenas um download, o valor obtido foi muito abaixo dos outros.

Em 2007 foi lançado o Niagara-2, no qual houve melhoria de desempenho do processador, e diminuição ou manuteção no consumo de energia) [13].

O Niagara-2 tem 8 cores de 1.4 GHz, sendo capaz de executar 8 threads de hardware por core. Isso foi possível colocando uma segunda unidade de execução (EXU) em cada core. Assim, é possível que mais de uma thread esteja em execução simultaneamente em cada core, desde que todas as partes requeridas pela instrução estejam disponíveis em dobro no pipeline (a unidade de load/store, por exemplo, não está, logo duas instruções de load/store simultâneas não são possíveis).

Cada core matém o padrão da cache L1 do Niagara (8KB de data cache, 16KB de instruction cache), e a cache L2 aumentou para 4MB. Além disso, há uma floating point unit por core, corrigindo (ou amenizando) um dos maiores defeitos do Niagara. Também é possível desligar cores ou threads de um core, a fim de economizar energia.

A Figura 7 compara o desempenho do Niagara a outros processadores. Pode-se observar que o processador supera de modo bastante expressivo o desempenho das demais arquiteturas para os benchmarks: SPECJBB05, SPECWeb05 e TPC. No caso do SPECWeb05, uma das justificativas para uma diferença de desempenho tão significativa é que Solaris é bastnate otimizado para Java (linguagem de programação utilizada neste benchmark). Entretanto, devido as limitações de operações de ponto flutuante, apreseta desempenho sofrível no SPECFPRate.

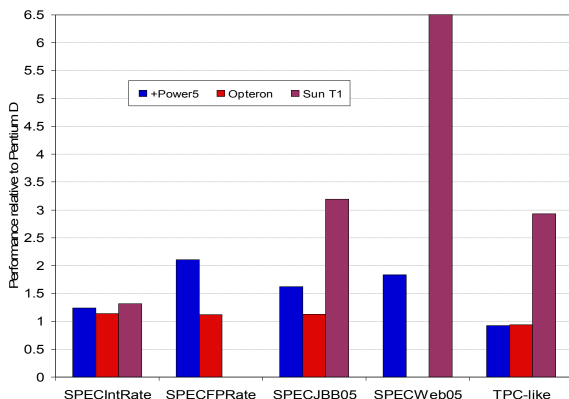


Figura 7. Comparação de desempenho utilizando benchmarks SPEC e TPC, comparando o Niagara ao Power 5 e ao Opteron.

Comparando o Niagara ao Xeon (MultiThreading), considerando o contexto de aplicação servidores web, para grande dos casos o Niagara é o mais indicado, conforme apresentado na Figura 8.

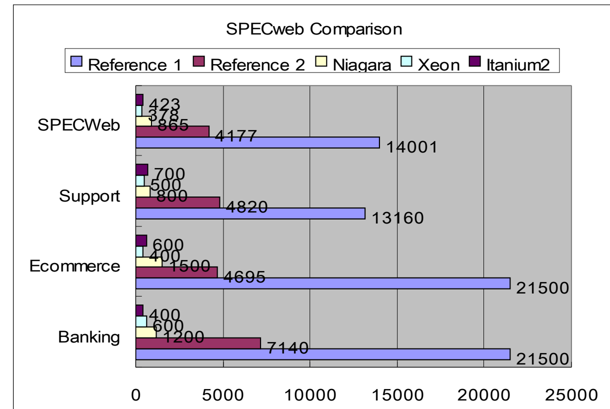


Figura 8. Comparação de desempenho utilizando benchmarks SPECWeb. A referência 1 consiste em um Niagara (1.2GHz CPU, Sun Java Web Server 6.1 + JSP) e a referência 2 é umXeon (2x3.8GHz CPU w/ HT, 2MB L2, Zeus + JSP)

.

4. APLICAÇÕES

Há diversos trabalhos que descrevem a eficiência ou a limitação na aplicação de tecnologias multi-threading a diversos contextos. Estudos demonstram restrições da aplicação de paralelismo em nível de thread em programas não numéricos [14] e outros discutem o real ganho em utilizar TLP em aplicativos Desktop [15], por exemplo.

Uma das principais motivações para a TLP foi melhorar o desempenho de servidores, como descrito anteriormente e exemplificado com o Niagara da Sun. Entretanto, há outras aplicações onde a tecnologia também provém ganhos. Elas são abordadas a seguir.

4.1 Aplicativos Desktop

Atualmente os multiprocessadores estão sendo amplamente comercializados em computadores de propósito geral. Entretanto, questiona-se até que ponto a TLP é aproveitada na execução de aplicações que exigem interatividade com o usuário, em oposição a servidores web.

Estudos [15] apresentam que usar dois processadores ao invés de um é uma forma de reduzir a duração de execução de aplicativos do dia-a-dia de usuários comuns de computadores. A TLP permite, em média, redução em 22% do tempo de execução (resultados variam entre 8% e 36%). Ainda, estas melhorias correspondem 16%-72% da máxima redução alcançavel por processadores dual-core (50%)[15]. Ou seja, utilizar dois processadores e TLP pode ser um meio eficiente e eficaz para melhorar o desempenho mesmo de aplicativos baseados na interatividade com o usuário.

Utilizar uma arquitetura Dual Core para executar um player de MP3 em background pode aumentar o tempo de resposta em 29%. Apesar do segundo processador eliminar a maior parte do overhead de tarefas em background, ele não absorve tais tarefas completamente.

Um aspecto importante apresentado nestes estudos é que não é vantajoso utiilizar mais de dois proessadores, para a maior parte das aplicações testadas.

Para os testes foi utilizado GNU Linux, o qual foi apresentado como uma plataforma SMP. Acredita-se que removendo o lock global do kernel seria possível melhorar ainda mais o paralelismo em nível de threads, mas que o maior pontencial é na reescrita de aplicativos visando o processamento multi-threading.

4.2 Aplicativos multimidia

Em contraposição a utilização de TLP em aplicações não adaptadas, como apresentado previamente, há testes de implementações específicas com suporte a TLP visando atender diversas aplicações, tais como codificação de video, processamento de imagens e de audio.

A literatura [16] apresenta que é possível reduzir drasticamente o tempo de execução de um codificador de MPEG2 utilizando TLP, onde são relatadas reduções de 96% para a utilização de 32 processadores (contextos) e uma faixa de 80 quadros ou imagens.

5. DESAFIOS

5.1 Paralelismo na Camada de Software

Para que se aproveite os benefícios propiciados pela TLP, é importante que existam programadores habilitados e que saibam usufruir da arquitetura.

A TLP pode ser aplicada em divrsos níveis de camada de software. A seguir são apresentadas algumas possíveis abordagens.

5.1.1 Sistema Operacional

Ao invés de todos os softwares terem que se preocupar com o paralelismo, a resposabilidade pode ser levada ao sistema operacional. Considerando um servidor que possui múltiplos cores, e que cada um pode rodar várias threads. Qual seria o melhor modo de aproveitar esta arquitetura para executar quatro threads simultâneas? Tudo depende do funcionamento delas. Se elas compartilharem memória, o ideal é que estejam próximas, caso contrário podem estar distantes, de modo a aproveitar o máximo de recursos independentes possíveis. Ainda considerando a memória, o programa de gerenciar o tráfego memória para CPU é bastente complexo neste tipo de proesador. Este tipo de problema, claramente, deve ser tratado pelo SO, e não pelas aplicações de nível mais alto.

Os sistemas operacionais recentes já oferecem suporte a multithreading (Unix, Solaris, Linux e Windows). Entretanto, ainda são necessários mais estudos e implementações para adequação dos OS existentes. Talvez uma refatoração bastante séria seja necessária.

Grande parte das aplicações é serial, não sendo vantajoso o recurso de TLP obeservando-as individualmente. Entretanto, os sistemas operacionais podem distriutir um conunto deste tipo de aplicação para ser executado paralelamente, lidando com as threads. Bons resultados já são obtidos com esta abordagem.

5.1.2 Compiladores

Hoje alguns compiladores, como o GCC, permitem a geração de código de máquina otimizado para suporte multi-threading, bastando para isso passar informações específicas na hora de compilar o programa.

Esta abordagem é intimamente dependente da linguagem, do compilador e do sistema operacional.

5.1.3 Softwares de infraestrutura

Aplicativos como banco de dados e web-servers tendem a ser projetados para dar suporte a threads, pois de um modo geral eles precisam oferecer alto desempenho.

Apache 2.0, por exemplo, foi estruturado de modo bastante inteligente, permitindo que sejam executados vários processos contendo algumas threads, ou poucos processos com várias threads. Entretanto, o Apache tende a ser reestruturado conforme programadores forem aprendendo a usufruir desta tecnologia, e novas demandas surgirem.

5.1.4 Camada da Aplicação

A princípio, por definição, todos os aplicativos de alto processamento de dados irão poder usufruir do paralelismo para obter melhores resultados. Softwares de processamento de imagens, videos, cálculos para física e matemática tem potencial para aproveitar esta tecnologia. Entretanto, um limitante é que programação baseada em threads é complicada.

Para se aproveitar ao máximo os chips TLP, é necessário uma grande evolução nas ferramentas de desenvolvimento, depuração e teste de software. Para bons programadores, é simples realizar "unit-tests" na maior parte das linguagens (JUnit aos programadores Java) para aplicações que rodam sobre uma única thread. Entretanto, quando o programa passa a ser paralelo, há um problema pois é difícil estabelecer um framework mental de como prever testes com múltiplas threads. Primeiro seria importante definir *design patterns*, para que então fosse possível desenvolver ferramentas de auxílio a testes e depuração.

Neste aspecto, a escolha da linguagem de programação pode afetar o resultado. É importante que a linguagem escolhida tenha um bom suporte para código paralelo multi-thread, facilitando que problemas de paralelismo, tal como a disputa de recursos, sejam encontrados.

5.2 Desafios e abordagens de hardware

Multiprocessadores é uma área ampla e diversa, sendo que grande parte dos desenvolvimentos são bastante recentes. Até recentemente havia mais casos de fracassos do que sucessos na área.

Um dos desafios em multiprocessadores é definir qual abordagem é mais adequada: processadores simétricos ou assimétricos. Ao longo deste artigo abordamos apenas arquiteturas simétricas, mas há implementações, como o Cell Hypervisor, na qual há 8 cores de uma categoria e um núcleo mais poderoso (Power) que repassa tarefa para os demais processadores.

Outras questões que merecem destaque [1] são o custo de comunicação entre processsadores e a troca de dados. Ambas são dependentes das arquiteturas adotadas – seja de comunicação entre processadores, seja da hierarquia de memória empregada. Em ambas situações surge latência e existem uma série de abordagens para tentar contornar tais problemas. Entretanto, estamos longe de termos achado uma solução definitiva.

6. CONCLUSÃO

Neste artigo apresentamos a importante abordagem de paralelismo a nível de thread, que tem sido amplamente empregada em processadores comerciais nos últimos anos. Foram analisadas as abordagens de implementação SMT e CMT.

Foi possível estudar como grandes fabricantes de processadores implementaram multithreading em suas arquiteturas, sendo que foram analisadas as tecnologias HyperThreading da Intel e a Niagara da Sun, exemplos comerciais das tecnologias SMT e CMT, respectivamente.

Observou-se que a eficiência da utilização destas tecnologias a servidores com alta carga de trabalho, em especial em circustâncias onde há níveis elevados de TLP e níveis relativamente baixos de ILP. Ainda assim constatou-se que o uso efetivo de TLP ainda está intimamente relacionado às implementações das aplicações.

Sem dúvidas o paralelismo em nível de threads e as arquiteturas multicore são uma grande oportunidade de pesquisa, mas também trazem consigo desafios, tanto na esfera de hardware quanto para o desenvolvimento de software.

7. REFERÊNCIAS

[1] Hennessy, J. L. and Patterson, D. A. 2007. Computer Architecture: A Quantitative Approach..Morgan Kaufmann Publishers, Inc. San Mateo, CA. Forth edition, 1995.

[2] Mitchell, N., Carter, L., Ferrante, J., Tullsen, D. 1999. ILP versus TLP on SMT. Supercomputing ACM/IEEE 1999 Conference. (Nov. 1999), 37-37.

[3] Flynn, M. 1972. Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948.

[4] Tullsen, D. M.; Egger S., and Levy, H. M. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. ISCA-22*, 1995.

[5] Kalla, R., Sinharoy B., and J. Tendler. 2003. Power5: Ibm's next generation power microprocessor. In *Proc. 15th Hot Chips Symp*, pages 292–303, August 2003.

[6] Marr D. T., Binns F., Hill D. L., G. Hinton, Koufaty D. A., Miller J. A. , and Upton M. 2002. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, Feb. 2002.

[7] Skadron Y. L., Brooks K., Zhigang Hu D. 2005. Performance, energy, and thermal considerations for SMT and CMP architectures. *High-Performance Computer Architecture*, 2005. HPCA-11. 11th International Symposium , pages 71- 82, February 2005.

[8] Heo S., Barr K., and Asanovic K.. 2003. Reducing power density through activity migration. In *Proc. ISLPED'03*, Aug. 2003.

[9] Marr D., Binns F., Hill D., Hinton G., Koufaty D., Miller J. Upton M. 2002. Hyper-Threading Technology Architecture and Microarchitecture.ntel Technology Journal, vol.3, issue 1, ITJ 2002

[10] Nagarajayya, N. 2005. Improving Application Efficiency Through Chip Multi- Threading. DOI = http://developers.sun.com/solaris/articles/chip_multi_thread.html. Acessado 15/06/2009.

[11] Kanter, D. 2005. Niagara II – The Hydra Returns. DOI = http://www.realworldtech.com/page.cfm?ArticleID=RWT090406012516. Acessado 15/06/2009.

[12] De Gelas, J. 2005. SUN's UltraSparc T1 - the Next Generation Server CPUs. DOI = http://www.anandtech.com/printarticle.aspx?i=2657 Acessado 15/06/2009.

[13] Shankland, S.2006. Sun doubles thread performance with Niagara 2. DOI = http://news.zdnet.co.uk/hardware/0,1000000091,39281614,00.htm Acessado 15/06/2009.

[14] Nakajima A., Kobayashi R., Ando H. and Shimada T. 2006. Limits of Thread-Level Parallelism in Non-numerical Programs. IPSJ Digital Courier, Volume 2. Pages 280-288. May 2006.

[15] Flautner, K., Uhlig, R., Reinhardt, S., and Mudge, T. 2000. Thread-level parallelism and interactive performance of desktop applications. SIGPLAN Not. 35, 11 (Nov. 2000), 129-138. DOI= http://doi.acm.org/10.1145/356989.357001

[16] Jacobs, T.R. Chouliaras, V.A. Mulvaney, D.J. 2006. Thread-parallel MPEG-2, MPEG-4 and H.264 video encoders for SoC multi-processor architectures. Transactions on Consumer Electronics, IEEE. Volume 52, Issue 1. Pages 269-275. Feb. 2006

Loop Restructuring e Vetorização

Bruno Cardoso Lopes
RA 023241
Instituto de Computação, Unicamp
Campinas, São Paulo
bruno.cardoso@gmail.com

ABSTRACT

Compiladores que geram código vetorial existem desde os antigos computadores Cray. A abordagem de vetorização é recorrente hoje em dia com as arquiteturas modernas e seus conjuntos de extensões multimídia. Este artigo introduz as arquiteturas vetoriais e em seguida apresenta um histórico de diversas técnicas de re-escrita de loops como base para a apresentação das técnicas de vetorização utilizadas na área de compiladores nos últimos 10 anos; vetorizações em laços e geração de SIMDs através da detecção de *Superword Level Paralelism* (SLPs).

General Terms

Compilers architecture parallelism

1. INTRODUCTION

Transformações em loops fazem parte das principais áreas de pesquisa de otimização em compiladores, e também dos principais enfoques da área de vetorização de código. As técnicas de vetorização são aplicadas como passos de transformações em compiladores e tentam descobrir operações vetoriais automaticamente. A aplicação destas técnicas só é possivel através das informações obtidas com as transformações nos loops.

Processadores vetoriais utilizam instruções SIMD (Single Instruction, Multiple Data) operando em vários fluxos de dados com uma única instrução. Assim, é possível com uma única instrução realizar operações sobre todos elementos de um vetor de uma única vez, como um incremento de um em todos os elementos desse vetor de maneira atômica.

Afim de suportar operandos fonte e destino vetoriais, estes processadores possuem registradores vetoriais, com tamanhos dependentes de implementação. O uso adequado desses registradores leva a ótimos desempenhos.

Desde 1994 processadores de propósito geral vêm sendo fabricados com extensões de multimídia. O pioneiro foi o MAX-1 fabricado pela HP, e em 1997 chegaram massivamente no mercado com o lançamento do Pentium MMX, o primeiro processador da Intel com extensões de multimídia.

Processadores vetoriais possuem grande similaridade com as extensões multimídia dos processadores de propósito geral - ambos conjuntos de instruções são SIMD. Por causa dessa semelhança, os mecanismos tradicionais de vetorização podem também ser aplicados a estas extensões.

Extensões de multimídia são os conjunto de operações vetoriais encontrados nos dias de hoje, portanto, todas as referências a códigos vetoriais serão feitas com este foco.

2. CÓDIGO VETORIAL

Instruções vetoriais geralmente possuem dois operandos vetoriais, ou um operando vetorial e um escalar. Um vetor pode ser definido pela posição de início, a distância (passo ou *stride*) entre os elementos do vetor e tamanho do vetor (em número de elementos). As duas primeiras propriedades estão geralmente implícitas nos próprios registradores, e o tamanho em algum registrador especial.

2.1 Motivação

Algumas operações utilizam muito menos bits do que os disponíveis em hardware. Operações com cores representam bem o espaço de operações realizadas com menos bits do que o necessário, supondo que um canal de cor (RGB) ocupa 8 bits:

Figure 1: Soma Vetorial

Soma A soma de dois canais, utiliza apenas 8 bits das fontes e destino, não necessitando do tamanho inteiro da palavra (figura 1).

Saturação Incremento de um em um canal com valor 255, permanece 255 se a soma for saturada.

Alem da subutilização de bits, em um caso comum é necessário mais do que uma instrução para realizar operações sobre um conjunto de pixels, sendo ainda pior se for necessário acesso freqüente a memória para realizar estas operações.

Uma maneira eficiente em hardware de realizar essas operações utiliza vetorização. Se cada canal for mapeado para uma posição de um registrador vetorial, é possível realizar somas saturadas entre registradores vetoriais, e efeitos de overflow são considerados apenas para cada canal. Assim, é possível realizar operações sobre vários conjuntos de pixels com apenas uma instrução.

Vários outros tipos de operações vetoriais também podem ser realizadas otimizando a execução em hardware, operações horizontais e empacotamento são duas bastante comuns :

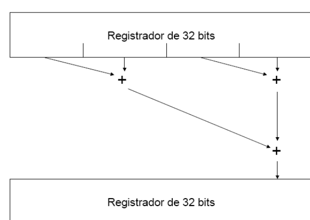


Figure 2: Soma Horizontal

Horizontalidade Realiza a soma de todos os elementos de um vetor e armazenada o resultado em um escalar. (figura 2).

Empacotamento Converte dados da memória ou escalares para o formato vetorial - no exemplo de cores, cada canal RGB do pixel pode ser colocado em uma posição diferente do vetor.

2.2 Conjuntos de Instruções

Vários fabricantes acoplam extensões de multimídia no seus conjuntos de instruções. A tabela 1 mostra algumas das várias arquiteturas e suas extensões multimídia.

intel x86	MMX, SSE/2/3, SSSE3, SSE4.1/4.2 AVX[1]
amd x86	3DNow, SSE/2/3, SSSE3, SSE4a, SSE5[2]
cell	SPU ISA
mips	MIPS-3D ASE
powerpc	Altivec

Table 1: Características da Série 8

3. VETORIZAÇÃO MANUAL

A maneira mais comum de aplicação de vetorização em códigos é através da utilização explícita de código vetorial. Isto pode ser feito de 2 formas:

1. Inserção da codificação de rotinas ou trechos de código em assembly. Estas rotinas são invocadas através de linguagens como C através de símbolos definidos externamente, de maneira que em tempo de link edição, o endereço real destas rotinas é conhecido. Outra abordagem é com a utilização de inline assembly diretamente dentro de rotinas em C.
2. Utilização de intrinsics do compilador. O Compilador pode disponibilizar funções intrinsics genéricas que são convertidas para código vetorial de qualquer arquitetura (desde que a mesma possua extensões multimídia), ou disponibiliza intrinsics específicos para cada arquitetura.

Em ambas abordagens, o compilador deve ter um suporte básico as instruções vetoriais. O trecho de código 1 exemplifica a utilização de intrinsics.

Trecho 1 Exemplo de utilização de intrinsics

```
#define ALIGN16 __attribute__((aligned(16)))

__m128 a, b, c;
float inp_sse1[4] ALIGN16 = { 1.2, 3.5, 1.7, 2.8 };
float inp_sse2[4] ALIGN16 = { -0.7, 2.6, 3.3, -4.0 };
...
a = _mm_load_ps(inp_sse1);
b = _mm_load_ps(inp_sse2);
c = _mm_add_ps(a, b);
```

4. VETORIZAÇÃO AUTOMÁTICA

A maneira mais conhecida e pesquisada de vetorizar aplicações é através da vetorização de laços[4]. Muitas vezes os laços apresentam oportunidades ideais onde o código vetorial pode substituir uma laço inteiro. Mas antes de chegar nos laços ideais, é necessário realizar diversas análises e transformações nos mesmos para obter a vetorização. Estas análises ocorrem na árvore de representação intermediária gerada pelo compilador, e todas dependem de uma análise base; a análise de dependência de dados.

4.1 Dependência de dados

A análise de dependência permite ao compilador maiores oportunidades de rearranjo de código, o programa é analisado para achar restrições essenciais que previnem o reordenamento de operações, sentenças, ou iterações de um loop.

Os três tipos de dependência utilizados para esta análise são:

- Flow ou Direta, quando uma variável é definida em uma sentença e utilizada em uma subseqüente.
- Anti, usada em uma sentença e definida em uma subseqüente.
- Output, definida em uma sentença e redefinida em uma subseqüente.

As dependências podem ser classificadas como *loop carried*, quando ocorrem entre iterações de um loop ou *loop independent* caso contrário. Grafos de dependência podem ser construídos para facilitar a visualização e implementação.

[1]Advanced Vector Extensions, fazerá parte do novo processador de 32nm Sandy Bridge, com previsão de lançamento em 2010

[2]Previsto para 2010 no processador bulldozer

Trecho 2 Programa Simples

```
(1) a = 0
(2) b = a
(3) c = a + d
(4) d = 2
```

No trecho de código 2 temos uma dependência de fluxo entre as sentenças 1-2 e 1-3 e uma anti-dependência entre 3-4 (figura 3).

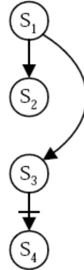


Figure 3: Grafo de dependência

5. VETORIZAÇÃO DE LAÇOS

Compiladores antigos tinham poucas regras sobre os tipos de construções para os quais eles poderiam gerar código. Isto ocorreu até o momento em que pesquisadores começaram a usar grafos de dependência para encontrar laços vetorizáveis[1].

Apenas laços seqüenciais com dependência acíclicas podem ser vetorizados, uma vez que a execução vetorial preserva relação de dependência entre sentenças que aparecem uma após a outra seqüencialmente no código fonte. Assim, boa parte do foco para vetorização auxilia na quebra dessas dependências.

5.1 Strip-mining

Cada arquitetura possui seu próprio tamanho de registradores vetoriais. Com o objetivo de aproveitar o tamanho desses registradores a técnica de *strip mining* pode ser utilizada. O *strip mining* decompõe um único laço em novos 2 laços não-aninhados; o segundo laço (chamado de *strip loop*) caminha entre passos de iterações consecutivas enquanto o primeiro caminha entre iterações únicas em um passo.

Trecho 3 Redução em Fortran

```
forall i = 1 to N do
  a[i]  = b[i] + c[i]
  ASUM += a[i]
```

O código 4 é a versão vetorizada do código 3. Após a aplicação de strip mining, o *strip loop* caminha entre passos de tamanho 64, uma vez que este é o tamanho do registrador vetorial para o exemplo.

Os trechos de código 3 e 4 são exemplos de redução de vetores de dados, operação bastante comum em códigos multimídia. Se a arquitetura suportar instruções de redução então o compilador pode utilizar estas instruções diretamente,

Trecho 4 Redução sem o intrinsic *forall*

```
v4 = vsub v4, v4
for i = 1 to N by 64 do
  VL = max(N-i+1, 64)
  v1 = vfetch b[i]
  v2 = vfetch c[i]
  v3 = vadd v1, v2
  vstore v3, a[i]
  v4 = vadd v3, v4
for i = 0 to min(N-1,63)
  ASUM = ASUM + v4[i]
```

basta tomar o cuidado de acumular os resultados de vários passos e depois reduzir novamente. Se a arquitetura não possuir estas instruções, ainda existe uma abordagem mais eficiente do que um loop seqüencial para resolver o problema.

5.2 Analise de Condicionais

Um dos problemas durante a vetorização é o tratamento de sentenças condicionais. Uma das maneiras de resolver é utilizar vetor de bits, afim de armazenar o resultado de operações de comparação para vetores. Com o vetor de bits calculado, instruções com predicado podem ser utilizadas junto a esse registrador, podendo gerar fluxo condicional sem branches.

Trecho 5 Loop com condicional

```
forall i = 1 to N do
  a[i] = b[i] + c[i]
  if a[i] > 0 then
    b[i] = b[i] + 1
```

O trecho 5 contêm uma verificação que pode resolvida pelo compilador com um código resultante semelhante ao do código 6.

Trecho 6 Código vetorial condicional

```
for i = 1 to N by 64 do
  VL = max(N-i+1, 64)
  v1 = vfetch b[i]
  v2 = vfetch c[i]
  v3 = vadd v1, v2
  vstore v3, a[i]
  m1 = vcmp v3, r0
  r2 = addiu r0, 1
  v1 = vaddc v1, r2, m1
  vstorec v1, b[i], m1
```

Outras abordagem podem ser também adotadas, se houver apenas um vetor de bits, a condição pode estar implícita na própria instrução. No caso de não existirem store condicionais, ambos caminhos podem ser gerados, conseguindo ainda assim aplicar a vetorização.

5.3 Expansão de escalares

A presença de escalares dentro de um laço gera ciclos no grafo de dependência (existe no mínimo a dependência de saída), isso ocorre tanto para variáveis de indução quanto para escalares temporários. Para ser possível vetorizar é necessário remover estes ciclos, removendo as váriaveis de indução e escalares.

As variáveis de indução são removidas por substituição, e os escalares podem ser expandidos. A expansão de escalares é realizada guardando o escalar em um registrador vetorial, assim cada posição do registrador deverá conter o valor do escalar em diferentes iterações, não havendo necessidade de utilizar a memória.

5.4 Dependências cíclicas

Laços com ciclos de dependência são mais complicados de vetorizar e várias abordagens podem ser aplicadas para extrair o que for possível de vetorização. A primeira e mais direta abordagem com dependência cíclica se trata de uma simples otimização; Para ciclos com dependência de saída, o ciclo pode ser ignorado se o valor sendo guardado for invariante no loop e igual em todas as sentenças.

5.5 Loop fission

Aplicando *loop fission* é possível vetorizar laços parcialmente. Isso é feito separando o laço em dois novos laços, um deles conterá o trecho de código com a dependência cíclica e o outro poderá ser vetorizado. Considere o trecho de código 7 e o grafo de dependência da figura 4:

Trecho 7 Dependência cíclica

```
for i = 1 to N do
  a[i] = b[i] + c[i]
  d[i+1] = d[i] + a[i]*c[i]
  c[i+1] = c[i+1]*2
```

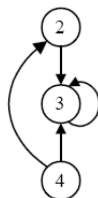


Figure 4: Grafo de dependência

Temos um laço com dependência cíclica na terceira sentença. O trecho 8 contêm o *loop fission* necessário.

Trecho 8 Loop fission

```
for i = 1 to N do
  c[i+1] = c[i+1]*2
  a[i] = b[i] + c[i]
for i = 1 to N do
  d[i+1] = d[i] + a[i]*c[i]
```

Quando há a oportunidade de se aplicar loop fusion porem existe um temporário que ficaria em loops diferentes, como o código 9, pode se aplicar algo semelhante à expansão de escalares, mas dessa vez um registrador vetorial não será suficiente, é necessário alocação na memória. Para contornar isso, se aplica *strip mining* antes do *loop fission*, o resultado pode ser visto no trecho de código 10.

Outro impedimento que pode ocorrer dificultando a vetorização em laços é a presença de condicionais. Podem aparecer dois tipos de condicionais; as que estão no ciclo de dependência e as que não estão. No primeiro caso é necessário

Trecho 9 Dependência cíclica com temporário

```
for i = 1 to N do
  TEMP = b[i] + c[i]
  d[i+1] = d[i] + TEMP * c[i]
  c[i+1] = c[i+1] * 2
```

ter suporte especial em hardware para resolver recorrência booleana, ou seja, inviável. No segundo caso, deve-se colocar a condição em um registrador escalar, expandi-lo e vetorizar como explicado algumas seções atrás.

Trecho 10 Loop fission e strip mining aplicados

```
for i = 1 to N by 64 do
  for v = 0 to min(N-I, 63) do
    a[v] = b[i+v] + c[i+v]
    c[i+v+1] = c[i+v+1]*2
  for v = 0 to min(N-I, 63) do
    d[i+v+1] = d[i+v] + a[v]*c[i+v]
```

5.6 Falsos ciclos

Existem vários truques que podem ser aplicados que permitem a vetorização independente da presença de ciclos. Diversos ciclos existem devido a anti-dependências, principalmente quando essa relação é da sentença consigo mesma. Isso ocorre devido a granularidade de visualização do grafo de dependência (essa anti-dependência não existe de fato). Considere o trecho de código 11:

Trecho 11 Falsos ciclos

```
(1) for i = 1 to N do
(2)  a[i] = a[i+1] + b[i]
```

A figura 5 ilustra a esquerda o grafo de dependência com alta granularidade e a direita o com baixa.

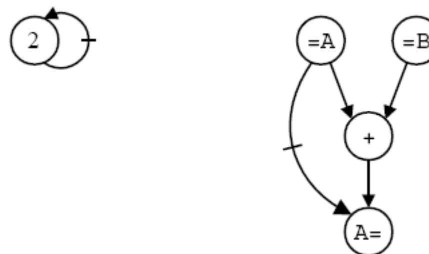


Figure 5: Diferentes granularidades

Refinando a granularidade o compilador consegue eliminar dependências, uma vez que elimina as dependências falsas geradas por uma granularidade alta. A vetorização pode ser conseguida aplicando-se uma ordenação topológica para reordenar o grafo de baixa granularidade, e garantindo que no código gerado em da 12, o fetch de $a[i+1]$ ocorra antes do store de $a[i]$ preservando a relação de anti-dependência.

Na presença de ciclos verdadeiros, o compilador pode vetorizar expressões parciais, basta decidir se o esforço compensa, e caso sim, vetorizar o que for possivel, deixando o resto como estava.

Trecho 12 Sem falsos ciclos

```
for i = 1 to N by 64 do
  VL = min(N-i+1, 64)
  v1 = vfetch a[i+1]
  v2 = vfetch b[i]
  v3 = vadd v1, v2
  vstore v3, a[i]
```

5.7 Dependência cruzada

A técnica de *Index set splitting* pode ser utilizada quando ocorre dependência cruzada, ou seja, um dos operando vetoriais contem um índice crescente e algum outro, decrescente. Suponha trecho de código 13:

Trecho 13 Dependência cruzada

```
for i = 1 to N do
  a[i] = b[i] + c[i]
  d[i] = (a[i] + a[n-i+1])/2
```

O compilador deve descobrir onde esses índices se encontram e separar o conjunto de índices neste ponto. Se aplicarmos esta técnica no código acima, obtemos o código 14, onde ambos os loops podem ser vetorizados.

Trecho 14 Laços resultantes

```
(1) for i=1 to (n+1)/2 do
      a[i] = b[i] + c[i]
      d[i] = (a[i] + a[n-i+1])/2
(2) for i=(n+1)/2+1 to N do
      a[i] = b[i] + c[i]
      d[i] = (a[i] + a[n-i+1])/2
```

5.8 Dependência em tempo de execução

Quando não se conhece um dos índices de acesso a uma posição de um array durante o tempo de compilação não se sabe a direção da dependência (código 15).

Trecho 15 k desconhecido em tempo de compilação

```
for i = 1 to N do
  a[i] = a[i-k] + b[i]
```

Uma maneira de resolver o problema, é fazer com que o compilador gere 2 versões do laço, a primeira para valores $0 < k < N$ e a outra caso contrário (código 16). O laço da condição (1) pode ser vetorizado posto que não há dependência de fluxo e dependência cíclica, já (2) não pode ser vetorizado.

Trecho 16 Dois novos laços

```
(1) if k <= 0 or k >= N then
      forall i=1 to N do
        a[i] = a[i-k] + b[i]
(2) else
      for i=1 to N do
        a[i] = a[i-k] + b[i]
```

5.9 Casos mais simples

Na presença de laços aninhados, se o grafo de dependência não conter ciclos, todos os laços podem ser paralelizados, a transformação fica equivalente a uma atribuição de arrays. O trecho de código 17 é transformado no código 18.

Trecho 17 Laços aninhados

```
for i = 1 to N do
  for j = 2 to M do
    a[i,j] = b[i,j-1] + c[i,j]
    b[i,j] = b[i,j]*2
```

Trecho 18 Codigo vetorizado

```
b[1:n,2:m] = b[1:n,2:m]*2
a[1:n,2:m] = b[1:n,1:m-1] + c[1:n,2:m]
```

5.9.1 loop interchanging

Quando um dos laços possuir ciclos de dependência, pode-se aplicar *loop interchanging*. Um exemplo simples dessa técnica é apresentado no trecho 19, o laço original (1) é tranformado em (2). *Loop interchanging* troca a ordem dos laços e pode levar a dependência para o laço de fora, quebrando ciclos. Esta técnica permite escolher o melhor laço para vetorização.

Trecho 19 Exemplo de *Loop interchanging*

```
(1) do i = 1,n
      do j = 1,m
        do k = 1,p
          C[i,j] = C[i,j] + A[i,k] * B[k,j]
(2) do j = 1,m
      do k = 1,p
        do i = 1,n
          C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

5.9.2 Loop collapsing

Trecho 20 Laço em Fortran

```
Real A(100,100), B(100,100)
do I = 1, 100
  do J = 1, 90
    A(I,J) = B(I,J) + 1
  enddo
enddo
```

Laços vetorizados são eficientes quando os limites do laço são largos. Afim de aumenta-los, a técnica de *loop collapsing* pode ser aplicada unificando os laços em apenas um, com um tamanho bastante longo. Os trechos de código 20 e 21 mostram respectivamente o código original e o resultado alcançado utilizando *loop collapsing*.

Trecho 21 Laço em Fortran após *loop collapsing*

```
Real A(100,100), B(100,100)
Real AC(10000),  BC(10000)
Equivalence (A,AC),(B,BC)
do IJ = 1, 9000
  AC(IJ) = BC(IJ) + 1
enddo
```

6. SLP : SUPERWORD LEVEL PARALELISM

Apesar das tecnologias de vetorização explicadas na última seção serem bem entendidas, elas são complexas e frágeis (muitas funcionam apenas em casos muito específicos). Outro ponto negativo é que elas são incapazes de localizar paralelismo do tipo SIMD em blocos básicos. SLP[2] é uma técnica que não realiza extração de vetorização através de paralelismo de laços, ao invés de ter laços como alvo, ela ataca blocos básicos.

SLP é um tipo de paralelismo onde os operandos fontes e destino de uma operação SIMD são empacotados em uma mesma unidade de armazenamento. A detecção é feita com a coleta de sentenças isomórficas em um bloco básico, entenda-se por isomórficas as sentenças que possuem as mesmas operações na mesma ordem. Estas sentenças são empacotadas em uma mesma operação SIMD. A figura 6 ilustra um exemplo de como funciona este empacotamento.

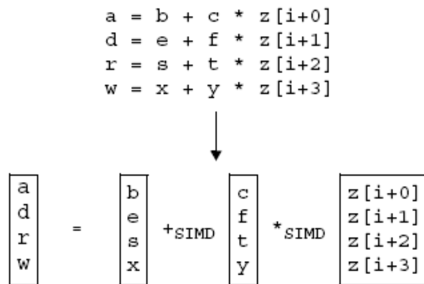


Figure 6: Sentenças isomórficas compactadas

6.1 Notações

- Pack é uma tupla que contem sentenças isomórficas independentes em um bloco básico.
- PackSet é um conjunto de Packs.
- Pair é um Pack de tamanho 2, onde a primeira sentença é considerada o elemento esquerdo (left element) e o outro elemento direito (right element)

6.2 Algoritmo

O Algoritmo de detecção e geração de SLPs, é feita executando-se várias otimizações em determinada ordem. Primeiro, loop unrolling é usando para transformar paralelismo de vetores em SLP (figura 7 e 8), em seguida alignment analysis tenta determinar o endereço de alinhamento de cada instrução de load e store (e os anota para posterior utilização).

```
for (i=0; i<16; i++) {
  localdiff = ref[i] - curr[i];
  diff += abs(localdiff);
}
```

Figure 7: Laço original

Após estas computações, todas as otimizações comuns devem ser executadas, terminando com scalar renaming (removendo dependências de entrada e saída que podem proibir a paralelização) - representações intermediárias que utilizam SSA não necessitam deste último passo.

Os primeiros candidatos a empacotamento são as referências adjacentes de memória, ou seja, acessos a arrays como os da figura 8. Cada bloco básico é analisado em busca de tais sentenças, a adjacência é determinada utilizando-se as informações anotadas de alinhamento e análise de arrays. A primeira ocorrência de cada par de sentenças com acessos adjacente de memória é adicionado ao PackSet.

```
for (i=0; i<16; i+=4) {
  localdiff0 = ref[i+0] - curr[i+0];
  localdiff1 = ref[i+1] - curr[i+1];
  localdiff2 = ref[i+2] - curr[i+2];
  localdiff3 = ref[i+3] - curr[i+3];

  diff += abs(localdiff0);
  diff += abs(localdiff1);
  diff += abs(localdiff2);
  diff += abs(localdiff3);
}
```

Figure 8: SLP alcançado após unrolling e renaming

Com o PackSet inicializado mais grupos podem ser adicionados, isso é feito achando-se mais candidatos. Estes devem cumprir os seguintes requisitos:

- Produzirem operandos fonte para outras sentenças na forma empacotada.
- Utilizar dados já empacotados como operandos.

Estes candidatos podem ser escolhidos varrendo os conjuntos def-use e use-def dos elementos já presentes no PackSet. Se a varredura encontrar sentenças novas que podem ser empacotadas, elas são incorporadas ao PackSet se comprarem as seguintes regras :

- As sentenças são isomórficas e independentes.
- A sentença à esquerda ainda não é esquerda de nenhum par, o mesmo vale para direita
- Informação de alinhamento consistente
- O tempo de execução da nova operação SIMD tem que ser estimadamente menor do que a versão seqüencial.

Quando todos os pares escolhidos segundo as regras acima são escolhidos, eles podem ser combinados em grupos maiores. Dois grupos podem ser combinados quando a sentença esquerda de um é igual à sentença direita de outro (prevenindo também a repetição de sentenças).

A análise de dependência antes do empacotamento garante que todas as sentenças em um grupo podem ser executadas em paralelo de maneira segura. No entanto, apesar de bastante raro, dois grupos podem produzir uma violação de dependência (através de ciclos). Se isso ocorrer, o grupo contendo a sentença mais recentemente não escalonada, é dividido durante a etapa de escalonamento.

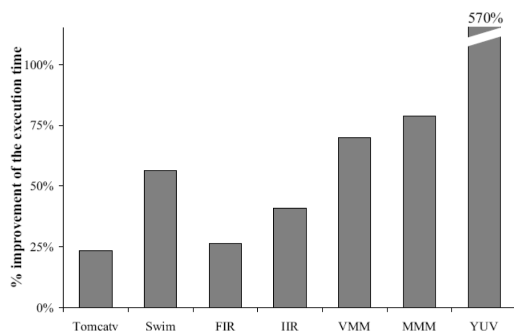


Figure 9: Ganho de desempenho utilizando SLP

As figuras 9 e 10 contem dados de desempenho da utilização de SLP. O processador utilizado chama-se SUIF[3] e o alvo utilizado foi o microprocessador da Motorola MPC7400 com extensões multimídia Altivec.

Benchmark	Speedup
swim	1.24
tomcatv	1.57
FIR	1.26
IIR	1.41
VMM	1.70
MMM	1.79
YUV	6.70

Figure 10: Speedup utilizando SLP

7. CONCLUSÃO

Vetorização foi o primeiro método de automaticamente encontrar paralelismo em laços seqüenciais. Sua utilização foi freqüente em maquina vetoriais como os Crays e hoje tem voltado na forma de extensões multimídia nos processadores modernos. As técnicas mais comuns durante muito tempo foram as baseadas em laços, e técnicas mais novas, como SLP, também tentaram endereçar o problema como uma abordagem diferente. Muitos compiladores modernos ainda não tem suporte a vetorização, e apesar de ser uma área já bastante investigada, a implementação de mecanismos de vetorização em compiladores é uma tarefa árdua mas tem historicamente obtido ótimos desempenhos.

8. REFERENCES

[1] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *Int. J. Parallel Program.*, 28(4):347–361, 2000.

[2] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 145–156, New York, NY, USA, 2000. ACM.

[3] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *Int. J. Parallel Program.*, 28(4):363–400, 2000.

[4] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood, California, 1996.

Arquitetura do Processador Cell Broadband Engine

Kleber Sacilotto de Souza
RA: 024249

RESUMO

Este trabalho apresenta uma introdução à arquitetura do processador Cell Broadband Engine (Cell BE). Essa arquitetura foi densenvolvida em conjunto pelas empresas Sony, Toshiba e IBM e apresenta um novo conceito de arquitetura multicore. O Cell BE possui um núcleo chamado de POWER Processing Element (PPE) com suporte a duas threads e oito núcleos chamados de Synergistic Processing Elements (SPEs). Também são apresentados algumas aplicações desta arquitetura e alguns dados sobre seu desempenho.

Categories and Subject Descriptors

D.1.2 [**Computer Systems Organization**]: Processor Architecture – Multiple Data Stream Architectures (Multiprocessors).

General Terms

Performance, Design.

Keywords

STI, Cell, Cell BE, Cell BEA, CBEA, arquitetura, multiprocessadores, supercomputadores

1. INTRODUÇÃO

O processador Cell Broadband Engine (Cell BE ou somente Cell), é a primeira implementação da Cell Broadband Engine Architecture (CBEA) [3], que é uma arquitetura desenvolvida em conjunto pela Sony, Toshiba e IBM (também conhecido por STI) com o objetivo de prover processamento de alto-desempenho com baixo consumo de energia e com uma boa relação custo/benefício para uma ampla gama de aplicações [6]. O Cell preenche o abismo entre processadores de propósito geral e processadores de alto-desempenho especializados. Enquanto o objetivo dos processadores de propósito geral é alcançar um bom desempenho numa vasta gama de aplicações, e o objetivo de um hardware especializado é obter o melhor desempenho em uma única aplicação, o objetivo do Cell é obter alto desempenho em workloads críticos para jogos, multimídia, aplicações que exigem uma grande largura de banda[11] e aplicações científicas.

O Cell BE inclui um POWER Processing Element (PPE) e oito Synergistic Processing Elements (SPEs). A arquitetura Cell BE foi desenvolvida para ser usada por uma ampla variedade de modelos de programação e permite que as tarefas sejam divididas entre o PPE e os oito SPEs.

O design do Cell BE foi focado em melhorar as relações desempenho/área e desempenho/consumo de energia. Esses objetivos foram alcançados basicamente na utilização de núcleos poderosos, porém simples, que fazem um uso mais eficiente da área com menos dissipação de potência. Suportados por um barramento com uma larga banda de dados, estes núcleos podem trabalhar tanto independentemente como em conjunto. Com um suporte a um número grande de acessos simultâneos dos núcleos a memória, a largura de banda da memória também pode ser usada mais eficientemente. A filosofia do design do Cell BE é de algum modo similar à tendência de ter vários núcleos de propósito geral no mesmo chip, entretanto no Cell BE os núcleos são apenas muito mais simples, mas poderosos.

2. ARQUITETURA DO CELL BE

2.1 Visão Geral

O Cell BE implementa um multiprocessador em um único chip com nove processadores operando em um sistema de memória compartilhado e coerente. Ele inclui um POWER Processing Element (PPE) de propósito geral de 64-bits e oito Synergistic Processing Elements (SPEs) interconectados por um barramento de alta velocidade, coerente em nível de memória, chamado de Element Interconnect Bus (EIB). Tanto o PPE quanto os SPEs são arquiteturas RISC com instruções de formato fixo de 32 bits. Os endereços de memória do sistema são representados tanto para o PPE quanto para os SPEs em 64 bits que podem endereçar teoricamente 2^{64} bytes, embora na prática nem todos esses bits são implementados em hardware. A figura 1 mostra a uma visão em alto nível da implementação do Cell.

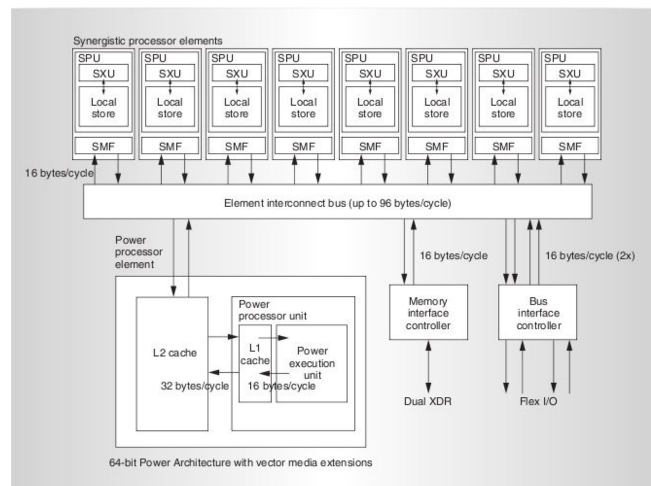


Figura 1. Implementação do processador Cell BE

Um die do processador Cell BE possui aproximadamente 241 milhões de transistores, numa área de $235mm^2$. A freqüencia mais comum utilizada é de 3,2GHz, porém freqüencias maiores

que 4GHz já foram alcançadas [5]. A figura 2 mostra a foto de um die do processador Cell BE onde os elementos principais estão identificados.

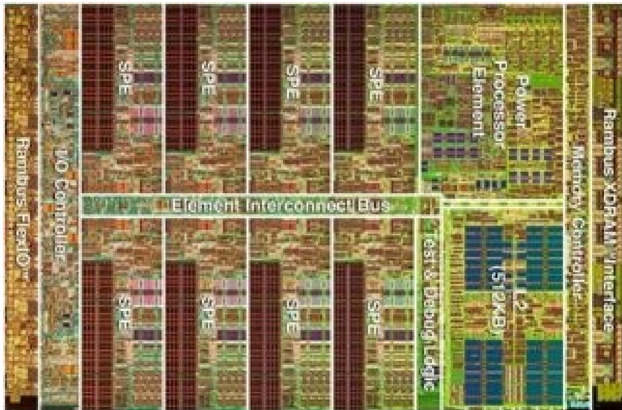


Figura 2. Foto de um die do processador Cell BE

2.2 O Power Processing Element (PPE)

O PPE consiste em um POWER Processing Unit (PPU) conectado a uma cache L2. O PPE é o processador principal do Cell BE e é responsável em executar o sistema operacional e coordenar os SPEs. O PPE é desenvolvido baseado na arquitetura Power de 64 bits da IBM com extensões vetoriais de multimídia (VMX, ou Vector Multi-media Extension) para operações SIMD (single instruction, multiple data). Ele é completamente compatível com a especificação da Arquitetura Power de 64 bits e pode executar sistemas operacionais e aplicações de 32 ou 64 bits. Além disso, o PPE inclui uma unidade de gerenciamento de memória (MMU, ou memory management unit) e uma unidade AltiVec que possui um pipeline completo para operações em ponto flutuante de precisão simples (o AltiVec não suporta vetores de ponto flutuante de precisão dupla).

O PPE possuiu dois níveis de memória cache. O nível 1 (L1) possui duas cachês de 32KB, uma para instruções a outra para dados, e o nível 2 (L2) possui uma cache unificada de 512KB (instruções e dados). O tamanho da linha de cache é de 128 bytes.

O PPU é um processador de execução em ordem, despacho duplo e com suporte a duas threads. Um diagrama do pipeline do PPU é mostrado na figura 3.

O PPE pode fazer o fetch de quatro instruções por vez, e despachar duas. Para melhorar o desempenho do seu pipeline em ordem, o PPE utiliza pipelines com execução atrasada e permite execução fora de ordem limitada de instruções de load. Isso permite o PPE obter algumas vantagens da execução fora de ordem sem aumentar significativamente sua complexidade.

O PPE acessa a memória principal com instruções de load e store que movem os dados entre a memória principal e o conjunto de registradores privado, sendo que este conteúdo pode ser armazenado em cache. O método de acesso à memória do PPE é semelhante às tecnologias convencionais de processadores. Os SPEs, diferentemente do PPE, precisam utilizar comandos de DMA para acessar a memória principal. Mais detalhes sobre este método serão mostrados na próxima seção.

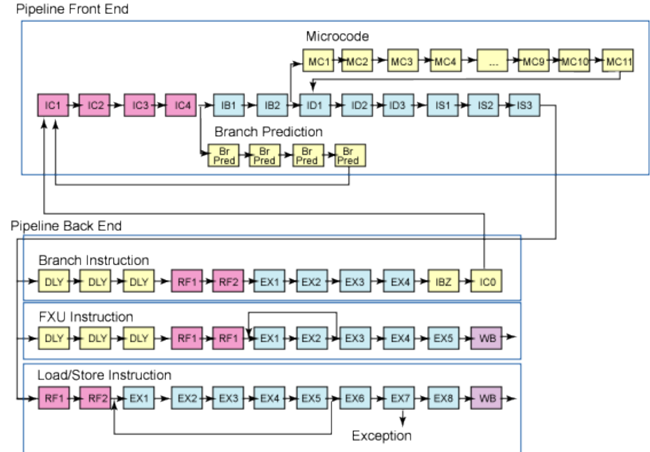


Figura 3. Pipeline do PPU

2.3 Os Synergistic Processing Elements (SPEs)

O SPE tem um design modular que consiste em uma Synergistic Processing Unit (SPU) e um Memory Flow Controller (MFC). Um SPU é um elemento de computação com suporte a SIMD e 256KB de armazenamento local dedicado (LS, ou Local Store), que possui um espaço de endereçamento de 32 bits [11]. O MFC contém um controlador DMA com uma MMU associada, assim como uma Unidade Atômica (Atomic Unit) para tratar das operações de sincronização com outras SPUs e com a PPU.

O PPE é mais competente que os SPEs em tarefas intensivas em controle e mais rápido na troca de tarefas. Os SPEs são mais competentes em tarefas de computação intensiva e mais lentos na troca de tarefas. Entretanto, todos os elementos de processamento são capazes de realizar ambos os tipos de funções. Essa especialização é um fator significante na melhoria em ordem de magnitude no desempenho, na área ocupada pelo chip e na eficiência no consumo de energia que o Cell BE alcança em relação aos processadores para PC convencionais

Uma SPU é um processador de execução em ordem, com despacho duplo e com um banco de registradores com 128 registradores de 128 bits usados tanto para operações de ponto flutuante quanto operações de inteiros. A SPU opera diretamente sobre as instruções e os dados em seu armazenamento local dedicado, e conta com uma interface para acessar a memória principal e outros armazenamentos locais. Esta interface, que é o MFC, executa independentemente da SPU e é capaz de traduzir endereços e realizar transferências DMA enquanto a SPU continua a execução do programa.

O suporte SIMD das SPUs podem realizar operações sobre dezoito inteiros de 8 bits, oito inteiros de 16 bits, quatro inteiros de 32 bits, ou quatro números em ponto flutuante de precisão simples por ciclo. A 3,2GHz, cada SPU é capaz de realizar até 51.2 bilhões de operações com inteiros de 8 bits ou 25.6GFLOPs com precisão simples. A figura 4 mostra as principais unidades funcionais de uma SPU: (1) uma unidade de ponto flutuante para multiplicação de inteiros e de ponto flutuante de precisão simples

e dupla; (2) uma unidade de ponto fixo par para aritmética, operações lógicas, e deslocamento de palavras; (3) uma unidade de ponto fixo ímpar para permutações, embaralhamentos, e rotação de quatro palavras; (4) uma unidade de controle para seqüenciamento de instruções e execução de desvios; (5) uma unidade de aramazenamento local para loads e stores; também fornece instruções para a unidade de controle; (6) uma unidade de transporte canal/DMA que é responsável em controlar as entradas e saídas através do MFC.

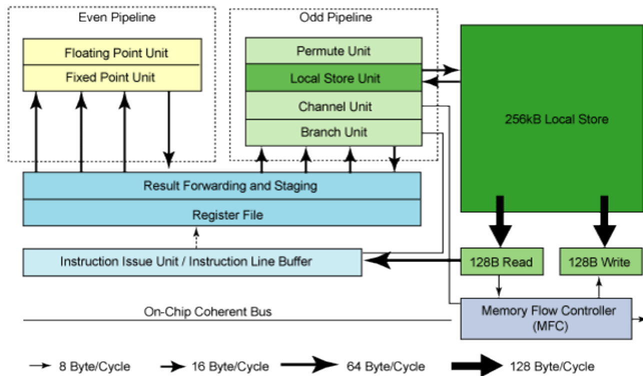


Figura 4. Unidades funcionais da SPU

Como mostra a figura 4, cada unidade funcional é atribuída a um dos dois pipelines de execução. As unidades de ponto flutuante e de ponto fixo estão no pipeline par enquanto que o resto das unidades funcionais está no pipeline ímpar. A SPU pode despachar e completar até duas instruções por ciclo, uma em cada um dos pipelines de execução. Um despacho duplo ocorre quando um grupo de instruções de um fetch tem duas instruções que podem ser despachadas, uma que é executada por uma unidade no pipeline par e a outra por uma unidade no pipeline ímpar.

Há três tipos de fetch de instruções: flush-initiaded fetches, inline fetches e hint fetches. Para fazer o fetch de uma instrução, a lógica de fetch de instruções lê 32 instruções de uma vez e armazena no seu buffer de instruções (ILB, ou instruction line buffer), do qual duas instruções por vez são enviadas à unidade lógica. Quando os operandos estiverem prontos, a lógica de despacho envia as instruções para as unidades funcionais para execução. Os pipelines das unidades funcionais variam de dois a sete ciclos. Instruções de hint fazem o preload de intruções no ILB.

Os SPEs são elementos de processamento independentes, cada um executando seus próprios programas ou threads individuais. Cada SPE tem acesso completo à memória compartilhada, incluindo o espaço de memória dedicado ao memory-mapped I/O (MMIO) implementado pelas unidades de DMA. Há uma dependência múltipla entre o PPE e os SPEs. Os SPEs dependem do PPE para executar o sistema operacional, e, em muitos casos, a thread de alto nível que controla a aplicação. O PPE depende dos SPEs para prover a maior parte do processamento responsável pelo desempenho da aplicação.

Os SPEs acessam a memória principal com comandos DMA que movem dados e instruções entre a memória principal e o armazenamento local (LS). O fetch das instruções e as instruções de load e store de um SPE acessam seu LS privado ao invés da memória principal compartilhada, e o LS não tem um cache associado. A organização do armazenamento em três níveis (banco de registradores, LS e memória principal), com transferências DMA assíncronas entre o LS e a memória principal, é uma mudança radical da arquitetura convencional e dos modelos de programação. Esta organização explicitamente paraleliza a computação com a transferência de dados e instruções que alimentam a computação e o armazenamento do resultado da computação em uma memória principal.

Uma das motivações para essa mudança radical é a latência das memórias, medida em ciclos de processador, que tem crescido em ordem de centena de vezes entre os anos 1980 e 2000. O resultado é que o desempenho das aplicações é, na maioria dos casos, limitado pela latência da memória ao invés do poder de processamento ou largura de banda. Quando um programa seqüencial sendo executado em uma arquitetura convencional executa uma instrução de load que dá miss na cache, a execução do programa pode ser interrompida por centenas de ciclos. (Técnicas como threading em hardware podem tentar esconder esses stalls, mas isso não ajuda no caso de aplicações que possuem somente uma thread.) Comparados com essa penalidade, os poucos ciclos que levam para configurar uma transferência DMA para um SPE é uma penalidade bem menor, especialmente considerando o fato de os controladores DMA de cada um dos SPEs podem gerenciar até 16 transferências DMA simultaneamente. A antecipação eficiente da necessidade de DMA pode fornecer uma entrega just-in-time de dados, o que pode reduzir ou até eliminar completamente os stalls. Processadores convencionais, até mesmo com uma larga e custosa especulação, conseguem obter, no melhor caso, apenas alguns acessos simultâneos à memória.

Um dos métodos de transferência DMA suporta uma lista, como uma lista *scatter-gatter*, de transferências DMA que é construída no armazenamento local do SPE. Portanto, o controlador DMA do SPE pode processar a lista de maneira assíncrona enquanto o SPE realiza operações em dados transferidos anteriormente. As transferências DMA podem ser iniciadas e controladas pelo SPE que está fornecendo ou recebendo os dados, ou alguns casos, pelo PPE ou outro SPE.

2.4 O Element Interconnect Bus (EIB)

O Element Interconnect Bus (EIB) no Cell BE permite a comunicação entre o PPE, os SPEs, a memória principal localizada externamente ao chip, e I/O externo (veja figura 5). O EIB consiste em um barramento de endereço e quatro anéis de dados de 16 bytes de largura, dos quais dois os dados são transmitidos em sentido horário, e dois em sentido anti-horário. Cada anel pode potencialmente permitir até três transmissões de dados concorrentes desde que seus caminhos não se sobreponham. O EIB opera a metade da velocidade do processador.

Cada requisitante do EIB começa com um número inicial pequeno de créditos de comandos para enviar requisições ao barramento. O número de créditos é o tamanho do buffer de comandos dentro do EIB para aquele requisitante em particular. Um crédito de comando é utilizado para cada requisição ao barramento. Quando um slot se abre no buffer de comandos assim que uma requisição anterior progride no pipeline de requisições do EIB, o EIB devolve o crédito ao requisitante.

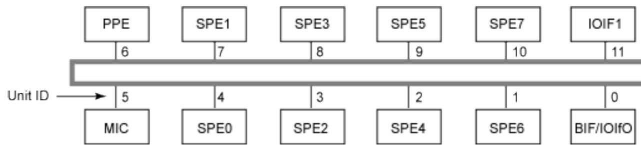


Figura 5. EIB e seus elementos

Quando um requisitante precisa de um anel de dados, ele envia uma única requisição para o árbitro do barramento de dados do EIB. O árbitro arbitra entre os múltiplos requisitantes e decide qual anel de dados é concedido a qual requisitante e quando. Para o controlador de memória é dada a maior prioridade para prevenir stall de dados no requisitante da leitura dos dados, enquanto todos os outros são tratados igualmente com uma prioridade em *round-robin*. Qualquer requisitante do barramento pode usar qualquer um dos quatro anéis de dados para enviar ou receber dados. O árbitro de dados não concede um anel de dados a um requisitante se a transferência cruzar mais do que a metade da volta do anel para chegar ao seu destino, ou se interferir em outra transferência de dados que esteja em progresso no momento.

Cada unidade conectada ao EIB pode enviar e receber simultaneamente 16B de dados a cada ciclo do barramento. A largura de banda máxima de todo o EIB é limitada a taxa máxima a qual endereços são monitorados através de todas as unidades no sistema, a qual é um por ciclo do barramento. Como cada requisição pode potencialmente transmitir até 128B de dados, o pico de largura de banda no EIB a 3,2GHz é 128Bx1,6GHz = 204,8GB/s.

A largura de banda sustentada pelo EIB será menor do que o pico de largura de banda devido a vários fatores: a distância relativa entre o destino e a origem, o potencial de novas transmissões interferirem em aquelas que já estejam em progresso, o número de processadores Cell BE no sistema, se as transmissões de dados são para ou da memória ou entre armazenamentos locais nos SPEs, e a eficiência do árbitro de dados.

Larguras de banda reduzidas podem resultar quando: (1) todos os requisitantes acessam o mesmo destino ao mesmo tempo, como memória no mesmo armazenamento local; (2) todas as transmissões são para o mesmo destino e deixam dois dos quatros anéis ociosos; (3) há um grande número de transferências parciais de linhas de cache diminuindo a eficiência do barramento; e (4) todas as transferências de dados passam por 6 unidades até chegarem ao seu destino, inibindo as unidades que estão no caminho de utilizarem o mesmo anel [3].

2.5 O Subsistema de Memória

O Memory Flow Controller (MFC) é o mecanismo de transferência de dados. Ele provê o método primário para transferência de dados, proteção, e sincronização entre a memória principal e o armazenamento local associado, ou entre o armazenamento local associado a outro armazenamento local. Um comando MFC descreve a transferência a ser executada. Um dos objetivos arquiteturais principais do MFC é realizar estas transferências de maneira mais rápida possível, deste modo maximizando o throughput geral do processador.

Comandos que transferem dados são chamados de comandos MFC DMA. Estes comandos são convertidos em transferências DMA entre o domínio do armazenamento local e o domínio do armazenamento principal. Cada MFC pode suportar múltiplas transferências DMA ao mesmo tempo e pode manter e processar múltiplos comandos MFC. Para alcançar isso, o MFC mantém e processa filas de comandos MFC. Cada MFC provê uma fila para a SPU associada (fila de comandos MFC SPU) e uma fila para outros processadores e dispositivos (fila de Proxy de comandos MFC). Logicamente, um conjunto de filas MFC é sempre associada a cada SPU em um processador Cell BE [9].

O Memory Interface Controller (MIC) do chip do Cell BE é conectado à memória externa RAMBUS XDR através de dois canais XIO que operam a uma freqüência máxima efetiva de 3,2GHz (400 MHz, Octal Data Rate). Ambos os canais RAMBUS podem ter oito bancos operando concorrentemente e um tamanho máximo de 256MB, em um total de 512MB.

O MIC possui filas separadas de requisições de leitura e escrita para cada canal XIO operando independentemente. Para cada canal, o árbitro do MIC alterna o despacho entre filas de leituras e escritas após o mínimo de oito despachos de cada fila ou até que a fila fique vazia, o que for mais curto. Às requisições de leitura de alta prioridade são dadas prioridade sobre leituras e escritas.

Escritas de 16B ou mais, mas com menos de 128B, pode ser escritas diretamente na memória usando uma operação masked-write, mas escritas menores que 16B precisam de uma operação read-modify-write. Devido ao número pequeno de buffers para operação read-modify-write, a parte de leitura da operação read-modify-write é dada alta prioridade sobre leituras normais, enquanto que a parte de escrita é dada prioridade sobre as escritas normais.

2.6 Flexible I/O Controller

Há sete links RAMBUS RRAC FlexIO para transmissão e cinco para recepção e possuem uma largura de 1B cada. Esses links podem ser configurados como duas interfaces lógicas. Com os links FlexIO operando a 5GHz, a interface de I/O provê uma largura de banda bruta de 35GB/s de saída e 25GB/s de entrada. Uma configuração típica pode ter uma interface de I/O configurada com uma largura de banda bruta de 30GB/s de saída e 20GB/s de entrada; e a outra interface de I/O com largura de banda bruta de 5GB/s de saída e 5GB/s de entrada.

Dados e comandos enviados para a interface de I/O são enviados como pacotes. Além do comando, resposta, e dados, cada pacote pode carregar informações tais como tag de dados, tamanho dos

dados, id do comando, e informações de controle de fluxo, além de outras informações. Devido a esses overheads, e tempos de chegada de comandos e dados não ótimos, a largura de banda efetiva nas duas interfaces é tipicamente menor, e varia de 50% a 80% da largura de banda bruta.

3. PROGRAMANDO PARA O CELL BE

3.1 Conjunto de Instruções

O conjunto de instruções para o PPE é uma versão entendida do conjunto de instrução da Arquitetura PowerPC. As extensões consistem em extensões multimídia vector/SIMD, algumas poucas mudanças e adições às instruções da Arquitetura PowerPC, e funções intrínsecas C/C++ para as extensões multimídia vector/SIMD.

O conjunto de instruções para os SPEs é um conjunto de instruções SIMD novo, o *Synergistic Processor Unit Instructions Set Architecture*, acompanhado de funções intrínsecas C/C++. Ele também possui um conjunto único de comandos para gerenciar transferências DMA, eventos externos, troca de mensagens intra-processador, e outras funções. O conjunto de instruções para os SPEs é similar ao das extensões multimídias vector/SIMD do PPE, no sentido que eles operam sobre vetores SIMD. Entretanto, os dois conjuntos de instruções vetoriais são distintos. Programas desenvolvidos para o PPE e para os SPEs são geralmente compilados por compiladores diferentes, gerando código para dois conjuntos de instruções completamente diferentes.

3.2 Domínios de Armazenamento e Interfaces

O processador Cell BE possui dois tipos de domínios de armazenamento: um domínio principal de armazenamento e oito domínios LS das SPEs, como mostrado na figura 6. Além disso, cada SPE possui uma interface de canal para comunicação entre o seu SPU e o seu MFC.

O domínio principal de armazenamento, o qual é todo o espaço de endereços efetivo, por ser configurado por software sendo executado em modo privilegiado no PPE para ser compartilhado por todos os elementos do processador e dispositivos memory-mapped do sistema. O estado de um MFC é acessado por sua SPU associada através da interface de canal. Este estado também pode ser acessado pelo PPE e outros dispositivos, incluindo outros SPEs, por meio dos registradores de MMIO do MFC no espaço de armazenamento principal. O LS de uma SPU pode também ser acessada pelo PPE e outros dispositivos através do espaço de armazenamento principal de uma maneira não coerente. O PPE acessa o espaço de armazenamento principal através do seu PowerPC processor storage subsystem (PPSS).

Cada MFC possui uma unidade de synergistic memory management (SMM) que realiza o processamento de informações de tradução de endereços e de permissão de acesso que são fornecidas pelo sistema operacional sendo executado no PPE. Para processar um endereço efetivo fornecido por um comando DMA, o SMM usa essencialmente o mesmo mecanismo de tradução de endereços e proteção é que utilizado pela unidade de gerenciamento de memória (MMU) no PPSS do PPE. Portanto, transferências DMA são coerentes com relação ao armazenamento do sistema, e os atributos do armazenamento do sistema são controlados pelas tabelas de página e segmento da Arquitetura PowerPC.

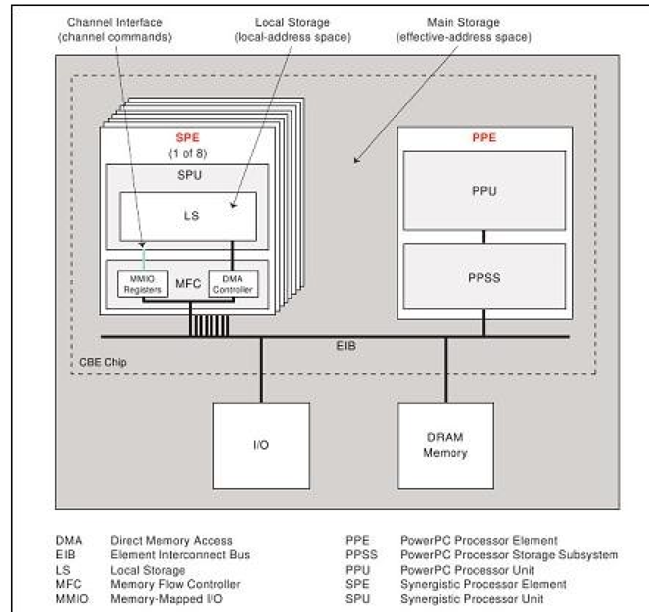


Figura 6. Domínios de armazenamento

3.3 Ambiente de Execução

O PPE executa aplicações e sistemas operacionais desenvolvidos para a Arquitetura PowerPC, o que inclui instruções da Arquitetura PowerPC e instruções da extensão multimídia vector/SIMD. Para utilizar todas as características do processador Cell BE, o PPE precisa executar um sistema operacional que suporta todas essas características, como multiprocessamento utilizando os SPEs, acesso as operações da extensão multimídia vector/SIMD, o controlador de interrupções do Cell BE, e todas as outras funções fornecidas pelo processador Cell BE.

Uma thread principal sendo executada no PPE pode interagir diretamente com uma thread em um SPE através do LS do SPE. Ela pode interagir indiretamente através do espaço de armazenamento principal. A thread do PPE pode também se comunicar através de uma caixa de mensagens e sinalização de eventos que são implementados em hardware.

O sistema operacional define o mecanismo e as políticas para selecionar um SPE disponível para agendar a execução de uma thread SPU. O sistema operacional também é responsável pelo carregamento em tempo de execução, passagem de parâmetros aos programas sendo executados no SPE, notificação de erros e eventos do SPE, e suporte a depuração.

3.4 Modelos de Programação

Devido a flexibilidade natural do Cell BE, há várias possibilidades para a utilização de seus recursos. Nas próximas seções veremos alguns dos possíveis modelos [10].

3.4.1 *Fila de tarefas*

O PPE mantém uma fila de tarefas em memória, agenda as tarefas para serem executadas nos SPEs e monitora o progresso das tarefas. Cada SPE roda um "mini kernel" cujo papel é buscar uma tarefa na fila de tarefas, executá-la, e sincronizar com o PPE.

3.4.2 *Multi-tarefa gerenciada pelos SPEs*

O kernel e o agendamento é distribuído entre os SPEs. A sincronização é feita através do uso de exclusão mútua e semáforos, como em um sistema operacional convencional. Tarefas prontas para serem executadas por um SPE são mantidas em um *pool*. Os SPEs utilizam memória compartilhada para organizar a comunicação entre SPEs.

3.4.3 *Processamento em fluxo*

Neste modelo cada SPE executa um programa distinto. Os dados chegam em um fluxo de entrada e são enviados para os SPEs. Quando um SPE termina o processamento, os dados de saída são enviados a um buffer de fluxo de saída.

3.5 Exemplo

O exemplo a seguir mostra um simples exemplo de um "Hello world!" onde um programa principal, que é executado no PPE, cria uma thread para cada SPU (executada pelo Código 1) e espera o retorno de cada uma delas (Código 2) [7].

Este exemplo utiliza a libspe (SPE Runtime Management Library) que é uma biblioteca C/C++ que fornece uma API (Application Programming Interface) para aplicações acessarem os SPEs do Cell BE [1].

Código 1: Código para o SPE

```
#include <stdio.h>

int main(unsigned long long speid,
         unsigned long long argp,
         unsigned long long envp)
{
      printf("Hello world (0x%llx)\n",
             speid);
      return 0;
}
```

4. DESEMPENHO

A Tabela 1 mostra um resumo dos dados de desempenho obtidos em [3]. Para uma vasta gama de aplicações, o Cell BE possui desempenho igual ou significativamente melhor que um processador de propósito geral (GPP, General Purpose Processor). Para aplicações que podem tirar vantagem do pipeline SIMD das SPUs e do paralelismo no nivel de thread da PPU os resultados serão similares.

Código 2: Código para o PPE

```
#include <stdio.h>
#include <libspe.h>
#include <sys/wait.h>
extern spe_program_handle_t hello_spu;

int main (void)
{
      speid_t speid[8];
      int status[8];
      int i;
      for (i=0;i<8;i++)
            speid[i]  =  spe_create_thread
(0, &hello_spu, NULL, NULL, -1, 0);
      for (i=0;i<8;i++)
      {
            spe_wait(speid[i],
&status[i], 0);
            printf ("status = %d\n",
            WEXITSTATUS(status[i]));
      }
      return 0;
}
```

Tabela 1. Desempenho do Cell comparado a um processador de propósito geral

Algorithm	3.2 GHz GPP	3.2 GHz Cell	Cell Perf Advantage
Matrix Multiplication (S.P.)	24 Gflops (w/SIMD)	200 GFlops* (8SPEs)	8x
Linpack (S.P.)	16 GFlops (w/SIMD)	156 GFlops* (8SPEs)	9x
Linpack (D.P.): 1kx1k matrix	7.2 GFlops (IA32/SSE3)	9.67 GFLops* (8SPEs)	1.3x
Transform-light	170 MVPS (G5/VMX)	256 MVPS** (per SPE)	12x
TRE	1 fps (G5/VMX)	30 fps* (Cell)	30x
AES encryp. 128-bit key	1.03 Gbps	2.06Gbps** (per SPE)	16x
AES decryp. 128-bit key	1.04 Gbps	1.5Gbps** (per SPE)	11x
TDES	0.12 Gbps	0.16 Gbps** (per SPE)	10x
DES	0.43 Gbps	0.49 Gbps** (per SPE)	9x
SHA-1	0.85 Gbps	1.98 Gbps** (per SPE)	18x
mpeg2 decoder (CIF)	----	1267 fps* (per SPE)	--
mpeg2 decoder (SDTV)	354 fps (IA32)	365 fps** (per SPE)	8x
mpeg2 decoder (HDTV)	----	73 fps* (per SPE)	--

Notas: * Medidas por hardware ** Restultados de simulações

5. APLICAÇÕES

Nesta seção veremos algumas das possíveis aplicações do processador Cell BE.

5.1 Processamento de vídeo

Algumas empresas possuem planos de lançar adaptadores PCI-E baseados no Cell para decodificação de vídeo em tempo real.

5.2 Blade Server

Em 2007 a IBM lançou o Blade Server QS21 que gera 1,02 Giga FLOPS por watt, com picos de desempenho de aproximadamente 460 GFLOPS, o que o faz uma das plataformas mais eficientes em consumo de energia da atualidade.

5.3 Console de videogames

O console de videogame da Sony PlayStation 3 contém a primeira aplicação do Cell BE a entrar em produção, com clock de 3,2GHz e contendo sete dos oito SPEs operacionais, para aumentar o número de processadores aproveitados no processo de produção. Apenas seis destes sete SPEs são acessíveis aos desenvolvedores, sendo que um é reservado pelo sistema operacional.

5.4 Supercomputadores

O mais recente supercomputador da IBM, o IBM Roadrunner, é um híbrido processadores de propósito geral CISC Opteron e processadores Cell. Em junho de 2008 este sistema entrou como número 1 da lista Top 500 como o primeiro computador a rodar a velocidades de petaFLOPS, conseguindo sustentar uma velocidade de 1,026 petaFLOPS durante a execução do benchmark Linkpack.

Os supercomputadores que utilizam os processadores Cell também são eficientes no consumo de energia. Dentre as 10 primeiras posições da lista dos 500 supercomputadores mais eficientes em consumo de energia, as 7 primeiras são ocupadas por sistemas que utilizam processadores Cell [4].

5.5 Computação em Cluster

Clusters de consoles PlayStation 3 podem ser uma alternativa a sistemas mais caros e sofisticados baseados em Blade Servers, como mostrado em [2].

6. PowerXCell 8i

Em 2008 a IBM anunciou uma versão revisada do Cell BE chamada de PowerXCell 8i. O PowerXCell 8i suporta até 16GB de memória principal além de um aumento substancial no desempenho de operações de ponto-flutuante de precisão dupla alcançando um pico de 12,8GFLOPS por SPE e mais de 100GFLOPS utilizando todos os núcleos [8]. O supercomputador da IBM, Roadrunner, consiste em 12.240 processadores PowerXCell 8i, além de 6.562 processadores AMD Opteron.

7. REFERÊNCIAS

[1] Arevalo, A., Matinata, R. M., Pandian, M., Peri, E., Kurtis, R., Thomas, F. e Almond, C. 2008. Programming the Cell Broadband Engine™ Architecture: Examples and Best Practices http://www.redbooks.ibm.com/redbooks/pdfs/sg247575.pdf

[2] Buttari, A., Luszczek, P., Kurzak, J., Dongarra, J., e Bosilca, G. 2007. A Rough Guide to Scientific Computing On the PlayStation 3 http://www.netlib.org/utk/people/JackDongarra/PAPERS/scop3.pdf

[3] Chen, T., Raghavan, R., Dale, J. e Iwata, E. Cell Broadband Engine Architecture and its first implementation http://www.ibm.com/developerworks/power/library/pa-cellperf/?S_TACT=105AGX16&S_CMP=LP 19/06/2009

[4] The Green500 List http://www.green500.org/lists/2008/11/list.php 25/09/2009

[5] IBM - Cell Broadband Engine Overview http://publib.boulder.ibm.com/infocenter/ieduasst/stgv1r0/topic/com.ibm.iea.cbe/cbe/1.0/Overview/L1T1H1_02_CellOverview.pdf 25/06/2009

[6] IBM - The Cell project at IBM Research http://www.research.ibm.com/cell/

20/06/2009

[7] IBM - Hands-on: The Hello World Cell Program, PPE vs SPE http://www.cc.gatech.edu/~bader/CellProgramming.html 22/06/2009

[8] IBM - PowerXCell 8i processor product brief http://www-03.ibm.com/technology/resources/technology_cell_pdf_PowerXCell_PB_7May2008_pub.pdf 25/06/2009

[9] Komornicki, Dr. A, Mullen-Schulz, G. e Landon, D. 2009. Roadrunner: Hardware and Software Overview. http://www.redbooks.ibm.com/redpapers/pdfs/redp4477.pdf

[10] Mallinson, D. e DeLoura, M. 2005. CELL: A New Platform for Digital Entertainment http://www.research.scea.com/research/html/CellGDC05/index.html 22/06/2009

[11] Sony - Synergistic Processor Unit Instruction Set Architecture, Version 1.2 http://cell.scei.co.jp/pdf/SPU_ISA_v12.pdf

A História da família PowerPC

Flavio Augusto Wada de Oliveira Preto*
Instituto de Computação
Unicamp
flavio.preto@students.ic.unicamp.br

ABSTRACT

Este artigo oferece um passeio histórico pela arquitetura POWER, desde sua origem até os dias de hoje. Através deste passeio podemos analisar como as tecnologias que foram surgindo através das quatro décadas de existência da arquitetura foram incorporadas. E desta forma é possível verificar até os dias de hoje como as tendências foram seguidas e usadas. Além de poder analisar como as tendencias futuras na área de arquitetura de computadores seguirá.

Neste artigo também será apresentado sistemas computacionais que empregam comercialmente processadores POWER, em especial os videogames, dado que atualmente os três videogames mais vendidos no mundo fazem uso de um chip POWER, que apesar da arquitetura comum possuem grandes diferenças de design.

Diferenças de design que não implicam na não conformidade com a PowerPC ISA, que é o conjunto de instruções da arquitetura PowerPC. Este conjunto que é aberto e mantido por uma instituição ao invés de uma única empresa. Permitindo assim que qualquer empresa fabrique chips compatíveis com a arquitetura POWER.

E desta forma o artigo permite que o leitor conheça os detalhes de arquitetura, político e historicos da família POWER.

1. INTRODUÇÃO

Na década de 70 um dos maiores problemas computacionais era o chaveamento de ligações telefônicas. Entretando nesta época a grande maioria dos computadores eram CISC (complex instruction set computer) e possuiam um conjunto de instruções extenso, complexo e muitas vezes redundante. Essa tendência de computadores CISCs decorria do fato do surgimento do transistor e do circuito integrado. Para resolver o problema de chaveamento telefônico a IBM iniciou o desenvolvimento do IBM 801, que tinha como objetivo principal atingir a marca de uma instrução por ciclo e 300 ligações por minuto.

O IBM 801 foi contra a tendência do mercado ao reduzir drásticamente o número de instruções em busca de um conjunto pequeno e simples, chamado de RISC (reduced instruction set computer). Este conjunto de instruções eliminava instruções redundantes que podiam ser executadas com uma combinação de outras intruções. Com este novo conjunto reduzido, o IBM 801 possuia metade dos circuitos de seus contemporâneos.

Apesar do IBM 801 nunca ter se tornado um chaveador telefônico, ele foi o marco de toda uma linha de processadores RISC que podemos encontrar até hoje: a linha POWER.

Descendente direto do 801, a arquitetura POWER (Performance Optimization With Enhanced RISC) surgiu em 1990, dando origem a uma séries de processadores que equipariam desde workstations até grandes servidores. O seu objetivo era fornecer complementos ao 801 que era bastante simples e faltava unidades aritméticas de ponto flutuante e processamento paralelo.

Baseado na arquitetura POWER e fruto da aliança Apple-IBM-Motorola (AIM), surgiu a linha de processadores PowerPC (POWER Performance Computing). Projetado para ser empregado desde de dispositivos embarcados até grandes servidores e mainframes teve sua primeira aparição comercial no Power Macintosh 6100.

Desde 1993, quando o PowerPC foi criado, o ecossistema que o abriga evoluiu continuamente, dando origem a produtos mais modernos, ao mesmo tempo que as empresas do ecossistema se alteravam. A Apple atualmente não está mais envolvida no projeto e hoje em dia emprega processadores X86 em sua linha de produtos. Já a Motorola separou sua divisão de semicondutores numa nova empresa: a Freescale Semiconductors.

Entretanto, uma grande vantagem que mantem vivo o ecossistema da arquitetura PowerPC é o fato dela ser aberta. Ou seja o ISA (Instruction Set Architecture) é definido por um consórcio e disponibilizada para que qualquer um possa projetar e fabricar um processador compatível com o PowerPC.

Outra grande vantagem competitiva da arquitetura PowerPC é que sua simplicidade herdada do 801 permite que o core

*RA032883

do CPU seja extremamente pequeno, liberando espaço no circuito para que seja adicionado outros componentes que o desenvolvedor desejar, como por exemplo coprocessadores, cache e controladores de memória.

Isso tudo resultou numa das mais bem sucedidas linhas de processadores que nos dias de hoje pode ser encontrado desde os videogames mais vendidos até os mais potentes super computadores existente.

2. A HISTÓRIA DO *POWERPC*

2.1 O projeto 801

Em 1974 a IBM inicou um projeto para um processador capaz de lidar com o roteamento de até 300 ligações telefônicas por minuto. Para este número era estimado que fosse necessário cerca de 20000 instruções por ligação, e por consequência para 300 ligações seria necessário um processador de 12 MIPS. Este valor era extremamente alto para a época, entretanto foi notado que os processadores existentes possuiam diversas instruções que nunca eram usadas ou eram usadas muito raramente.

Apesar de 1975 o projeto de telefonia ter sido cancelado sem sequer ter produzido um protótipo, A idéia de um processador com um conjunto de instruções bastante reduzido mostrou-se bastante promissora. De modo que os trabalhos de pesquisa continuaram sobre o codinome "Cheetah" no prédio número 801 do Thomas J. Watson Research Center.

Os pesquisadores estavam tentando determinar se era viável uma máquina RISC manter múltiplas instruções por ciclo e quais alterações deveriam ser feitas sobre o projeto 801 original. Para isso, foi imaginado unidades separadas para branch, artimética de ponto fixo e aritimética de ponto flutuante.

Em 1985 iniciou o projeto de uma segunda arquitetura RISC no Thomas J. Watson Research Center de codinome"AMERIC" para ser usada nas series RS/6000. Este projeto resultou na arquitetura POWER[2].

2.2 POWER

Apresentado como o processador RISC da série RS/6000[1] em fevereiro de 1990. Iniciava-se aí uma longa saga de processadores POWER até os dias de hoje.

A arquitetura fortemente baseada na 801, com um design RISC, buscava superar as limitações de seu predecessor. Para isso deveria incluir uma unidade de ponto flutuante, um sistema de pipeline para execução de mais de uma instrução ao mesmo tempo.

Inicialmente com 32 registradores de 32 bits, e posteriormente os modelos 64 bits incluiam mais 32 registradores de 64 bits e armazenamento de dados no formato *big-endian.*

2.2.1 POWER1

Disponibilizado incialmente nos servidores IBM RS/6000 POWERserver com clocks de 20, 25 ou 30 Mhz, foi logo seguido de uma pequena atualização para POWER1+ e depois para POWER1++. Essas atualizações possuiam clocks mais elevados devido a evolução nos processos de fabricação de semicondutores, chegando até 62.5 Mhz.

A arquitetura do POWER1 é baseada em um CPU de 32-bits superscalar de duas vias. Contém três unidades de execução, uma de aritmética de ponto-fixo (FXU), uma de aritmética deponto flutuante (FPU) e uma de branch (BU). O espaço de enderaçamento físico era de 32-bits, entretanto o endereçamento virtual era de 52 bits para beneficiar o desempenho das aplicações. O cache contava com 8 KB e um *2-way set associative* para instruções e 32 ou 64 KB em *4-way set associative* com 128 bytes por linha para dados.

2.2.2 POWER2

O processador POWER2 foi lançado em 1993 com um projeto aprimorado do POWER1. Com clocks que variavam de 55 até 71.5 Mhz e uma unidade aritmética de ponto fixo adicional e uma unidade de ponto flutuante adicional. Um cache maior e algumas instruções novas. A implementação usada para o POWER2 era de multi-chip, sendo composto de 8 chips.

Em 1996 foi lançado o POWER2 Super Chip, ou simplesmente P2SC, como uma implementação em apenas um chip do POWER2. Além detsa alteração, tambem foram melhorados os caches e o clock chegou a 135 MHz. Essa versão foi empregada para construção do super-computador de 30 cores, Deep Blue da IBM, que derrotou o campeão mundial de xadrex, Garry Kasparov, em 1997.

2.2.3 POWER3

Foi o primeiro multiprocessador simétrico (SMP) 64-bits e mesmo assim totalmente compatível com o conjunto de instruções POWER. O POWER3 foi lançado em 1998 depois de um longo atraso de quase 3 anos.

Disponível inicialmente com o clock de 200 MHz, possuia três unidades de ponto-fixo e duas e unidades de ponto-flutuantes capazes de realizar operações de multiplicação e adição em apenas um ciclo para uma dada precisão. Com um design super-escalar capaz de executar instruções fora de ordem em seu pipeline 7 estágios para inteiros e 8 estágios para load/store.

Para otimizar possuia registradores ocultos para efetuar *register renaming* tanto para operações de propósito geral como operações em ponto flutuante. Além de um cache otimizado para aplicações científicas e técnicas. Com uma capacidade dobrada que agora atingia 64Kb com linhas de 128-bytes.

2.2.4 POWER4

Com um clock que rompeu a barreira de 1Ghz, em seu lançamento em 2001 era considerado o chip mais poderoso do mercado. Com uma arquitetura que herdava todas as características do POWER3, incluindo a compatibilidade com o conjunto de instruções PowerPC, porém em um design totalmente diferente.

Cada chip possuia dois cores de 64-bits PowerPC que trabalhavam a mais de 1Ghz dando origem a tendência de chips multi-cores. Com sua capacidade super escalar, o POWER4 é capaz de executar mais de 200 instruções ao mesmo tempo.

Cada core é dotado de duas unidaestades de ponto-flutuante, duas unidades de load/store, duas unidades de ponto-fixo, uma unidade de branch e uma unidade de registrador condicional.

Antes de extinguir a linha foi lançado o POWER4+ que atingia velocidades de 1.9GHz e consumuia menos energia, devido a novas tecnologias de fabricação.

2.2.5 POWER5

Subsequentemente foi lançado em 2003 o POWER5, porém a única maneira de se ter acesso a um destes chips era através da aquisição de um dos sistemas da IBM ou de seus parceiros. Este chip buscava competir no mercado empresarial high-end contra o Intel Itanium 2 e o Sun UltraSPARC IV.

A primeira inovação do POWER5 era a capacidade de multithreading, ou seja, de executar mais de uma thread em um core. Portanto, o processador POWER5 de core duplo podia executar até quatro threads virtuais. Com o controlador de memória, caches L1, L2 e L3 no próprio chip o POWER5 o processador evitava a necessidade outros chips.

Outro grande recurso implementado foi o de virtualização assistida por hardware que permitia a execução de até 256 LPAR (Logical Partitions).

Inicialmente com clocks entre 1.5 e 1.9 GHz, recebeu atualizações de tecnologia e teve mais uma versão chamada de POWER5+. Esta versão possui clocks de até 2.2GHz, quatro cores por chips e um consumo menor de energia por core.

2.2.6 POWER6

O processador POWER6 surgiu na IBM com o projeto de nome eCLipz. no qual ipz seria um acrônimo para a iSeries, pSeries e zSeries (respectivamente as linhas de servidores de medio porte, de pequeno porte e main frames). Este acrônimo sugeria que o POWER6 seria o processador de convergência para essas linhas.

Lançado em junho de 2007 com clocks que chegam até 4.7 GHz, embora alguns protótipos chegaram a 6 GHz. O processador manteve o projeto de dois núcleos com caches L1 de 128KB em 8-set-associative. Além disso o L2 possui 4 MB e o L3 32 MB.

2.3 PowerPC

PowerPC[5] significa POWER Performance Computing e surgiu em 1993 como um derivado da arquitetura POWER. Fruto da aliança entre Apple, IBM e Mortola (tambem conhecida como AIM), o PowerPC era baseado no POWER porem com uma série de diferenças. Por exemplo, enquanto o POWER é big-endian, o PowerPC possui suporte tanto para big-endian como para little-endian. O foco original do PowerPC, assim como no POWER, é no desempenho das operações de ponto-flutuante e multiprocessamento.

Apesar dessas modificações, a PowerPC mantém grande compatibilidade com a arquitetura POWER, fato que pode ser comprovado ao verificar que muitas aplicações rodam em ambos sem recompilação ou com pequenas recompilações.

Com a proposta de flexibilidade do PowerPC, no qual ele se propõe a equipar desde dispositivos embarcados até grandes computadores, podemos avaliar o sucesso com a presença do PowerPC nos três videogames mais vendidos hoje (Microsoft X-Box 360, Nintendo Wii e Sony Playstation 3), assim como em grandes servidores IBM BladeServers até o super computador IBM Blue Gene que figura entre os cinco mais poderosos no TOP 500.

2.3.1 PowerPC 400

A família 400 é uma família para dispositivos embarcados que mostra a flexibilidade da arquitetura PowerPC de se adaptar e ser empregados em dispositivos bastante específicos. Disponíveis em diversas tecnologias e potências, hoje é possível encontrar chips trabalhando desde 200 MHz até 800 MHz em dispostivos que vão desde eletrodomésticos até switches gigabit de rede.

2.3.2 PowerPC 600

Apesar de 600 ser um número maior que 400, o PowerPC 601 foi o primeiro chip PowerPC. Ele pode ser considerado o elo entre as arquiteturas POWER e PowerPC, que pode ser verificado com o alto grau de compatibilidade com o ISA da POWER assim como com o barramento Motorola 88110.

2.3.3 PowerPC 700

Surgiu em 1998, sendo que o PowerPC 750 foi o primeiro processador produzido a base de cobre no mundo. A família ganhou notoriedade ao ser usado como o processador da família G3 da Apple. Porem logo foi ofuscado pelo G4, ou Motorola 7400. Disponível em sua época com clocks de até 1GHz e caches L2 de 1MB

2.3.4 PowerPC 900

Com um grande poder computacional, podendo executar até 200 instruções ao mesmo tempo em clocks maiores que 2 GHz e com um baixo consumo de energia, a familia 900 podia ser vista como uma versão single core do POWER4.

Porém esta família já empregava as instruções 64-bits além de capacidades SIMD (Single Instruction Multiple Data) para aumentar o desempenho de aplicações computacionamente intensivas, como multimídia e gráficas.

Desta forma, logo foi adotado pela Apple em sua linha G5. E devido ao baixo consumo de energia de algumas linhas de processadores logo pode ser visto também em dispositivos embarcados.

3. OS LIVROS E A ESPECIFICAÇÃO

A especificação da arquitetura da família POWER pode ser encontrada em um conjunto de livros chamado Power ISA (Instruction Set Architecture). Este conjunto atualmente é composto por cinco livros e apêndices. Esta especificação é aberta e mantida pela Power.org, de forma que qualquer empresa que deseja fabricar um chip compatível com a família POWER possui a documentação necessária de referência.

Através de comites de experts a Power.org desenvolve os padrões abertos e as especificações, além de promover boas práticas, educar e certificar, de modo que a arquitetura Power

possa evoluir e aumentar a adoção da arquitetura pela industria de eletrônicos.

3.1 Power.org

A Power.org é uma comunidade de hardware aberto que gerencia e mantém as especificações da arquitetura POWER. Fundada em dezembro de 2005 pela IBM e mais 14 outras organizações, dentre elas Chartered, Cadence e Sony, a Power.org triplicou seu tamanho desde então. Em fevereiro de 2006 a Freescale Semiconductor também se juntou a Power.org, trazendo uma grande representatividade da comunidade de dispositivos embarcados.

3.2 PowerPC ISA

Como fruto do trabalho da Power.org, é disponibilizado para o público o PowerPC ISA[6], um conjunto de livros que detalha o conjunto de instruções que um chip POWER deve implementar. Desta forma, para o fabricante projetar um chip POWER, basta que ele implemente as instruções detalhadas no conjunto de livros do PowerPC ISA.

Este conjunto é bastante extenso e atualmente divido em 5 livros:

3.2.1 Livro I

Também conhecido como *User Instruction Set Architecture*, cobre o conjunto de instruções disponíveis e recursos relacionados para o programador da aplicação. Neste conjunto estão inclusos também as instruções APU, incluindo a extensão de vetorização (SIMD) conhecida como Altivec.

3.2.2 Livro II

Também conhecido como *Virtual Enviroment Architecture* define o modelo de armazenagem virtual, incluindo desde operações atômicas, cache, controle de cache, modelo de memória até o armazenamento compartilhado e instruções de controle de memória pelo usuário.

3.2.3 Livro III-S

O Livro III é dividido em dois livros. Ambos tratam de *Operating Environment Architecture*. Ou seja a arquietetura POWER vista pelo lado do sistema operacional, como por exemplo tratamento de excessões, interrupções, gerenciamento de memória, debug e funções especiais de controle.

O Livro III-S basicamente trata das instruções de supervisão usada para implementações de propósito geral e servidor.

3.2.4 Livro III-E

Já o Livro III-E, que derivou do antigo livro PowerPC Livro E definine as instruções de supervisão usada em aplicações embarcadas.

3.2.5 Livro VLE

Finalmente o Livro VLE, conhecido como *Variable Length Encoded Instruction Architecture* apresenta instruções e definições alternativas aos Livros I, II e III. Os propósitos dessas instruções são para obter uma maior densidade de instruções e para aplicações bem específicas e de baixo nível.

4. O PODER DOS VIDEOGAMES

Atualmente todos os três principais videogames disponíveis no mercado possuem processador compatível com a família PowerPC. Este é um marco historico num mundo dominado pro processadores MIPS desde os primeiros consoles de videogame de 16-bits.

4.1 Nintendo Wii

O Nintendo Wii é um console de videogame da chamada sétima geração. Seus principais concorrentes são o Microsoft X-Box 360 e o Sony Playstation 3. Lançado em dezembro de 2006 logo se tornou um sucesso de venda batendo seus rivais devido a duas grandes armas: seu baixo preço (metade de um Playstation 3) e seu controle capaz de detectar movimentos com precisão.

O núcleo de processamento é um processador baseado no PowerPC chamado de"Broadway" desenvolvido pela IBM[4]. Rodando a 729 MHz e com baixo consumo de energia, especulações dizem que o "Broadway" é uma evolução do "Gekko", o processador que equipava o Nintendo Gamecube (console predecessor do Wii).

Fabricado atualmente na tecnologia de 90 nm SOI (Silicon on Insulator), conta com registradores inteiros de 32-bits e suporte a extensão PowerPC de 64-bits, além de instruções SIMD. Como se trata de um projeto de processador para um hardware específico, não existem muitas informações disponíveis sobre detalhes do processador.

4.2 Sony Playstation 3

Projetado para ser o mais avançado videogame disponível no mercado e lançado em novembro de 2006 dispunha do processador mais avançado do mercado: o Cell [3].

O Cell, ou formalmente conhecido como Cell Broadband Engine Architecture, é um processador multi-cores projetado com conceitos totalmente inovadores pela aliança Sony-Toshiba-IBM. O design dele contempla dois tipos de cores: baseado em PowerPC e baseado em um modelo inovador chamado Synergistic.

Neste design os elementos de processamento PowerPC (PPE) e os elementos de processamento Synergistic (SPE) estão interligados por um bus de interconeção de elementos (EIB) que possui a forma de um anel. Este anel possui diversos canais que podem ser reservados para estabelecer canais de comunicação entre os elementos de processamento, garantindo uma comunicação com alto through-put.

O PPE suporta rodar até duas threads por core e funciona como controlador para os SPEs. Seu projeto é muito semelhante a um processador PowerPC, entretanto com diversas unidades simplificadas em busca de permitir um clock de operação maior.

Já os SPEs são processadores independentes com a memória embutida em seu core. Ou seja, os dados necesários para execução estão na memória interna dele e portanto não há a necessidade de efetuar acesso a memória externa de acesso aleatório. Esta característica evita stalls devido a acesso compartilhado de memória , que em conjunto com sua arquitetura otimizada para operações de ponto flutuante aumenta

drasticamente o desempenho das aplicações, especialmente as científicas e gráficas.

Cada SPE possui 128 registradores de 128-bits cada além de uma memória RAM interna de 256 KB que roda na velocidade do SPE. Os dados e códigos são carregados através de DMA pelo PPE para o SPE e este inicia a execução.

O projeto do Cell utilizado pelo Playstation 3 utiliza 1 PPE e 7 SPE, apesar de que o projeto inicial previa 8 SPE. Devido a baixa confiabilidade no processo de fabricação um dos cores é sempre desabilitado, restando apenas 7 para uso no Playstation 3.

Os SPE podem muitas vezes serem vistos como hardwares programáveis para determinadas funções específicas. Por exemplo, num Cell, a decodificação de video pode ser divida em estágios e cada estágio pode ser programado em um SPE. Depois programa-se o EIB para conectar as SPE na ordem correta e a decodificação de video pode ser feita sem gastar processamento do PPE.

Estas características fizeram o Cell ser o processador multimídia mais avançado atualmente e diversas aplicações vislumbradas. Como por exemplo decodificação e processamento de TV digital em tempo real.

4.3 Microsoft X-Box 360

O X-Box 360 é o segundo videogame produzido pela Microsoft. Lançado em Novembro de 2005, quase um ano antes de seu principal competidor o Playstation 3 ele dispunha de um processador totalmente novo desenvolvido pela IBM chamado de Xenon.

O Xenon é um processador compatível com o PowerPC contendo 3 cores. Estes cores são versões levemente modificadas do PPE que equipa o processador Cell [7].

Embora o desenvolvimento do Cell tenha se iniciado antes do Xenon, o projeto do Xenon teve a vantagem de utilizar os cores já quase prontos que foram desenvolvidos para o Cell. Desta forma o processador ficou pronto quase um ano antes do Cell e deu uma vantagem comercial ao X-Box 360 contra o Playstation 3 da Sony.

5. FUTUROS LANÇAMENTOS

A arquitetura PowerPC continua bastante ativa no desenvolvimento de novos processadores compatíveis. Atualmente existem três grandes projetos que estão nos holofotes da mídia. Um processador para dispositivos embarcados e o tão aguardado POWER7 para grandes máquinas.

5.1 POWER7

Apesar de seu desenvolvimento ser sigiloso, a IBM confirmou que um supercomputador chamado "Blue Waters" está sendo construído para a universidade de Illinois e este supercomputador empregará chips POWER7 com 8 cores.

A especificação publicada do "Blue Waters" caracteriza cada POWER7 com quatro cores e cada core capaz de rodar quatro threads. Além disso o POWER7 utiliza um design com módulos de dois chips, chegando aos 8 cores anunciados e 32 threads por chip.

Estes cores serão feitos na tecnologia de 45nm e seu clock atingirá 4 GHz, isso torna o POWER7 duas vezes mais rápido que os chips POWER6 atuais.

O projeto "Blue Waters" pretende empregar 38900 chips POWER7 cada um com 8 cores rodando a 4 GHz. Não só o quantidade de processador é monstruosa, mas também a memoria: o sistema conta com cerca de 620 TB de memória RAM. Isso tudo para bater a marca de 10 petaflops de pico.

5.2 e700

Projetado pela Freescale Semiconductors com a promessa de ser da família da nova geração de 64 bits de alto desempenho para dispositivos embarcados, entranto não foi liberado muitas informações sobre ele exceto que a família contará com processadores com clocks desde 667 MHz até 3 GHz.

6. CONCLUSÃO

Através desse passeio pela história da arquitetura POWER podemos ver como um projeto que se iniciou na metade da década de 70 em busca de uma solução para telefonia terminou 35 anos depois como uma arquitetura altamente empregada nos mais diversos sistemas, desde sistemas embarcados até supercomputadores.

Também é notável a incorporação de novas tecnologias a linha POWER ao longo dos anos. Surgindo o primeiro processador numa forma bem simples, e foi ganhando com o tempo capacidade superescalar através de pipelines e multiplas unidades de processamento (aritimético, branches, load e store, etc.), processamento 64 bits, operações vetoriais (SIMD), multiplos cores, multiplas threads e virtualização.

Diversos supercomputadores foram construídos e estão sendo construídos empregando chipes POWER, além da notável dominação no mercado de videogames, expulsando os tradicionais MIPS. Hoje em dia podemos encontrar máquinas POWER na grande maioria das grandes empresas, bancos e instituições de pesquisas devido ao seu alto poder de processamento.

Lembramos também o curioso fato que a arquitetura POWER não está no monopólio de apenas uma grande empresa e sim em uma instituição, a Power.org, que através de seus comitês determina os rumos da PowerPC ISA. Desta forma, inúmeras empresas ao redor do mundo desenvolve chips POWERs que atendam necessidades específicas do mercado.

Desta forma, concluímos que é possível uma arquitetura sobreviver ao longo de décadas evoluindo e acompanhando o mercado, produzindo processadores novos de altíssimo desempenho a cada geração sem esbarrar no problema da arquitetura legado.

7. REFERENCES

[1] R. H. J. e. a. Anderson, S.; Bell. *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*. IBM Corp, 1998.

[2] J. Cocke and V. Markstein. The evolution of risc technology at ibm. *IBM Journal of Research and Development*, 34:4–11, 1990.

[3] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[4] IBM. Ibm ships first microchips for nintendo's wii video game system. *IBM Press room*, 1996.

[5] C. e. e. a. . May. *The PowerPC Architecture: A Specification for A New Family of RISC Processors.* Morgan Kaufmann Publishers, 1994.

[6] Power.org. *Power ISA v2.06 Now Available.*

[7] D. Takahashi. Learning from failure - the inside story on how ibm out-foxed intel with the xbox 360.

A Arquitetura VLIW

Conceitos e Perspectivas

Thomaz Eduardo de Figueiredo Oliveira
Instituto de Computação - Unicamp
thomaz.oliveira@students.ic.unicamp.br

RESUMO

Ao final da década de 2000 os projetistas enfrentam sérios obstáculos para manter o aumento contínuo de performance nos microprocessadores. Nos paradigmas atuais, aumentar a performance resulta normalmente em aumento de consumo energético, tornando impraticável a implementação na nova área de sistemas embutidos. O artigo apresenta uma filosofia de arquitetura denominada VLIW, que expande o paralelismo ao compilador, delegando a este tarefas antes realizadas por hardware. Esta nova perspectiva gera uma grande simplificação nos componentes de hardware e maior fator de paralelismo em nível de instrução.

1. INTRODUÇÃO

O aumento contínuo da performance dos computadores enfrenta no final da década de 2000 sérios obstáculos: os microprocessadores chegaram em seus limites físicos; os multiprocessadores precisam superar as dificuldades de comunicação, coerência e divisão de recursos; e os superescalares lidam com problemas para reduzir o overhead proveniente da busca de instruções paralelas. Especialistas na área entendem que novos paradigmas devem ser construidos para que a evolução da performance continue.

Assim, a arquitetura VLIW foi concebida, visando uma nova forma de não só produzir, como aumentar o nível do paralelismo em nível de instruções (ILP).

O artigo apresentará uma breve história da arquitetura, assim como suas características principais. Logo, mostraremos algumas comparações entre os paradigmas atuais, alguns problemas gerados pela aquiterura a serem resolvidos e alguns exemplos de implementações práticas. Por fim, concluimos com a situação atual do VLIW.

2. HISTÓRIA

No inicio da década de 80, um pesquisador chamado Joseph Fisher estava ciente das limitações do ILP nas arquiteturas da época, que eram capazes de gerar no máximo duas a três operações paralelas.

Fisher compreendeu que a dificuldade não estava na execução de múltiplas operações simultâneamente (já haviam máquinas que processavam muitas instruções de uma vez, porém a codificação era manual), mas na capacidade do hardware encontrar paralelismo no código.

Com isso, ele cria o VLIW (Very Long Instruction Words), que na teoria seria capaz de executar de dez a trinta instruções em um ciclo.

Juntamente com a criação da arquitetura, Fisher desenvolve o escalonamento por traçado, uma técnica de compilação que processa o código para encontrar um caminho que tenha uma frequência de execução altíssima, independente da entrada do programa.

Esta técnica permitiu a busca de instruções paralelas além dos blocos básicos.

Fisher desenvolve então, uma máquina denominada ELI-512 junto com um compilador chamado Bulldog, enfrentando alguns obstáculos como o problemas dos desvios e o acesso múltiplo à memória.

Em seguida lança o processador comercial Multiflow, que não foi muito aceito no mercado da época (apenas cem unidades vendidas) e vai à falência.

Alguns anos mais tarde, grandes empresas como HP e Philips passam a se interessar pela arquitetura e convidam Fisher e sua equipe a prosseguir com o desenvolvimento do VLIW.

3. CARACTERÍSTICAS

Com as grandes mudanças que a arquitetura recebeu desde o seu nascimento, muitos autores encontram dificuldades em definir claramente as características do VLIW. Podemos listar alguns principios básicos que os projetos precisam seguir:

- As instruções do VLIW consistem em operações paralelas, definidas em tempo de compilação, ou seja, o hardware presume que o conjunto de operações geradas pelo compilador podem ser executadas ao mesmo tempo. Estas instruções podem ser padronizadas, com cada operação específica em uma determinada posição.

- Muita informação do hardware é visível ao compilador no momento de gerar as instruções paralelas. Desta forma, para garantir que o programa funcione corretamente, é necessário compilar o mesmo para cada implementação específica.

- O hardware confia no paralelismo gerado pelo compilador, portanto, ele não verifica dependências entre instruções nem recursos disponíveis em tempo de execução.

- Algumas peculiaridades não são expostas ao compilador, deixando a responsabilidade ao hardware. Como exemplo, o tratamento de interrupções.

A filosofia do VLIW portanto, resume-se em infundir paralelismo em toda a arquitetura, extendendo tembém o compilador como parte desta. Com isso, os problemas de identificar unidades funcionais disponíveis, dependências e especulação passam a ser resolvidos na compilação.

A possibilidade de enxergar o compilador como parte da arquitetura revela algumas vantagens:

- O compilador possui uma janela de visualização do código muito maior que o hardware de despacho de instruções, podendo assim buscar paralelismo em trechos maiores do programa, como loops; além de ser capaz de identificar variáveis que são constantes no progrma, podendo especular em desvios condicionais.

- A complexidade no compilador é menos custosa no geral, pois o trabalho é realizado apenas uma única vez. Ao contrário do escalonamento em hardware, que precisa sempre buscar ILP nos programas.

- O projeto do hardware pode ser mudado com maior facilidade, e menos esforço. Simuladores também podem ser mais confiávies, pois o hardware não modificará a maneira como o codigo será executado.

Estas vantagens diminuem significantemente o custo do hardware, tanto no âmbito financeiro como no de consumo energético.

Quando a instrução longa chega no hardware, cada operação é diretamente alocada para cada unidade funcional responsável. Se não houver operações suficientes, a unidade fica ociosa no ciclo em questão.

Muitas implementações de VLIW foram feitas nos últimos anos, algumas caracteristicas foram adicionas e outras retiradas. Algumas vezes, os autores inclusive enxergaram a necessidade de nomear seus projetos baseados no VLIW de maneira distinta, como exemplo, a arquitetura EPIC desenvolvida em conjunto entre Intel e HP.

4. COMPARAÇÕES

A arquitetura VLIW tem bastante semelhanças com o modelo RISC. Na verdade, muitos afirmam que o VLIW é apenas uma implementação da arquitetura RISC, pois usa os mesmos conjuntos de instruções simples e regulares, além de utilizar-se de técnicas de pipeline.

As diferenças se encontram no tamanho das instruções e o número de unidades funcionais, pois o VLIW pode processar um número maior de instruções RISC por vez. Com isso, o pipeline de uma arquitetura VLIW é distinto, que deve ter várias unidades para decodificação/leitura de registros para dar suporte às várias operações contidas na instrução.

Tabela 1: Comparação VLIW e Superescalares

	Superescalares	VLIW
Instruções	Formadas por operações escalares únicas	Formadas por múltiplas operações escalares
Escalonamento	Dinâmico por hardware	Estático pelo compilador
Num. instr. despachadas	Definida dinâmicamente pelo hardware, levando em conta as dependências e os recursos disponíveis	Determinado estaticamente pelo compilador
Ordem	Permite execução fora de ordem	Execução apenas em ordem

Outro fato importante é que a unidade de despachamento de instruções é bem mais simples nas arquiteturas VLIW, assim como os decodificadores e seu buffer de reordenação. Com isso, muitos outros registradores podem ser alocados.

A maior peculiaridade no VLIW porém, é o desenvolvimento conjunto de técnicas de compilação que sejam cooperativas com a arquitetura. Os superescalares por sua vez, executam aplicações cujos compiladores foram projetados para reduzir tamanho de código e tempo de execução. Estas características serviriam para processadores escalares, mas prejudicam arquiteturas que buscam maior fator de paralelismo.

Os dois esquemas podem ser comparados nas Figuras 1 e 2. Nesta comparação, o mesmo fator de ILP foi atingido nos dois casos. A diferença é o meio como são alcançados. Enquanto no Superescalar, são componentes de hardware que buscam as instruções, no VLIW este trabalho é feito pelo compilador, antes do tempo de execução.

5. PROBLEMAS

Como qualquer arquitetura, o modelo VLIW também possui alguns problemas.

Primeiramente, as técnicas atuais dos compiladores não são capazes de gerar instruções paralelas para qualquer tipo de aplicação. Programas muito interativos por exemplo, não podem ter suas instruções paralelizaveis. Isso irá gerar instruções grandes com poucas operações, tornando ociosas muitas unidades funcionais.

Além disso, o fato do compilador produzir instruções bastante extensas, que podem chegar até a 512 bits, consequentemente gera códigos extensos, tornando um obstáculo em sistemas com restrição de memória.

Outra observação é que algum overhead precisa ser introduzido para sincronizar as várias unidades funcionais, ou seja,

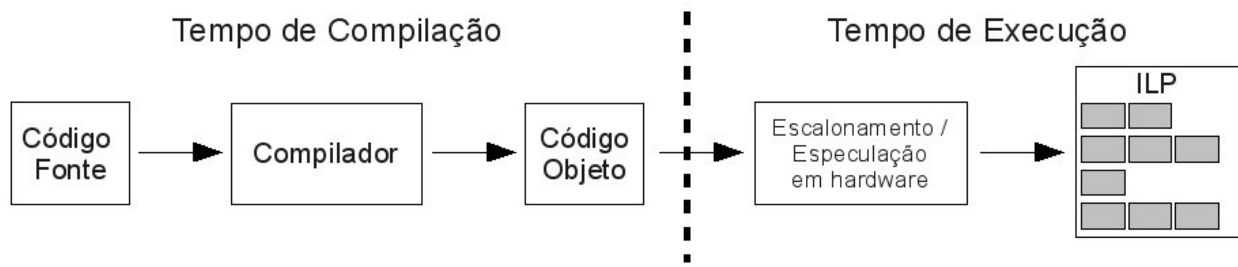


Figura 1: Esquema Superescalar

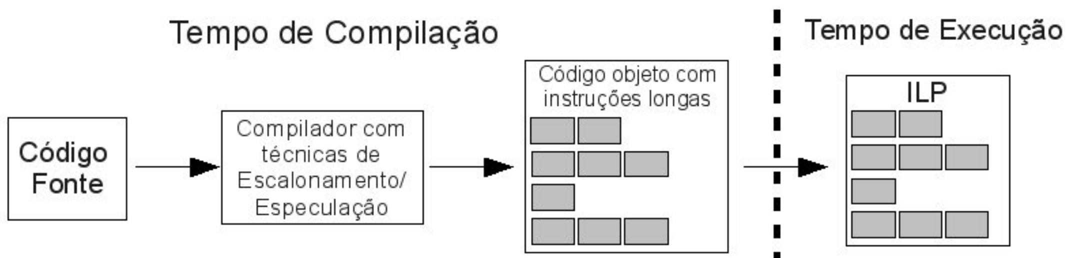


Figura 2: Esquema VLIW

dependendo do fator de paralelização, esta quantidade pode se tornar considerável.

Finalmente, como a arquitetura se apoia no compilador, um mesmo aplicativo não pode ser executado em máquinas VLIW com diferentes fatores de paralelismo. A razão é que o número de operações em cada instrução seriam diferentes em máquinas distintas. As novas gerações de VLIW, permitem maior flexibilidade de execução, no entanto, as unidades funcionais continuariam ociosas.

6. MERCADO E SITUAÇÃO ATUAL

No mercado de desktops e servidores o VLIW foi pouco aceito, as poucas empresas que lançaram processadores para este mercado (Multiflow e Cydrome) não obtiveram sucesso e logo faliram.

Grande parte do fracasso do VLIW deve-se à chamada ”inércia de conjunto de instruções” que domina as maquinas no mercado, ou seja, os usuários são desejosos em protejer seus investimentos feitos em software compilado historicamente em arquiteturas CISC ou RISC com instruções simples. Assim, é preferível manter o programa com um grau limitado de eficiência a fazer grandes investimentos para mudar a arquitetura.

Porém, o mercado de sistemas embarcados não se incluem neste esquema, pois são os próprios fabricantes que consomem os softwares, o que permite maior facilidade para mudança de arquiteturas.

Este mercado justamente é o mais promissor para o VLIW, que traz algumas soluções para os problemas peculiares destes sistemas, como a flexibilidade e facilidade de projetar o hardware, além do menor consumo energético.

Algumas empresas tem projetos envolvendo a arquitetura. Entre elas a Philips que produz sistemas embarcados para celular. Podemos citar também a HP, que tem o próprio Joseph Fisher como diretor de sua divisão de pesquisa.

7. PROCESSADORES

Nesta parte iremos descrever alguns processadores VLIW produzidos para o mercado de sistemas embutidos.

7.1 Transmeta Crusoe e Efficeon

Os processadores da Transmeta possuem uma caracteristica bastante interessante, o conjunto de instruções é implementado em software ao invés de ser colocado no hardware. Assim, é possível emular qualquer arquitetura, inclusive a x86.

Usando este princípio, a empresa desenvolveu dois processadores VLIW, o Crusoe (com instruções de 128 bits) e o Efficeon (com 256 bits).

Seus processadores são famosos por garantir economia de energia, sendo implementados em vários produtos.

7.2 Philips TriMedia

Produzido pela NXP, divisão de semicondutores criada pela Philips, os processadores TriMedia são especializados em processar dados multimedia. Eles contam com 5 a 8 operações em uma instrução. A redução de hardware no processador permitiu a alocação de 128 registradores de 32-bits.

Estes processadores também são muito usados no mercado, como na área de aparelhos celulares.

7.3 Tensilica Xtensa LX2

O processador Xtensa LX2 emite instruções de 32 ou 64 bits. Através de sua arquitetura denominada FLIX (Flexible Length Instruction Xtension) garante flexibilidade nas

instruções. Otimizando assim a performance e o consumo de energia.

Outra grande vantagem é o compilador XCC, que gera paralelismo diretamente de programas escritos na linguagem C ou C++.

8. CONCLUSÃO

Pode-se afirmar que dentre as formas de obter paralelismo – pipelining, processadores múltiplos, superescalares – a arquitetura VLIW é a mais simples e barata; porém contem algumas limitações, agora relacionadas ao compilador, ou seja, para que arquitetura seja eficiente serão necessárias pesquisas para improvisar técnicas de compilação

Além disso, acredita-se que mesmo os compiladores em algum momento encontrarão limitações para encontrar instruções paralelas. Para isso, novos paradigmas de programação seriam necessários para que o paralelismo seja pensado desde o desenvolvimento do aplicativo.

9. REFERENCIAS

[1] Joseph A. Fisher, *Very Long Instruction Word architectures and the ELI-512*. International Symposium on Computer Architecture. Proceedings of the 10th annual international symposium on Computer architecture: 140-150, New York, USA.

[2] Joseph A. Fisher, *Retrospective: Very Long Instruction Word Architectures and the ELI- 512*. Hewlett-Packard Laboratories, Cambridge, Massachusetts.

[3] *An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture*. Philips.

[4] John L. Hennessy, David A. Patterson, *Arquitetura de Computadores: Uma abordagem Quantitativa*. Editora Campus, Terceira Edição, 2003.

[5] Joseph A. Fisher, Paolo Faraboschi, Cliff Young, *Embedded Computing, A VLIW Approach to Architecture, Compilers and Tools*. Elsevier, 2005.

O Processador Intel Core i7

Bruno Teles
Instituto de Computação
Universidade Estadual de Campinas
Av. Albert Einstein, 1251
55-19-35215838
seletonurb@gmail.com

ABSTRACT
Neste paper, são descritas características do processador quadcore da Intel Core i7, como seu datapath, gerenciamento de memória, gerenciamento de energia e ganhos de desempenho que podem ser obtidos com as inovações.

Categorias e Descritores de Temas
C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) – *Interconnection architectures, Multiple-instruction-stream, multiple-data-stream processors (MIMD).*

Termos Gerais
Measurement, Documentation, Performance, Design.

Palavras-Chave
Microarquitetura, Nehalem, quadcore, memória cache, consumo de energia, multithread.

1. INTRODUÇÃO
No fim de 2008, a Intel lançou seu novo processador quad-core. Sob o nome de Core i7, este é o primeiro processador baseado na micro-arquitetura Nehalem. Os processadores Intel Core i7 oferecem uma série de inovações em desempenho quad-core e contam com avançadas tecnologias de processador. Esta nova micro-arquitetura não prevê uma mudança tão radical quanto a passagem de Netburst (Pentium 4) para Core2, pelo menos não em um nível tão baixo. O que não significa que não venha a trazer ganhos significativos em desempenho.

Tomando como base os núcleos Core2, a Intel se questionou sobre o que se poderia ser feito para obter processadores ainda melhores. Além de melhorias na já excelente micro-arquitetura, uma das principais mudanças, ou pelo menos a que chama mais atenção é que finalmente a controladora de memória foi integrada ao processador, aposentando o esquema de FSB e Northbridge, aproveitando a oportunidade para a implementação de um novo barramento; serial, ponto-a-ponto, bi-direcional e de baixa latência: o QPI (Quick Path Interconnect); para conexão do processador com o chipset ou outros processadores. Adicionado a isto, será feita menção à retomada do HyperThreading, à utilização do Turbo Boost e à organização da memória, cruciais para a compreensão do funcionamento desta arquitetura.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'04, Month 1–2, 2004, City, State, Country.
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

2. A ARQUITETURA NEHALEM
2.1 Visão Geral
A micro arquitetura Nehalem substitui a arquitetura Core2 em praticamente todas as frentes: desde processadores para dispositivos móveis de baixíssimo consumo até potentes servidores, passando principalmente pelo desktop. Esta micro arquitetura toma como base o excelente núcleo da micro arquitetura Core2, inovando fundamentalmente nos componentes ao redor do núcleo, com destaque para a controladora de memória e a interconexão por QPI.

Apesar de obsoleto para os padrões atuais, o AGTL+ (sigla de Assisted Gunning Transceiver Logic, o FSB utilizado pela Intel) é muito versátil. Sem ele não seria possível criar tão facilmente processadores dual-core e quad-core como os Pentium D e os Core 2 Quad. Sua origem está ancorada em uma característica do GTL que permite o compartilhamento do FSB por mais de um processador, até quatro processadores podem ser instalados sobre o mesmo FSB. Sobre cada FSB podem ser instalados até quatro processadores (chips), não importando quantos núcleos cada processador tenha.

Com dois processadores, o gargalo não é tão evidente. Dessa Maneira, pode-se instalar dois chips dentro do mesmo encapsulado, dobrando o número de núcleos sem grandes dificuldades técnicas (com exceção de consumo e aquecimento, que também são dobrados). O principal benefício disso é otimizar a produção. Assim, em vez de ter linhas separadas para processadores dual-core e quad-core, por exemplo, fabrica-se apenas um tipo de chip dual-core que podem dar origem tanto a processadores dual-core como quad-core. Se um Core 2 Quad fosse feito a partir de um único chip com os quatro núcleos, sua produção seria muito menor e seu preço muito maior.

Essa estratégia funciona muito bem em desktops e workstations com apenas um processador e razoavelmente bem em máquinas com dois processadores, embora exigindo o uso de um potente (literalmente, devido ao alto consumo elétrico) chipset com dois FSBs e controladora de memória de quatro canais. Já que além de potentes, os núcleos Core2 contam com enormes caches L2, equipados com agressivos sistemas de prefetch para atenuar a latência no acesso à memória.

Como contam com controladora de memória integrada e interconexão por QPI (barramento serial, ponto a ponto, bi-direcional e de baixa latência), os processadores Nehalem não terão empecilhos com a limitação do acesso ao chipset (onde fica a controladora de memória).

Desenvolvida com foco na modularidade, a micro arquitetura Nehalem permite que sejam criados diversos tipos de processadores, com características mais adequadas a cada segmento. Entre a biblioteca de componentes pode-se escolher de dois a oito núcleos, quantos links QPI forem necessários e até um processador gráfico.

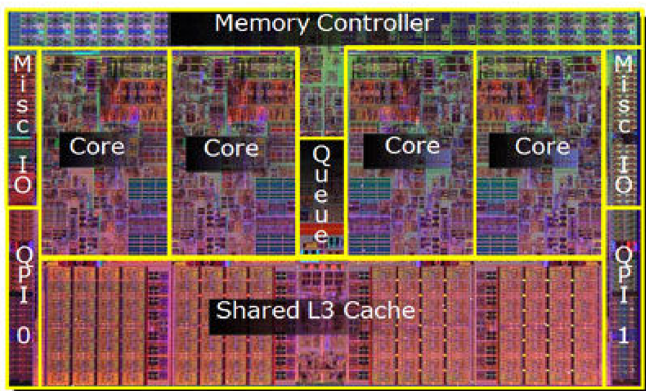


Figura 1. Foto do die do processador Nehalem.

Um ponto que merece destaque é a volta do Hyper-Threading (discutida mais detalhadamente na próxima seção), que em uma micro arquitetura de pipeline curto será muito mais útil que no Pentium4, contando ainda com o benefício da maior facilidade em trazer dados até o núcleo, tanto em função dos caches maiores, mais decodificadores e menor latência no acesso à memória graças à controladora de memória integrada. Como a tendência é o aumento da paralelização dos programas, a capacidade de executar mais threads ao mesmo tempo pode trazer ótimos ganhos em desempenho. O restante são alguns artifícios implementados para ganhar desempenho, como os feitos na revisão do Core2 de 65nm para 45nm:

- Aumento dos buffers de instruções para acomodar o maior número de instruções devido à implementação do Hyper-Threading, a reconstrução de instruções divididas entre linhas de cache foi acelerada, as primitivas de sincronização de processos foram aceleradas (importante já que cada núcleo pode trabalhar com dois threads simultaneamente).

- O previsor de desvios agora conta com dois estágios. O segundo estágio possui um histórico maior, portanto é mais lento, mas pode realizar previsões mais precisas. Normalmente a previsão do primeiro nível já é consistente o suficiente para adivinhar que caminho seguir e com a adição do segundo nível se a previsão do primeiro nível não for confiável, passa-se ao segundo. Assim diminuem ainda mais as chances de uma previsão incorreta e da necessidade de retornar até o desvio para continuar a execução, desperdiçando vários ciclos e ainda impondo uma penalização de outros tantos ciclos para reorganizar o pipeline. O que, por coincidência, também foi acelerado com a ajuda do "Renamed Return Stack Buffer", que guarda cópias dos dados já calculados. Dessa maneira, em caso de previsão incorreta, menos dados deverão ser recalculados.

- Mais algumas instruções foram incluídas ao conjunto SSE4, com foco em processamento de texto útil para servidores de bancos de dados.

- E por fim, a TLB (Table Look-a-side Buffer, tabela para consulta rápida de endereços de memória) agora possui um segundo nível, com 512 entradas.

2.2 A Organização da Memória no Nehalem

Devido à integração da controladora de memória e ao aumento do número de núcleos dentro do chip, o arranjo de memórias cache foi refeito. O cache L1, com 32KB para dados e 32KB para instruções, foi mantido inalterado, mas o cache L2 mudou radicalmente. Com apenas dois núcleos faz sentido ter um grande cache L2 compartilhado, pois a concorrência no acesso é baixa. Mas com 4 núcleos (ou mais) a concorrência seria muito maior, prejudicando o desempenho. Por isso, foi incluído um cache L2 para cada núcleo (pequeno, mas de baixa latência), enquanto o grande cache compartilhado por todos os núcleos passa a ser o L3. Nos primeiros processadores, destinados a máquinas desktop, workstations e servidores com um ou dois processadores, o cache L3 terá respeitáveis 8MB. O cache L2 (256KB), que deve permanecer constante para todas as versões, mas o tamanho do cache L3 deve variar conforme o perfil do processador produzido, podendo também ser eliminado em processadores de baixo custo. Outro detalhe interessante na primeira geração da Nehalem é que a controladora de memória conta com três canais. A Intel anuncia que o ganho em largura de banda de memória de uma máquina Dual Nehalem sobre um Dual Harpertown ("Core2 Xeon" de 45nm com FSB1600) atual será superior a quatro vezes. E é bom ver que a Intel abandonou as memórias FB-DIMM em favor das DDR3, que consomem menos e têm latência menor. Nas plataformas Intel atuais, as memórias (FB-DIMM) e o northbridge (com dois ou quatro FSBs e controladora de memória de 4 canais) representam uma parcela considerável do consumo elétrico da máquina. Com a integração da controladora de memória no processador, o chipset deixará de consumir tanta energia, passando a um simples controlador PCI Express. Enquanto que as memórias deixarão de consumir cerca de 12W por módulo, passando para apenas 5W.

2.3 Consumo de Energia

Nesta breve análise do consumo de energia, considera-se uma máquina com dois processadores quad-core e oito módulos de memória FB-DIMM para comparação. Cada processador consome até 120 W e cada módulo de memória 12 W. Somados aos quase 40 W do northbridge, tem-se aproximadamente 375 W, sem considerar o restante da máquina. Em uma máquina semelhante, baseada em processadores Nehalem, o consumo dos processadores deve permanecer o mesmo, mas o consumo do northbridge cai para níveis desprezíveis (10 a 15 W) e mesmo aumentando o número de módulos de memória para 12 (totalizando apenas 60 W, contra 96 W dos oito módulos FB-DIMM do caso anterior) o consumo do "conjunto-motriz" deve cair para cerca de 315 W. Neste caso já pode-se constatar uma redução de pelo menos 50w no consumo, que vem acompanhado de um sensível aumento no desempenho.

3. HYPER-THREADING

A tecnologia Hyper-Threading, desenvolvida pela própria Intel, é mais uma técnica criada para oferecer maior eficiência na utilização dos recursos de execução do processador. Esta tecnologia simula em um único processador físico dois processadores lógicos. Cada processador lógico recebe seu próprio controlador de interrupção programável (APIC) e conjunto de registradores. Os outros recursos do processador físico, tais como, cache de memória, unidade de execução, unidade lógica e aritmética, unidade de ponto flutuante e barramentos, são compartilhados entre os processadores lógicos. Em termos de software, significa que o sistema operacional pode enviar tarefas para os processadores lógicos como se estivesse enviando para processadores físicos em um sistema de multiprocessamento.

A nova micro-arquitetura marca o retorno do Hyper-Threading, que cria dois núcleos virtuais a partir de cada núcleo físico. Como os Core i7 são processadores quad-core, tem-se um total de 8 núcleos virtuais. O Hyper-Threading ou SMT (simultaneous multi-threading) tem muito a oferecer em processadores mais novos como o Core i7. Por se tratarem de núcleos 4-issue wide (isto é, a cada ciclo cada núcleo faz o fetch de 4 instruções para processar), o HT tem mais oportunidades de promover um melhor aproveitamento de suas unidades de execução. Ainda mais porque há uma cache maior e uma largura de banda de memória bem mais alta. Isto é ilustrado na imagem abaixo.

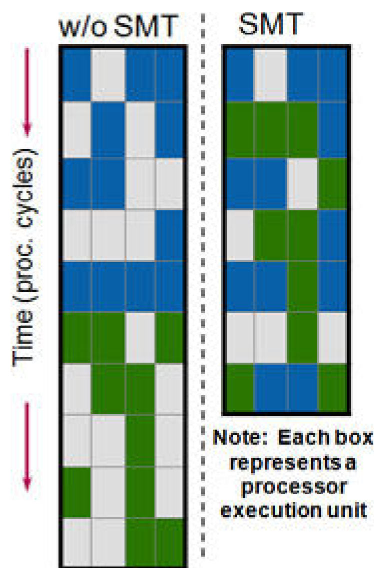


Figura 2. Cada uma das quatro unidades de execução em cada núcleo, sem o uso de HT e com o uso de HT.

Cada espaço representa uma unidade de execução. Em vez de esperar que um bloco de instruções seja executado, para então passar ao seguinte, o núcleo físico pode receber mais instruções simultaneamente com dois núcleos virtuais. Se estas forem de tipos diferentes, sua execução pode ser agendada ao mesmo tempo em que outras instruções são executadas, em outras unidades; proporcionando um significativo ganho em desempenho.

Alguns testes de desempenho feitos pela Intel mostraram que esta abordagem pode apresentar ganhos em performance em algumas aplicações específicas no Core i7. Testes mostraram que o ganho sobre operações com inteiros e com pontos flutuantes apresentaram ganhos de 13% e 7%, respectivamente. O 3DMark Vantage CPU, um simulador de física e IA apresentou um ganho de 34% quando comparado à simulação sem HT.

4. TECNOLOGIA TURBO BOOST

A tecnologia Turbo Boost, também desenvolvida pela Intel, concerne ao controle de energia e freqüência de operação de acordo com a necessidade de uso dos núcleos. De forma automática, ela permite que os núcleos do processador trabalhem mais rápido que a frequência básica de operação quando estiverem operando abaixo dos limites especificados para energia, corrente e temperatura. A tecnologia Intel Turbo Boost é ativada quando o sistema operacional (SO) solicita o estado de desempenho mais elevado.

A frequência máxima da tecnologia Intel Turbo Boost depende do número de núcleos ativos. O tempo que o processador gasta no estado da tecnologia Turbo Boost depende da carga de trabalho e do ambiente operacional, proporcionando o desempenho de que você precisa, quando e onde for necessário. Os elementos que podem definir o limite superior da tecnologia Turbo Boost em uma determinada carga de trabalho são os seguintes: número de núcleos ativos, consumo estimado de corrente, consumo estimado de energia e temperatura do processador.

Quando o processador estiver operando abaixo desses limites e a carga de trabalho do usuário exigir desempenho adicional, a frequência do processador aumentará dinamicamente 133 MHz em intervalos curtos e regulares até ser alcançado o limite superior ou o máximo upside possível para o número de núcleos ativos. Por outro lado, quando algum desses limites for alcançado ou ultrapassado, a frequência do processador cairá automaticamente 133 MHz até que ele volte a operar dentro dos seus limites.

O modo Turbo Boost explora a economia de energia através do aumento de frequência de um único núcleo, caso necessário. Esta ação corrobora com a preocupação da Intel em sempre melhorar a performance das aplicações usadas hoje em dia. Como boa parte das aplicações hoje em dia ainda não são multithreaded, de forma que tirem o maior proveito de Hyper-Threading, o 'overclocking' do modo Turbo irá fazer com que estas aplicações sejam executadas em menos tempo.

5. QUICKPATH INTERCONNECT

Uma importante mudança no projeto da CPU foi a troca do antigo barramento FSB (Front Side Bus), que compartilhava acessos entre a memória e a I/O, pelo novo barramento QPI (QuickPath Interconnection), que é projetado para aumentar a largura de banda e diminuir a latência.

O QPI utiliza dois caminhos para a comunicação entre a CPU e o chipset, como pode ser visto na Figura 2. Isto permite que a CPU faça a operação de transmissão e recepção dos dados de I/O ao mesmo tempo, isto é, os datapaths de leitura e escrita para esta função são separados. Cada um destes caminhos transferem 20 bits por vez. Destes 20 bits, 16 são utilizados para dados e os

restantes são usados para correção de erro CRC (Cyclical Redundancy Check), que permite ao receptor verificar se os dados recebidos estão intactos.

Além disso, o QPI trabalha com uma freqüência de 3.2 GHz transferindo dois dados por ciclo (uma técnica chamada DDR, Double Data Rate), fazendo o barramento trabalhar como se estivesse operando a uma taxa de 6.4GHz. Como 16 bits são transmitidos por vez, tem-se uma taxa teórica máxima de 12.8 GB/s em cada um dos caminhos. Comparado ao FSB, o QPI transmite menos bits por ciclo de clock mas opera a taxas muito maiores. Uma outra vantagem em ralação ao FSB, é que como o FSB atende à requisições de memória e de I/O, há sempre mais dados sendo transferidos neste barramento comparados ao QPI, que atende apenas às requisições de I/O. Por isso, o QPI fica menos ocupado, e assim, maior largura de banda disponível. Por último, o QPI utiliza menos ligações do que o FSB

Uma característica incorporada ao QPI são os modos de energia que ele pode assumir chamados de L0, L0s e L1. O L0 é o modo no qual o QPI está em funcionamento pleno. O estado L0s indica os fios de dados e os circuitos que controlam estes fios estão desativados para economia de energia. E em L1 todo o barramento está desativado, economizando ainda mais energia. Naturalmente, o estado L1 necessita de um tempo maior para reativação do que o L0s.

Existe também uma técnica introduzida para aumentar a confiabilidade do QPI. O QuickPath permite que cada caminho de 20 bits ainda seja dividido em outros quatro de 5 bits. Esta divisão melhora a confiabilidade principalmente em ambientes servidores.

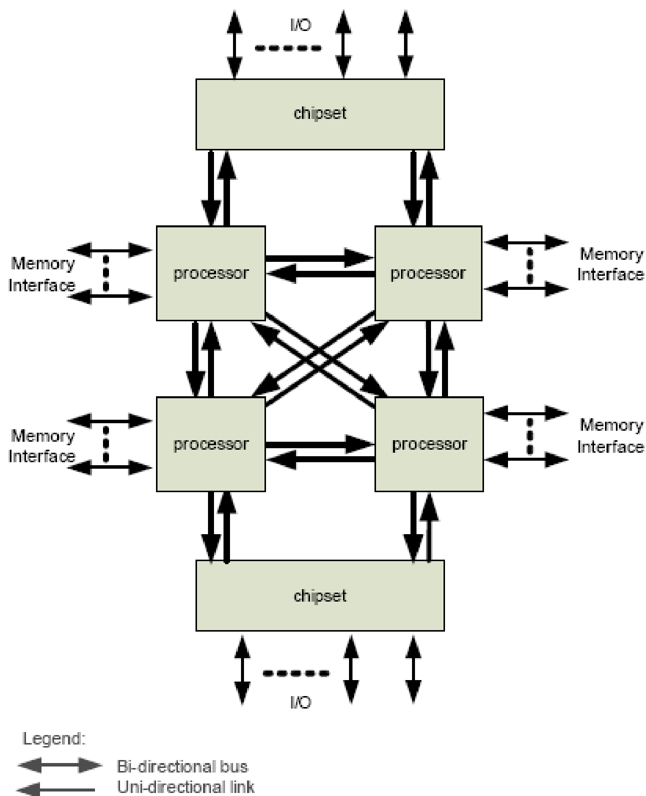


Figura 3. QuickPath Interconnect da Intel.

Quando esta funcionalidade é implementada, o receptor de dados pode perceber que a conexão entre ele e o transmissor foi danificada, e assim, desativar a porção do barramento que foi danificada e operar com a transmissão de menos bits por vez. Isto diminui a taxa de transmissão mas por outro lado o sistema não falha.

5.1 Comunicação em Camadas

Teoricamente, o barramento QPI deveria ser chamado de uma conexão ponto-a-ponto, pois conecta apenas dois dispositivos. Entretanto, vale ressaltar que os dados são enviados em paralelo através das várias conexões ponto-a-ponto existentes.

Assim como se faz em redes de computadores, a comunicação do barramento é feita por pacotes, que são quebrados em múltiplas transferências que ocorrem em paralelo, e possui cinco camadas, descritas brevemente a seguir:

- física: são os próprios fios que transportam o sinal, assim como o circuito e lógica necessários para realizar a transmissão e recebimento de 0s e 1s. A unidade de transferência na camada física é de 20 bits, chamada de Phit (Physical Unit).
- de enlace: responsável por tornar confiável a transmissão e o fluxo de controle.
- de roteamento: decide o caminho a ser percorrido pelo pacote na malha.
- de transporte: possui uma avançada capacidade de roteamento para que a transmissão fim-a-fim seja confiável.
- de protocolo: conjunto de regras de alto nível para a troca de pacotes de dados entre os dispositivos.

5.2 Coerência de Cache

Uma outra característica importante do QPI é a implementação de um protocolo de monitoração para manter a coerência de cache entre todos os controladores de cache no Core i7. O protocolo utilizado é uma versão modificada do conhecido protocolo MESI com a introdução de um novo estado F (forward). Este estado foi introduzido a fim de permitir a limpeza de linhas de encaminhamento de cache para cache. As características de todos estes estados estão resumidas na tabela a seguir.

Tabela 1. Os estados da cache no protocolo MESIF.

Estado	Limpo /Sujo	Pode Escre-ver?	Pode Encaminhar?	Pode Mudar para…
M-Modificado	Sujo	Sim	Sim	-
E–Exclusivo	Limpo	Sim	Sim	MSIF
S–Compartilhado	Limpo	Não	Não	I
I-Inválido	-	Não	Não	-
F-Encaminhar	Limpo	Não	Sim	SI

6. GERENCIAMENTO DE MEMÓRIA

Com a controladora de memória dentro do processador, os núcleos não devem percorrer o longo caminho do FSB cada vez que necessitarem um dado da memória RAM. Aproveitando a ocasião da inclusão de vários núcleos (2, 4 ou mais) no mesmo chip, a Intel optou por implementar uma controladora de memória com 3 canais. Com uma largura efetiva de 192 bits e usando memórias DDR3 (apenas, memórias DDR2 não são suportadas), a oferta de banda de memória atinge níveis bem maiores que os convencionais.

A integração de 4 núcleos (com possibilidade para mais núcleos, conforme a necessidade/possibilidade por questões energéticas/ térmicas) no mesmo chip, requer uma reorganização na estrutura de cache. O grande cache L2 compartilhado do Core2 funciona muito bem quando há apenas 2 núcleos por chip, mas 4 núcleos disputando acesso ao cache L2 pode e tronar um gargalo. Então, transportou-se o cache compartilhado para um nível superior e entre eles criou-se um cache L2; razoavelmente pequeno, mas de latência baixíssima (para um cache L2), para diminuir a concorrência pelo grande cache L3, de 8MB; compartilhado por todos os núcleos.

Os caches dos Core2 e Nehalem são organizados de forma inclusiva. Assim, cada nível superior guarda uma cópia do nível anterior. O cache L2 de cada núcleo possui uma cópia do cache L1 e o cache L3 guarda uma cópia de cada cache L2; portanto, dos 8MB, sobram efetivamente 7MB (já que 1MB é reservado para cópia dos quatro caches L2 de 256KB). Este sistema requer cuidados para que seja mantida a consistência dos dados; pois cada vez que um cache é atualizado, suas cópias também devem ser atualizadas. Porém, facilita o compartilhamento de dados entre os núcleos, já que todos os dados presentes nos caches L1 e L2 de todos os núcleos são encontrados no cache L3.

O cache L2 é exclusivo e de comportamento chamado "victim-cache", pois só recolhe as "vítimas" do cache L1 (dados eliminados por falta de espaço). O cache L3 também é um "victim-cache", mas não é inclusivo nem exclusivo. Não guarda cópias dos demais caches, mas permite o compartilhamento de dados. Se mais de um núcleo precisar de um mesmo dado, é mantida uma cópia no cache L3 e esta é marcada com uma flag de compartilhamento para que não seja apagada por já constar em outros níveis superiores. A vantagem deste sistema é que quando um núcleo requisitar um dado à controladora de memória, este "sobe" diretamente ao cache L1, enquanto os outros caches se re-organizam, abrigando as vítimas dos níveis superiores. Porém, antes disso, cada núcleo deve requisitar aos demais se já possuem em cache o dado em questão, antes de pedi-lo à controladora de memória. Se algum núcleo o tiver, pode enviar a outro núcleo pelo crossbar e uma cópia é guardada no cache L3.

Dois aspectos importantes da estrutura de cache da Intel são a garantia de muita banda e latências excelentes. A latência do cache L1 teve que aumentar de 3 para 4 ciclos, devido à implementação do SMT (Hyper Threading) já que ambos núcleos virtuais compartilham o mesmo cache L1. Mas a latência do cache L2 caiu consideravelmente, dos 23 ciclos do cache L2 do "Penryn" (Core2 de 45nm) para apenas 10 ciclos. O cache L3 é um caso à parte, como se encontra em outra região do processador (na parte "uncore" da figura 1, onde ficam também a controladora de memória e o controlador do QPI), que segue um clock próprio e tem relação com o clock da memória; mas também é muito rápido. E a controladora de memória é especialmente eficiente, obtendo altíssimas taxas de transferência mesmo em condições pouco favoráveis, como utilizando memórias DDR3 de clock relativamente baixo (DDR3-1066, por exemplo).

7. CONCLUSÃO

Esta análise sobre as principais características que fazem do Core i7 um processador diferenciado mostrou como a indústria dispõe aos consumidores uma vasta gama de novas funções visando o ganho de desempenho em uma curta janela de tempo. Seja pelo uso do Turbo Boost, que utiliza lógica para otimizar o consumo de energia; seja pelo QPI, uma conexão ponto-a-ponto de alta velocidade, a Intel conseguiu atacar de forma abrangente os principais problemas que concernem ao multicore. Outro aspecto relevante a ser mencionado aqui é a volta do Hyper-Threading corroborando com a afirmação de que na computação algumas abordagens são recorrentes.

Mais importante do que as inovações que o Core i7 proporcionou é a consolidação de uma era na indústria que é voltada ao design com a replicação de núcleos dentro do processador, iniciada com a geração do Core Duo. Os ganhos em desempenho já obtidos em relação aos Core Duo de até 30% são apenas uma amostra do potencial deste quad-core: os ganhos poderão ser efetivamente notados quando boa parte das aplicações forem voltadas para este tipo de processador.

8. REFERÊNCIAS

[1] Intel® Core™ i7 Processor Extreme Edition Series and Intel® Core™ i7 Processor Datasheet - Volume 1. Document # 320834-002. Acessado em 02/07/2009 em http://www.intel.com/design/corei7/documentation.htm

[2] Intel® Core™ i7 Processor Extreme Edition Series and Intel® Core™ i7 Processor Datasheet - Volume 2. Document Number: 320835-002. Acessado em 02/07/2009 em http://www.intel.com/design/corei7/documentation.htm

[3] An introduction to Intel® QuickPath Interconnect white paper. Document Number: 320412-001US. Acessado em 02/07/2009 em http://www.intel.com/technology/quickpath/introduction.pdf

[4] Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors. Acessado em 02/07/2009 em http://www.intel.com/portugues/technology/turboboost/index.htm

[5] Stallings, William, Arquitetura e Organização de Computadores. Editora Prentice Hall, Quinta Edição, 2003.

[6] Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors. Acessado em 02/07/2009 em http://download.intel.com/design/processor/applnots/320354.pdf?iid=tech_tb+paper

Multicores: Uma visão geral e novos desafios

Júlio Reis, RA: 044415
Instituto de Computação
Universidade Estadual de Campinas

juliocesardosreis@gmail.com

RESUMO

A fim de manter o desempenho exponencial dos processadores ao longo dos anos, as empresas encontraram-se incapazes de continuar a melhorar efetivamente o desempenho dos microprocessadores da maneira típica — por encolhimento dos transistores e mais encapsulamento destes em *chips* com um único núcleo. Esta abordagem mostrou-se inviável, pois gera muito calor e usa muita energia. Em resposta a isso, desenvolveram aumentando no desempenho através da construção de *chips* com múltiplos núcleos (multicore). Esta nova arquitetura de processadores é projetada para melhorar o desempenho computacional e minimizar o calor (aumentando a dissipação de energia com uma solução de gerenciamento térmico) através da integração de dois ou mais núcleos de processadores em um único *chip*. Estes microprocessadores com múltiplos núcleos são uma importante inovação na evolução dos processadores e esta tecnologia poderá revolucionar o modo de se desenvolver aplicações computacionais. Assim, o objetivo deste trabalho é desenvolver uma visão geral desta nova tecnologia, mostrando suas características, potencialidades, mas também os desafios introduzidos por ela que ainda não foram resolvidos de maneira adequada na literatura.

Palavras-Chaves

Multicores; microprocessadores; Paralelismo.

1. INTRODUÇÃO

Até alguns anos atrás, a comunidade de hardware (processador) traduziu a Lei sobre densidade de transistores de Moore diretamente em ganho de desempenho em um único processador como resultado do aumento de freqüência nestes. Ultimamente, essa tradução tem sido prejudicada pelos efeitos de freqüência em consumo de energia e geração de calor. A nova realidade é que o desempenho por processador é essencialmente estático, e o aumento no desempenho é dado por um aumento do número de núcleos de processador em um *chip* [1]. Para Gorder [2] o que impulsionou a indústria para a tecnologia de múltiplos núcleos não foi apenas a necessidade de uma maior velocidade de processamento, mas a necessidade de menos calor e mais eficiência em energia. Menos calor, porque os *chips* mais rápidos aquecem mais rapidamente do que os *coolers* podem arrefecer-los, e mais eficiência em energia, pois *chips* de apenas um núcleo gastam muita energia para concluir o trabalho.

De acordo com Agarwal *et al.* [3], a tecnologia de múltiplos núcleos refere-se a um único *chip* com vários "motores de processamento" visivelmente distintos, cada um com comando independente. Esta tecnologia melhora o desempenho através de diversas partes de uma aplicação em paralelo e *chips* com apenas um núcleo, por outro lado, realizam tarefas de maneira serial [4]. Processadores com múltiplos núcleos representam uma grande evolução na tecnologia da informação. Este importante desenvolvimento está chegando em um momento em que as empresas e os consumidores estão começando a exigir os benefícios oferecidos por estes processadores, devido ao crescimento exponencial de dados digitais e da globalização da Internet. Estes processadores acabarão por se tornar o modelo de computação pervasiva, pois eles oferecem o desempenho e vantagens em produtividade que vão além das capacidades dos processadores de apenas um núcleo [5].

Os processadores com múltiplos núcleos também irão desempenhar um papel central na condução de importantes avanços em segurança para os computadores pessoais (PCs) e tecnologias de virtualização, que estão sendo desenvolvidas para proporcionar uma maior proteção, utilização dos recursos, e valor para o mercado comercial da computação [5]. Consumidores gerais também terão acesso a um melhor desempenho, o que irá expandir significativamente a utilidade dos seus PCs domésticos e sistemas computacionais na mídia digital. Estes processadores terão o benefício de oferecer desempenho sem ter de aumentar os requisitos de potência, que se traduz em maior desempenho por *watt*. Colocar dois ou mais núcleos em um único processador abre um mundo de novas possibilidades importantes. A próxima geração de aplicações de *software* provavelmente será desenvolvida utilizando o recurso destes processadores por causa do desempenho e eficiência que podem proporcionar, em comparação com os processadores com um único núcleo. Seja para estas aplicações ajudarem as empresas profissionais de animação a produzirem filmes mais realistas de maneira mais rápida por menos dinheiro, ou para criar maneiras de tornar o PC mais natural e intuitivo aos usuários, a disponibilização generalizada de processadores com múltiplos núcleos mudará o universo da computação para sempre [5].

Neste cenário, o objetivo deste trabalho é desenvolver uma visão geral da tecnologia de múltiplos núcleos, elucidando desde aspectos de sua evolução, os tecnológicos, de performance e também os desafios desta nova tecnologia. Para isso o trabalho está estruturado da seguinte maneira: A seção 2 apresenta um breve relato sobre a evolução dos processadores e o surgimento da tecnologia de múltiplos núcleos; a seção 3 objetiva apresentar as principais considerações e algumas técnicas sobre esta tecnologia; a seção 4 visa observar questões sobre desempenho de arquiteturas com diversos núcleos; na seção 5 é apresentado os

desafios e questões em aberto sobre esta tecnologia e por fim na seção 6 o trabalho é concluído.

2. EVOLUÇÃO DOS PROCESSADORES E A TECNOLOGIA MULTICORE

A Intel fabricou o primeiro microprocessador, o 4-bit 4004, no início dos anos 1970. Pouco tempo depois eles desenvolveram o 8008 e 8080, ambos de 8 bits, e a Motorola seguiu o exemplo com os seus 6800 que foi equivalente ao da Intel 8080. As empresas, em seguida, fabricaram os microprocessadores de 16-bits. Motorola teve seu 68000 e a Intel o 8086 e 8088. Esta série seria a base para o Intel 80386 de 32-bit e mais tarde a sua popular linha Pentium que estava no primeiro PC para os consumidores. Cada geração de processadores crescia menor, mais rápido, dissipando mais calor e consumindo mais energia [6]. Desde a introdução do Intel 8086 e através do Pentium 4 um aumento no desempenho, a partir de uma geração para a seguinte, foi visto como um aumento na freqüência do processador. Por exemplo, o Pentium 4 variou de velocidade (freqüência) de 1,3 a 3,8 GHz em 8 anos. Enquanto o tamanho físico do *chip* diminuiu, o número de transistores por *chip* aumentou junto com a velocidade, impulsionando assim o aumento de dissipação de calor [7].
Historicamente, os fabricantes de processador têm respondido à demanda por maior poder de processamento, principalmente através da disponibilização de processadores mais rápidos. No entanto, o desafio do gerenciamento de energia e refrigeração para os processadores poderosos de hoje levou a uma reavaliação desta abordagem. Com o calor aumentando progressivamente mais rápido que a velocidade em que os sinais deslocam-se pelo processador, conhecido como velocidade de *clock*, um aumento no desempenho pode criar um aumento ainda maior no calor [8].

Esta discussão tem inicio em 1965, quando Gordon Moore observou que o número de transistores disponíveis nos semicondutores dobraria num período de 18 a 24 meses. Agora conhecida como a lei de Moore, esta observação tem orientado *designers* de computador nos últimos 40 anos. A métrica mais comumente utilizada na medição do desempenho da computação é o CPU (freqüência de *clock*) e ao longo dos últimos 40 anos esta tendeu a seguir a lei de Moore. Segundo Akhter e Roberts [9], embora a melhoria linear do fluxo de instruções e da velocidade de *clock* são metas que se valem lutar, arquitetos de computador podem tirar proveito desta melhoria em maneiras menos óbvias. Por exemplo, em um esforço para tornar mais eficiente o uso dos recursos do processador, os arquitetos têm utilizado técnicas de paralelismo em nível de instrução. Paralelismo em nível de instrução (ILP) dá a capacidade de reorganizar as instruções em uma ótima maneira para eliminar os *stalls* do *pipeline*. Para que esta técnica seja eficaz, múltiplas e independentes instruções devem ser executadas. No caso da execução em ordem, dependências entre instruções podem limitar o número de instruções disponíveis para a execução, reduzindo a quantidade de execução em paralelo que pode ocorrer. Uma abordagem alternativa que tenta manter as unidades de processamento de execução ocupada é a reordenação das instruções para que instruções independentes possam executar simultaneamente. Neste caso, as instruções são executadas fora da ordem do programa original.

Com a evolução do *software*, aplicações tornaram-se cada vez mais capazes de executar várias tarefas em simultâneo. A fim de apoiar o paralelismo, várias abordagens, tanto em *software* e *hardware* tem sido adotadas. Uma abordagem para enfrentar cada vez mais a natureza concorrente dos *softwares* modernos envolve usar uma preemptiva, ou fatia de tempo, e um sistema operacional multitarefa. Processamento com fatia de tempo permite desenvolvedores esconder latências associadas com I/O trocando a execução de múltiplos processos. Este modelo não permite a execução paralela. Apenas um fluxo de instrução pode ser executado em um processador em um único ponto no tempo [9].

Ainda de acordo com Akhter e Roberts [9], outra abordagem para resolver o paralelismo de processos é aumentar o número de processadores físicos no computador. Sistemas multiprocessadores permitem verdadeira execução em paralelo; múltiplos processos são executados simultaneamente em vários processadores. A perda de vantagem neste caso é o aumento do custo global do sistema. O próximo passo lógico para o processamento de processos simultâneos é o processador com múltiplos núcleos. Em vez de apenas reutilizar recursos dos processadores em um processador com um único núcleo, fabricantes de processadores aproveitaram melhorias na fabricação da tecnologia para implementar dois ou mais “núcleos de execução” independentes dentro de um único processador. Estes núcleos são essencialmente dois processadores em um único *chip*. Estes núcleos têm o seu próprio conjunto de execução e seus recursos arquitetônicos (veja a seção 3).

Um desempenho global de processamento excelente pode ser conseguido através da redução da velocidade de *clock*, enquanto se aumenta o número de núcleos de processamento e, conseqüentemente, a redução da velocidade pode conduzir a uma baixa produção de calor e maior eficiência do sistema. Por exemplo, ao passar de um único núcleo de alta velocidade, que gera um aumento correspondente no calor, para múltiplos núcleos mais lentos, que produzem uma redução correspondente no calor, pode-se melhorar o desempenho das aplicações, reduzindo simultaneamente o calor. Uma combinação que prevê dois ou mais núcleos com mais baixa velocidade poderia ser concebido para proporcionar excelente desempenho, reduzindo ao mínimo o consumo de energia e proporcionando menor produção de calor do que configurações que dependem de um único *clock* de alta velocidade [8]. Aumentar a velocidade de *clock* causa mudanças mais rápidas entre transistores e, assim, geram mais calor e consomem mais energia.

Segundo Geer [10], quando um *chip* com um único núcleo executa vários programas, ele atribui uma fatia de tempo para trabalhar em um programa e, em seguida, atribui diferentes fatias de tempo para outros. Isto pode causar conflitos, erros ou baixo desempenho quando o processador tem de efetuar múltiplas tarefas em simultâneo. Se você tiver várias tarefas que deverão executar todas ao mesmo tempo, poderemos notar que haverá uma melhora significativa em desempenho com múltiplos núcleos. Por exemplo, os *chips* poderiam usar um núcleo distinto para cada tarefa. Devido aos núcleos estarem nos mesmos *chips*, eles podem compartilhar componentes arquitetônicos, tais como elementos de memória e gerenciamento de memória. Assim, eles têm menos componentes e sistemas de custos mais baixos do que executar em múltiplos processadores distintos em *chips* distintos. Além disso, a sinalização entre os núcleos pode ser mais rápida e consumir menos eletricidade do que em uma abordagem em separado.

Acelerar a freqüência do processador executou o seu curso na primeira parte desta década; arquitetos de computador precisavam de uma nova abordagem para melhorar o desempenho. Adicionando um núcleo de processamento adicional no mesmo *chip* seria, em teoria, conduzir a duas vezes o desempenho e dissiparia menos calor, embora, na prática, a velocidade real de cada núcleo é mais lento do que o mais rápido processador de núcleo único. Em Setembro de 2005, constatou-se que o consumo de energia aumenta em 60% com cada aumento de 400MHz de velocidade, mas a abordagem de dois núcleos significa que você pode receber uma melhora significativa no desempenho, sem a necessidade de execução em baixas taxas de *clock* [11].

Se os núcleos utilizarem o mesmo número de transistores, mais núcleos poderão ser adicionados em um *chip* conforme o número de transistores disponíveis aumenta. De fato, usando um corolário da Lei de Moore, podemos dizer que o número de núcleos em um *chip* irá duplicar a cada 18 ou 24 meses. Dado que o *dual* e *quad* núcleos de liderança comercial já estão em uso generalizado hoje, e protótipos de investigação de 16 núcleos nas universidades provavelmente estarão disponíveis, é altamente provável que veremos processadores com 1000 núcleos na primeira parte da próxima década [3].

Mais importante do que isso é que processadores com múltiplos núcleos oferecem uma tecnologia imediata e eficaz para resolver os desafios de projeto de processadores de hoje, aliviando os subprodutos de calor e consumo de energia que existem quando continuamente se avança com processadores de maior freqüência e com apenas um núcleo. Ela ajudará a romper com as limitações de desempenho dos processadores de apenas um núcleo e fornecer capacidade de desempenho para enfrentar os *softwares* mais avançados de amanhã [5]. Isto ocorrerá, pois, estes processadores têm o potencial de executar aplicações de forma mais eficiente do que processadores de um núcleo único, fornecendo aos usuários a capacidade de continuar trabalhando mesmo durante a execução das tarefas que mais demandam processamento em segundo plano, como pesquisar uma base de dados, renderizar uma imagem 3D, mineração de dados, análise matemática e servidores *Web*.

A *Advanced Micro Devices* (AMD) [5] diz que a crescente complexidade dos *softwares*, bem como o desejo dos usuários para executar várias aplicações ao mesmo tempo, irá acelerar a adoção generalizada de sistemas baseados em processadores com múltiplos núcleos. Isso dará a aplicações comerciais a capacidade de lidar com grandes quantidades de dados e com mais usuários de forma mais rápida e eficiente. Os consumidores irão experiênciar mais funcionalidades e funcionalidades mais ricas, especialmente para aplicações como mídia digital e criação de conteúdos digitais.

Por fim, Merritt [12] argumenta que a tecnologia de múltiplos núcleos não é um novo conceito, uma vez que a idéia tem sido utilizada em sistemas embarcados para aplicações especiais, mas, recentemente, a tecnologia tornou-se integrar com as empresas Intel e AMD que tem disponibilizado comercialmente muitos *chips* com múltiplos núcleos. Em contraste com as máquinas de dois e quatro núcleos disponíveis comercialmente a partir de 2008, alguns especialistas acreditam que até 2017 processadores embutidos poderiam dispor de 4.096 núcleos, servidores poderão ter 512 núcleos e *chips* de *desktop* poderiam utilizar 128 núcleos. Esta taxa de crescimento é surpreendente considerando que os *chips* dos *desktops* atualmente estão no linear de quatro núcleos e um único núcleo tem sido utilizado ao longo dos últimos 30 anos [6]. Na próxima seção será descrito melhor sobre a arquitetura destes processadores.

3. ARQUITETURA MULTICORE: PRINCIPAIS CONSIDERAÇÕES TECNOLÓGICAS

Embora os fabricantes de processadores possam diferir um do outro em relação às arquiteturas dos microprocessadores, veja com detalhes em [6] diversos aspectos relacionados às diferenças entre as implementações específicas de arquiteturas de múltiplos núcleos de processadores da Intel, AMD e outros fabricantes. No entanto, as arquiteturas de múltiplos núcleos precisam aderir a determinados aspectos básicos. A configuração de um microprocessador pode ser visto na Figura 1.

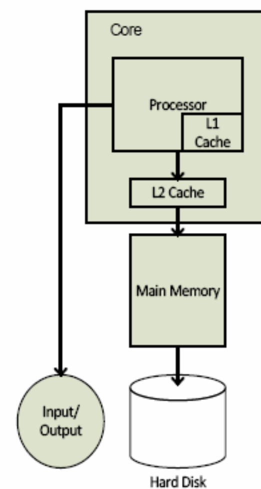


Figura 1. Configuração de um Processador Moderno Genérico [6]

As arquiteturas de computadores podem variar sobre diversos aspectos (Veja Figura 2). Em específico nas arquiteturas de múltiplos núcleos, além do modelo de memória compartilhada e modelo de memória distribuída que será discutido a seguir (Veja Figura 3), eles podem variar quanto ao modelo de cachê também. Assim, dependendo da arquitetura, estes processadores podem ou não compartilhar uma *cache*. As diferentes arquiteturas de processadores estão destacadas na Figura 2.

Em relação à Figura 1, mais próximo do processador está a *cache* de nível 1 (L1), que é uma memória muito rápida que é utilizada para guardar dados freqüentemente utilizados pelo processador. A *cache* de Nível 2 (L2) esta fora do *chip*, é mais lenta do que a *cache* L1, mas ainda muito mais rápida que a memória principal. A *cache* L2 é maior que a *cache* L1 e é utilizada para a mesma finalidade. A Memória principal é muito grande e mais lenta do que a *cache*, e é utilizado, por exemplo, para armazenar um arquivo atualmente sendo editado em um editor de texto. A maioria dos sistemas tem entre 1GB para 4GB de memória principal, em comparação com cerca de 32KB de L1 e 2MB de *cache* L2. Finalmente, quando os dados não estiverem localizados na *cachê* ou na memória principal, o sistema deve recuperá-lo a partir do disco rígido, o que leva exponencialmente mais tempo do que a partir da leitura da memória principal.

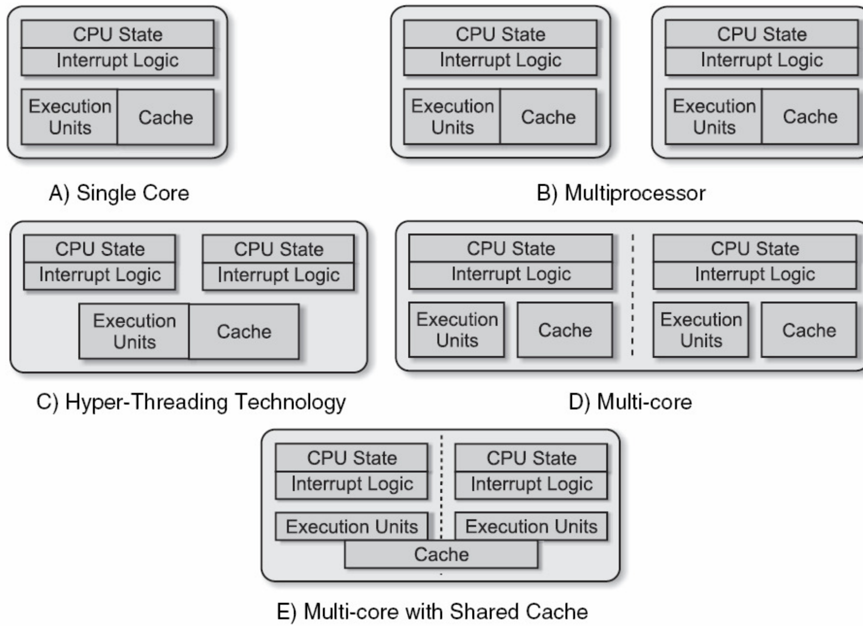


Figure 2: Comparação entre arquiteturas de um único núcleo, com múltiplos processadores e múltiplos núcleos [9]

Se estabelecermos dois núcleos lado a lado, pode-se ver que um método de comunicação entre os núcleos, bem como para a memória principal será necessário. Isso é normalmente realizado usando um único barramento de comunicação ou uma rede de interconexão. O método por barramento é usado com um modelo de memória compartilhada, enquanto que a abordagem de interconexão de rede é utilizada com um modelo de memória distribuída. Após cerca de 32, núcleos o barramento estará sobrecarregado com a quantidade de processamento, comunicação e concorrência, o que leva a diminuição do desempenho; portanto, uma comunicação pelo barramento tem uma capacidade limitada de expansão. Observe a Figura 3, que ilustra os dois tipos de comunicação possíveis [6].

4. O DESEMPENHO DOS MULTICORE

Segundo Geer [10], de acordo com o professor assistente Steven Keckler da Universidade do Texas, um *chip* com dois núcleos executando várias aplicações é cerca de 1,5 vezes mais rápido comparável com um *chip* com apenas um núcleo. Os *chips* de múltiplos núcleos não necessariamente rodam tão rápido como o melhor desempenho dos modelos com apenas um núcleo, mas melhoraram o desempenho global através do tratamento de mais trabalhos em paralelo, conforme ilustra a Figura 4. *Chips* de múltiplos núcleos são a maior mudança no modelo de programação desde que a Intel introduziu a arquitetura 386 de 32-bit. Além disso, estes processadores são uma maneira de estender a lei de Moore.

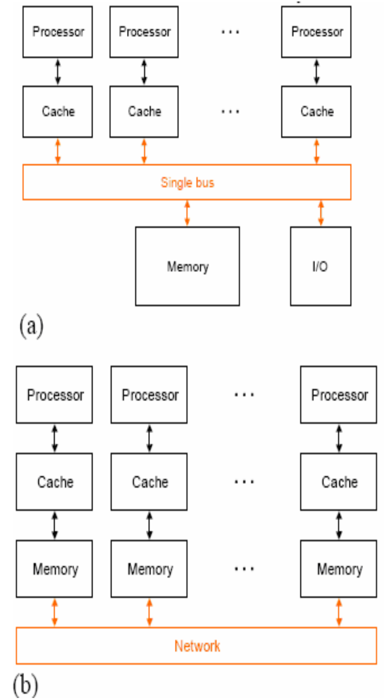


Figura 3: Modelo de memória compartilhada e modelo de memória distribuída [6]

A melhor maneira de apreciar os benefícios obtidos por uma implementação com múltiplos núcleos sobre a abordagem de apenas um único núcleo é comparar a forma como os transistores seriam utilizados. Por exemplo, suponha que há um único núcleo de tecnologia de 90-nm que ocupa uma área A do *chip* e que as porções do processador e a memória *cache* ocupam cada uma a metade da superfície do *chip*. Vamos chamar isto de caso base. Com a tecnologia de 65-nm, os arquitetos têm o dobro do número de transistores para a mesma área A. Para tirar proveito do dobro de transistores, arquitetos podem manter a mesma arquitetura do CPU e triplicar o tamanho da *cache* (Caso 1). Alternativamente, os arquitetos podem dobrar o número de núcleos e manter o mesmo tamanho da *cache* para cada núcleo (Caso 2).

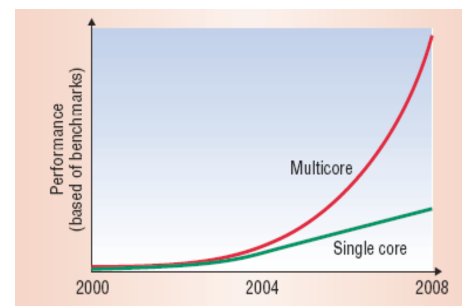


Figura 4. *Chips* com múltiplos núcleos têm um melhor desempenho [10]

Podemos aplicar um modelo simplificado para demonstrar as implicações de desempenho das duas alternativas. Suponhamos que a taxa de perda de trabalho para o cenário base é de 1% e a

latência da memória principal para uma *cache miss* é de 100 ciclos. Também assumimos que cada uma das instruções idealmente executa em um ciclo. Começando com o cenário base, as instruções por *clock* (IPC) será 0,5 (IPC pode ser calculado como 1 / (1 + 0,01 × 100) = 0.5). Supondo que a carga de trabalho tem grande paralelismo, o IPC dobra, aplicando o caso 2 (2 × 1 / (1 + 0,01 × 100) = 1), uma vez que a taxa de execução de instruções é o dobro do cenário base. Por outro lado, assumindo que a latência de memória não mudou, o IPC do Caso 1 será 0,62, ou 1 / (1 + 0,006 × 100), que mostra um desempenho mais baixo do que a alternativa de múltiplos núcleos (Caso 2). Este cálculo é obtido a partir de uma regra comumente utilizada para taxas de *cache miss* que sugere que a taxa de *miss* segue a regra de raiz quadrada. Assim, a taxa de *cache miss* no Caso 1 será de 1%/*sqrt*(3) = 0,6%. O Caso 2 também contribui para demonstrar que *caches* oferecerem retornos decrescentes.

Se o argumento de múltiplos núcleos é válido, devemos ser capazes de reduzir o tamanho da *cache* e colocar mais núcleos no mesmo *chip* para um melhor desempenho. Vamos tentar três núcleos, cada um com um terço do tamanho da *cache* do cenário de base. A taxa de *cache miss* será de 1%×*sqrt* (3) = 1,7%, e o IPC aumenta ainda mais, para 1,1, ou 3 × 1 / (1 + 0,017 × 100). No entanto, utilizando o mesmo tamanho de *chip*, a tendência é continuar? Com quatro núcleos, há pouco espaço para *cache*, mas presume-se que se pode espremer em uma *cache* do que um décimo sexto do tamanho do cenário de base para cada núcleo. A taxa de *miss* será de 1%×*sqrt* (16) = 4%, e o IPC cai para 0,8, ou 4 × 1 / (1 + 0,04 × 100). Observe que, neste exemplo, o Caso 3 tem o número ideal de cores e tamanho de cache. Isto demonstra que o balanço de recursos em um núcleo é de grande importância em arquiteturas de múltiplos núcleos [13].

Uma profunda e bem desenvolvida análise de medida de desempenho dos processadores de múltiplos núcleos foi desenvolvida por Muthana *et al.* [14], na qual diversas considerações são feitas em diversos aspectos. Foi observado que o desempenho de um processador com múltiplos núcleos é muito superior à de um processador com apenas um único núcleo, e uma taxa de banda de 1 *terabyte/s* é possível com uma considerável economia de energia. Capacitores embutidos com uma densidade de capacitância de *1uF/cm2* podem ser fabricados e a combinação desses capacitores em paralelo podem proporcionar dissociação para futuros processadores. Já sobre a dissipação de calor, comparado com vários *chips* com apenas um único núcleo, os *chips* com múltiplos núcleos são mais fáceis de resfriar, pois estes processadores são mais simples e utilizam menos transistores. Isto significa que eles usam menos energia e dissipam mais calor global [2].

Contudo, devido à limitada largura de banda e esquema de gerenciamento de memória que são mal adaptados a supercomputadores, o desempenho dessas máquinas estaria um nível abaixo com mais núcleos podendo até diminuir. Embora o número de núcleos por processador esteja aumentando, o número de ligações a partir do *chip* para o resto do computador não está. Então, devido a isso, é um problema manter todos os núcleos alimentados com dados a todo tempo. A chave para resolver esse gargalo pode ser uma melhor integração entre memória e processador [15]. Observe a Figura 6 que ilustra este cenário.

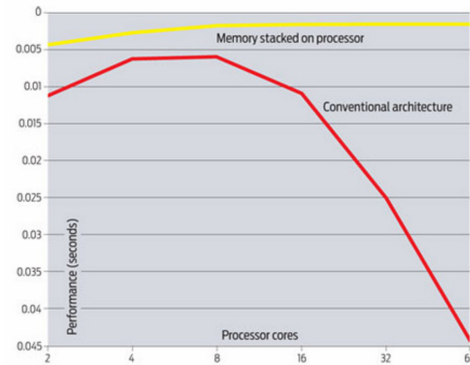


Figura 6. Mais núcleos por *chip* deixará alguns programas mais lentos ao menos que exista um grande aumento na taxa de memória [15]

Segundo Agarwal *et al.* [3], nos dias de *design* de apenas um único núcleo, arquitetos aumentariam o tamanho dos recursos do núcleo (por exemplo, tamanho da *cache*), conforme ocorreu com os transistores, que aumentaram em cada sucessiva geração da tecnologia. Arquiteturas com múltiplos núcleos proporcionaram uma maior escolha de inclusão de outros núcleos. Desta forma, tendo mais espaço, aumentando o número de núcleos e mantendo o tamanho dos recursos relativamente pequenos, poderar-se resultar em mais desempenho do que aumentar o tamanho dos recursos e manter o número de núcleos constante. Esta nova dimensão para melhorar o desempenho e eficiência em energia em múltiplos núcleos nos obriga a repensar na arquitetura do processador e isto é capturado por um princípio simples chamada de "*Kill Rule*". Este princípio afirma que o tamanho de um recurso deve ser aumentado somente se, para cada 1% de aumento na área central, exista pelo menos um aumento de 1% no desempenho do núcleo.

Já sobre como medir o desempenho destes processadores, em Gal-On e Levy [15] é feito uma discussão sobre como desenvolver a medição do desempenho dos processadores com múltiplos núcleos a partir de diversos tipos de *benchmarks* e de distintos critérios, tais como banda de memória e escalabilidade. Os autores explicitam que estes benchmarks poderiam ser baseados em arquiteturas de memória distribuída ou compartilhada. Ainda, eles poderiam consistir em núcleos homogêneos ou heterogêneos e poderiam empregar uma variedade de tecnologias interligadas. Podendo a tecnologias de múltiplos núcleos terem abordagens muito diferenciadas, então, *benchmarks* sobre elas precisam ser altamente diferenciados também.

Por fim, mesmo com desempenhos apurados e sendo esta uma arquitetura sofisticada, ainda há diversos desafios e questões em aberto nas arquiteturas de múltiplos núcleos, tópico que será mais bem discutido na próxima seção.

5. DESAFIOS E QUESTÕES EM ABERTO

Embora a tecnologia de múltiplos núcleos ofereça oportunidades para melhorias no desempenho de processamento e de eficiência em energia, ela também traz muitos novos desafios de programação e de *design*, devido a indústria de processadores avançar rumo a eficiência em energia, desempenho e programabilidade. Curiosamente, a eficiência e o desempenho não são apenas as maiores oportunidades oferecidas pela tecnologia de múltiplos núcleos, mas também, o mais importante desafio que se

tem refere-se ao número de núcleos que vão além dos projetos de único dígito de hoje. Outro desafio envolve o modelo de programação necessário para a utilização eficiente de múltiplos núcleos em um *chip* [13] (veja a subseção 5.5). Além disso, ter múltiplos núcleos em um único *chip* dá origem a alguns problemas como: gestão de energia e temperatura, que são duas preocupações que podem aumentar exponencialmente com a adição de múltiplos núcleos; coerência em memória cache, além do uso de processador com múltiplos núcleos no seu pleno potencial que é uma outra questão [6]. Observe a seguir alguns desafios e problemas relacionados a esta tecnologia.

5.1 Energia e Temperatura

O calor sofre a influência de um conjunto de fatores, dos quais dois são: densidade do processador e velocidade de *clock*. Outros controladores incluem tamanho da *cache* e tamanho do núcleo em si. Em arquiteturas tradicionais, o calor gerado a cada nova geração de processadores tem crescido a uma taxa maior do que a velocidade. Em contraste, usando uma cache compartilhada (em vez de separar *caches* dedicadas para cada núcleo do processador) e processadores de baixa velocidade, processadores de múltiplos núcleos podem ajudar os administradores a minimizar o calor, mantendo um elevado desempenho global [8].

Segundo Schauer [6], se dois núcleos forem colocados em um único *chip* sem qualquer modificação o *chip,* em teoria, consumiria duas vezes mais energia e geraria uma grande quantidade de calor. No caso extremo, se um processador sobre-aquecesse o computador poderia até queimar. Levando isso em conta, para cada projeto, os múltiplos núcleos são executados a uma menor freqüência visando reduzir o consumo de energia. Para combater o consumo de energia desnecessário, muitos projetos também incorporaram uma unidade de controle que tem o poder de parar núcleos não utilizados ou limitar a quantidade de energia. Ao alimentar núcleos não utilizados e usar "*clock gating"* a quantidade de perda no *chip* é reduzido. Para diminuir o calor gerado por múltiplos núcleos em um único *chip*, este é arquitetado para que o número de pontos quentes não cresça e não se torne muito grande, de modo que o calor seja espalhado por todo o *chip*.

5.2 Coerência de *Cache*

Coerência de *cache* é uma preocupação em um ambiente de múltiplos núcleos devido as *caches* L1 e L2 distribuídas. Uma vez que cada núcleo tem sua própria *cache*, a cópia dos dados naquela *cache* pode não ser sempre a versão mais atualizada. Por exemplo, imagine um processador com duplo núcleo no qual cada núcleo trouxe um bloco de memória em sua *cache* privada. Um núcleo escreve um valor para um local específico e, quando o segundo núcleo tenta ler o valor da sua *cache,* não vai ter a cópia atualizada, a menos que a sua entrada na *cache* seja invalidada e um *cache miss* ocorra. Este *cache miss* força a entrada da *cache* do segundo núcleo ser atualizada. Se esta política de coerência não estava correta, dados inválidos poderiam ser lidos e resultados inválidos seriam produzidos, podendo deixar errado o resultado final do programa errado. Em geral, existem dois regimes de coerência de *cache:* o protocolo *snooping* e um protocolo baseado em diretório. O protocolo *snooping* só funciona com um sistema baseado em barramento, usa um número de estados para determinar se precisa atualizar as entradas da *cache* e tem controle sobre a escrita ao bloco. O protocolo baseado em diretório pode ser usado em uma rede arbitrária e é, portanto, escalável para vários processadores, ou núcleos, em contraste com o *snooping* que não é escalável. Neste esquema é utilizado um diretório que detém informações sobre a localização de memória que estão sendo compartilhadas em múltiplas *caches* e que são utilizadas exclusivamente por um núcleo da *cache*. O diretório sabe quando um bloco precisa ser atualizado ou cancelado [6].

5.3 Sistema de Memória

Muitos aplicativos não são capazes de tirar pleno proveito da velocidade das CPU atuais, pois os seus desempenhos são ditados pela latência de memória (por exemplo, os programas que passam a maior parte do de seu tempo em listas ligadas). Se muitas cópias em paralelo são executadas destes programas,,a latência pode ser superada — os núcleos coletivamente pode proporcionar cargas pendentes suficientes para "esconder" a latência do sistema de memória subjacentes. De fato, sistemas de múltiplos núcleos foram concebidos com esse trabalho em mente [1].
De acordo com Agarwal e Levy [13], existem potenciais problemas de desempenho devido ao gargalo da banda de memória relacionada com as instruções e dados. Processadores com múltiplos núcleos podem resolver um aspecto deste problema por meio da distribuição de memória *cachê* dentro dos núcleos dos processadores. Do mesmo modo, a taxa da memória principal pode ser aumentada por meio da aplicação de vários portos de memória principal e um grande número de bancos de memória. Assim, nos múltiplos núcleos, o chamado gargalo da taxa de memória não é realmente um problema de memória. Pelo contrário, é uma questão de interconexão e o problema reside na forma como as unidades de memória fazem interface com os núcleos. A interface entre os bancos de memória e os núcleos é a interconexão, o que inclui tanto a rede *on-chip* quanto os pinos.

Soluções anteriores para a questão da interconexão escalável de redes aplicam-se à questão de memória, utilizando-se pacotes simples de transporte de memória ao contrário de dados processador-a-processador. Assim redes *on-chip* que são escaláveis aliviam o problema da taxa de memória. Os pinos que permitem o acesso a um processador com múltiplos núcleos acessarem a *offchip* DRAM também fazem parte da rede de interconexão. Em conclusão, interfaces de memória serial de alta velocidade darão um impulso significativo a curto prazo na taxa de pinos disponíveis para a memória e ao longo prazo, porém, taxas *off-chip* irão continuar a ser significativamente mais caras do que a largura de banda *on-chip*. Portanto nossos modelos de programação necessitam se adaptar para substituir comunicação baseada em memória *off-chip* com comunicação direta processador a processador *on-chip* [13].

5.4 Sistema de Barramento e Redes de Conexão

Que tipo de interconexão é o mais adequado para processadores com múltiplos núcleos? Uma abordagem baseada em barramento é melhor do que uma interconexão de rede? Ou existe um híbrido como a malha de rede que funciona melhor? A questão permanece. Embora o desempenho e a eficiência em energia sejam as principais vantagens dos processadores com múltiplos núcleos versus a abordagem de um único processador, eles também são desafiadores para melhorar à medida que o número de núcleos aumenta para além dos dígitos simples e se move para dezenas ou mesmo centenas de núcleos. O principal desafio de

lidar com o desempenho de um maior número de núcleos refere-se à rede que liga os vários núcleos em si e a memória principal. Sistemas atuais com múltiplos núcleos dependem de barramento ou de anéis para a sua interligação. Estas soluções não são escaláveis e, portanto, se tornarão um gargalo para o desempenho. Uma topologia comum interliga um barramento igualmente compartilhado entre todos os núcleos que estão ligados a ela. Arbitragem para a utilização do barramento é uma função centralizada e apenas um núcleo é permitido usá-lo em um determinado ciclo. Alternativamente, um anel é uma ligação unidimensional entre núcleos locais e a arbitragem é uma função de cada um dos interruptores. A malha, uma extensão bidimensional de comutação por pacotes, é ainda uma outra topologia de interconexão. Esta solução funciona bem com a tecnologia 2D VLSI (escala muito grande de integração) e também é a interconexão mais eficaz quando escalar a um grande número de núcleos. A malha pode ser escalar a 64 núcleos ou mais, pois a sua taxa de bisseção aumenta à medida que são adicionados mais núcleos, e sua latência aumenta apenas como a raiz quadrada do número de núcleos. (a taxa de bisseção é definida como a largura de banda entre as duas metades iguais dos múltiplos núcleos).

Contrastando isto com a topologia de barramento que apenas pode tratar cerca de oito núcleos, e o anel é viável até cerca de 16 núcleos. Assim para ambos (barramento e anel), a taxa de bisseção é fixa mesmo quando são adicionados mais núcleos, bem como a latência aumenta em proporção ao número de núcleos [13]. Portanto, memória extra será inútil se a quantidade de tempo necessário para os pedidos à memória não melhorar também. Reprojetar a interligação entre os núcleos de rede é um grande foco dos fabricantes de *chips*. Uma rede mais rápida significa uma menor latência na comunicação inter-núcleos e em transações de memória. [6]

5.5 Programação Paralela

De acordo com Geer [4], para *chips* com um único núcleo, programadores escrevem aplicações e estes *chips* priorizam tarefas na ordem em que elas devem ser realizadas para fazer a atividade designada mais eficiente. O sistema operacional, em seguida, atribui as tarefas a executar seriadamente no processador. Desenvolvedores que escrevem aplicações para *chips* com múltiplos núcleos devem dividir em tarefas que o sistema operacional pode atribuir; que decorrem paralelamente em vários núcleos. Cada núcleo tem a sua própria capacidade de multi-processamento e, portanto, pode dividir em partes as suas próprias tarefas que podem funcionar em paralelo. Uma questão-chave para os programadores é como uma atividade pode ser dividida em sub-tarefas.

Seguindo a lei de Moore, o número de núcleos em um processador poderia ser definido para duplicar a cada 18 meses. Isto só faz sentido se o *software* em execução nestes núcleos levar isso em conta. Em última análise, os programadores precisam aprender a escrever programas em paralelos que podem ser divididos e executados simultaneamente em múltiplos núcleos, em vez de tentar explorar o *hardware* de *chips* com um único núcleo para aumentar paralelismo de programas seqüenciais. O desenvolvimento de *software* para processadores com múltiplos núcleos traz algumas preocupações, tais como: Como é que um programador garante que uma tarefa de alta prioridade receba prioridade no processador, e não apenas em um núcleo? Em teoria, mesmo se um processo tivesse a mais alta prioridade dentro do núcleo em que está executando, poderia não ter uma alta prioridade no sistema como um todo. No entanto, como garantir que todo o sistema pára e não apenas o núcleo em que uma aplicação está a executar? Estas questões devem ser abordadas juntamente com o ensino de boas práticas de programação de paralelismo para desenvolvedores [6].

Embora o objetivo da programação para múltiplos núcleos seja clara, fazê-la não é necessariamente fácil. Uma das principais preocupações para os programadores é que algumas atividades, tais como os relacionados com gráficos, não se divide facilmente em sub-funções, como ocorre naturalmente em pontos de início e término. Nestes casos, o programador deve executar complexas transformações que produzem pontos. Assim uma aplicação pode ser dividida em tarefas menores, dividindo o *software* em partes distintas que são menores e mais granulares do que em sub-funções difícil de dividir. Outra opção seria mudar todas as estruturas de dados [4].

Para Schauer [6], o último e mais importante problema será usar multi-processos, ou outras técnicas de processamento paralelo, para obter o máximo desempenho de processadores com múltiplos núcleos. Reconstruir aplicações para serem multi-processos implica em um completo retrabalho pelos programadores na maioria dos casos. Os programadores têm que escrever aplicações com sub-rotinas capazes de serem executadas em diferentes núcleos, o que significa que as dependências de dados terão de ser resolvidas ou contabilizadas (por exemplo, latência na comunicação ou utilizando uma *cache* compartilhada). As aplicações devem ser balanceadas. Se um núcleo está sendo usado muito mais do que outro, o programador não tirará plena vantagem do sistema com múltiplos núcleos.

5.6 Núcleos Homogêneos e Heterogêneos

Arquitetos têm debatido se os núcleos em um processador com múltiplos núcleos devem ser homogêneos ou heterogêneos, e ainda não há uma resposta definitiva. Núcleos homogêneos são todos exatamente o mesmo: freqüências equivalentes, tamanhos de *cachê*, funções, entre outras características. No entanto, cada núcleo de um sistema heterogêneo pode ter uma diferente função, freqüência, modelo de memória, etc. Há um aparente *trade-off* entre a complexidade do processador e sua customização. Núcleos homogêneos são mais fáceis de produzir, uma vez que o mesmo conjunto de instrução é usado em todos os núcleos e cada núcleo contém o mesmo *hardware*. Mas eles são os mais eficientes para o uso nas tecnologias de múltiplos núcleos? Cada núcleo, em um ambiente heterogêneo, poderia ter uma função específica e executar o seu próprio conjunto de instrução especializado. Um modelo heterogêneo poderia ter um grande núcleo centralizado construído para tratamento genérico e rodar um sistema operacional, um núcleo para gráficos, um núcleo de comunicações, um núcleo para reforço matemático, um de áudio, um criptográfico, etc. [16].

Alderman [17] discute algumas vantagens em diversos aspectos de desempenho em uma arquitetura heterogênea. Multiprocessadores heterogêneos (ou assimétricos) apresentam oportunidades únicas para melhorar a taxa de processamento do sistema e de reduzir a energia de processamento. Heterogeneidade *on-chip* permite ao processador uma melhor execução dos recursos correspondentes a cada uma das necessidades da aplicação e endereça um espectro muito mais amplo de cargas - de baixo para alto paralelismo de processos com alta eficiência.

Pesquisas recentes em microprocessadores heterogêneos identificaram vantagens significativas sobre os processadores com uma abordagem de múltiplos núcleos homogêneo em termos de energia e taxa de processamento, além de enfrentar os efeitos da lei de Amdahl sobre o desempenho de aplicações paralelas. Em um *chip*, no entanto, nenhum núcleo único precisa ser ideal para o universo das aplicações. A utilização da heterogeneidade explora este princípio. Inversamente, uma concepção homogênea na verdade agrava ainda mais o problema, criando um único projeto universal e, em seguida, replicando esta solução em todo o *chip*. O modelo heterogêneo é mais complexo, mas pode ter benefícios em eficiência, energia e calor que superam a sua complexidade. Com os principais fabricantes de ambos os lados desta questão entre a abordagem homogênea e heterogênea, este debate de prós e contras se estenderá ao longo dos próximos anos [6].

6. CONCLUSÃO

A tecnologia de múltiplos núcleos encapsulados em um único processador já é realidade na computação nos dias atuais. Objetiva-se com estes melhorar o desempenho das aplicações a partir de melhores potências e eficiência térmica, ou seja, ter o potencial de prover melhor desempenho e escalabilidade sem um correspondente aumento em consumo de energia e calor, conforme seria o caso do aumento de freqüência de *clock* nos processadores com apenas um único núcleo [8].

Os processadores com múltiplos núcleos são bem sofisticados e estão definitivamente arquitetados a aderir a um moderado consumo de energia, além de um melhor controle da dissipação de calor e melhores protocolos de coerência de *cache*. Apesar das claras evidências de seu uso hoje, já no mercado, e das potencialidades desta tecnologia, diversos desafios ainda necessitam ser desbravados tais como: sistema de barramento e interconexão com a memória, programação paralela e arquiteturas homogêneas verso as heterogêneas; conforme apresentado neste trabalho. Além disso, diversas questões arquiteturais sobre estes processadores permanecem em aberto e ainda não foram resolvidas de maneira plausível pela literatura.

Observa-se que a tecnologia de múltiplos núcleos em um único processador poderá revolucionar o modo de se desenvolver aplicações, no entanto o desempenho dependerá muito do modo como as aplicações aproveitarão os recursos e vantagens (por exemplo, paralelismo) em sua total capacidade. Há relativamente poucas aplicações (e mais importante, poucos programadores com este conhecimento) para escrever níveis de paralelismo. Finalmente, segundo Schauer [6], processadores com múltiplos núcleos são uma importante inovação na evolução dos microprocessadores e com programadores qualificados e capazes de escrever aplicações paralisáveis, a eficiência poderá aumentar drasticamente. Nos próximos anos, poderá haver muitas melhorias para os sistemas computacionais, e essas melhorias vão proporcionar programas mais rápidos e uma melhor experiência computacional para os usuários.

7. REFERENCIAS

[1] Kaufmann, R. & Gayliard, B. 2009. **Multicore Processors**. Dr. Dobb's Journal. June 2009, http://www.ddj.com/architect/212900103

[2] Gorder, P. F. 2007. **Multicore Processors for Science and Engineering**. Computing in Science & Engineering. vol. 9, issue: 2. pp. 3-7.

[3] Agarwal, Anant; Levy, Markus. 2007. **The KILL Rule for Multicore**. Design Automation Conference. DAC '07. 44th ACM/IEEE, vol., no., pp.750-753, 4-8 June 2007.

[4] Geer, David. 2007 "**For Programmers, Multicore Chips Mean Multiple Challenges,**" Computer, vol. 40, no. 9, pp. 17-19, Aug. 2007, doi:10.1109/MC.2007.311

[5] AMD. 2005. **Multi-Core Processors: the next evolution in computing**. AMD, June 2009, http://multicore.amd.com/Resources/33211A_Multi-Core_WP_en.pdf

[6] Schauer, Bryan. 2008. **Multicore Processors: a necessity**. June 2009, http://www.csa1.co.uk/discoveryguides/multicore/review.pdf?SID=sfjgtitfdsj6h998ji91ns5nd4

[7] Knight, W. "**Two Heads Are Better Than One**", IEEE Review, September 2005

[8] Fruehe, John. 2005. **Multicore Processor Technology.** June 2009, http://www.dell.com/downloads/global/power/ps2q05-20050103-Fruehe.pdf

[9] Akhter, Shameem & Roberts, Jason. 2006. **Multi-Core Programming: Increasing Performance through Software Multi-threading**. Intel Press; 1st edition. 360 pages.

[10] Geer, David. 2005. **Chip Makers Turn to Multicore Processors.** Computer volume 38, issue 5, May 2005 pp. 11–13.

[11] Knight, W. 2005. "**Two Heads Are Better Than One**", IEEE Review

[12] Merritt, R. 2008. "**CPU Designers Debate Multi-core Future**", EETimes Online, June 2009, http://www.eetimes.com/showArticle.jhtml?articleID=206105179

[13] Agarwal, Anant & Levy, Markus. 2007. **Going multicore presents challenges and opportunities.** Embedded Systems Design. June 2009, http://www.design-reuse.com/articles/15618/going-multicore-presents-challenges-and-opportunities.html

[14] Muthana, P.; et al. 2005. **Packaging of Multi-Core Microprocessors: Tradeoffs and Potential Solutions**. Electronic Components and Technology Conference

[15] Gal-On, Shay & Levy, Markus. 2008. "**Measuring Multicore Performance,**" Computer, vol. 41, no. 11, pp. 99-102, Nov. 2008, doi:10.1109/MC.2008.464

[16] Alderman, R. 2007. "**Multicore Disparities**", VME Now, June 2009, http://vmenow.com/c/index.php?option=com_content&task=view&id=105&Itemid=46

[17] Kumar, R.; Tullsen, D. M.; Jouppi, N. P.; Ranganathan, P. 2005. **Heterogeneous chip multiprocessors**. Computer. volume 38, issue 11, Nov. 2005, pp. 32 – 38.

Introdução à Arquitetura de GPUs

Marcos Vinícius Mussel Cirne
RA: 045116
Instituto de Computação - IC / UNICAMP
marcosvcirne@gmail.com

RESUMO
Este relatório fará uma abordagem sobre o funcionamento das GPUs, explorando-se os conceitos básicos, suas aplicações e alguns detalhes sobre as suas arquiteturas. Além disso, será mostrado todo o histórico da evolução destas unidades, desde os anos 70 até os dias atuais, bem como os principais fatores que fizeram com que as GPUs evoluíssem rapidamente ao longo dos anos, obtendo um crescimento exponencial na sua performance geral.

1. INTRODUÇÃO
As GPUs (do inglês *Graphics Processing Unit*) são processadores especializados que basicamente efetuam operações ligadas a aplicativos gráficos 3D. São usadas com frequência em sistemas embarcados, jogos, consoles de videogames, estações de trabalho, etc. Elas também são bastante eficientes na manipulação de tarefas de Computação Gráfica em geral, conseguindo obter um desempenho superior ao de CPUs de propósito geral, devido à sua estrutura altamente paralelizada. Em computadores pessoais, elas podem estar presentes nas placas de vídeo ou então integradas à placa-mãe (tal como ocorre nos notebooks).

Basicamente, as GPUs são dedicadas para o cálculo de operações de ponto flutuante, destinadas a funções gráficas em geral. Os algoritmos de renderização normalmente trabalham com uma quantidade significativa dessas operações, as quais são executadas de uma maneira eficiente, graças ao uso de microchips próprios que contém um conjunto de operações matemáticas especiais. O bom desempenho desses microchips garante a eficiência das GPUs de uma forma geral.

Outras operações implementadas nas GPUs são aquelas que lidam com operações primitivas, como traçado de retas e círculos, desenhos de triângulos, retângulos, etc. GPUs mais modernas também fazem uso de operações relacionadas a vídeos digitais, bastante úteis para a Computação Gráfica em 3D.

Atualmente, existem várias empresas que atuam no mercado de GPUs pelo mundo. As duas principais são a NVIDIA, com a linha GeForce, e a ATI, com a linha Radeon. Além destas, a Intel também se destaca no ramo, mas as suas GPUs geralmente são fabricadas de forma integrada.

2. HISTÓRICO
Nos anos 70, foram criados os primeiros computadores da família Atari 8-bits (também chamados de *home computers*). Eles eram fabricados com chips ANTIC, que processavam instruções de display e eram responsáveis pelo "background" das telas gráficas, e CTIA, presentes nos modelos Atari 400 e 800 e realizavam operações de adição de cores e sprites[1] aos dados processados pelos chips ANTIC. Os chips CTIA foram posteriormente substituídos pelos chips GTIA nos modelos seguintes.

Nos anos 80, o Commodore Amiga foi um dos primeiros computadores de produção em massa a incluir um blitter[2] em seu hardware de vídeo. Além dele, o sistema 8514 da IBM foi uma das primeiras placas de vídeo para PC que implementavam primitivas 2D em hardware. Nessa época, o Amiga era considerado um acelerador gráfico completo, justamente pelo fato de ele transferir todas as funções alusivas à geração de vídeos para hardware. No entanto, uma CPU de propósito geral era necessária para lidar com todos os aspectos de desenho do display.

Já nos anos 90, continuava-se a evolução dos aceleradores 2D. Uma vez que as técnicas de fabricação se tornavam cada vez mais avançadas, a integração de chips gráficos seguia os mesmos passos. Foram surgindo uma série de API's (*Application Programming Interfaces*) que lidavam com um conjunto de tarefas. Enrte eles, estavam a biblioteca gráfica *WinG*, que aumentava a velocidade e a performance dos ambientes para o Windows 3.x, além de fazer com que jogos desenvolvidos para DOS fossem portabilizados para a plataforma Windows, e também o *DirectDraw*, que provia aceleração de hardware para jogos 2D em Windows 95 e nas versões posteriores.

Alguns anos mais tarde, a renderização de gráficos 3D em tempo real era bastante comum em jogos de computador e de alguns consoles, o que exigiu uma demanda cada vez maior

[1]Imagens ou animações em 2D / 3D que são integradas em uma cena maior.
[2]Dispositivo que realiza rápidas transferências de uma área de memória para outra.

Figura 1: Placa de vídeo 3dfx Voodoo3.

de placas gráficas aceleradoras 3D. Era o auge dos chamados consoles de videogames da quinta geração, tais como o PlayStation e o Nintendo 64. No tocante aos computadores, surgiram alguns chips gráficos 3D de baixo custo, como o S3 Virge, o ATI Rage e o Matrox Mystique, mas nenhum deles obteve muito sucesso. Isso se deve ao fato de esses chips serem basicamente aceleradores 2D com algumas funções de geração de gráficos 3D embutidas. No início, a performance obtida com gráficos 3D era obtida somente por meio de algumas placas dedicadas a funções de aceleração 3D, mas tendo uma baixa aceleração 2D como um todo. Uma dessas placas era a *3dfx Voodoo*, a primeira placa gráfica comercial, fabricada em 1995, e que era capaz de realizar mapeamento de textura em geometrias e possuía o algoritmo de Z-Buffer[3] implementado em hardware.

Em 1999, a NVIDIA lançou o modelo GeForce 256, considerada a primeira placa gráfica do mundo. Ela se destacou na época pelo alto número de pipelines, pela realização de cálculos de iluminação e de geometria e pela adição de recursos para compensação de movimentos de vídeo MPEG-2. Além disso, possibilitou um avanço considerável no desempenho em jogos e foi o primeiro acelerador gráfico compatível com o padrão DirectX 7.0. O sucesso desta placa foi tanto que causou a queda de seus concorrentes diretos no mercado, entre eles a 3dfx, fabricante da Voodoo3.

A partir de 2000, as GPUs incorporavam técnicas de *pixel shading*, onde cada pixel poderia ser processado por um pequeno programa que incluía texturas adicionais, o que era feito de maneira similar com vértices geométricos, antes mesmo de serem projetados na tela. A NVIDIA foi a primeira a produzir plcasa com tais características, com o modelo GeForce 3. Em meados de 2002, a ATI introduziu o modelo Radeon 9700, que foi a primeira placa do mundo compatível com o acelerador DirectX 9.0.

Em 2005, foi criado o barramento PCIe, que melhorava ainda mais o desempenho das transferências de dados entre GPU e CPU. A partir dele, foram criadas as tecnologias SLI, da NVIDIA, e CrossFire, da ATI, as quais permitiam interligar múltiplas GPUs em um único sistema, o que se assemelha à concepção de CPUs com múltiplos núcleos (cores).

Mesmo com toda essa evolução, uma vez que o poder de processamento das GPUs aumentava ao longo dos anos, elas exigiam cada vez mais energia elétrica. As GPUs da alta performance normalmente consomem mais energia do que as CPUs atuais.

[3]Gerenciamento de coordenadas de imagem em 3D. Muito utilizado para a solução de problemas de visibilidade de objetos em uma cena. Também conhecido como *depth buffering*.

Figura 2: Placa de vídeo ATI Radeon HD 4870.

Figura 3: Placa de vídeo GeForce GTX 295.

Nos dias de hoje, as GPUs paralelas tem disputado espaço com as CPUs, principalmente com a criação de técnicas como a GPGPU (*General Purpose Computing on GPU*), que consiste basicamente em fazer com que as GPUs executem tarefas de alto processamento gráfico, o que até certo tempo atrás era realizado pelas CPUs. Essa técnica possui diversas aplicações em processamento de imagens, álgebra linear, reconstrução 3D, entre outras áreas. Existem também uma forte demanda por parte dos adeptos da GPGPU para melhorias no design do hardware, com o intuito de tornar os modelos de programação mais flexíveis.

3. PIPELINES GRÁFICOS

Todos os dados processados pelas GPUs seguem o que se chama de pipeline gráfico, desenvolvido para manter uma alta frequência de computação por meio de execuções paralelas. O pipeline gráfico convencional, mostrado na figura 4, é composto por uma série de estágios, executados através de parâmetros definidos por uma API associada. Inicialmente, a aplicação envia à GPU, via barramento, um conjunto de vértices a serem processados. Em seguida, são definidas as primitivas geométricas utilizadas para o processo de rasterização, o qual consiste na conversão dessas primitivas em conjuntos de pixels. Em seguida, a GPU faz uma combinação dos vértices para gerar fragmentos para cada um dos pixels da imagem de saída, onde cada fragmento é definido por um conjunto de operações de rasterização, que incluem mapeamento de textura, combinação de cores, recortes, etc. O resultado dessa operação é então encaminhado ao framebuffer, também conhecido como memória de vídeo, que se encarrega de armazenar e transferir para a tela os dados processados nos estágios anteriores do pipeline.

Com o surgimento das placas gráficas programáveis, houve também uma mudança na concepção geral do pipeline gráfico. Alguns estágios do pipeline gráfico podem ser substituídos por programas que manipulam vértices e fragmentos. No entanto, quando um programa desse tipo é implemen-

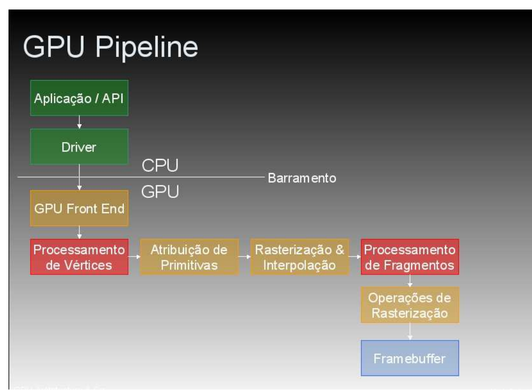


Figura 4: Esquema geral do pipeline gráfico.

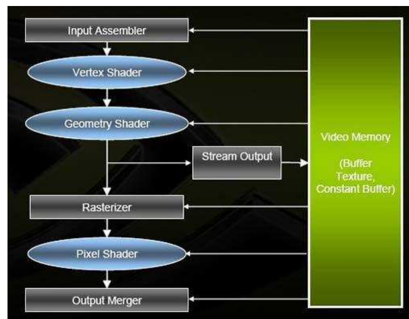


Figura 5: Novo pipeline gráfico.

tado, ele deve também implementar todos os estágios do pipeline gráfico que ele substitui.

Além disso, um novo estágio foi adicionado ao pipeline gráfico programável: o estágio de geometria, responsável por todo o processo de renderização de uma cena. Dessa forma, o novo pipeline adquire os estágios de *vertex shader*, *geometry shader* e *pixel shader*. O primeiro shader é normalmente usado para adicionar efeitos especiais em objetos presentes em um ambiente 3D. O segundo serve para gerar novas primitivas geométricas a partir dos dados recebidos do vertex shader. E o último tem a função de adicionar efeitos de luz e sombreamento aos pixels de uma imagem 3D.

A adição do estágio de geometria divide o pipeline em duas partes, como mostrado na figura 5. Um deles é o estágio de *Stream-Output*, responsável pela alocação de vértices oriundos do geometry shader em um ou mais buffers na memória, podendo também gerar novos vértices para o pipeline. O outro consiste no processo de rasterização descrito anteriormente, mas agora gerando uma entrada para o pixel shader.

Na época de lançamento do DirectX 10, as GPUs eram implementadas de forma a atender aos seus requisitos. Com isso, foi introduzida uma nova unidade, definida como *unified shader*, inicialmente introduzido no chip GeForce 8800 GTX, e que agrega todos os outros shadres citados anteriormente, conforme exibido na figura 6.

Recentemente, foi especulado um novo pipeline gráfico programável com a inclusão de novos shaders, graças à implementação do Direct3D 11, que provavelmente será lançado no segundo semestre de 2009. O esquema desse pipeline está representado na figura 7.

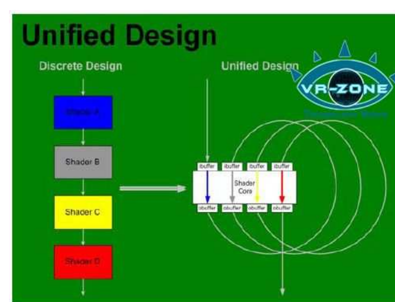


Figura 6: Agregação dos shaders em uma única unidade: o *unified shader*.

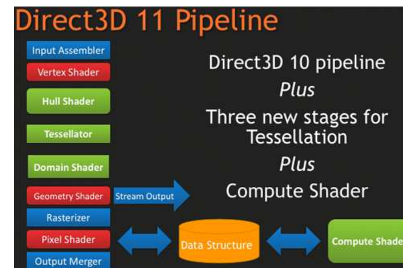


Figura 7: Pipeline do Direct3D 11, com a inclusão de novos shaders (e consequentemente, novos estágios).

4. CARACTERÍSTICAS DO HARDWARE

As GPUs foram projetadas com a ideia de executar todo tipo de operação gráfica. Normalmente, na execução de aplicações gráficas, uma mesma operação é feita repetidas vezes com vários dados diferentes, fazendo com que a arquitetura das GPUs fossem desenvolvidas para operar grandes conjuntos de instruções de maneira paralela. Para tornar esse paralelismo eficiente, o hardware foi implementado com base nos pipelines gráficos, conforme descrito na seção 3. Os pipelines, por sua vez, foram então implementados em várias unidades independentes. Desta forma, as GPUs conseguem manter altas frequências de computação em execuções paralelas de instruções.

Outra característica importante a ser ressaltada é o fato de o hardware das GPUs ser especializado, obtendo uma eficiência muito maior do que os hardwares de propósito geral (como as CPUs). Ao longo dos anos, a disparidade entre GPUs e CPUs, em relação à performance, foi se tornando cada vez maior, como mostra o gráfico da figura 8.

Uma segunda vantagem que as GPUs levam sobre as CPUs é que elas utilizam a maioria de seus transistores para a computação e muito pouco para a parte de controle, obtendo-se um poder de computação maior, mas tornando os fluxos de programas um pouco mais limitados.

Outro aspecto importante a se considerar é com relação ao acesso à memória. As GPUs procuram maximizar o th-

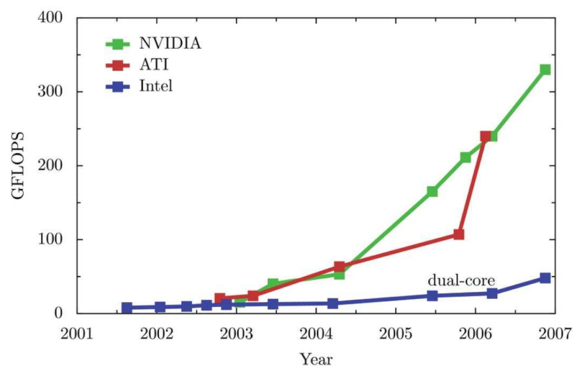


Figura 8: Evolução das performances dos processadores Intel e das placas de vídeo ATI e GeForce.

roughput, ao passo que as CPUs priorizam a latência no acesso. Isso ocorre porque a computação das GPUs tira mais proveito do chamado princípio da localidade do que a computação genérica. Nesse caso, valoriza-se mais a eficiência na transferência de um determinado conjunto de elementos, em detrimento do tempo gasto para acessá-los.

Atualmente, a GPU é o tipo de hardware mais barato para a computação de alto desempenho, quando a tarefa a ser processada se aproveita do seu alto grau de paralelismo. No entanto, mesmo que ela esteja se tornando cada vez mais programável, ainda é necessário um conjunto de fatores que proporcionem a melhor performance possível para uma determinada aplicação.

Dada essa eficiência das GPUs, podemos então imaginar que uma aplicação que é executada em uma CPU normal tem uma performance melhor quando executada em uma GPU. Esta é uma concepção leviana, já que nem toda aplicação pode ser diretamente adaptada para o uso em uma GPU, uma vez que é necessária a remodelagem para um problema gráfico, o que não é possível em boa parte dos casos.

5. MODELO DE PROGRAMAÇÃO

Um dos motivos pelos quais as CPUs não conseguem uma eficiência tão boa quanto as GPUs está no modelo de programação utilizado. As CPUs são desenvolvidas sob um modelo de programação serial, o qual não explora muito o conceito de paralelismo e os padrões de comunicação nas aplicações. Em contrapartida, as GPUs utilizam, como base, o modelo de programação por streams[4], que estrutura os programas de uma forma que se obtenha uma alta eficiência na computação e na comunicação.

No modelo de programação por streams, os dados podem ser simples (inteiros, pontos flutuantes) ou complexos (pontos, triângulos, matrizes de transformação). As streams também podem ter um tamanho qualquer, sendo que as operações executadas sobre as streams são mais eficientes se elas forem longas (contendo centenas de elementos ou mais). As operações que podem ser feitas nas streams são as de cópia de streams, construção de substreams a partir delas, indexação e computação por meio de kernels[5].

A eficiência na computação proveniente deste modelo está no alto grau de paralelismo dos dados na aplicação, onde os elementos das streams podem ser processados a partir de um hardware especializado em tal aspecto. Dentro de cada elemento processado, pode-se explorar um paralelismo a nível de instrução (ILP). Uma vez que as aplicações são constituídas de múltiplos kernels, eles também podem ser processados em paralelo, utilizando-se um paralelismo a nível de tarefa (TLP).

A adoção deste modelo no desenvolvimento de GPUs reflete uma série de tendências. A primeira delas é a habilidade de se concentrar uma vasta quantidade de dados a serem computados em um simples *die* do processador, bem como o uso eficiente desses componentes. Outra tendência está associada à queda no custo da produção de GPUs, fazendo com que elas se tornassem uma parte de um desktop padrão. Por fim, temos a adição de uma programabilidade de alta precisão ao pipeline gráfico, completando-se uma transição de um simples processador especializado para um poderoso processador programável que pode desempenhar uma grande variedade de tarefas.

6. DESAFIOS FUTUROS

O primeiro desafio a se considerar diz respeito às tendências da tecnologia. Cada nova geração de hardware se tornará um novo desafio para os fabricantes de GPUs, pois nesse caso é necessário integrar novos recursos de hardware de maneira eficiente, com o objetivo de melhorar a performance e a funcionalidade. Um número cada vez maior de transistores é adicionado às placas gráficas, proporcionando maior exploração de paralelismo e novas funcionalidades ao pipeline gráfico.

Além disso, também é necessário garantir a eficiência na comunicação. O aumento do custo desse fator influencia nas microarquiteturas dos futuros chips. Desta forma, os desenvolvedores devem planejar o tempo exigido para o envio de dados ao longo dos chips.

Outro desafio para a evolução das GPUs é com relação ao consumo de energia, um fator que tem se tornado bastante crítico no desenvolvimento das GPUs atuais, uma vez que cada nova geração de hardware apresenta uma demanda de energia cada vez maior. É esperado então um maior gerenciamento de energia nos estágios do pipeline, bem como sistemas de resfriamento (cooling) ainda mais sofisticados.

Por fim, temos a questão da programabilidade e das funcionalidades que as GPUs oferecem. Mesmo com todo o progresso que se conquistou até os dias de hoje, a programabilidade em GPUs ainda está um pouco longe do ideal. Uma ideia para a evolução destes fatores é melhorar a flexibilidade e as funcionalidades das unidades programáveis atuais (de vértices e de fragmentos). Possivelmente, os seus conjuntos de instruções podem se convergir, tornando os seus controles de fluxos mais generalizados. Podemos ver também um

[4]Conjuntos ordenados de dados do mesmo tipo.

[5]Operador que age sobre streams, produzindo um outro conjunto de streams como saída. No caso do pipeline gráfico, deve-se implementar kernels para o processamento de vértices, de primitivas geométricas, e assim por diante.

compartilhamento de hardware programável entre esses dois estágios, de forma a utilizar melhor os seus recursos. Outra opção é expandir a programabilidade para unidades diferentes.

7. CONCLUSÃO

Mesmo sendo a GPU um componente recente no mundo da computação, ela vem mantendo uma evolução cada vez mais rápida e se tornando a cada dia mais útil para diversas aplicações gráficas. Isso pode ser evidenciado pela quantidade de modificações que ela sofre entre uma versão e outra, exigindo novas modelagens de seus componentes (ou até mesmo sobre as tecnologias utilizadas) e também novas implementações, visando melhoras ainda mais significativas na sua performance.

A introdução do conceito de programação genérica em placas gráficas (GPGPU) foi uma grande revolução para o uso das GPUs. Fabricantes de GPUs vem investindo bastante em melhorias na arquitetura de modo a tornar a programação cada vez mais simples e acessível aos programadores convencionais. Assim, esse conceito pode prover um ganho na eficiência do processamento de programas que não constituem essencialmente um problema gráfico.

É importante ressaltar também que essa evolução das GPUs, incorporando cada vez mais poder de processamento e capacidade nos seus chips, pode provocar algum conflito entre os seus fabricantes e os fabricantes de CPUs. Futuramente, não se sabe se as próximas gerações de computadores constituirão de CPUs que incorporarão funcionalidades básicas das GPUs ou então o contrário. Questões como esta são um bom desafio para os futuros projetistas, dentre uma série de outros desafios inerentes ao ramo tecnológico.

8. REFERÊNCIAS

[1] NVIDIA Corporation. Geforce256 – The World's First GPU, 1999.

[2] Randy Fernando, GPGPU: General-Purpose Computation on GPUs, 2004.

[3] Randy Fernando & Cyril Zeller, Programming Graphics Hardware, 2004.

[4] Artigo da Wikipedia – Graphics Processing Unit.

[5] NVIDIA Developer Zone – CPU Gems 2

[6] David Luebke (NVIDIA Research) & Greg Humphreys (University of Virginia), How GPUs work – artigo publicado em 2007, no site Computer Magazine, da IEEE Computer Society.

Trace Scheduling

Vitor Monte Afonso
046959
vitor@las.ic.unicamp.br

ABSTRACT
Existem diversas técnicas usadas por compiladores para otimizar código, tornando-o mais rápido ou até diminuindo (compactando) o código, como é o caso da técnica *trace scheduling*. Neste artigo esse método de otimização é descrito, além disso, são apresentados os algoritmos existentes.

Termos Gerais
Otimização global de código, compactação de microcódigo, paralelização de instruções

Palavras-chave
Trace scheduling, compactação global de microcódigo

1. INTRODUÇÃO
Para um bom aproveitamento dos recursos computacionais diversas técnicas de otimização de código são empregadas. Uma forma de fazer isso é com o desenvolvedor indicando que conjuntos de instruções podem ser executados em paralelo, mas esse método é muito difícil e custoso. Por isso é necessário que esses processos de otimização sejam empregados pelo compilador.

Uma forma de alcançar isso é utilizando a compactação de microcódigo. Esta consiste de transformar código seqüencial (vertical), em código horizontal, através da paralelização das microoperações.

Essa compactação pode ocorrer de duas maneiras, localmente ou globalmente. O método local consiste da otimização dentro dos blocos básicos. Blocos básicos são seqüências de instruções que não possuem saltos, excetuando-se a última instrução. Entretanto os blocos básicos não costumam possuir muitas instruções, fazendo com que essa otimização seja bastante limitada.

Isso mostra a importância do método global, no qual vários blocos básicos são levados em consideração de uma única vez, ao invés de trabalhar um a um. Pesquisas mostram que dessa forma é possível obter mais paralelismo [2], [3].

Estudos passados mostram que esse é um problema NP-completo [1], então achar sua solução ótima é computacionalmente inviável. Para isso foi criada a técnica de *trace scheduling* que apesar de não chegar na solução ótima, se aproxima consideravelmente dela.

O objetivo deste trabalho é apresentar essa técnica e mostrar os algoritmos utilizados nela. Ele está organizado da seguinte forma, na seção 2 são apresentadas algumas definições, na seção 3 são apresentados os algoritmos e na seção 4 as conclusões.

2. CONCEITOS NECESSSÁRIOS
Durante a compactação de microcódigo trabalhamos com microoperações (MOPs) e grupos de MOPs. MOPs são a operação mais básica com a qual o computador trabalha.

Uma ou mais MOPs podem formar micriinstruções (MIs), que é a menor unidade com a qual o algoritmo de *trace scheduling* trabalha. A união das Mis forma P, o programa que estiver sendo procesado.

Existe também a função *compacted* : P → {*true, false*}. No início da compactação todas as Mis são inicializadas com *false*. Se *compacted*(m) é *false*, m é chamado de MOP.

As funções *readregs* e *writeregs* : P → conjunto de registradores, definem respectivamente os registradores lidos e escritos pelas microinstruções.

Há ainda a função *resource compatible* : P → {*true*,*false*}, que define se um determinado conjunto de microinstruções pode ser executado em paralelo, ou seja, o processador possui recursos suficientes para todas.]

Durante o processo de compactação algumas MOPs podem mudar de posição. Para que essa mudança não acarrete em alterações na semântica do programa algumas regras devem ser seguidas.

Definição 1: Dada uma certa seqüência de MIs (m1, m2, m3, ..., mt), é definida a ordem parcial (<<). Se mi << mj mi tem precedência de dados sobre mj. Ou seja, se mi escreve em um registrador e mj lê esse valor, mi << mj e mj não pode ler esse valor enquanto mi não escrevê-lo. Além disso, se mj lê um registrador e mk escreve nele posteriormente, mj << mk e mk não pode escrever no registrador enquanto ele não for lido por mj.

Definição 2: A ordem parcial gera um grafo direcionado acíclico (DAG) que é chamado de grafo de precedência de dados, cujos nós são Mis e existe uma aresta de mi para mj se mi << mj.

Definição 3: Dado um DAG gerado a partir de P, a função sucessores: P → conjuntos de P define os sucessores de uma MI da seguinte forma. Se mi,mj pertencem a P e i<j, mj pertence ao sucessores(mi) se existe uma aresta de mi para mj.

Definição 4: Tendo um P com um grafo de precedência de dados, definimos a compactação ou scheduling como um particionamento de P em conjuntos disjuntos S = (S1, S2, ..., Su) seguindo certas propriedades. Para cada k, 1 ≤ k ≤ u, *ressource_compatible*(Sk) = true. Além disso, se mi << mj, mi está em Sk e mj está em Sh, com k<h.

3. TRACE SCHEDULING

3.1 Método Menu
Muitos programadores realizam a compactação de microcódigo manualmente, quando acham que é possível. A tabela 1 mostra um menu com regras para movimentação de código que poderia

ser usado implicitamente por um programador que faria essa compactação.

A tabela é feita utilizando conceitos de grafos de fluxo. Definições de grafos de fluxos podem ser vistas em [4] e [5].

Regra	DE	PARA	Condições
1	B2	B1 e B4	MOP livre no início de B2
2	B1 e B4	B2	Cópias da MOP estão livres no fim de B1 e B4
3	B2	B3 e B5	MOP livre no fim de B2
4	B3 e B5	B2	Cópias da MOP estão livres no início de B3 e B5
5	B2	B3 (ou B5)	MOP livre no fim de B2 e os registradores modificados pela MOP estão mortos em B5 (ou B3)
6	B3 (ou B5)	B2	MOP livre no início de B3 (ou B5) e os registradores modificados pela MOP estão mortos em B5 (ou B3)

Tabela 1. Menu com regras para movimentação de código

Esse método foi automatizado por técnicas anteriores de compactação de microcódigo [6], [7]. Isso é feito basicamente da seguinte forma:

1. Apenas código sem laços é tratado;
2. Os blocos básicos são tratados separadamente;
3. É feita ordenação nos blocos básicos;
4. Movimentações de blocos para blocos tratados anteriormente são consideradas e são feitas se forem salvar ciclos.

Esse método automatizado possui alguns problemas.

- Quando uma MOP é movida, podem ser criadas mais possibilidades de movimentação. Isso causa uma grande quantidade de buscas na árvore, deixando o processo bastante pesado;
- Operações em movimentações são pesadas e são repetidas muitas vezes;
- Para localizar movimentações que resultam em grandes melhorias é necessário passar antes por movimentações que não ajudam ou até pioram o código.

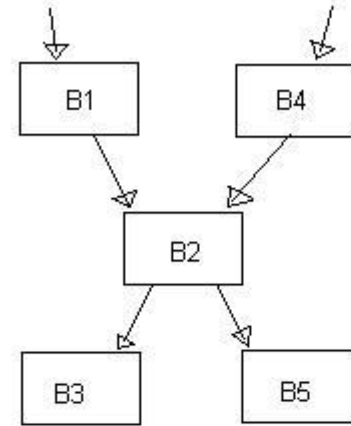


Figura 1. Grafo de fluxo

3.2 Trace Scheduling

Os problemas encontrados pelo método menu são resolvidos na técnica de trace scheduling. Esta, ao invés de operar sobre blocos básicos, opera sobre *traces.* Traces são seqüências de instruções sem ciclos que, para determinado conjunto de dados, são executadas continuamente. Assim, a técnica avalia vários blocos básicos por vez, podendo localizar mais possibilidades de movimentação de código.

A definição que segue será usada pelo algoritmo.

Definição 5: Dada a função *followers*: P → subconjuntos de P e uma microinstrução m, o conjunto *followers*(m) é composto pelas microinstruções que podem ser executadas após a execução de m. Se mi está no *followers*(mj), então dizemos que mj é líder de mi. Se o *followers*(m) possui mais de uma micoinstrução, então m é chamada de *jump* condicional.

Um *trace* é portanto uma seqüência de microinstruções distintas (m1, m2, m3, ..., mt) tal que para cada j, $1 \leq j \leq t-1$, mj+1 está no *followers*(mj). Este *trace* forma um DAG da seguinte forma.

Definição 6: Dado um *trace* T = (m1, m2, m3, ..., mt), usamos a função de sucessores para gerar um DAG chamado grafo de precedência de dados de um trace. Isso é feito de forma análoga aos blocos básicos usando *readregs*(m) e *writeregs*(m), com excessão das arestas de *jumps* condicionais. Se m é um *jump* condicional, os registros em *condreadregs*(m) são tratados como sendo de *readregs*(m).

Então, o *trace scheduling* para códigos sem ciclos, é feito da seguinte maneira. Dado P, selecionamos o caminho cuja execução é mais provável dentre as MOPs que ainda não foram compactadas, a partir de uma aproximação de quantas vezes cada MOP é executado para um conjunto de dados, depois construímos o DAG e fazemos a compactação.

Após esse processo, algumas inconsistências podem ter sido inseridas, por isso é necessário executar o algoritmo de *bookkeep* descrito em [8].

3.3 Código com Loops

Códigos com loops devem ser tratados com cuidado porque em geral estas são as partes que executam com maior freqüência nos programas.

Definição 7: Loops são conjuntos de microinstruções que ligam um bloco a um bloco anterior no grafo de fluxo. Mais detalhes podem ser vistos em [5].

Assumindo que o grafo em questão é redutível e os loops estão ordenados. A figura 2 mostra um exemplo de grafo redutível.

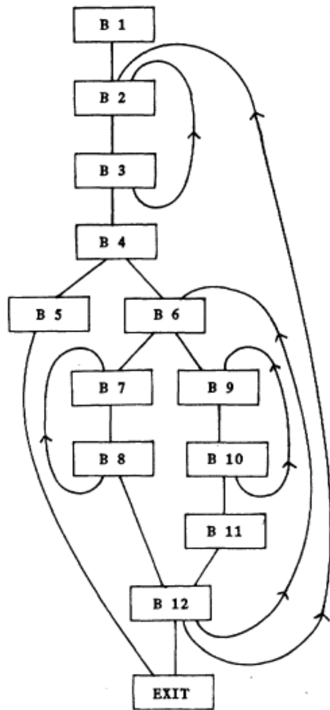


Figura 2. Grafo redutível

Existem duas formas de modificar o algoritmo de *trace scheduling* de forma que ele trate códigos com loops.

No primeiro caso, basta compactar os loops na ordem do mais interno para o mais externo, As arestas de retorno são tratadas como *jumps* para *exit*. O último loop a ser tratado é o P.

O método acima não se aproveita de certos casos em que a compactação pode ser consideravelmente mais efetiva, que são apresentados a seguir.

- Deve-se considerar trocas entre MOPs que estão antes e que estão depois de loops;
- Em certos casos de loops que executam poucas vezes, uma boa estratégia é mover certos MOPs para dentro do loop quando podem ser feitos sem ciclos adicionais e são invariantes de loop.

A técnica mais eficiente trata cada loop compactado como um MOP do loop de nível superior. O escalonador não tem ciência dessa representação, então quando um desses loops aparece no *trace*, as operações que estão antes e depois dele poderão ser alteradas sem problemas.

A definição a seguir é importante para podermos mover MOPs para dentro de loops também.

Definição 8: Dado um loop L, sua representação como MOP lr e um conjunto de operações N, o conjunto [lr] U N é *resource compatible* se N não possui representações de loop, cada operação de N é invariante com relação a R e a adição das operações de N no escalonador de L não o tornam mais pesado.

Nesse caso os loops também começam a ser compactados a partir do mais interno, e os loops vão sendo cada um na sua vez substituídos por MOPs representativos. Transferências de controle de e para o loop são tratadas como *jumps* de e para a instrução que representa o loop.Após isso o algoritmo se comporta como no caso em que não são tratados loops. Quando um representante de loop aparece no *trace*, ele é incluído no DAG.

Assim que o DAG está terminado, o processo de *scheduling* continua até que algum representante de loop esteja pronto. Ele é então considerado para inclusão em cada novo ciclo C. A inclusão é feita se a microinstrução representativa for *resource compatible* com as operações que já estão em C.

Após o fim do escalonamento os representantes são substituíddos pelos loops completos incluindo as possíveis modificações que tenham sido feitas. O algoritmo de *bookkeeping* é então executado normalmente. Esse método possibilita a movimentação de MOPs para antes, depois e até para dentro de loops.

3.4 Melhorias Para o Algoritmo de Trace Scheduling

3.4.1 Reduzindo Espaço

O algoritmo apresentado consome uma grande quantidade de memória devido ao algoritmo de *bookkeeping*. Essa perda de espaço se deve mais especificamente ao espaço necessário para gerar escalonamentos mais curtos e espaço usado porque o escalonador toma decisões arbitrárias, que as vezes acabam consumindo mais espaço em memória do que é necessário.

Medidas para redução do consumo de espaço podem ser tomadas juntamente com a compactação de código, a partir da adição de arestas no DAG para reduzir a duplicação, ao custo de flexibilidade.

Se a probabilidade de um bloco ser executado está abaixo de um certo limite e um escalonamento curto não é crítico, o processo de adição destas arestas segue da seguinte forma:

- No caso de um bloco terminado em *jump* condicional, adicionamos uma aresta de cada MOP que está acima do *jump* e escreve em um registrador que está vivo. Esta aresta irá ajudar a evitar que sejam feitas cópias em blocos que não são muito utilizados;
- Se o início do bloco é um ponto de reunião do *trace*, adicionamos uma aresta para cada MOP livre no início do bloco, de cada MOP que está em um bloco anterior no *trace* e não possui sucessores em blocos anteriores. Isso deixa o ponto de reunião limpo e sem cópias;
- Arestas de todos os MOPs que estão acima de pontos de reunião para a representação do ciclo previnem cópias inapropriadas..

3.4.2 Task Lifting

Antes de compactar um *trace* que sai de um outro *trace* já compactado, pode ser possível mover MOPs que estão no início do *trace* novo para buracos no escalonamento do outro. Essa técnica é explicada com mais detalhes em [9].

4. CONCLUSÕES

No artigo foi apresentada a técnica de *trace scheduling* [8] proposta por Fisher em 1981 que é utilizada para realizar compactação de microcódigo. A técnica possui alguns problemas mas sua utilização é bastante útil. Foram apresentadas também maneiras de aprimorar o algoritmo.

5. REFERÊNCIAS

[1] S. S. Yau, A. C. Schowe and M. Tsuchiya, "On Storage Optimization of Horizontal Microprograms," Proceedings of 7th Annual Microprogramming Workshop, Sept. 30-Oct. 2, 1974, pp. 98-106

[2] C. C. Foster and E. M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution," IEEE Trans. Comput., vol. C-21, pp.1411-1415, Dec. 1972

[3] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Trans. Comput., vol. C-21, pp. 1405-1411, Dec. 1972

[4] A. V. Aho and J. D. Ullman, Principles of Compiler Design. Reading,MA: Addison-Wesley, 1974.

[5] M. S. Hecht, Flow Analysis of Computer Programs. New York: El-sevier, 1977.

[6] S. Dasgupta, "The organization of microprogram stores," ACM Comput. Surveys, vol. 11, pp. 39-65, Mar. 1979.

[7] M. Tokoro, T. Takizuka, E. Tamura, and 1. Yamaura, "Towards an efficient machine-independent language for microprogramming," in Proc. II th Annu. Microprogramming Workshop, SIGMICRO, 1978, pp. 41-50.

[8] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Transactions on Computers, C-30(7): 478-490, July 1981

[9] J. A. Fisher, "The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with re-sources," Courant Math. Comput. Lab., New York Univ., U.S. Dep.of Energy Rep. COO-3077-161, Oct. 1979

Tecnologia Bluetooth e Aspectos de Segurança

André Ricardo Abed Grégio
R.A. 079779
Instituto de Computação
Unicamp

abedgregio@gmail.com

RESUMO

Bluetooth é uma tecnologia definida por um padrão especificado pelo Bluetooth Special Interest Group (SIG), cujo objetivo é prover um meio de baixo custo, consumo de energia e baixa complexidade de configuração para interligar dispositivos eletrônicos diversos, tais quais telefones celulares, notebooks, desktops, câmeras digitais, impressoras, PDAs e periféricos em geral. Neste trabalho é apresentada a tecnologia para redes sem fio de curto alcance denominada Bluetooth, abordando aspectos de sua arquitetura, protocolos e segurança do sistema.

Termos Gerais

Padronização, segurança, projeto

Palavras-Chave

Bluetooth, redes wireless, segurança de sistemas de informação

1. INTRODUÇÃO

Desde o surgimento das redes de comunicação de dados utilizando computadores, nos idos dos anos 60, tem-se buscado novas soluções para aumentar a mobilidade e conectividade dos sistemas, de forma a prover um alto grau de independência aos usuários. Tal independência é alcançada com a evolução das tecnologias de interfaces de rede, como as wireless, aliado à facilidade do uso. Neste aspecto, o hardware possui papel importantíssimo, pois é necessário garantir a confiabilidade e integridade dos dados em trânsito, bem como a robustez do dispositivo.

No que diz respeito às tecnologias sem fios para sistemas computacionais, há vários protocolos disponíveis para tratar da comunicação em diversas áreas de alcance. Os dispositivos infra-vermelhos (IrDA) são utilizados para comunicações simples, pois a tecnologia possui alcance limitado, baixa transmissão de dados e nenhuma necessidade especial de configuração por parte do usuário. Redes utilizando IrDA também são conhecidas como PAN, ou Personal Area Networks. Para transmissões de dados de longo alcance, há as WLANs – Wireless Local Area Networks – que podem trafegar mais dados em menos tempo, mas que necessitam de uma configuração, mesmo que mínima, para que fiquem disponíveis para operação. Para suprir o *gap* entre as duas tecnologias citadas, isto é, para se ter uma solução intermediária em termos de configuração facilitada e alcance razoável, foi desenvolvida a tecnologia Bluetooth [3].

A tecnologia Bluetooth permite cobrir uma distância maior em termos de dispositivos conectados e formar pequenas redes, enquanto que não necessita de conhecimentos especializados a fim de configurar os dispositivos.

Neste trabalho serão descritos os pormenores da tecnologia Bluetooth, tais como protocolo e arquitetura, bem como aspectos relacionados à segurança deste tipo de comunicação.

Na seção 2, um breve histórico da tecnologia Bluetooth será apresentado. Na seção 3 será descrita a arquitetura Bluetooth. Na seção 4, é explicada a pilha de protocolos do Bluetooth, enquanto que na seção 5 serão abordadas algumas características relacionadas à segurança. As considerações finais encontram-se na seção 6.

2. BREVE HISTÓRICO

Com objetivo de eliminar os cabos das conexões entre dispositivos e desenvolver um padrão que atendesse à demanda de interconectar não só computadores, mas outros dispositivos de comunicação (ex.: celulares e PDAs) e acessórios (ex.: controles remotos, fones de ouvido, mouses) de maneira eficiente e de baixo custo, cinco companhias (Ericsson, IBM, Intel, Nokia e Toshiba) se uniram para formar um consórcio (SIG – Special Interest Group) [6]. O consórcio foi nomeado Bluetooth, em homenagem ao Rei Viking Harald "Bluetooth" Blaatand, que unificou Noruega e Dinamarca durante seu reinado (940-981) e cuja estratégia era baseada no diálogo [4].

3. ARQUITETURA BLUETOOTH

3.1 Características

De maneira similar à tecnologia Wi-Fi 802.11, Bluetooth também utiliza a faixa de freqüência de 2.4 GHz, estando sujeita aos mesmos problemas de interferência ocorridos nesta faixa de freqüência [1].

A definição do protocolo Bluetooth abrange tanto dados como voz, e é para ser utilizado por dispositivos diversos, que vão desde celulares e PDAs, até microfones e impressoras. É dividido em três classes, onde a variação se dá na potência máxima e na área de cobertura estimada [4]. A tabela 1 a seguir ilustra as classes em que os equipamentos podem ser divididos:

Tabela 1: Potência e área de cobertura por classe

Classe	Potência máxima (mW)	Potência máxima (Dbm)	Área de cobertura estimada
1	100	20	100 m
2	2.5	4	10 m
3	1	0	1 m

3.2 Arquitetura

Um sistema Bluetooth pode ser composto por oito dispositivos separados por uma distância de 10 metros entre si, sendo um nó mestre e até sete nós escravos ativos. Podem haver até 255 outros nós na rede, colocados em estado suspenso pelo nó mestre, a fim de economizar bateria. Um dispositivo suspenso tem por função apenas responder a sinais de ativação ou sinalização (*beacon*) enviados pelo mestre, o qual pode suspender dispositivos e ativar outros, desde que obedecendo ao limite de 8 nós ativos [1].

O conjunto dos dispositivos interconectados da maneira supracitada é chamado de *piconet* [2], pois forma uma pequena rede de comunicação entre eles. A Figura 1 mostra um exemplo de *piconet.*

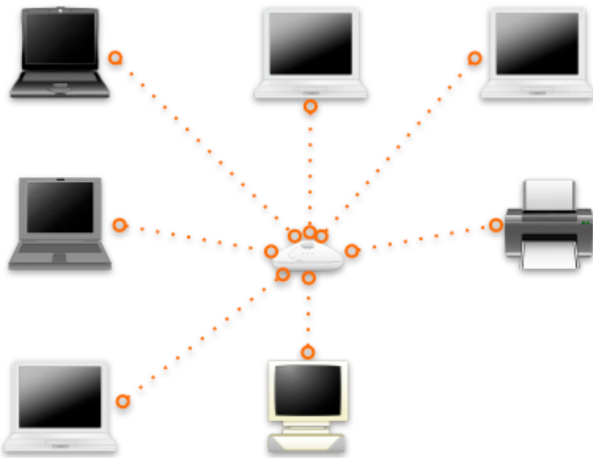


Figura 1: Vários nós escravos conectados a um mestre formando uma piconet

Caso um dos nós escravos seja configurado em modo *bridge* e permitir a interligação de uma ou mais *piconets*, forma-se uma *scatternet* [6]. Assim, a *scatternet* é a rede formada pela interconexão de diversas *piconets* presentes em um ambiente, como ilustrado na Figura 3.2.

A arquitetura mestre-escravo facilita a redução de custo e economia de bateria, uma vez que o mestre controla a comunicação entre os escravos, os quais apenas efetuam as ações que lhes são comandadas.

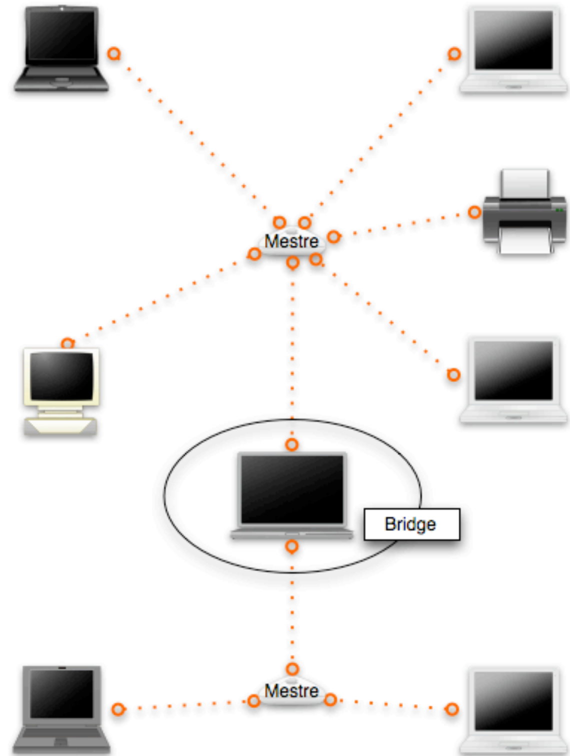


Figura 2: Uma scatternet formada por duas piconets

4. A PILHA DE PROTOCOLOS

O padrão Bluetooth é composto por diversos protocolos agrupados em camadas, conforme pode-se visualizar na Figura 3.

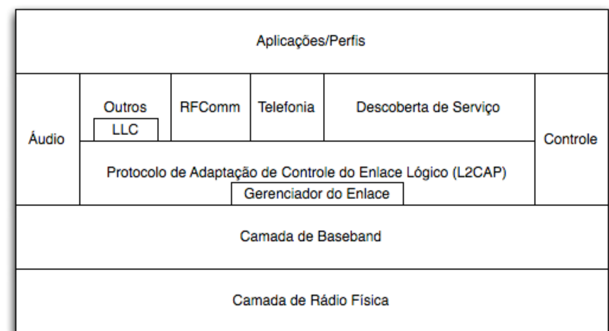


Figura 3: Arquitetura Bluetooth segundo padrão IEEE 802.15

A arquitetura básica dos protocolos que correspondem ao padrão Bluetooth é dividida em camada física de rádio; camada de *baseband*; uma camada (L2CAP) com alguns protocolos relacionados entre si e que, junto com a camada de *baseband*

corresponde à camada de enlace de dados; a camada de *middleware* e a camada de aplicações [1].

4.1 Camada de Rádio

A camada de rádio tem por função movimentar os bits entre nós mestre e escravo. Nesta camada ocorre a emissão dos sinais e alocação de canais, na qual o mestre estabelece a seqüência de salto entre os canais e todos os nós da piconet saltam ao mesmo tempo, através de FHSS (*Frequency Hopping Spread Spectrum*). A banda utilizada (ISM 2.4 GHZ) é dividida em 79 canais de 1 MHz e a taxa de dados é de aproximadamente 1 Mbps.

4.2 Camada de Baseband

A função da camada de *baseband* é transformar um fluxo de bits em frames, bem como definir alguns formatos importantes, como a divisão de *slots* de tempo para comunicação dos dispositivos mestre e escravos. Por padrão, um nó mestre de uma piconet define *slots* de 625 micro segundos, sendo que as transmissões do mestre são iniciadas nos *slots* pares e as dos escravos iniciam-se nos *slots* ímpares.

Para a transmissão dos frames, é estabelecido um link entre o mestre e o escravo, o qual pode ser assíncrono (ACL – *Asynchronous Connection-Less*) ou síncrono (SCO – *Synchronous Connection Oriented*).

Os links do tipo ACL são utilizados por dados enviados em intervalos irregulares e entregues seguindo a filosofia do melhor esforço, ou seja, não há garantias de que os dados cheguem íntegros ou que não haja necessidade de retransmissão devido à corrupção ou perda de pacotes.

Os links do tipo SCO são utilizados normalmente para dados que necessitam ser trafegados em tempo real e, portanto, possuem prazos críticos de tempo. Neste caso, é alocado um slot fixo para cada direção de tráfego e os frames não são retransmitidos em caso de perda.

4.3 Camada L2CAP

Nesta camada são tratados o gerenciamento de potência, autenticação e qualidade de serviço, correspondendo ao estabelecimento dos canais lógicos entre os dispositivos. Para encapsular os detalhes de transmissão enviados para camadas superiores, há o protocolo de adaptação de controle do enlace lógico (L2CAP – *Logical Link Control Adaptation Protocol*), que nomeia a camada. Suas três principais funções são:

- Receber pacotes das camadas superiores e dividí-los em frames para transmissão, a fim de que sejam remontados no destino, sendo que os pacotes devem respeitar o limite de 64 KB;
- Gerenciar a multiplexação e demultiplexação dos pacotes provenientes de origens distintas, determinando qual protocolo de camada superior irá tratá-los quando da remontagem destes;
- Gerenciar os requisitos de qualidade de serviço tanto durante o estabelecimento dos links como na operação normal.

4.4 Camada de Middleware

Na camada de *middleware* repousam protocolos relacionados com outros padrões – como *Wireless Application Protocol* (WAP), *Transmission Control Protocol* (TCP), *Internet Protocol* (IP) – e protocolos de terceiros ou de interesse, como o protocolo de descoberta de serviço, o qual possibilita que dispositivos obtenham informações sobre serviços disponíveis em outros dispositivos Bluetooth.

4.5 Camada de Aplicações e Perfis

Diferentemente de outros protocolos de rede, como por exemplo o 802.11, a especificação do Bluetooth elege aplicações específicas para ser suportadas e provê suas respectivas pilhas de protocolos. São 13 as aplicações definidas pelo Bluetooth SIG, e estas aplicações são denominadas **perfis**, os quais estão identificados e descritos na Tabela 2 a seguir.

Tabela 2: Perfis definidos pelo Bluetooth SIG como aplicações suportadas pelo protocolo

Nome	Descrição
Acesso genérico	Procedimentos para gerenciamento do enlace
Descoberta de serviço	Protocolos para descoberta de serviços oferecidos
Porta serial	Reposição para cabo de porta serial
Troca de objetos genérica	Define relacionamento cliente-servidor para movimentação de objeto
Acesso à LAN	Protocolo entre computador móvel e LAN fixa
Conexão discada	Permite a um laptop efetuar chamadas via telefone móvel
Fax	Permite a um aparelho móvel de fax conversar com um telefone móvel
Telefonia sem fio	Conecta um aparelho de telefone sem fio e sua estação-base local
Intercom	Walkie-talkie digital
Headset	Comunicação *hands-free*
Push de objetos	Provê troca simples de objetos
Transferência de arquivos	Provê transferência de arquivos geral
Sincronização	permite a um PDA sincronizar com outro computador

5. SEGURANÇA EM BLUETOOTH

As tecnologias wireless possuem problemas inerentes de segurança devido ao meio físico não ser auto-contido (como um cabo de rede) e a comunicação dar-se por intermédio de ondas eletromagnéticas as quais podem ser facilmente interceptadas.

Os riscos a que as comunicações Bluetooth estão submetidos podem ser divididos basicamente em [4]:

- Captura de tráfego (escuta);
- Negação de serviço;
- Forja de identidade;
- Configuração padrão e força bruta
- Acesso não autorizado;

Estes riscos, quando explorados ou combinados, causam problemas de violação de integridade, confidencialidade ou disponibilidade da *piconet* e dos dados dos dispositivos/usuários conectados a ela. A seguir, serão detalhados os riscos citados.

5.1 Captura de Tráfego

Uma vez que o meio de transmissão das comunicações envolvendo Bluetooth é o não guiado, a captura de tráfego torna-se uma tarefa simples. Basta se utilizar de uma ferramenta de análise de tráfego, mais conhecida pelo nome de *sniffer* e escutar o tráfego em fluxo.

Existem ferramentas para escuta de tráfego de rede tradicionais para uso em sistemas *unix-like* e plataforma Microsoft, como *tcpdump* e *Wireshark*, bem como ferramentas próprias para escuta em Bluetooth, como o *hcidump*.

A captura de tráfego pode fornecer informações sensíveis sobre a comunicação, tais quais as características deste tráfego, seqüências de comandos, senha (PIN) e arquivos (fotos, dados) transmitidos [5].

Nos dispositivos cabeados, a dificuldade da escuta está no acesso físico aos equipamentos. Nos dispositivos *wireless*, a dificuldade é centrada na distância entre o atacante e o alvo, sendo que nos equipamentos Bluetooth esta distância chega a cerca de 250 metros.

5.2 Negação de Serviço

Ataques de negação de serviço são comuns em redes de computadores e, se bem orquestrados, muito difíceis de serem evitados. Este tipo de ataque consiste na indisponibilização do uso do equipamento, fazendo-o deixar de comunicar-se com a rede ou com outros dispositivos temporariamente.

No caso da tecnologia Bluetooth, uma negação de serviço pode simplesmente consistir do envio de pacotes um pouco maiores ou do envio massivo de pacotes, fazendo com que o dispositivo alvo não consiga tratar o tráfego e comece a descartá-lo, bem como toda e qualquer conexão estabelecida ou tentativas de estabelecimento enquanto sob ataque.

Outra maneira de causar negação de serviço em Bluetooth é baseada na geração de ruído, ou *jamming*, seja através da identificação da seqüência de saltos na mesma ordem de freqüência dos dispositivos envolvidos, seja através do preenchimento de boa parte do espectro com tráfego de ruído.

5.3 Forja de Identidade

Para que um dispositivo Bluetooth estabeleça uma comunicação com outro e troque dados, é feita uma autenticação entre eles, na qual é combinada uma senha que pode ser memorizada nas partes envolvidas. Quando essa autenticação é feita, também é estabelecida uma relação de confiança entre os dispositivos, e a falsificação das credenciais de um dos dispositivos por um terceiro pode fazer com que este tenha acesso ao dispositivo alvo, fazendo-se passar pela parte autorizada.

Tanto a forja de identidade como a captura de tráfego podem levar ao roubo de informações, uma vez que na captura de tráfego, escuta-se a transmissão e é possível obter credenciais que possibilitem a forja de identidade.

5.4 Configuração Padrão e Força Bruta

Tratam-se de problemas de autenticação que permitem que um dispositivo seja acessado com suas próprias credenciais.

No caso da configuração padrão, o atacante tenta utilizar um PIN pré-determinado pelo fabricante para aquele tipo de equipamento, identificado através de uma varredura por dispositivos Bluetooth e suas características. Caso o dispositivo esteja configurado para aceitar conexões sem a solicitação de confirmação e o usuário não tiver trocado o seu PIN, o ataque será concretizado com sucesso.

No ataque de força bruta, o atacante tenta descobrir o PIN através do esgotamento de possibilidades, tentando todas as combinações possíveis dentro de um escopo específico.

5.5 Acesso Não Autorizado em Redes

Usuários mal intencionados podem configurar seus computadores pessoais ou mesmo concentradores Bluetooth para permitir acesso de dispositivos Bluetooth à rede cabeada ou *wi-fi* de uma organização. Com isso, um atacante sem acesso físico ou credenciais para acessar uma rede corporativa poderia se utilizar dessa ponte Bluetooth para efetuar o acesso e obter informações sensíveis. Este tipo de ataque é dificultado devido ao alcance limitado da tecnologia Bluetooth, mas é facilitado devido ao fato de que normalmente há pouca ou nenhuma monitoração de redes Bluetooth nas organizações em geral.

6. CONSIDERAÇÕES FINAIS

A tecnologia Bluetooth alcançou altos níveis de popularidade, uma vez que está presente em diversos dispositivos do dia-a-dia. Sua flexibilidade permite que seja utilizada em diversas atividades, como sincronismo de bases de dados em geral (agendas de telefone, fotos, músicas, entre outros), formação de redes ponto a ponto, possibilidade de acesso discado, integração de equipamentos em uma pequena rede (headset, impressora, celular, PC) e acesso à redes IP, por exemplo.

O Projeto Bluetooth alcançou seu objetivo de prover uma tecnologia versátil para comunicação e de baixo custo e consumo

de energia sem abrir mão da facilidade de utilização e configuração.

Enquanto que a arquitetura e protocolos do padrão Bluetooth são bem definidos e limitados em escopo, este tipo de rede padece das mesmas vulnerabilidades encontradas nas redes sem fio padrão 802.11, devido à inerência de problemas no meio não-guiado.

Espera-se que, com a cada vez mais crescente utilização de dispositivos Bluetooth, sejam desenvolvidas mais métodos de segurança e formas para monitoração das redes Bluetooth, visando a autenticação e auditoria do ambiente.

7. REFERÊNCIAS

[1] Tanenbaum, A. S. Computer Networks, 4th Edition. Prentice Hall, 2003. ISBN 0-13-066102-3.

[2] Kurose, J. F. and Ross, K. W. Computer Networking: A Top-Down Approach Featuring the Internet, 2nd Edition. Addison Wesley, 2003. ISBN 0-201-97699-4.

[3] Peikari, C. and Fogie, S. Wireless: Maximum Security. Sams, 2002. ISBN 0-672-32488-1

[4] Rufino, N. M. Segurança em Redes sem Fio. Novatec, 2005. ISBN 85-7522-070-5.

[5] Rufino, N. M. Bluetooth e o Gerenciamento de Riscos. 2005. In: 5a Reunião do Grupo de Trabalho em Segurança (GTS), São Paulo, 2005. Disponível em: ftp://ftp.registro.br/pub/gts/gts0105/05-gts-bluetooth.pdf (acessado em 02/07/2009).

[6] Bluetooth SIG. Specification of the Bluetooth System v. 1.2. 2003. Disponível em: http://www.bluetooth.com (acessado em 02/07/2009).

Conjunto de Instruções Multimídia

Jonathas Campi Costa
Instituto de Computação
Universidade Estadual de Campinas - Unicamp
Campinas, Brasil
RA: 085380
jon.costa@gmail.com

ABSTRACT
Apresenta-se neste artigo uma visão geral dos diferentes conjuntos de instruções multimídia existentes no mercado de processadores. São abordados os principais conceitos da tecnologia por detrás do conjunto de instruções bem como seus principais representantes; além de análises de desempenho e abordagens de implementação.

General Terms
SIMD theory, MMX, SSE, 3DNow!, Altivec.

1. INTRODUÇÃO
Durante os anos 90 houve um grande aumento no uso da computação como suporte as operações multimídia, isto é, o uso do computador na criação de informação multimídia (video, imagem, som, etc.); aliado a esse fato, as *workstations* e os computadores pessoais eram utilizados cada vez mais como instrumentos de cálculos avançados. Analisando essa tendência, os principais fabricantes de processadores utilizaram uma idéia já conhecida para atender a uma nova demanda: o uso de instruções vetoriais.

A implementação de uma arquitetura vetorial completa (a presença de registradores vetoriais em todo os estágios do *pipeline*) sobre uma arquitetura i386, por exemplo, se possível (devido a falta de flexibilidade na execução de códigos de propósito geral) ainda seria altamente complexa e custosa, do ponto de vista operacional, logo a solução encontrada pelos fabricantes de processadores foi a implementação de um subconjunto das operações tipicamente existentes em uma arquitetura puramente vetorial[9], sobre uma arquitetura do tipo SISD [3]. Para tal conjunto de operações foi dado o nome de Conjunto de Instruções Multimídia. É importante notar que existem diferenças significativas entre as instruções multimídia e vetorial [9]; *e.g.* o número de elementos em uma instrução vetorial não está presente no código da operação (*opcode*) como nas instruções multimídia, e sim em um registrador separado.

Analisando mais atentamente esse conjunto de operações multimídia podemos classifica-la, segundo a classificação proposta por Flynn [3], como pertencentes a um hardware do tipo SIMD, isto é, *Single Instruction Multiple Data*; processadores em que uma mesma instrução é aplicada sobre diferentes fluxos de dados, empacotados (o conceito de empacotamento de dados será analisado mais adiante). Essas instruções permitem ao hardware a operação simultânea de diferentes ALUs (*Arithmetic Logic Unit*), ou equivalentemente, a divisão de uma grande ALU em muitas ALUs menores que podem executar paralelamente [9].

A idéia dos projetistas de hardware foi unir o melhor de dois mundos, ou seja, unir o paralelismo existente em nível de instruções das máquinas tipo SISD com o paralelismo no nível dos dados, típico da máquinas SIMD.

O uso de instruções multimídia, referenciado de agora em diante como instruções SIMD também, pode ser visto como uma forma de aproveitamento de situações em que o paralelismo está presente e pode ser utilizado. Como um exemplo do uso de instruções SIMD podemos citar a coerência espacial em aplicações de computação gráfica [4].

Em aplicações de computação gráfica, tipicamente aplicações de *rasterização* e processamento de imagem, a coerência espacial está muito presente, isto é, a probabilidade de que o conjunto de pixels vizinhos a um certo pixel em questão possua atributos diferentes é muito pequena [4]. Logo, quando desejamos aplicar uma instrução sobre a imagem, a mesma instrução será aplicada ao mesmo conjunto de pixels com iguais propriedades, portanto utilizando uma única instrução sobre o mesmo conjunto de dados. Se o conjunto de pixels suportado pela operação em questão for de cardinalidade n, podemos dizer que a instrução SIMD possui n unidades funcionais onde cada unidade opera sobre um pixel a mesma instrução.

Um exemplo mais comum é o uso de instruções SIMD para aritmética de vetores; como um vetor pode ser decomposto por suas coordenadas, pode-se efetuar operações aritméticas como soma, subtração, etc., sobre as diferentes coordenadas dos vetores envolvidos nas operações. Por exemplo, para a soma de dois vetores: $\vec{Z} = \vec{X} + \vec{Y}$ pode ser executada diretamente sobre as coordenadas dos vetores: $z_i = x_i + y_i$, onde cada soma será efetuada por uma unidade funcional distinta mas a partir da mesma instrução, *i.e.*, a instrução de soma.

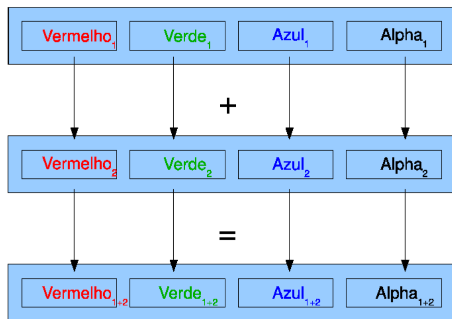


Figure 1: Diagrama representando a soma entre dois pixels diferentes utilizando os registradores vetoriais.

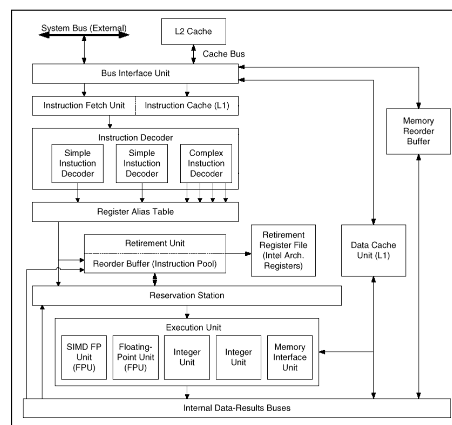


Figure 2: Diagrama do pipeline básico de execução da arquitetura P6. Figura retirada de [1].

Nos dois exemplos citados acima podemos observar claramente a maior vantagem do uso das instruções SIMD: a diminuição da latência no acesso a memória ao ler todos os dados necessários uma única vez e efetuar a mesma operação sobre eles[5].

2. REGISTRADORES VETORIAIS

A base da arquitetura vetorial e das instruções SIMD são os registradores vetoriais. Um registrador vetorial é um registrador em que os dados estão organizados na forma de um vetor, isto é, os dados podem ser comparados aos valores dos escalares que compõem as coordenadas de um vetor. Assim, enquanto que em arquiteturas do tipo SISD, a CPU opera sobre escalares um a um, em uma arquitetura do tipo SIMD a CPU opera sobre uma linha desses escalares, todos do mesmo tipo, executando uma mesma operação sobre todos, como uma unidade.

Esses vetores são representados em um formato de dados chamado: empacotado (*packed data*). Por empacotado podemos entender que os dados são agrupados em diferentes formatos, por exemplo, para um registrador vetorial de 128-bits, podemos empacotar os dados como 4 inteiros de 32-bits cada, ou 8 inteiros de 16-bits cada.

Utilizando dessa abordagem de organização dos dados, é possível efetuar operações sobre os dados de forma eficiente (a latência no acesso aos dados é diminuída, como anteriormente exeplicado). Como abordado anteriormente, a soma de dois pixels pode ser efetuada em uma operação de adição apenas, bastando organizar os elementos do pixel (cores vermelha, verde, azul e o canal de composição) em um registrador vetorial. Podemos observar tal arranjo na Figura 1.

3. ARQUITETURA PARA INSTRUÇÕES MULTIMÍDIA

Em geral, a adição das instruções multimídia é efetuada através da alteração do estágio de execução das arquiteturas escalares [11, 9, 5, 1], incluindo uma unidade especializada para a execução das instruções SIMD. Podemos observar a presença dessas unidades nas arquiteturas dos processadores da família P6 da Intel (Figura 2), na arquitetura do processador Athlon da AMD (Figura 3) e na arquitetura do processador PowerPC 970 (Figura 4), por exemplo.

Uma implementação interessante foi a do primeiro conjunto de instruções multimídia da Intel, o MMX [18]. As instruções MMX foram implementadas sobre a unidade de ponto flutuante já disponível nos primeiros membros da arquitetura P5, isto é, os registradores vetoriais foram implantados sobre os registradores de ponto flutuante logo, os registradores MMX, como veremos mais adiante, que eram implementados com largura de 64-bits para trabalhar com dados em precisão inteira, eram representados internamente como números em ponto flutuante inválidos, já que os registradores de ponto flutuante da arquitetura P5 possuiam largura de 80-bits. Isso era uma forma de diferenciar o conteúdo dos registradores também.

No início, cada fabricante de processadores criou e implementou seu próprio conjunto de instruções SIMD, como por exemplo os conjuntos MAX, VIS, MDMX, etc; enquanto que na arquitetura i386 esse conjunto de instruções acabou por tornar-se um padrão de mercado, o padrão SSE; apesar de atualmente existirem algumas variações, como veremos mais adiante.

Para determinar quais seriam as melhores intruções a implementar, os fabricantes de processadores selecionaram um conjunto de aplicações multimídia que melhor representava o que eles acreditavam ser um conjunto representativo de aplicações multimídia geral [2]. Analisando essas aplicações, criaram, além das instruções básicas de aritmética e instruções de manipulação lógica e de alinhamento, instruções para suportar operações comuns a muitas das aplicações. Essas operações variam em número e complexidade de fabricante para fabricante.

Em geral, podemos dividir o conjunto de instruções SIMD implementadas pelos fabricantes em quatro grandes grupos:

- Instruções aritméticas

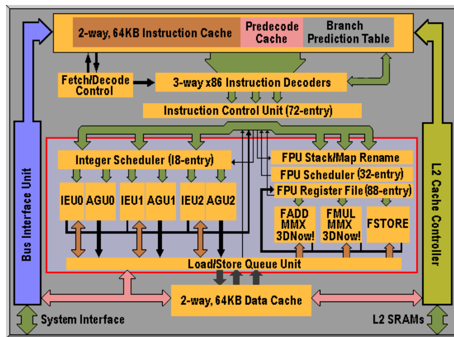


Figure 3: Diagrama da arquitetura do Athlon. Figura retirada de [10].

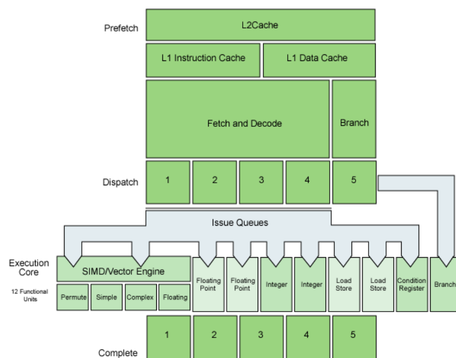


Figure 4: Diagrama da arquitetura do PowerPC 970. Figura retirada de [13].

  - Podemos dividir as instruções aritméticas em dois subgrupos: as de ponto flutuante e as de precisão inteira. Aqui estão incluídas as principais operações aritméticas, *e.g.*: saturação (*clampf*), módulo, soma, subtração, divisão e multiplicação (alguns fabricantes implementam essas duas últimas operações apenas através de *shifts* para esquerda e direita, respectivamente [6]). Em ponto flutuante podemos citar também operações específicas para arredondamento e conversão.

- Instruções lógicas
- Instruções para conversão de dados e reordenação
  - Instruções para organizar os dados em pacotes de 8-, 16-, 32-, 64-, ou 128-bits, e reordenação dos dados dentro de um pacote.
- Instruções de memória.
  - Instruções para acesso, leitura e escrita e, em alguns casos, instruções de armazenamento parcial através do uso de máscaras [6].

A seguir são apresentadas exemplos de implementações de instruções SIMD.

4. PRINCIPAIS REPRESENTANTES

Como previamente abordado, cada fabricante de processadores optou por implementar seu próprio conjunto de instruções multimídia. A seguir apresentamos as principais implementações e algumas de suas características mais importantes.

4.1 MAX-1

MAX é um acrônimo para *Multimedia Acceleration eXtensions*, um conjunto de instruções multimídia desenvolvidas para a arquitetura PA-RISC da Hewlett-Packard. Foi o primeiro conjunto de instruções multimídia disponível para o público em geral, em 1994. Projetadas através da análise de uma aplicação MPEG [6], as instruções mais frequentes foram divididas em simples primitivas e implementadas em hardware.

As instruções MAX operam sobre tipos de dados SIMD de 32-bits, formados por múltiplos inteiros de 16-bits alinhados e armazenados (empacotados) em registradores de propósito geral. Assim, uma mesma instrução pode ser executada sobre 2 inteiros de 16-bits cada. Podemos observar as principais instruções MAX na tabela 1.

4.2 VIS

VIS é um acrônimo para *Virtual Instruction Set*, conjunto de instruções SIMD para os processadores `UltraSPARC I` desenvolvidos pela Sun Microsystems em 1995. Assim como a maioria dos conjuntos de instruções multimídia, foi desenvolvido motivado pela necessidade da melhora de desempenho de aplicações multimídia como MPEG e visualização computacional [7]. Como premissa básica no desenvolvimento foi aplicada que qualquer potencial instrução deveria ser executada em um único ciclo de *clock* ou deveria ser facilmente *pipelined*[14].

Table 1: Principais Instruções MAX

Instrução	Descrição
HADD	Soma paralela com aritmética modular.
HADD,ss	Soma paralela com saturação e sinal.
HADD,us	Soma paralela som saturação sem sinal.
HSUB	Subtração paralela com aritmética modular.
HSUB,ss	Subtração paralela com saturação e sinal.
HSUB,us	Subtração paralela com saturação sem sinal.
HAVE	Média paralela.
HSHLADD	*Shift* paralelo para esquerda e soma saturada com sinal.
HSHRADD	*Shift* paralelo para direita e soma saturada com sinal.

As instruções VIS operam sobre tipos de dados SIMD de 64-bits, formados por dados de precisão inteira (2 inteiros de 32-bits ou 4 inteiros de 16-bits). Para efetuar suas operações, utiliza a já presente unidade de ponto flutuante, maximizando o paralelismo no nível de instruções e maximizando o número de registradores utilizados.

Entre as principais características, podemos citar:

- duas instruções VIS podem ser lidas e decodificadas por ciclo de *clock*,
- as instruções VIS são totalmente *pipelined*,
- as instruções VIS possuem baixa latência.

4.3 VIS 2.0

A extensão das instruções VIS [8]. Implementadas primeiramente nos processadores `UltraSPARC III` veio como uma solução para um problema recorrente do VIS, a reordenação dos dados dentro de um pacote SIMD (armazenamento dos dados no registrador SIMD).

Assim como seu ancestral, possui uma execução totalmente *pipelined*, como exceção das instruções de raíz quadrada e divisão. É composta de um *pipeline* de 12 estágios e 32 registradores SIMD. Efetua operações sobre dados de precisão inteira somente, a versão VIS 3.0 pretende corrigir essa depenência.

4.4 MMX

É o primeiro conjunto de instruções multimídia da Intel. Apareceu pela primeira vez na arquitetura P5, a arquitetura dos primeiros Pentiums, em 1997. Assim como os conjuntos de instruções anteriormente descritos, as instruções MMX foram desenvolvidas a partir do estudo de um conjunto de aplicações multimídia: MPEG-1, MPEG-2, audio, reconhecimento e compressão de voz, jogos 3D, modens, etc. Analisando essas aplicações, um conjunto de características críticas para a execução foram obtidas e então transformadas para instruções MMX [14].

Uma das principais premissas no desenvolvimento das instruções MMX foi a necessidade de não alterar o hardware existente até o momento para que os sistema operacionais da época pudessem executar no novo processador com as novas instruções. Assim, as instruções MMX foram mapeadas na arquitetura e nos registradores de ponto flutuante existente,

Table 2: Principais Instruções MMX

Instrução	Descrição
PADD-	Soma paralela com aritmética modular.
PADDS-	Soma paralela com saturação e sinal.
PADDSU-	Soma paralela som saturação sem sinal.
PSUB-	Subtração paralela com aritmética modular.
PSUBS-	Subtração paralela com saturação e sinal.
PSUBUS-	Subtração paralela com saturação sem sinal.
PACKSSWB	Empacotamento.

o que significa que o uso de instruções MMX e de ponto flutuante não eram permitidas ao mesmo tempo.

As instruções MMX definem então um modelo SIMD simples e flexivel capaz de trabalhar com dados de precisão inteira empacotados em registradores SIMD de 64-bits [18]. É composta por 8 registradores SIMD de 64-bits cada, nomeados de MMX0 até MMX7, com dois tipos de acesso aos dados: mode de acesso de 64-bits e modo de acesso de 32-bits. É capaz de armazenar 8 bytes, ou 4 palavras de 16-bits cada, ou 2 palavras de 32-bits cada.

Seu conjunto de instruções é composto por 47 intruções agrupadas nas seguintes categorias: transferência de dados, aritmética, comparação, conversão, desempacotamento, lógica, *shift* ou deslocamento e *Empty MMX state instructions - EMMS*[1].

Alguns das principais instruções podem ser vistas na tabela 2. Na tabela 2, um instrução do tipo XXX- pode ser interpretada como atuando sobre bytes (B), palavras de 16-bits (W), e palavras duplas de 32-bits (D). Por exemplo, PADDD é uma soma paralela em registradores SIMD de 64-bits contendo dois dados de 32-bits cada.

4.5 3DNow!

A resposta da concorrente AMD as instruções MMX da Intel em 1998. A AMD licenciou o conjunto de instruções e registradores MMX da Intel e adicionou suporte particionado a tipo de dados em ponto flutuante [14]. Como isso, em um processador AMD com suporte a instruções multimídia do tipo 3DNow! é possível efetuar operações em ponto flutuante em paralelo no formato SIMD.

Com a introdução dos processadores Athlon, em 1999, a AMD introduziu 19 novas instruções ao seu conjunto 3DNow!, batizando-o de AMD Enchanced 3DNow!.

4.6 SSE

O conjunto de instruções multimídia SSE da Intel é a evolução natural das instruções MMX [18]. Introduzido em 1999 com o Pentium III sua principal diferença em relação as instruções MMX é a capacidade de trabalhar com dados em ponto flutuante além de uma nova unidade (um estado arquitetural) para execução de operações SSE. Essa nova unidade reduziu a complexidade da implementação bem como

[1] Essa instrução esvazia o estado MMX do processador e deve ser chamada ao final de uma rotina utilizando instruções MMX e antes de iniciar outras rotinas que utilizem instruções de ponto flutuante.

da programação, permitindo aos desenvolvedores utilizar SSE e MMX ou x87 concorrentemente.

As instruções SSE são compostas por 8 registradores SIMD de 128-bits cada, nomeados de XMMX0 até XMMX7, em modo não-64-bits [18], e 16 registradores XMM em modo 64-bits. Capaz de trabalhar com 4 dados em ponto flutuante (*IEEE single precision floating-point*) empacotados.

Um fato interessante é que, apesar da definição dos registradores de 128-bits de largura, presente no SSE, as unidades de execução presentes no Pentium III possuem largura de 64-bits. A unidade de execução SSE do Pentium III traduz a instrução SSE de 128-bits em dois pares de micro-ops de 64-bits cada [14].

4.7 SSE2
É a evolução do conjunto de instruções SSE. Introduzidas inicialmente em 2000, com o processador Pentium IV, ela extendeu o conjunto de instruções em precisão inteira do MMX para registradores de 128-bits, com o auxílio de 68 novas instruções que utilizam os mesmo registradores XMM0-7, do original SSE [14, 18]. Sua principal funcionalidade foi a adição da capacidade de trabalhar com dados em ponto flutuante de precisão dupla, destinados a aplicações científicas, na maior parte do do tempo.

4.8 SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 e SSE4a
O conjunto de instruções SSE3 foi introduzido com o processador Pentium IV com suporte a *Hyper-Threading*, enquanto que as instruções SSE4 foram introduzidas nos processadores de da geração de 45*nm* [18].

A principal diferença entre as instruções SSE2 e SSE3 é a inclusão de novas instruções para aritmética horizontal [18], *i.e.*, artimética entre partes diferentes de registradores SIMD, e não todo o registrador; e instruções que melhoram a sincronização entre agentes multi-thread.

As instruções SSSE3 adicionaram 18 novas instruções para aritmética horizontal, intruções para acelerar o cálculo do produto vetorial, instruções para alinhamento dos dados, e para execução de valores absolutos entre outras [18].

O conjunto de instruções SSE4 é composto por dois subconjuntos, o das instruções SSE4.1 e SSE4.2. Seu desenvolvimento teve como motor impulsionador o desempenho de aplicações em mídia, imagem e 3D. SSE4.1 adicionou instruções para melhorar a vetorização via compilador e melhorou significantemente o suporte a computação de palavras de largura dupla empacotadas [18]. SSE4.2 adicionou instruções para melhorar o desempenho em aplicações com uso de *strings* e *Application-target accelerator (ATA) instructions*[2].

As instruções SSE4a são a implementação de 4 instruções do conjunto SSE4 da Intel mais 2 novas instruções, pela concorrente AMD na arquitetura do K10.

4.9 Altivec
É o conjunto de instruções SIMD para os processadores Power desenvolvidos em conjunto entre a Motorola e a IBM [20]. As instruções Altivec são compostas de 32 registradores de 128-bits que armazenam dados fontes para as unidades Altivec. Esses registradores oferecem suporte a 16 dados paralelos de 8-bits cada em precisão inteira (como ou sem sinal) ou caracteres, ou 8 dados paralelos de 16-bits cada em precisão inteira (com ou sem sinal) ou 4 dados paralelos de 32-bits em precisão inteira e ponto flutuante. Possui um conjunto extenso de instruções, totalizando mais de 150 diferentes instruções.

4.10 Outras extensões
Entre as outras extensões existentes podemos citar: *MIPS V ISA Extension*, *MDMX - Mips Digital Media Extension*, *MIPS-4D ASE - Application Specific Extension* todos para processadores MIPS, *MVI - Motion Video Instructions* para processadores Alpha e *NEON* para processadores ARM.

5. DESEMPENHO
O ganho de desempenho em aplicações utilizando o conjunto de instruções multimídia é, em geral, avalidado através da comparação do algoritmo utilizando as instruções multimídia contra aquele que não as utiliza [15]. O que a literatura mostra [19, 16] é que, em geral, pode-se obter melhoras de 2 a 5 vezes em desempenho. Tal variação no ganho de desempenho médio é fruto da implementação do algoritmo, da otimização no acesso a memória e também nas características do problema a ser paralelizado (vetorizado) [17].

6. CONCLUSÃO
O conjunto de instruções multimídia trouxe aos computadores pessoais e as *workstations* a vantagem dos computadores vetoriais: o paralelismo dos dados. Mesmo que, divididas em vários sabores (implemetações de fabricante para fabricante), um pabrão para a arquitetura i386 foi estabelecido com as instruções SSE. Em geral, possuem um ganho de desempenho da ordem de 2 a 5 vezes mas, deve-se levar em conta que esse ganho de desempenho é fortemente baseado na habilidade dos programadores já que os compiladores atuais são imaturos no sentido de utilizar toda a capacidade SIMD dos atuais processadores do mercado.

É importante notarmos que as primeiras instruções multimídia surgiram na época em que os processadores não dispunham de desempenho o sucifiente para as aplicações multimídia existentes, mas hoje com o avanço das atuais GPUs fica a pergunta: quanto de energia criativa os projetistas de processadores devem inserir no projeto dos processadores atuais com instruções multimídia se as principais aplicações multimídias são hoje aceleradas pelas GPUs? Uma possível resposta para essa pergunta pode ser o uso de inúmeros processadores com conjuntos de instruções multimídia como um único processador de propósito gráfico; por exemplo a arquitetura Larrabee da Intel [12].

7. REFERENCES
[1] J. K. And. Pentium iii processor implementation tradeoffs, 1999.
[2] G. Erickson. Risc for graphics: A survey and analysis of multimedia extended instruction set architectures, 1996.

[2] Acelera o desempenho de softwares na procura de padrões de bits.

[3] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21, 1972.

[4] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, second edition in c edition, 1996.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, fourth edition, 2006.

[6] R. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15:22–32, 1995.

[7] S. Microsystems. *VIS Instructions Set User's Manual*. Sun Microsystems, 1997. Manual.

[8] S. Microsystems. *The VIS Instructions Set*. Sun Microsystems, 2002. A White Paper.

[9] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, fourth edition, 2009.

[10] PCTECHGUIDE. Amd athlon, 2009. [Online; accessed 20-June-2009].

[11] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE Micro*, 20(4):47–57, 2000.

[12] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, August 2008.

[13] C. Shamieh. Understanding 64-bit powerpc architecture: Critical considerations in 64-bit microprocessor design, 2004. [Online; accessed 20-June-2009].

[14] N. Slingerland and A. J. Smith. Multimedia extensions for general purpose microprocessors: a survey. Technical Report UCB/CSD-00-1124, EECS Department, University of California, Berkeley, Dec 2000.

[15] J. Stewart. An investigation of simd instruction sets. Technical report, School of Information Technology and Mathematical Sciences, University of Ballarat, Nov 2005.

[16] C. D. W. F. Stratton. Optimizing for sse: A case study, 2002. [Online; accessed 20-June-2009].

[17] D. Talla, L. K. John, and D. Burger. Bottlenecks in multimedia processing with simd style extensions and architectural enhancements. *IEEE Transactions on Computers*, 52(8):1015–1031, 2003.

[18] I. A. Team. *Intel Architecture Software Developer's Manual*. Intel Corporation, 2007.

[19] H.-M. C. Trung Hieu Tran and S.-B. Cho. Performance enhancement of motion estimation using ss2 technology. In *World Academy of Science, Engineering and Technology*, volume 30, July 2008.

[20] Wikipedia. Altivec — wikipedia, the free encyclopedia, 2009. [Online; accessed 15-June-2009].

Memórias Transacionais

João Batista Correa G. Moreira, RA: 087331
Instituto de Computação
UNICAMP
Campinas, São Paulo
joao@livewire.com.br

ABSTRACT

O surgimento de arquiteturas computacionais multiprocessadas impõe novas dificuldades no desenvolvimento de software. Dentre estas dificuldades está a necessidade de sincronização dos fluxos de execução de forma a garantir acessos ordenados à memória compartilhada. A sincronização tradicionalmente é realizada com o uso de travas de exclusão mútua ou semáforos, porém estas estruturas são ineficientes e difíceis de usar. As memórias transacionais surgem como alternativa para a sincronização de um software paralelo, fornecendo uma abstração simples e garantia de consistência dos dados.

Keywords

Memória Transacional, Arquitetura de Computadores, Programação Paralela

1. INTRODUÇÃO

A evolução dos processadores seguiu por muito tempo um modelo baseado no aumento da freqüência do *clock* para conseguir um maior poder de processamento. Este modelo de evolução, no entanto, esbarrou em limitações físicas que impossibilitam a alimentação e o resfriamento adequado dos processadores que estariam por vir, exigindo novas alternativas para o desenvolvimento de computadores mais velozes. Buscando solucionar este problema, a indústria passou a desenvolver os chamados processadores *multicores*, com dois ou mais núcleos em um único *chip*. Esta solução mostra-se interessante uma vez que possibilita a continuidade da evolução dos processadores pelos próximos anos.

Para que os novos processadores sejam devidamente aproveitados, o software que eles executam precisa fazer uso adequado dos núcleos adicionais. Tradicionalmente um software é desenvolvido para executar seqüencialmente, o que exige a compreensão de novos paradigmas de programação por parte dos desenvolvedores que pretendem escrever aplicações paralelas. Além disto, se comparado a um software seqüencial equivalente, fatores como concorrência e não-determinismo tornam o software paralelo muito mais difícil de se programar. Os impactos gerados pelas arquiteturas paralelas e os problemas envolvidos na sua programação são descritos em [24].

O desenvolvimento de um software paralelo, como é feito hoje, deixa a cargo do programador todo o trabalho de sincronização dos fluxos de execução (*threads*). A sincronização das *threads* garante que o acesso aos dados compartilhados seja realizado de forma correta, garantindo a execução consistente do software. Os mecanismos utilizados pelos programadores para evitar que uma *thread* interfira inadequadamente na execução das demais *threads* concorrentes são travas de exclusão mútua (*locks*) e os semáforos.

Estas estruturas, *locks* e semáforos, controlam a execução de seções críticas nas diferentes *threads* de um programa. As seções críticas de um software dizem respeito a trechos de código que, quando executados em paralelo, podem tentar acessar recursos compartilhados de forma desordenada e levar a execução a estados inconsistentes de memória. Conforme descrito em [13], estes mecanismos não oferecem uma abstração adequada ao uso uma vez que todos os detalhes de implementação relacionados a sincronização são utilizados de forma explícita. Esta característica torna tais mecanismos difíceis de se usar corretamente, levando freqüentemente a problemas comuns em um software paralelo, como enfileiramentos e bloqueios mútuos (*deadlocks*).

Com o objetivo de facilitar o desenvolvimento de um software paralelo, surgiram os sistemas de memória transacional (*Transactional Memory, TM*). Esta proposta de modelo de programação fornece uma abstração semelhante aos sistemas de transações utilizados em bancos de dados, onde propriedades como atomicidade, consistência e isolamento são garantidas. Desta forma, os sistemas de memória transacional encarregam-se da sincronização de um software paralelo, possibilitando a execução de operações corretas de leitura e escrita na memória sem a necessidade do uso de mecanismos como *locks* e semáforos.

Com o uso em larga escala dos processadores *multicore*, o interesse pelas memórias transacionais aumentou bastante, resultando em uma grande quantidade de pesquisas sobre o assunto. Diferentes sistemas de memória transacional tem sido desenvolvidos e testados. Alguns destes sistemas consistem em bibliotecas de software, sendo chamados *Software Trans-*

actional Memories (*STM*). Outros sistemas contam com recursos de hardware para diminuir o impacto e atingir um melhor desempenho, compondo o que é chamado de *Hardware Transactional Memory* (*HTM*). Sistemas de memória transacional que combinam recursos oferecidos pelo *Hardware* com técnicas em *Software* são chamados *Hybrid Transactional Memories* (*HyTM*)

2. TRANSAÇÕES

Segundo [13], uma transação é uma seqüência de operações cuja execução é vista de maneira instantânea e indivisível por um observador externo. O conceito de transação é amplamente utilizado em sistemas de gerenciamento de banco de dados, garantindo quatro propriedades necessárias: Atomicidade, Consistência, Isolamento e Durabilidade.

Considerando o modelo de programação paralela tradicional, pode se dizer que uma transação é equivalente às seções críticas definidas com o uso de *locks* e semáforos, garantindo que uma *thread* não interfira no fluxo de execução das demais, porém evitando serializações desnecessárias e definidas com uma sintaxe mais simples e fácil de compreender.

Quando duas transações tentam acessar os mesmos locais de memória simultaneamente, diz-se que estas transações entraram em conflito e as operações realizadas por uma delas não deve ser aplicada ao sistema. Esta característica é implementada através do uso de mecanismos de execução especulativa ou por mecanismos de *roll-back*. Em uma execução especulativa, as transações são executadas e seus efeitos são armazenados em um *buffer*, só sendo aplicados ao sistema caso a transação seja concluída sem ocorrência de conflitos. Nos mecanismos de *roll-back*, uma transação é executada e os seus efeitos são aplicados diretamente ao sistema, porém o estado de memória anterior ao início da transação é armazenado de forma que o sistema possa recuperá-lo caso um conflito seja detectado. Alguns mecanismos de *roll-back* utilizam *logs* que armazenam apenas as operações executadas na transação em vez de guardar todo o estado de memória anterior a transação.

Por **Atomicidade** entende-se que todas as operações de uma transação executam com sucesso ou nenhuma das operações é executada. Ao final de uma transação, todos os seus efeitos são aplicados ao sistema através de um *commit* efetivo, ou então são descartados através de um *abort*.

Consistência garante que as mudanças geradas por uma transação não levarão o sistema a um estado incorreto. Por estado correto pode-se entender um estado em que todos os dados armazenados em memória são valores certos.

Isolamento garante que cada uma das transações produzirá um resultado correto, independente de quantas e quais transações sejam executadas simultaneamente. Esta propriedade garante que o sistema não apresente erros gerados pelo uso incorreto de memória compartilhada, como pode ocorrer em um software paralelo que não possua um sistema adequado de sincronização

Durabilidade assegura que os resultados gerados por uma transação concluída sejam permanentes e estejam disponíveis para as próximas transações. Esta propriedade não é importante para sistemas de memória transacional, uma vez que os dados em memória são considerados transientes.

Um bloco de código atômico precisa ser delimitado de forma a identificar quais instruções fazem parte da transação especificamente. Um exemplo de bloco de código em que se define uma transação pode ser visualizado na Figura 1:

```
atomic {
        if (flag) x.function();
        y = true;
}
```

Figure 1: Definição de um bloco atômico

3. MEMÓRIA TRANSACIONAL

A idéia geral por trás dos sistemas de memória transacional consiste no uso de estruturas para definição das transações. O sistema de memória transacional se encarrega por executar as transações, detectar possíveis conflitos e gerencia-los, caso existam. Conforme descrito em [13], cada sistema de memória transacional possui um conjunto de características próprias, especificando como este sistema lida com as transações. Os mecanismos utilizados na detecção e tratamento de conflitos, no isolamento de memória, bem como estruturas para definição das transações podem variar bastante em cada implementação.

Algumas das classificações das principais características de um sistema de memória transacional são:

- Detecção de conflitos
  - *Eager* : O sistema detecta conflitos existentes entre transações durante sua execução, isto é, antes da realização dos *commits*.
  - *Lazy* : O sistema detecta os conflitos quando uma transação encerra sua execução e tenta realizar o *commit*.
- Isolamento
  - Fraco: Um acesso a memória executado fora de uma transação não entra em conflito com acessos a memória executados dentro de uma transação, podendo eventualmente corromper a execução do programa. Este modelo induz todos os acessos a memória compartilhada a serem feitos dentro de uma transação.
  - Forte: Converte todas as operações fora de uma transação em pequenas transações individuais, executando todos os acessos a memória compartilhada como se fossem transações. Neste modelo, referências de memória externas a uma transação entram em conflito com referências de memória executadas dentro de uma transação.
- Gerenciamento de versões
  - *Eager*: Mantém o estado de memória anterior a transação em um *buffer* ou *log*, aplicando os valores computados na transação imediatamente. Esta estratégia beneficia operações de *commit*, porém torna operações de *abort* mais complicadas e lentas.

– *Lazy*: Mantém o estado de memória anterior até o momento do *commit* efetivo. Este modelo beneficia operações de *abort*, porém aumenta o custo das operações de *commit*.

A Figura 2 ilustra um fluxograma da execução de uma transação em um sistema com detecção de conflitos *Lazy*.

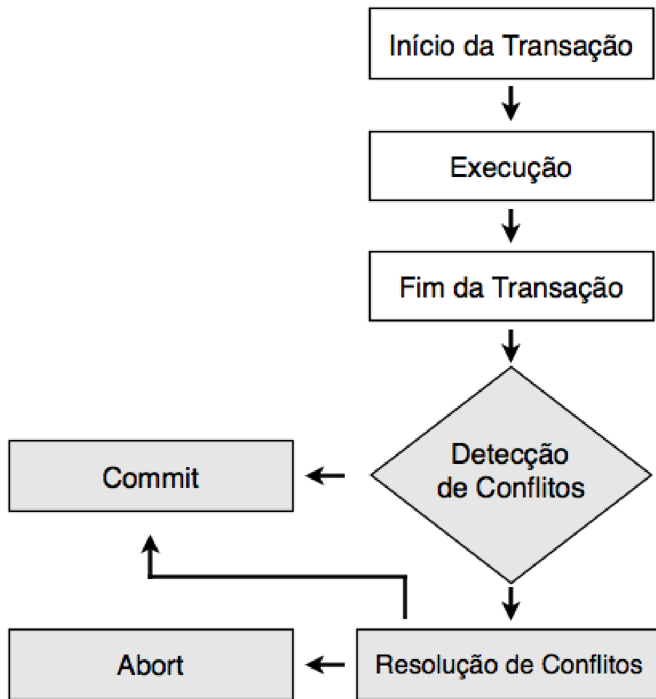


Figure 2: Fluxo de uma transação em um Sistema TM com detecção de conflitos *Lazy*

Outra importante característica de um sistema de memória transacional é o gerenciamento de contenção. O gerenciador de contenção é a parte do sistema responsável por lidar com os conflitos, aplicando diferentes políticas para decidir qual das transações conflitantes será abortada e qual delas efetuará o *commit*. Estas políticas, chamadas políticas de resolução de conflitos, consistem em regras que auxiliam na decisão a respeito de *aborts* e *commits* sobre transações conflitantes, de forma que esta escolha minimize o custo decorrente do conflito. Algumas políticas de gerenciamento de contenção são [4]:

- *Committer Wins*: Quando o conflito é detectado na fase de *commit* de uma transação, esta transação sempre é efetivada e as demais transações conflitantes são abortadas.
- *Requester Wins*: Quando uma transação realiza acesso a um endereço de memória compartilhado que esteja em uso por outra transação, esta continua executando e as demais transações conflitantes são abortadas.
- *Requester Stalls*: O sistema bloqueia transações que tentam realizar acesso a um endereço de memória compartilhado em uso por outra transação. Caso um potencial *deadlock* seja detectado, o sistema aborta esta transação.

Em um sistema de memória transacional, espera-se que o gerenciador de contenção seja capaz de garantir o fluxo contínuo do programa. Em [13] afirma-se que a política de gerenciamento de contenção possa ser especificada pelo programador para cada uma das transações ou ainda ser escolhida automaticamente pelo sistema de memória transacional.

A escolha ideal do sistema de memória transacional e, consequentemente, das suas características, depende diretamente da aplicação em que será utilizado. Um exemplo são os sistemas de memória transacional com detecção de conflitos *Eager*. Estes sistemas evitam que as transações executem de maneira desnecessária, detectando os conflitos de maneira antecipada. Por outro lado, estes sistemas impõem maior complexidade ao modelo de memória transacional utilizado, exigindo mais recursos computacionais. Normalmente a escolha do sistema de detecção de conflitos é feita com base nos *throughputs* observados em execuções com cada um deles.

3.1 Software Transactional Memory

Um STM implementa transações não duráveis para a manipulação de dados compartilhados entre *threads*. A maioria dos STMs pode ser executada em um processador convencional, porém pouco se conhece a respeito do custo computacional adicional introduzido ao se utilizar tal recurso. As principais vantagens relacionadas ao uso de STMs são:

- Software é mais flexível que o hardware, permitindo variadas implementações de sofisticados algoritmos
- É mais fácil modificar e desenvolver software que hardware
- STMs são mais fáceis de se integrar com os sistemas de software existentes
- STMs possuem poucas limitações intrínsecas impostas por capacidades físicas do hardware, como *caches*

Quando um software é desenvolvido com o uso de uma STM, as seções críticas do seu código são demarcadas, identificando-as como transações. Parte do sistema é responsável por realizar a detecção de conflitos entre transações. Quando conflitos são detectados o gerenciador de contenção seleciona quais transações deverão ser abortadas e quais transações serão concluídas. Ao utilizar um sistema de memória transacional, o desenvolvedor só se preocupa com definir quais as seções críticas do seu código. Todo o trabalho adicional é realizado pelo sistema. Este modelo genérico de memória transacional, em que se baseiam a maioria das implementações, é descrito em [13].

O termo *Software Transactional Memory* surgiu em [21], publicado em 1995, onde se descreveu a primeira implementação de STM. Este sistema exigia que uma transação declarasse todos os acessos a memória que seriam realizados em seu início, solicitando automaticamente a posse dos *locks*

referentes aos dados. Este mecanismo introduziu um custo computacional significativo ao software.

O sistema de memória transacional RSTM é descrito em [20]. Este sistema foi feito para programas implementados na linguagem C++ e que utilizam *pthreads* no seu processo de paralelização. Este sistema implementa o conceito de objetos transacionais, ou seja, se um objeto é transacional ele pode executar transações com a garantia de que, se as mesmas são abortadas, os valores alterados durante sua execução serão restaurados. Quando duas ou mais transações entram em conflito, um sistema de gerenciamento de contenção seleciona uma para ser concluída, enquanto força as demais a aguardar ou reexecutar. O sistema de gerenciamento de contenção do sistema RSTM é baseado no algoritmo Polka, que privilegia transações que já executaram uma maior quantidade de instruções.

Em [7], um sistema de memória transacional chamado TL2 é descrito. Este sistema baseia-se na idéia de manter um *clock* global de versões de cada uma das variáveis que são acessadas no escopo de uma transação. Sempre que uma leitura ou escrita é realizada, um algoritmo de verificação de versão é seguido, garantindo a consistência dos dados compartilhados. O *clock* global funciona como um *lock* que é adquirido e incrementado no final de uma execução especulativa de uma transação. As transações que só executam leituras são bastante facilitadas neste sistema, uma vez que só é necessário verificar se o *clock* global não foi acessado entre o início e o fim da transação.

Em [8] descreve-se um sistema de memória transacional que também utiliza *locks*. Este sistema, chamado TinySTM, utiliza um algoritmo similar ao utilizado no sistema TL2, descrito em [7], exceto por adquirir os *locks* sempre que uma variável é acessada. A execução de *benchmarks* demonstrou que esta característica proporcionou maior *throughput* em relação aos testes realizados com o sistema TL2. Em [8] também é demonstrada uma técnica utilizada para configuração dos parâmetros do sistema de memória transacional, onde um algoritmo de subida da colina é utilizado a partir de vários valores aleatórios.

3.2 Hardware Transactional Memory

Sistemas de memória transacional em Hardware consistem em sistemas que suportam a execução de transações em dados compartilhados, respeitando as propriedades Atomicidade, Consistência e Isolamento. O principal objetivo destes sistemas é atingir melhor eficiencia com menor custo. As principais vantagens relacionadas ao uso de HTMs são:

- HTMs conseguem executar aplicações com menor custo adicional que STMs.
- HTMs apresentam menor consumo de energia.
- HTMs existem de maneira independente aos compiladores e as características de acesso a memória.
- HTMs fornecem alta eficiência e isolamento forte sem a necessidade de grandes modificações nas aplicações

A execução comum de um programa baseia-se num modelo tradicional em que um contador de programa aponta para uma instrução, esta instrução é lida pelo processador, decodificada, executada e, quando concluida, o contador de programa é incrementado apontando para uma nova instrução que passará pelo mesmo ciclo. Em processadores que suportam execuções paralelas, cada uma das threads executando no processador executa a sequência descrita.

Apesar de manter o modelo de execução sequencial, os processadores executam diversas tarefas em paralelo. O processador aplica ao código sequencial vários mecanismos de controle de fluxo e previsão de dependências para poder executar suas instruções sequenciais em paralelo, fora da ordem prevista, e ainda assim produzir resultados corretos. Para cumprir esta tarefa, é necessário que o processador mantenha informações que possibilitem a correção do estado de execução no caso de uma predição incorreta.

HTMs baseiam-se em mecanismos que possibilitam a execução otimista (conflito não existe, a menos que seja detectado) de uma seqüência de instruções, detecção de eventuais conflitos no acesso aos dados e uso de *caches* para armazenar modificações temporárias que só serão visíveis após o *commit*. HTMs exploram vários mecanismos de hardware, como execução especulativa, *checkpoints* de registradores, *caches* e coerência de *cache*.

Segundo [13], os sistemas de memória transacional em hardware podem ser divididos em Abordagens percursoras, *Unbounded HTMs*, *Bounded/Large HTMs* e *Hybrid TMs/ Hardware Accelerated TMs*.

3.2.1 Abordagens percursoras

As abordagens percursoras consistem em trabalhos voltados para o suporte eficiente a paralelismo utilizando técnicas e implementações em hardware. Estas abordagens forneceram grande inspiração para posteriores desenvolvimentos de HTMs, servindo como base e fundamentação para consequentes pesquisas.

Em 1988 Chang e Mergen publicaram dois artigos [6, 17] descrevendo o *IBM 801 Storage Manager System*, em que suporte a *locks* para transações em bancos de dados seriam suportadas pelo hardware. Este é o primeiro caso conhecido de implementação de suporte a *locks* e transações em hardware. Caso um acesso a memória não encontrasse os respectivos *locks* em um determinado estado, uma função em software seria automaticamente invocada para realizar o gerenciamento do *lock* e garantir o sucesso da transação. Esta proposta incluía novos registradores ao processador para manter informações sobre as transações.

Knight descreve em [11] um sistema de paralelização especulativa de programas com apenas um fluxo de execução. O compilador seria responsável por dividir o programa em blocos de código chamados transações, assumindo que não exista nenhuma dependência de dados entre eles. Estas transações seriam executadas especulativamente e conflitos poderiam ser detectados através das *caches*.

O *Oklahoma Update protocol*, proposto por Stone et. all, é descrito em [23]. Este artigo propõe implementações em hardware para operações atômicas do tipo *read-modify-write* em um número limitado de locais de memória. Esta abor-

dagem é uma alternativa as seções críticas, cujo objetivo era simplificar a construção de blocos de código concorrentes e não bloqueantes, possibilitando a atualização automática de multiplas variáveis compartilhadas. Esta proposta incluía novas instruções que utilizariam novos registradores, chamados *Reservation Registers*, para apontar endereços e armazenar atualizações. Atualizações realizadas nestes endereços poderiam ser monitoradas através dos protocolos de coerência da *cache*.

Uma proposta similar ao *Oklahoma Update protocol* foi feita por Herlihy e Moss em [10]. Esta proposta incluía a adição de novas instruções e *cache* transacional para monitorar e armazenar dados transacionais. Este artigo é responsável pelo termo *Memória Transacional* e por iniciar o uso de mecanismos de *cache* para realizar sincronização otimista de dados compartilhados, ao contrário do *Oklahoma Update protocol*, que utilizava registradores no lugar de *caches* transacionais.

Ravi Rajwar e James R. Goodwin propuseram em [18] um sistema onde os *locks* de um software seriam reconhecidos e removidos pelo hardware, eliminando possíveis serializações. Este sistema executaria as transações de forma especulativa, isto é, todas as modificações geradas pela transação seriam escritas em um *buffer* local até que esta chegue ao fim. Quando a transação encerra, caso nenhum conflito seja detectado, as operações realizadas no *buffer* são aplicadas nos verdadeiros endereços de memória do sistema, tornando as modificações visíveis para os demais processos que compartilham o hardware. Executar as transações de maneira especulativa é uma estratégia utilizada em vários sistemas de memória transacional até hoje, inclusive em STMs, como pode ser observado em [20, 7, 8].

3.2.2 Bounded/Large HTMs

Alguns sistemas de HTM permitem que uma transação utilize informações além das fornecidas pela cache de dados. No entanto estes sistemas não permitem que uma thread seja migrada para outro processador durante sua execução ou sobrevivam a mudanças de contexto. Estes sistemas são chamados *Bounded/Larges HTMS*.

Transactional Coherence and Consistency (TCC) é o modelo de memória compartilhada descrito em [9]. Neste modelo, todas as operações são executadas dentro de transações. As transações são divididas pelo programador ou pelo compilador e podem possuir dependencias de dados. O hardware TCC executa estas transações em paralelo e, quando uma alteração é realizada na memória, esta é informada através de *broadcasts* aos demais processadores. Neste modelo, uma transação é executada especulativamente sem a realização de qualquer tipo de solicitação de *lock* na cache. As modificações são armazenadas em um *buffer* local. Múltiplas transações podem escrever no mesmo local de memória armazenado na *cache* local de maneira concorrente, porém, quando uma transação estiver pronta para realizar o *commit*, seu processador solicita permissão. Esta permissão define qual transação realizará o *commit*, sendo que apenas uma transação poderá realizá-lo por vez. Uma vez realizado o *commit*, esta transação envia um *broadcast* informando os outros processadores quais escritas foram feitas na memória. Estes processadores comparam o conjunto de escritas recebido no *broadcast* com seus próprios conjuntos de escrita, se houver um conflito, estas transações são abortadas e reexecutadas. Se uma transação excede as informações das *caches* locais, ela entra em modo não especulativo, solicitando o *lock* global para evitar que outras transações realizem *commits*.

Large Transactional Memory é uma implementação de memória transacional descrita em [1]. Esta implementação permite que linhas da *cache* transacional sejam alocadas em uma região reservada de memória local, sem que a transação seja abortada. Esta característica permite que uma transação acesse conjuntos de dados maiores que o espaço disponível na *cache*.

Em [15] descreve-se o sistema LogTM, cujos dados transacionais não se restringem a utilizar a *cache* local, mas também são transportados para os demais níveis da hierarquia de memória. Esta implementação utiliza um sistema de *logs* em software para armazenar os valores de locais de memória atualizados em uma transação, de forma que estes valores possam ser recuperados por uma rotina em software se a transação for abortada. Esta abordagem favoreceu transações que realizam o *commit* com sucesso.

Em [5] descreve-se um sistema HTM que não utiliza *caches* e protocolos de coerência de *cache* para detectar os conflitos entre as transações. Uma nova implementação, chamada *Bulk* utiliza os endereços modificados na geração de uma assinatura que em seguida é enviada através de um *broadcast* para os demais processadores no momento do *commit*.

3.2.3 Unbounded HTMs

Os chamados *Unbounded HTMs* são sistemas de memória transacional que permitem que as transações sobrevivam a mudanças de contexto, podendo continuar sua execução ainda que esta tenha sido interrompida. Para atingir este objetivo, é necessário armazenar os estados das transações em um espaço de memória persistente, como o espaço de memória virtual.

Unbounded Transactional Memory (UTM) é o sistema descrito em [1]. Este sistema é capaz de sobreviver a mudanças de contexto e exceder limites de *buffer* em hardware. O UTM propõe modificações na arquitetura, ampliando-a de forma a desacoplar os estados de manutenção das transações e detecções de conflito das *caches*. Para manter o estado transacional fora do hardware, este sistema introduz uma estrutura de dados residente em memória chamada XSTATE, que armazena informações (como conjuntos de escrita e leituras) de todas as transações no sistema. Para manter a verificação de conflitos independente da coerência de *cache*, este sistema mantém bits de acesso com informações sobre processos em cada um dos blocos de memória. Através destas informações, que poderiam ser acessadas por uma transação, seria possível detectar conflitos.

Em [19] descreve-se o sistema TM *Virtual Transactional Memory (VTM)*. Neste sistema não seria necessário que programadores implementassem detalhes a respeito do hardware, através da virtualização dos recursos limitados, como *buffers* de hardware e períodos de escalonamento. Esta abordagem também possibilita que as transações sobrevivam a trocas de contexto e excedam os limites de memória impostos pelo hardware. Este mecanismo de virtualização é

semelhante a forma como memória virtual torna transparente o gerenciamento de memória física limitada para os programadores. O sistema VTM desacopla os estados de manutenção das transações e detecções de conflitos do hardware através do uso de estruturas de dados residentes na memória virtual da própria aplicação. Estas estruturas armazenam informações sobre transações que excederam os limites da *cache* ou ainda que excederam o tempo de escalonamento.

Em [26] uma implementação alternativa para o sistema VTM é proposta. Nesta implementação são descritos extensões em software e no conjunto de instruções para permitir que as transações VTM aguardem determinados eventos e reiniciar sua execução eficientemente através de comunicação com o escalonador, que é orientado a pausa-las temporariamente, porém compensa-las em seguida. Esta implementação adiciona novas instruções, novo suporte a interrupções de software e amplia as estruturas de metadados da implementação VTM original.

3.2.4 Hybrid/Hardware Accelerated TMs

Sistemas de memória transacional que se baseiam em STMs como um ponto de partida, porém utilizam mecanismos de HTM para manter a eficiencia são chamados Híbridos ou acelerados por hardware. Esta abordagem garante a flexibilidade, mantendo as transações desacopladas do hardware através do uso de implementações em software (STMs) aplicados a mecanismos limitados de hardware (HTMs).

O sistema *Lie* é proposto em [14] e descreve um sistema de memória transacional híbrido de hardware e software, em que as transações são executadas como transações de hardware porém, se estas transações não puderem ser executadas desta forma, elas são reiniciadas e executadas como uma transação em software. Neste sistema, duas versões do código das transações é gerado, um para transações em hardware e outro para transações em software. O código gerado para transações em hardware executa ainda algumas verificações em software, permitindo que estas transações detectem conflitos com transações que estejam executando em software.

Outro sistema de memória transacional em que as transações são inicialmente executadas como transações de hardware porém, caso não possam ser executadas como tal, são reiniciadas e executadas como uma transação de software, é descrito em [12]. Através desta abordagem, é possível manter a eficiência de uma transação em hardware sempre que possível, porém recorrer a transações de software se o hardware for insuficiente.

O sistema *Rochester Transactional Memory (RTM)*, descrito em [22], utiliza mecanismos HTM apenas para acelerar uma STM. Esta abordagem apresenta um novo conjunto de instruções através do qual é possível que uma STM controle cada um dos mecanismos da HTM, executando tarefas específicas da STM diretamente em hardware. Este sistema permite, por exemplo, que um software controle quais linhas de cache devem ser transacionalmente monitoradas e mantidas expostas aos protocolos de coerência de cache.

3.3 Análise de consumo de energia em Sistemas de Memória Transacional

A eficiência de uma memória transacional pode ser medida de diferentes formas. A principal delas é chamada *throughput*, e diz respeito a quantidade de transações bem-sucedidas em um intervalo definido de tempo. Outra medida importante é o *speedup*, que diz respeito ao quão mais veloz um programa executou em proporção ao seu tempo de execução anterior. Estudos mais recentes apresentam a medida do consumo de energia como um parâmetro adicional, levando em consideração se estes sistemas aumentam o diminuem o consumo médio de uma aplicação.

Em [3] a plataforma MPARM foi utilizada como simulador para análise de consumo de energia em MPSoCs (*Multiprocessor System on Chips*). Neste trabalho, modelos de consumo de energia foram incorporados a cada um dos componentes de hardware, possibilitando que a plataforma observe os valores de consumo de energia em cada um dos ciclos simulados.

Em [16], um estudo a respeito dos efeitos gerados pelo uso de memórias transacionais em relação ao consumo de energia é apresentado. O sistema proposto é semelhante ao sistema apresentado em [25], sendo capaz de identificar momentos de alta contenção e serializar a execução das transações para diminuir a quantidade de transações mal-sucedidas. Nos resultados apresentados, pode-se perceber que as memórias transacionais reduzem significativamente o consumo de energia, sendo que no sistema que utiliza serialização de transações em momentos de alta contenção este consumo foi ainda menor.

Em [2], uma abordagem a respeito de metodologias de medição do consumo de energia são expostas, demonstrando técnicas para diferenciar o consumo de energia médio da aplicação e o consumo introduzido pelo uso do sistema de memória transacional.

4. CONCLUSÃO

Como se pode observar, o uso de sistemas de memória transacional se mostra uma alternativa interessante para a programação de um software paralelo. Através desta metodologia é possível manter uma abstração de alto nível, evitando que os programadores sejam obrigados a lidar e entender detalhes intrínsecos ao hardware. Além de facilitar a programação através da introdução de uma abstração mais amigável, estes sistemas também possibilitam maior aproveitamento do recursos disponíveis, uma vez que diferentes *threads* não se bloqueiam a menos que um conflito seja realmente detectado, característica que não existe em implementações com *locks* e semáforos.

Por ser uma idéia relativamente nova, pode-se observar que existem diversas implementações que tiram proveito de características de hardware, software ou ambas. A grande quantidade de pesquisas, implementações e publicações a respeito do tema denotam ainda a importância deste novo campo, expondo o fato de que ainda não se encontrou um modelo ideal ou padrão para implementações de sistemas de memória transacional.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th Int. Symp. on High-Performance Computer Architecture*, pages 316–327, February 2005.

[2] A. Baldassin, F. Klein, P. Centoducatte, G. Araujo, and R. Azevedo. A first study on characterizing the energy consumption of software transactional memory. *Relatórios Técnicos IC*, April 2009.

[3] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41(2):169–182, 2005.

[4] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. *IEEE Micro*, 28(1):32–41, 2008.

[5] R. P. Case and A. Padegs. Architecture of the ibm system/370. *Commun. ACM*, 21(1):73–96, 1978.

[6] A. Chang and M. Mergen. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst.*, 6(1):28–50, February 1998.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. 2006.

[8] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[9] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News*, 32(2):102, 2004.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[11] T. F. Knight. An architecture for mostly functional languages. *ACM Lisp and Functional Programming Conference*, pages 105–112, August 1986.

[12] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 209–220. ACM Press, 2006.

[13] J. R. Larus and R. Rajwar. *Transactional Memory.* Morgan & Claypool Publishers, 2007.

[14] S. Lie. Hardware support for transactional memory. Master's thesis, Massachusetts Institute of Technology, 2004.

[15] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *12th Int. Symp. on High-Performance Computer Architecture*, pages 254–265, February 2006.

[16] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy-aware microprocessor synchronization: Transactional memory vs. locks. In *4th Workshop on Memory Performance Issues (held in conjunction with the 12th International Symposium on High-Performance Computer Architecture)*, 2006.

[17] G. Radin. The 801 minicomputer. In *1st Int. Symp on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, 1982.

[18] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[19] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[20] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 107–113, 2007.

[21] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995.

[22] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer, and M. F. Spear. Hardware acceleration of software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[23] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, 1993.

[24] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[25] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 169–178, New York, NY, USA, 2008. ACM.

[26] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. *1st ACM Sigplan Workshop on Languages, Compilers and Hardware Support for Transactional Computing*, 2006.

Earth Simulator: uma visão geral

Tiago J. Carvalho – RA:087343
Instituto de Computação
Universidade Estadual de Campinas
Campinas, São Paulo, Brasil
tiagojc@gmail.com

RESUMO

Esse trabalho traz uma visão do que é o "Earth Simulator" uma proposta de máquina com tecnologia vetorial que tem o objetivo de possibilitar a previsão de mudanças globais no ambiente. Dono de um sistema computacional com poder único, tem atraido pesquisadores do mundo inteiro. Dono de uma performance de execução de 35.86TFLOPS é considerado o computador mais rápido do mundo. Possui um hardware baseado em um sistema paralelo de memória distribuída e tem seu sistema operacional baseado no UNIX. Sempre utilizado simulações do meio ambiente, tem produzidos resultados únicos para simulações nas áreas de circulação oceânica global e reprodução de campos de ondas sísmicas, dentre outras.

Palavras chave

Earth Simulator, supercomputador, NEC, processamento vetorial

1. INTRODUÇÃO

O "Earth Simulator" é uma proposta de máquina especial feita pelo NEC com o mesmo tipo de tecnologia vetorial que a disponível no SX-6. O NEC é uma das maiores produtoras de computadores e eletrônicos no mundo. Ele é o segundo maior produtor de semicondutores (a Intel é a primeira) [1]. Ele também produz PCs e notebooks, controlando metade do marketing no japão. A decisão do NEC de se basear completamente em processadores vetoriais é uma nova tendência seguida no design de super computadores [2]. O "Earth Simulator" foi terminado em fevereiro de 2002 depois de cinco anos de desenvolvimento tendo a mais recente arquitetura na linha de computadores vetoriais, linha essa que começou com o Cray 1.

O objetivo do "Earth Simulator System (ESS)" é possibilitar a previsão de mudanças globais no ambiente devido ao seu enorme poder computacional, poder esse requerido em tais tipos de pesquisas [6]. O assunto de mudanças ambientais tem se tornado mais e mais importante com o passar das décadas, já que a poluição tem se tornado um sério problema que ameaça o clima do planeta em que vivemos. E o clima é exatamente onde o ESS vai dar suporte em pesquisas.

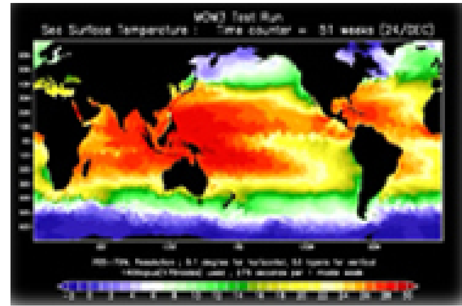


Figura 1: Tela do ESS

A possibilidade do uso de um sistema computacional com poder único tem atraído pesquisadores do mundo inteiro e estimulado o interesse na realização de projetos conjuntos na área das ciências e engenharia. Um número de pesquisas cooperativas entre o Japão e países europeus bem como os E.U.A. estão em andamento, resultando em grandes avanços na ciência e abrindo caminho para novos aspectos no ramo das simulações científicas.

2. DETALHES TÉCNICOS

Do ponto de vista da ciência da computação, os seguintes pontos tem que ser notados. Primeiro, o ESS demonstra a possibilidade e significancia de uma grande arquitetura paralela para computadores vetoriais. Segundo, o ESS tem empurrado o estado-da-arte nas linguagens de programação e facilita a questão do desenvolvimento de aplicações por suportar linguagens de alto-nível. Em terceiro lugar, a performance sustentável do "Earth Simulator" é 1000 vezes maior que o supercomputador mais usado em 1996 no campo de pesquisas climáticas. Sua performance de execução de 35.86TFLOPS foi aprovada pelo Linpack benchmark e o ESS foi registrado como o super computador mais rápido do mundo em 20 de junho de 2002 [5].

3. ARQUITETURA DO HARDWARE

O "Earth Simulator" é um sistema paralelo de memória distribuída que consiste de 640 nodos processadores (PNs) conectado por um barramento compartilhado de 640 x 640

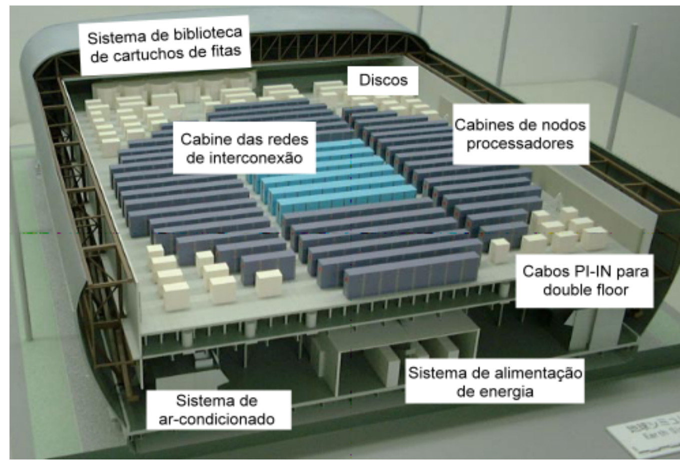


Figura 2: Um modelo do ESS. O local que abriga o ESS tem 50m x 65m x 17m e tem dois andares, além de incluir um sistema de isolação sísmica.

nodos internos [5]. Cada nodo é um sistema de memória compartilhado composto de de oito processadores aritméticos (APs), um sistema de memória compartilhada de 16GB, uma unidade de controle remoto (RCU), e um controlador de entrada e saída. O máximo de performance de cada AP é 8GFLOPS. O número total de processadores é 5120 e o máximo total de performance e capacidade total de memória são 40TFLOPS e 10TB, respectivamente. As demais especificações do hardware são mostradas na tabela 1.

Tabela 1: *A especificação de Hardware do Earth Simulator*

Número total de nodos processadores	640
Número de AP para cada nodo	8
Número Total de AP	5120
Máximo de performance para cada AP	8GFLOPS
Máximo de performance para cada PN	64GFLOPS
Máximo de performance do sistema total	40TFLOPS
Memória compartilhada de cada PN	16GB
Memória principal total	10TB

3.1 Processador

Cada AP contém uma unidade vetorial (VU), uma unidade de operações super-escalares (SU), e uma memória principal acessada pela unidade de controle, os quais são montados em um chip LSI operando numa freqüência de clock de 500MHz, parcialmente 1GHz.

A SU é um processador 4-way super escalar com uma cache de instruções de 64KB, uma cache de dados de 64KB, e 128 registradores escalares de propósito geral. A SU utiliza a previsão de "branch", "prefetching" de dados e execução de instruções "out-of-order".

A VU possui uma pipeline vetorial que pode manipular seis tipos de operações (add/shift, multiplicação, divisão, lógica, máscara e load/store), além de um haver um total de 8 registradores vetoriais e 64 registradores vetoriais de dados, cada um dos quais tem 256 elementos vetoriais.

Quando a adição e a multiplicação são realizadas concorrentemente, o máximo de performance será alcançado. Aqui o máximo de performance teórico pode ser computado por vários operadores padrões. Cada AP carrega 16 operações de ponto flutuante com um ciclo de máquina de 2 nseg pela utilização completa de um conjunto de pipelines - permite que ambos adição e multiplicação rode 8 operações. Isso indica 8 Gflops como máximo de performance, mas um vetor de comprimento suficientemente longo e uma alta velocidade na taxa transferência de dados entre o processador e a unidade de memória é essencial para garantir essa performance.

Dado que um elemento do vetor tem 8 bytes então, 8Gflops de performance requerem 64GB/s de taxa de transferência de dados, enquanto a taxa de transferência de dados com a unidade de memória é 32 GB/s para um processador. Isso implica que uma aplicação tem que ser computada intensivamente enquanto a freqüência do acesso à memória tem que ser mantida baixa para uma computação eficiente. A performance teórica máxima "F" pode ser derivada do número de instruções dentro de um laço "DO" com base na seguinte fórmula assumindo um vetor de comprimento suficientemente longo.

$$F = \frac{4\,(ADD + MUL))}{max\,(ADD, MUL, VLD + VST)} \quad (1)$$

onde ADD, MUL, VLD, e VST são o número de adições, multiplicações e operações vetoriais de "load" e "store" respectivamente. A tabela 3 sumariza como a performance sustentável é computada para diferentes operações padrão. Aqui o índice do laço "DO" mais interno é denotado como "n" e o do laço externo como "j". Note que o acesso aos vetores C e D é tomado como "load" escalar em termos de "n", e não é incluído na contagem de VLD.

No.	DO loop	VLD	VST	ADD	MUL	Gflops	Taxa Máxima
1	X(n) = X(n) + A(n)	2	1	1	0	4/3=1.33	17 %
2	X(n) = X(n) + A(n)*B(n)	3	1	1	1	8/4=2.00	25 %
3	X(n) = X(n) + P(n, j)*C(n)	2	1	1	1	8/3=2.67	33 %
4	X(n) = X(n) + P(n, j)*C(j) + P(n, j+1)*C(j+1) + P(n, j+2)*C(j+2) + P(n, j+3)*C(j+3) Y(n) = Y(n) + P(n, j)*D(j) + P(n, j+1)*D(j+1) + P(n, j+2)*D(j+2) + P(n, j+3)*D(j+3)	6	2	8	8	64/8=8.00	100 %

Figura 3: Estimação de performance de vetores

3.2 O Sistema de Memória

O sistema de memória (MS) em cada PN é simetricamente compartilhado pelos 8 APs. Isso é configurado por 32 unidades de memória principal (MMU) como 2048 bancos. Cada AP em um PN pode acessar 32 portas de memória quando um vetor de instruções "load/store" é carregado [8]. Cada AP tem uma taxa de transferência de dados de 32 GB/s com os dispositivos de memória, o que resulta em um "throughput" agregado de 256 GB/s por PN. Assim sendo, uma série de acessos a endereços de memória contínuos resultará na performance máxima, enquanto isso será reduzido particularmente com o acesso a um número constante de intervalos de memória porque o número de portas de memória disponível para memória para acesso concorrente é diminuído. Logo, o acesso à memória em um dos 32 endereços com um grande espaçamento para outro endereço irá resultar em uma diminuição da eficiência de 1 GB/sec. O RCU em cada PN é diretamente conectado a um barramento compartilhado de dois modos, enviando e recebendo, e controlando a comunicação de dados entre os nodos. A capacidade total de memória é de 10 TB

3.3 Rede de Interconexão

A rede de interconexão (IN) consiste de duas partes: uma é a unidade de controle de barramento entre nodos (XCT) que coordena as operações de switch; o outro é um barramento entre nodos de "switch"(XSW) que funciona como um caminho real para dados. XSW é composto de 128 switches separados, cada um dos quais tem uma banda de 1Gbits/s operações independentemente. Qualquer par desses switches e nodos são conectados por cabos elétricos. A taxa de transferência de dados teóricos entre os pares de PNs é 12.3GB/s. Um modelo de programação para trocas de mensagens com uma biblioteca de Interface de Passagem de Mensagens (MPI) é suportado pelos PNs e dentro deles. A performance da função Put da MPI foi medida. O throughput máximo e a latência da função Put da MPI são 11.63GB/s e 6.63 μseg, respectivamente. Também deveria ser notado que o tempo para sincronização de barreiras é apenas 3.3 μseg por causa do sistema de hardware dedicado para sincronização de barreiras dentro dos PNs.

4. SISTEMA OPERACIONAL E LINGUAGENS

O sistema operacional que roda em um nodo processador é um sistema baseado no UNIX para suportar computações científicas de larga escala. Compiladores de Fortran90, Fortran de Alta Performance (HPF), C e C++ são disponibilizados com suporte automático para vetorização e paralelismo. Uma biblioteca de transmissão de mensagens baseada em MPI-1 e MPI-2 também está disponível.

Em especial, é focado o HPF/ES por ser desenvolvido especialmente para o ES. Ele é um compilador HPF desenvolvido para o Earth Simulator pelo melhoramento em muitos aspectos do HPF/SX V2 feito pelo NEC. Possui algumas extensões únicas bem como algumas características do HPF 2.0, suas extensões aprovadas e HPF/JA para suportar uma programação paralela eficiente e portátil. As extensões únicas incluem diretivas de vetorização, otimização de características de comunicação irregular, entrada e saída paralela, e assim por diante. HPF/ES detecta comunicações do tipo SHIFT automaticamente e gera chamadas para rotinas em tempo de execução altamente otimizadas do tipo SHIFT de comunicação. No mais, a geração de "schedule" de comunicação é re-usada quando o mesmo padrão de comunicação é iterado na execução de laços. O ESS tem ainda uma hierarquia de três níveis de paralelismo que serão apresentadas nas próximas sub-seções:

4.1 Paralelismo de Nível 1

O primeiro nível de processamento paralelo é o processamento vetorial em um AP individual. Esse é o mais fundamental nível de processamento no ES. Vetorização automática é aplicada por compiladores para programas escritos em linguagens convencionais como Fortran 90 e C [9].

4.2 Pararelismo de Nível 2

O segundo nível é o de processamento paralelo com memória compartilhada processado em um PN individual. Programação paralela com memória compartilhada é suportada por "microtasking" e "OpenMP". A capacidade do "microtasking" é similar em estilo àquelas fornecidas por um supercomputador Cray, e as mesmas funções são realizadas pelo ES. "Microtasking" é aplicado de duas formas: uma (AMT) é a paralelização automática pelos compiladores e a outra

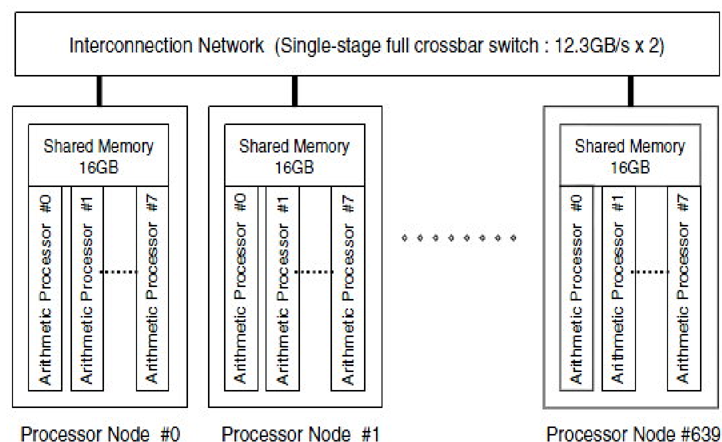


Figura 4: Esquema simplificado do ES

(MMT) é a inserção manual de diretrizes de "microtasking" antes do alvo dos loops.

4.3 Pararelismo de Nível 3

O terceiro nível é o processamento paralelo de memória distribuída que é compartilhado dentro das PNs. Um modelo de programação paralela com memória distribuída é suportado pela MPI. A performance desse sistema para funções de entrada (Put) da MPI foi medida com funções especificadas para o MPI-2.

5. APLICAÇÕES

Para aplicações práticas com performance média sustentável de 30% do máximo, o "Earth Simulator" tem produzido resultados únicos para simulações como "Kuroshio", "Gulf Stream" e "Agulhas Ring" na circulação oceânica global; furacões e início de raios na circulação atmosférica global; reprodução de campos de ondas sísmicas de toda terra na física de terra sólida; material de super-diamante de nano tubos de carbono em materiais científicos [7].

A figura 5 mostra os parâmetros de performance sustentável dos projetos rodando no Earth Simulator em 2002. A maior das performances foi condecorada com o prêmio "Gordon Bell" em 2002.

5.1 Simulação de terremotos

Para modelar a propagação de ondas sísmicas resultantes de grandes terremotos no Earth Simulator foram utilizados 1944 processadores segundo a implementação descrita em [3]. Esse tipo de simulação é baseado sobre o método de elementos espectrais, uma técnica com alto grau de elementos finitos com uma grande quantidade de matrizes diagonal exatas. É usada uma grande rede com 5.5 bilhões de pontos de rede (14.6 bilhões de graus de liberdade). Para isso é utilizada a complexidade total da Terra, isto é, uma velocidade de onda tridimensional e estrutura densa, um modelo de crosta 3D eliptico, bem como a topografia. Um total de 2.5 terabytes de memória é necessário. Tal implementação é puramente baseada sobre o MPI, com a vetorização de loops em cada processador. Obteve uma excelente taxa de vetorização de 99.3%, e foi alcançada uma performance de 5 teraflops (30% da performance máxima) em 38% da máquina. A resolução mais alta da malha permite a geração completa de calculos tridimensionais em períodos sísmicos com menos de 5 segundos.

5.2 Simulação de fluidos tridimensional para fusão científica

Dentre vários experimentos testados no "Earth Simulator", um código para simulação de plasma (IMPACT-3D) paralelizado com HPF, rodando em 512 nodos do "Earth Simulator" alcançou uma performance de 14.9 TFLOPS [5]. A performance máxima teórica dos 512 nodos foi de 32 TFLOPS, o qual representa 45% do máximo de performance que foi obtido com HPF. IMPACT-3D é um código de análise de implosões utilizando o esquema TVD, o qual apresenta uma compressão tridimensional e uma computação de fluido "Euleriano" com o esquema de cópia explícito de 5 pontos para a diferenciação espacial e o passo de tempo fracional para a integração de tempo.O tamanho da malha é 2048x2048x4096, e a terceira dimensão foi distribuída para a paralelização. O sistema HPF utilizado na avaliação é HPF/ES, desenvolvido para o "Earth Simulator" através do melhoramento do NEC HPF/SX V2 principalmente na escalabilidade de comunicação. A comunicação foi manualmente ajustada para dar a melhor performance utilizando extensões HPF/JA, os quais foram projetados para dar aos usuários um maior controle sobre paralelismo sofisticado e comunicações otimizadas.

5.3 Simulação de Turbulência

Outro tipo de simulação executada no "Earth Simulator" foram as simulações numéricas diretas de alta resolução (DNSs) de turbulência sem compressão, com número de pontos de "grid" superior a 4096^3 [9].As DNSs são baseadas nos métodos do espectro de Fourier, tal que as equações para conservação de massa são precisamente resolvidas. Nas DNSs baseadas no método espectral, a maior parte do tempo computacional é consumido calculando a transformada rápida de Fourier (FFT) em três dimensões (3D), a qual requer uma escala gigantesca de transferência de dados e tem sido o maior impecilho para uma computação de alta performance. Implementando novos métodos para possibilitar a 3D-FFT no ESS, a DNS alcançou 16.4 Tflops em 2048^3 pontos de "grid".

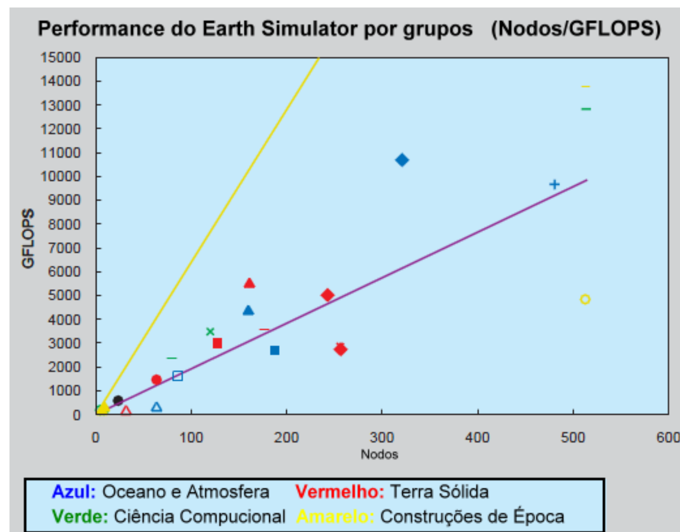


Figura 5: Perfomance teórica e sustentável do Earth Simulator. A linha amarela indica o máximo teórico e a linha rosa indica 30% da performance sustentável. As três maiores foram condecoradas com o prêmio Gordon Bell em 2002.

A DNS ainda produz um espectro de energia exibido em um completo subdomínio inerte, em contraste com as DNSs anteriores com baixa resolução e então fornece dados valiosos para o estudo de características universais de turbulência nos maiores números de Reynold.

5.4 Simulação da Atmosfera Global

Um modelo geral de circulação espectral atmosférico chamado AFES (AGCM para o "Earth Simulator") foi desenvolvido e otimizado para a arquitetura do Earth Simulator (ES) [8]. A performance sustentável de 26.58 Tflops foi conseguida para uma simulação de alta resolução (T1279L96) com AFES utilizando uma configuração com todos os 640 nodos do ES. A eficiência de computação resultante é de 64.9% da performance máxima, bem maior que das aplicações convencionais de estado atmosférico (clima) que têm em torno de 25-50% de eficiência para computadores vetoriais paralelos. Essa excelente performance prova que a efetividade do ES é um significantemente viável para aplicações práticas.

6. TRABALHOS FUTUROS

No dia 12 de maio de 2008 a corporação NEC anunciou que está sendo contratada para construir o "Novo Earth Simulator", um sistema de computador ultra veloz para a Agência Japonesa para Ciência e Tecnologia de Terra e Mar (JAMSTEC) [4]. O novo ESS será uma atualização do Earth Simulator já existente e irá introduzir novas características para possibilitar uma precisão melhor e análise de alta velocidade, além de projeções em escala global dos fenômenos ambientais. O sistema também será usado para produzir simulações numéricas para campos de pesquisa avançados que vão além do escopo de outros sistemas computacionais.

O novo sistema principalmente consiste de um sistema de supercomputador principal, unidades de sub-sistema e um sistema de gerenciamento de operações, e é projetado para alcançar uma performance máxima de 131TFLOPS (TFLOPS: um trilhão de operações de ponto flutuante por segundo). A performance de aplicações eficazes deverá ser aumentada para duas vezes aquela conseguida com o Earth Simulator existente. O novo sistema está previsto para ser instalado no Instituto para Ciências da Terra (JAMSTEC) de Yokohama.

7. CONCLUSÕES

O Earth Simulator tem contribuído de maneira extremamente importante para pesquisas relacionadas a fenômenos ambientais tais como aquecimento global, poluição atmosférica e marinha, El Niño, chuvas torrenciais dentre outros, uma vez que ele é capaz de executar tais simulações com alta velocidade desde que os mesmos sejam especialmente preparados para rodar no ES. Tais fatos são de grande importância para o desenvolvimento de atividades econômicas e sociais, para ajudar a resolver problemas ambientais além de melhorar o entendimento (de estudiosos da área) de fenômenos terrestres tais como placas tectônicas e terremotos. Ele possui um hardware dedicado e muito eficiente, que conta principalmente com computação vetorial para melhorar ainda mais o tempo e confiabilidade das simulações nele executadas. Assim sendo, é um dos supercomputadores de maior importância (se não o mais) nos dias atuais.

8. REFERÊNCIAS

[1] Nec. Internet: http://www.webopedia.com/TERM/N/NEC.html (acessado em 21-05-2009), 2001.

[2] M. Y. chief designer of NEC. Earth simulator system. Internet: http://www.thocp.net/hardware/nec_ess.htm (acessado em 19-05-2009), 2002.

[3] D. Komatitsch, S. Tsuboi, C. Ji, and J. Tromp. A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the earth simulator. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 4, Washington, DC, USA, 2003. IEEE Computer Society.

[4] NEC. Nec awarded contract for new earth simulator system. Internet:

http://www.nec.co.jp/press/en/0805/1601.html (acessado em 16-06-2009), 2008.

[5] H. Sakagami, H. Murai, Y. Seo, and M. Yokokawa. 14.9 tflops three-dimensional fluid simulation for fusion science with hpf on the earth simulator. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[6] T. Sato. The earth simulator: roles and impacts. *Parallel Comput.*, 30(12):1279–1286, 2004.

[7] T. Satu. The current status of the earth simulator. Internet: http://www.jamstec.go.jp/esc/publication/journal/jes_vol.1/pdf/JES1-2.pdf (acessado em 16-06-2009).

[8] S. Shingu, H. Takahara, H. Fuchigami, M. Yamada, Y. Tsuda, W. Ohfuchi, Y. Sasaki, K. Kobayashi, T. Hagiwara, S.-i. Habata, M. Yokokawa, H. Itoh, and K. Otsuka. A 26.58 tflops global atmospheric simulation with the spectral transform method on the earth simulator. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–19, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[9] M. Yokokawa, K. Itakura, A. Uno, T. Ishihara, and Y. Kaneda. 16.4-tflops direct numerical simulation of turbulence by a fourier spectral method on the earth simulator. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

Computação de Alto Desempenho usando Clusters

César Castelo Fernández*
Instituto de Computação, Universidade Estadual de Campinas, Unicamp
Av. Albert Einstein 1251, Cidade Universitaria, CEP 13083-970
Campinas, SP, Brasil
ccastelo@liv.ic.unicamp.br
RA: 089028

RESUMO
Neste artigo são apresentados todos os conceitos relativos aos Clusters, sendo que eles são configurações que oferecem um desempenho muito bom e são muito baratos. O cluster é formado por muitos computadores que são interligados para funcionar como um computador único, para o qual são necessários muitos componentes que garantem o correto funcionamento do cluster. No artigo são descritos todos esses componentes e ao final é apresentado um exemplo muito importante que descreve o uso da arquitetura de cluster no mundo real, através de uma aplicação usada no mundo inteiro.

Palavras chave
Clusters, Computação Alto Desempenho, Cluster do Google, Multi-processamento, Redes de Computadores, Software Paralelo, Sistemas Operacionais

1. INTRODUÇÃO
Um cluster é um conjunto de computadores interligados, instalados e programados de tal forma que os seus usuários tenham a impressão de estar usando um recurso computacional único [5]. São usados para fazer alguma tarefa específica onde é necessário muito processamento, o que sería inviável fazê-lo com um computador só.

Uma das principais motivações para a construção dos clusters é o alto custo de um computador com arquitetura multiprocessador. Atualmente, o custo de implementar um cluster de computadores é muito mais baixo do que comprar um computador com arquitetura multiprocessador com o mesmo poder de processamento, já que os clusters podem ser construídos com estações de trabalho muito baratas, ou até com computadores pessoais. Outro motivo importante é o uso cada vez maior do processamento paralelo em atividades acadêmicas, científicas e empresariais. As universidades ou centros de pesquisa montam os clusters para fazer as provas nas pesquisas desenvolvidas por eles. Os clusters são usados em muitas áreas da ciência, como física, química, mecânica, computação, matematica, medicina, biologia, etc, incluindo problemas como predição numérica do clima, simulação de semicondutores, oceanografia, modelagem em astrofísica, sequenciamento do genoma humano, análise de elementos finitos, aerodinânica computacional, inteligencia artificial, reconhecimento de padrões, processamento de imagens, visão computacional, computação gráfica, robótica, sistemas espertos, exploração sísmica, análise de imagens médicas, mecânica quântica, etc.

Atualmente, os clusters são muito usados no mundo inteiro. Pelo fato de serem muito fáceis de implementar, são usados tanto por empresas muito grandes, quanto por empresas menores, estudantes ou até pessoas nas suas casas. Existem muitas implementações de clusters que são muito conhecidas porque realmente tem um desempenho ótimo, como o cluster do Google, da Sun, do Oracle, etc, e o fato de serem usados por empresas tão importantes como essas ajudou também a torná-los mais populares.

Na construção de um cluster é importante levar em conta aspectos muito importantes incluíndo Hardware e Software, porque são essenciais para conseguir o melhor desempenho possível para uma dada arquitetura. No Hardware destacam-se elementos como as interfaces de comunicação entre computadores, o tipo de processador usado em cada computador, a quantidade de memória, etc; e no software é muito importante a escolha do Sistema Operacional, software para a sincronização, compiladores especiais; assim como também é fundamental o desenvolvimento de aplicações paralelas que possam aproveitar ao máximo o poder de processamento paralelo do cluster.

O trabalho está organizado da seguinte maneira: na secção 2 são apresentados os principais componentes de um cluster, considerando o software e o hardware; na seção 3 são descritas as vantagens de implementar um cluster; na seção 4 são descritas as opções para implementar um cluster. A seguir, é aprensentado um exemplo real sobre implementação de um cluster na seção 5; e na seção 6 são apresentadas as conclusões do trabalho.

2. ELEMENTOS DE UM CLUSTER
Os clusters tem a vantagem de dar para o usuário muita liberdade na hora de escolher os componentes que formarão o cluster, sendo que ele pode escolher um cluster fabricado

*B. Eng César Castelo Fernández

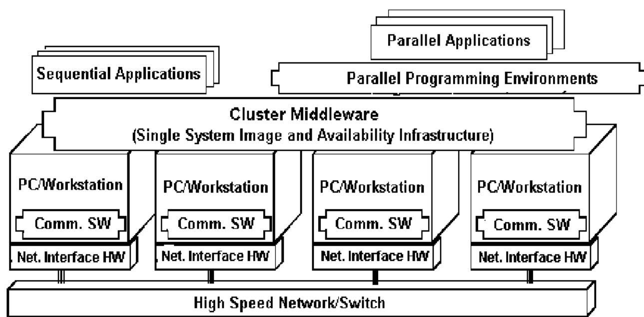


Figura 1: Arquitetura de um cluster.

completamente por um fabricante só, ou pode escolher as peças de diferentes marcas segundo as suas preferências.

Em geral, um cluster é composto por quatro elementos principais [9]: os computadores que são os nós, as redes de interligação (hardware), o conjunto de ferramentas para fazer programas a serem executados no cluster e o conjunto de programas para a administração do cluster (software). Esses quatro elementos são compostos de outros elementos, que serão estudados nas próximas seções [4]. Na Figura 1 pode-se observar a relação entre esses componentes.

2.1 Hardware

2.1.1 Computadores

O computador contém todos os dispositivos de hardware necessários para a execução dos programas, para o qual tem que armazenar a informação e usar interfaces para mostrar os resultados. Os principais componentes do computador são:

- Microprocessador: é o elemento que fornece o poder de processamento ao computador. É muito importante escolher um com bom desempenho, destacando também a quantidade de memória cache que tem. Os mais usados são as familias: Intel x86, Compaq Alpha systems, IBM Power PC Chip e SUN SPARC, porém, é possível implementar um cluster quase com qualquer microprocessador.
- Memória principal: aqui é onde está armazenada a informação usada pelos programas que estão em execução. A tecnologia mais usada atualmente é a DRAM e os fabricantes de memórias sempre estão aumentando a capacidade e velocidade de acesso. Elas são mais lentas do que a memória cache, mas tem muito mais capacidade.
- Motherboard: é o chip onde estão conectados todos os componentes do computador para formar um sistema único. Tem interfaces para a comunicação entre os componentes e cada vez estão sendo desenvolvidas interfaces mais rápidas e compatíveis com todo tipo de componentes, como o USB, PCI, etc.
- Armazenamento secundário: é onde a informação fica armazenada de maneira permanente. Sempre é muito maior do que a memória RAM, mas também é muito mais lenta. Este tipo de armazenamento não é fundamental para o cluster, porque está mais focado no processamento; até alguns nós no cluster tem apenas o espaço necessário para executar o sistema operacional, reduzindo os custos.

2.1.2 Multi-Processadores Simétricos

Além do paralelismo ser implementado distribuindo o processamento entre os diferentes computadores, também é importante usar em cada computador processadores paralelos, aumentando assim a capacidade de processamento total do sistema. Um multi-processador simétrico é formado por um conjunto de processadores e tem como característica principal o fato de todos os processadores terem acesso à mesma memória, ou seja, compartilhar a informação na memória. A sua principal vantagem é que todos os processadores tem a possibilidade de usar a memória completa, mas também é necessário que tenham mecanismos de escolha no hardware para o acesso à memória.

Atualmente este tipo de processadores não tem um custo muito elevado e, dependendo do processamento a ser feito pelo cluster, podem ser usados em todos os computadores do cluster, ou apenas em um servidor que é parte do cluster.

Também pode-se considerar um tipo de processadores muito parecidos, que são os Microprocessadores Múltiplos, que estão implementados como um único chip contendo vários processadores. Estes tem como vantagem que a comunicação entre eles é mais rápida porque estão no mesmo chip, além de poderem compartilhar a memória cache do chip, que é a memória mais rápida do computador. Atualmente este tipo de processadores é cada vez mais acessível e o seu poder de processamento é maior, assim como o número de processadores inclusos no mesmo chip.

2.1.3 Redes

Uma rede é uma combinação de meios físicos de transporte e mecanismos de controle para o transporte. A informação é enviada de um computador para outro usando mensagens, que podem ser muito pequenas ou muito grandes. Uma mensagem é um conjunto de bits que estão organizados segundo um formato, o qual permite que a mensagem seja corretamente interpretada. Quando a mensagem é enviada, é usada uma interface que faz a comunicação entre a aplicação e a rede. Essa interface aumenta na mensagem algumas informações que são usadas para encontrar o destino correto da mensagem quando é enviada pela rede.

Os parâmetros utilizados para medir o desempenho de uma rede são:

- Taxa de transferência: que é medida como a quantidade de informação que pode ser transmitida por unidade de tempo.
- Latência: que é o tempo total necessário para transmitir uma mensagem, a qual vai depender da taxa de transferência.

Existem muitas tecnologias para fazer a interligação entre os computadores na rede, as mais importantes são:

- Ethernet: é a tecnologia mais popular e barata, destacando-se dois: Fast Ethernet, que permite uma velocidade de 10 Mbps até 100 Mbps e ainda é a tecnologia mais usada nos clusters; e Gigabit Ethernet, que permite velocidades na ordem dos Gigabits por segundo e que são usadas somente quando é necessário uma alta taxa de transferência em aplicações como processamento de imagens de alta qualidade, consultas em tempo real, aplicações distribuídas, etc, sendo que também tem compatibilidade com as redes Fast Ethernet.

- Myrinet: é uma rede que usa interruptores de baixa latência e atinge velocidades de 640 Mbps até 2.4 Gbps. Uma de suas grandes limitações é que os interruptores que são usados para desenhar o caminho a ser percorrido pelos pacotes podem ser bloqueados, causando assim um desempenho mais baixo, mas essa dificudade tende a ser superada com a alta velocidade da rede.

- cLAN: é um conjunto de interruptores para clusters de alto desempenho, que oferece um desempenho muito bom na taxa de transferência e na latência, conseguindo velocidades de até 2.5 Gbps por cada um dos 13 portos disponíveis. O problema é que não tem uma especificação dos sinais que são enviados, nem das interfaces de comunicação, o que faz com que não sejam muito usadas.

- Scalable Coherent Interface: é uma interface desenvolvida pela IEEE para conectar sistemas de memória compartilhada com caches coerentes. É muito pouco usada porque as motherbords atuais não tem suporte para os mecanismos de coerência que são necessários para os SCI.

- Infiniband: Mistura conceitos usados na interface cLAN, com um conjunto de especificações detalhadas para conseguir produtos mais compatíveis. É uma interface que está sempre em desenvolvimento para atingir velocidades cada vez maiores.

2.2 Software

2.2.1 Sistema Operacional

É o software básico para a administração dos recursos do cluster, é fundamental na correta operação do cluster. Existem várias tarefas que tem que ser feitas pelo sistema operacional:

- Consulta: os trabalhos iniciados por diferentes usuários em diferentes lugares e com requisitos diferentes tem que ser armazenados pelo sistema operacional em uma fila de prioridades até que o sistema esteja na possibilidade de fazer os trabalhos para cada usuário.

- Planificação: Talvez seja o componente mais complexo do sistema, pois tem que administrar as prioridades aos trabalhos iniciados pelos usuários, levando em conta os recursos do sistema e as políticas estabelecidas pelo administrador do sistema. O planejador deve fazer um equilíbrio entre as aplicações que vão ser executadas, já que há aplicações que devem usar todos os nós do cluster. Embora outras apenas usem alguns nós, há outras que tem que ter respostas e visualizações em tempo real; algumas precisam ter a maior prioridade para terminar sua execução antes do que as outras.

- Controle de recursos: As aplicações deste tipo colocam as aplicações nos nós atribuidos pelo planificador, disponibilizam os arquivos necessários, começam, terminam e suspendem trabalhos. Notificam ao planificador quando os recursos estão disponíveis.

- Monitoração: Para um adequado controle do desempenho do cluster é necessário que sempre seja reportado o estado geral do cluster a algum lugar de controle centralizado, ou a algum dos nós do cluster. Os relatórios tem que incluir disponibilidade de recursos, estado das tarefas em cada nó e quão "saludáveis" estão os nós. Quando esta informação está disponível são executadas certas ações já programadas.

- Contabilidade: Em todo cluster sempre é necessário armazenar informações quantitativas sobre o funcionamento do sistema, seja para calcular os custos de operação para depois serem faturados aos clientes, ou simplesmente para medir o desempenho do sistema, através do análise da utilização do sistema, disponibilidade, resposta efetiva às exigências.

Tradicionalmente, os sistemas operacionais mais usados para implementar clusters são Linux, Windows e Solaris. A seguir serão explicadas as suas características mais importantes tomando como exemplo uma distribução de cada um deles [4]:

- Linux Redhat 7.2 (2.4.x kernel) [8]: É uma das distribuções mais populares do Linux, e atualmente a IBM investe muito dinheiro no desenvolvimento do sistema operacional, sendo que existe uma versão livre e outra que é vendida pela IBM, chamada Redhat High Availability Server, que é muito mais poderosa e foi desenvolvida para trabalhar com clusters. As principais vantagens do Redhat são: o código é livre para modificar, suporte para multi-tasking, suporte para muitos tipos de sistemas de arquivos.

  Oferece um sistema especial de arquivos chamado Parallel Virtual File System, que foi desenvolvido especialmente para oferecer alto desempenho em execuções em paralelo. A versão de Rredhat para clusters desenvolvida pela IBM é uma opção muito boa para fazer um cluster, sendo que oferece um custo baixo em comparação com outros productos no mercado, mas é muito bom para trabalhar com Servidores Web, Servidores FTP, firewalls, VPN gateways, etc. As características principais oferecidas são: alto desempenho e escalabilidade, flexibilidade, maior segurança, disponibilidade, baixo custo, suporte.

- Microsoft Windows 2000 [8]: É o sistema operacional que domina o mercado dos computadores pessoais atualmente e é baseado na arquitetura Windows NT, que é implementada para 32 bits, e é estável, suporta multi-tasking e multi-thread, tem suporte para diferentes arquiteturas de CPU (Intel x86, DEC Alpha, MIPS, etc), tem um modelo de segurança baseado em objetos, tem o sistema de arquivos NTFS e protocolos

de rede TCP-IP, IPX-SPX, etc. Um grande problema do Windows é o custo, porque para construir o cluster é necessário comprar licenças para cada nó.

Para o trabalho com cluster é usado o Windows 2000 Cluster Server (chamado de Wolfpack), que é formado pelos seguintes componentes: Database Manager (contém informação dos nós do cluster, tipos de recursos, grupos de recursos), Node Manager (controla o correto funcionamento de cada nó, fazendo testes entre diferentes nós para ativar ou desativar o nó quando alguma comunicação entre eles não pode ser completada), Event Processor (é o centro de comunicações do cluster e é responsável por comunicar aplicações e componentes do cluster quando é solicitado), Communication Manager (é responsável pela comunicação entre o Cluster Service de um nó com algum Cluster Service de outro nó) e Global Update Manager (é responsável por mudar o estado dos recursos do cluster e enviar notificações quando ocorrem as mudanças).

- SUN Solaris [1]: É um sistema operacional baseado em UNIX, que é multi-thread e multi-user. Suporta as arquiteturas Intel x86 e SPARC, possuindo suporte aos protocolos de rede TCP-IP e Remote Procedure Calls (RPC). Também tem recursos já integrados no kernel para suporte a Cluster.

  Para trabalhar com cluster existe um sistema operacional distribuído que é baseado no Solaris, que é o SUN Cluster, sendo que o sistema inteiro é oferecido ao usuário como uma imagem única, ou seja como se fosse um sistema operacional único e não um cluster. A arquitetura do SUN Cluster é formada pelos seguintes componentes: Object and Communication Support (é usado o modelo de objetos CORBA para definir os objetos e o mecanismo para RPC), Process Management (administra as operações dos processos tal que sua localização é transparente ao usuário), Networking (o sistema implementa sistema de rede para dar suporte à implementação do cluster) e Global Distributed File System (é implementado um sistema global de arquivos, para simplificar as operações de arquivos e a administração de processos).

2.2.2 *Middleware*

É o software que é aumentado no Sistema Operacional para conseguir que todos os computadores no cluster funcionem como um único sistema (chamado de Single System Image). Também tem uma capa de software que permite acesso uniforme aos nós, mesmo tendo diferentes sistemas operacionais. Este software é o responsável por prover alta disponibilidade do sistema. Destacam-se dois software Middleware:

- Bibliotecas para Comunicações Paralelas

  A única maneira de fazer a comunicação entre computadores no cluster é enviando mensagens entre eles, e é por isso que o maior objetivo do Middleware é conseguir portabilidade entre computadores que usam diferentes sistemas operacionais (i.e. conseguir uma interpretação correta das mensagens). Existem duas bibliotecas que são as mais populares: PVM (que foi criada para redes de workstations) e MPI (que é um padrão suportado por IBM, HP, SUN, etc e tem mais funcionalidade do que o PVM).

  PVM (Parallel Virtual Machine) [2] é um framework para o desenvolvimento de aplicações paralelas de maneira fácil e eficiente, administrando com transparência o envio de mensagens entre os computadores do cluster. O PVM é um modelo simples que oferece um conjunto de interfaces de software que podem ser facilmente implementadas para fazer tarefas como executar e finalizar tarefas na rede e sincronizar e comunicar estas tarefas. As tarefas que estão sendo executadas podem executar ou finalizar outras tarefas. Apresenta também características que o fazem tolerante a falhas.

  MPI (Message Passing Interface) [7] é um sistema padronizado e portável de troca de mensagens, que foi projetado para trabalhar com muitos tipos de arquitecturas de computadores e oferencendo interfaces para programação em Fortran, C e C++. Pode-se se dizer que é um padrão melhor do que o PVM e existem algumas razões: Possui muitas implementações livres de boa qualidade, oferece comunicação assíncrona completa, administra eficientemente buffers de mensagens, suporta sincronização com usuários de software proprietário, é altamente portável, é especificado formalmente, é um padrão. Existe uma nova versão do MPI chamada de MPI2, que incrementa suporte para entrada e saída paralelas, operações remotas de memória, administração dinâmica de processos e suporta threads.

- Bibliotecas para Desenvolvimento de Aplicações

  Um dos principais problemas para o desenvolvimento dos clusters é a dificuldade de construção de software paralelo, mas felizmente cada vez é mais comun seu desenvolvimento, mesmo para computadores que não sejam paralelos, pois igual representam uma melhoria no desempenho. Como as tecnologias de hardware estão constantemente mudando, as aplicações paralelas desenvolvidas devem ter a capacidade de executar-se em diferentes arquiteturas e por isso é recomendado desenvolver aplicações trabalhando com a camada baixa do middleware.

  Os modelos e linguagens de programação paralela devem ter algumas características para serem considerados adequados [6]: Facilidade de programação, Ter uma metodologia de desenvolvimento de software, independência de arquitetura, facilidade de entendimento, capacidade de ser eficientemente implementados, oferecer informação precisa sobre o custo dos programas.

  Um problema que está sempre presente nas aplicações paralelas é que algumas vezes são muito abstratas para conseguir maior eficiência, ou ao contrario, não são muito eficientes para ser menos abstratas; por isso podem ser classificadas segundo esses criterios [6]: nada explícito e paralelismo implícito; paralelismo explícito e descomposição implícita de tarefas; descomposição explícita de tarefas e mapeamento implícito de memória; mapeamento explícito de memória e comunicação implícita; comunicação explícita e sincronização implícita; e tudo explícito.

  Dois pacotes para desenvolvimento de software paralelo que são muito conhecidos são BSP e ARCH, os

quais são muito usados atualmente.

2.2.3 *Software de Redes [9]*

Os elementos mais importantes no software de redes são os seguintes:

- TCP-IP: A diferença dos supercomputadores, os clusters não tem protocolos de comunicações proprietários, senão que usam protocolos padrão como o TCP-IP, que é o padrão de fato nos sistemas de cluster. O protocolo IP é dividido conceitualmente em diferentes capas lógicas que tem como objetivo dividir a mensagem que vai ser transmitida em pacotes individuais de dados, chamados de Datagramas, que tem o seu tamanho limitado pela longitude do meio de transmisão. Depois de serem transmitidas individualmente, as mensagens são reconstruidas no computador de destino, o qual é indicado em cada pacote mediante o endereço IP do host de destino. Atualmente existem as versões IPv4 e IPv6 que tem 4 bytes e 16 bytes para representar o endereço do host de destino. Os serviços suportados pelo protocolo IP são TCP (Transmission Control Protocol) e UDP (User Datagram Protocol).

- Sockets: Os sockets são as interfaces de baixo nível entre as aplicações de usuário e o sistema operacional e o hardware no computador. Eles oferecem um mecanismo adequado para a comunicação entre as aplicações, tendo suporte para diferentes formatos de endereços, semântica e protocolos. Foram propostos no sistema operacional BSD 4.2 e agora existe uma API no Linux que oferece suporte aos sockets. Alguns programadores não gostam de trabalhar com sockets diretamente, mas com outras capas que fazem uma abstração dos sockets. A idéia básica nos sockets é a implementação de um arquivo que permita fazer operações read e write de um computador a outro como se fosse um arquivo que está no mesmo computador. Pode-se usar muitos tipos de implementações de sockets, mas na prática os mais usados são Connectionless Datagram Sockets (que são baseados no protocolo UDP) e outro tipo baseado no protocolo TCP, que é melhor do que o baseado em UDP porque oferece uma comunicação de ida e volta entre os elementos ligados; os sockets para o protocolo UDP tem o problema que algumas vezes as mensagens não são enviadas corretamente. Os sockets deixam para o conhecimento do programador como é o funcionamento dos mecanismos de troca de mensagens, pelo qual algumas vezes é comparado com a linguagem Assembler.

- Protocolos de Alto Nível: Os sockets não são muito usados pelos programadores que desenvolvem aplicações para os clusters, mas são usados pelos programadores de sistemas operacionais, sendo que os programadores de aplicações tem preferência por usar protocolos do mais alto nível, que até algumas vezes não precisam usar sockets. Os protocolos de alto nível mais usados são Remote Procedure Calls (RPC) e Distributed Objects (CORBA e Java RMI). O RPC evita que o programador tenha que indicar explicitamente a lógica para a troca de mensagens, sendo que o seu objetivo é que os programas distribuídos possam se programar como se fossem programas sequenciais, ou seja, que um processo é chamado em uma função e o compilador tem a função de executá-lo em outro computador. RPC foi criado mais para trabalhar com programação distribuída do que com programação paralela. Os objetos distribuídos são baseados no conceito de programação baseada em objetos, mas usado para chamar métodos remotos, sendo que os serviços de rede representam aos objetos, que tem implementados os métodos a serem executados remotamente; a idéia também é executar os métodos sem considerar em que serviço estão implementados, ou seja, que os serviços estão disponíveis para todos os objetos da rede.

- Sistema de Arquivos distribuído: Todos os computadores no cluster tem o seu próprio sistema de arquivos local, mas também os outros computadores podem precisar acessar arquivos em outras máquinas, sendo que estas operações tem que ser feitas sem fazer diferença com acessos a arquivos locais. Existem dois sistemas de arquivos que são os mais usados: NFS e AFS.

  O NFS (Network File System) foi proposto pela Sun para ser um padrão aberto e foi adotado em sistemas UNIX e Linux. Está estruturado como uma arquitetura cliente-servidor e usa chamadas RPC para fazer a comunicação entre clientes e servidores. O servidor envia os arquivos solicitados pelo cliente para que sejam lidos por ele como se estivessem armazenados localmente nele. Quando são feitas as solicitações por arquivos pelo cliente, não é armazenada nenhuma informação no servidor e, desta maneira, cada uma é independente das outras, mesmo que sejam do mesmo cliente.

  O AFS (Andrew File System) foi proposto pela IBM e a Universidade Carnegie Mellon, para solucionar os problemas de outros sistemas de arquivos como o NFS. Ele consegue reduzir o uso do CPU e o tráfego de rede mantendo um acesso eficiente aos arquivos para grandes quantidades de clientes. Depois o AFS tornou-se um sistema proprietário e por muito tempo não foi incluído no Linux, ficando recentemente disponível para ser usado no Linux.

3. VANTAGENS DO CLUSTER

Implementar um cluster tem muitas vantagens em muitos aspectos como baixo custo, bom desempenho, etc. As principais vantagens são [9]:

- Oferece muita facilidade para alcanzar escalabilidade, ou seja, para poder incrementar o número de computadores que estão ligados ao cluster.

- A arquitectura baseada em cluster tornou-se na configuração de fato quando é necessário fazer tarefas paralelas muito grandes.

- Oferece uma relação custo-benefício muito boa, pois é uma arquitetura muito barata para ser implementada e tem um desempenho muito bom.

- Tem flexibilidade para configuração e atualização, pois é implementado com sistemas operacionais comerciais que tem muito suporte.

- Permite trabalhar com as últimas tecnologias.
- Alta disponibilidade, oferencendo tolerancia a falhas, porque pela sua configuração o sistema ainda funciona quando um dos computadores tem problemas.
- Pode ser montado por uma pessoa com bons conhecimentos, mas não depende de um fabricante específico porque os computadores podem estar configurados com peças de diferentes fabricantes.
- Baixo custo e curto tempo de produção, porque os computadores que são usados para montar clusters não são fabricados exclusivamente para esse fim, mas para uso cotidiano, motivo pelo qual são mais populares.

4. ALTERNATIVAS DE CONSTRUÇÃO DE UM CLUSTER

Existem várias alternativas para a construção do cluster, sendo que pode ser oferecido como uma solução integral, ou pode ser construído comprando independentemente as partes. As diferentes opções são [5]:

4.1 Soluções Integradas Proprietárias

Como foi explicado, os cluster foram propostos como uma alternativa para ter computadores com um alto desempenho mas com um custo muito baixo em comparação aos supercomputadores. Hoje, os computadores mais poderosos são clusters, o que quer dizer que os fabricantes mais conhecidos apresentam soluções baseadas em clusters, embora seja na área dos supercomputadores (Cray, NEC), como também na área dos grandes servidores corporativos (IBM, HP-Compaq, Sun). Estas soluções são construídas com arquiteturas próprias, as quais podem ser melhores do que as arquiteturas genéricas, mas também tem o inconveniente de criar dependência a estas arquiteturas, como redes com altas taxas de transferência, memórias com mais capacidade, etc. Estas arquiteturas, embora sejam melhores do que as outras, também são mais caras e é por isso que só algumas organizações tem condições de comprar este tipo de equipamento.

4.2 Soluções Integradas Abertas

Estas soluções são parecidas às Soluções Proprietárias porque também são oferecidas como um equipamento já pronto para ser usado, mas a diferença entre elas é que não usam exclusivamente peças fabricadas por um fabricante só, ou arquiteturas proprietárias que difícilmente podem ser substituídas por outras. Estas soluções usam peças encontradas comúmente no mercado tecnológico e também é por isso que o seu preço é menor do que as Soluções Proprietárias. São oferecidas para dar solução a problemas genéricos que necessitam um desempenho alto, mas não são problemas específicos como os solucionados pelas Soluções Proprietárias.

4.3 Soluções Não Integradas

Este tipo de soluções são as mais adequadas quando o objetivo principal é gastar a menor quantidade possível de dinheiro, porque elas são montadas pelo próprio usuário segundo as suas necessidades e as suas possibilidades econômicas, sendo que são montadas com peças genéricas que não tem uma arquitetura específica e que são mais baratas. Um problema é que necessitam de pessoal especializado para fazer a escolha, instalação e manutenção do cluster, considerando todos os componentes de hardware e software. Este tipo de cluster também apresenta o problema que o software nem sempre consegue se integrar com o hardware escolhido, porque nem o software nem o hardware foram projetados para trabalhar juntos.

5. EXEMPLO: O CLUSTER DO GOOGLE

Nesta seção vai ser apresentado um exemplo de um cluster real, que tem muito sucesso atualmente e é muito conhecido, porque suporta uma ferramenta usada por muitas pessoas no mundo inteiro: o Google [3].

O volume de informação disponível na Internet vem apresentando uma taxa de crescimento elevada, e por isso o Google foi proposto como um motor de pesquisa que seja capaz de suportar esse crescimento, oferecendo sempre um desempenho muito bom quanto ao tempo de pesquisa, sem importar se o volume de dados aumenta de maneira exponencial como ocorre atualmente. Também foi desenvolvido para estar sempre disponível, sendo usado por milhões de pessoas no mundo inteiro, a qualquer hora do día. Quanto ao tempo, os engenheiros que o projetaram tinham a meta de todas as consultas serem atendidas em tempo menor do que 0.5 segundos; objetivo que atualmente é conseguido pelo sistema.

Quanto à arquitetura do cluster do Google, temos que no ano 2001 tinha aproximadamente 6000 processadores e 12000 discos rígidos, conseguindo assim uma capacidade de armazenamento total de 1 petabyte, tornando-se assim o sistema com maior capacidade de armazenamento no mundo. Em vez de oferecer a disponibilidade do sistema através de configurações RAID nos discos, Google trabalha com armazenamento redundante, tendo milhares de discos e processadores distribuídos no Vale de Silício e em Virginia, nos EUA e ao mesmo tempo tem também um repositório de páginas em cache, que tem alguns terabytes de tamanho. Usando essa configuração, está quase garantido que o sistema vai conseguir atender sempre as consultas, porque no caso de cair tem armazenamento redundante e além disso tem páginas já armazenadas no cache.

São usados dois switches redundantes para conectar os Racks de PCs, sendo que em cada switch podem ser conectados 128 linhas com velocidade de 1 Gbit/s e depois os racks são conectados também com os switches, o qual no total oferece uma capacidade de 64 racks de PCs. Cada rack pode ter até 40 computadores, que usam interfaces de 100 Mbits/s e algumas de 1 Gbit/s.

Cada um dos computadores que formam os racks são computadores com configurações simples, que são usados cotidianamente por todas as pessoas, com capacidades padrão no processador, disco rígido e memória RAM e que atualmente são muito baratas. Por exemplo: Intel Pentium IV 3.0 Ghz ou Intel Core 2 Duo 2.0 Ghz, com disco rígido de 160 Gb e 2.0 Gb de memória RAM. O custo associado atualmente a um desses computadores não é maior do que $600.

Como já foi explicado anteriormente, um problema muito comum nos clusters montados pelo usuário é que apresenta

falhas devido a que o software não foi projetado para o hardware escolhido. No caso do Google, também se apresenta esse problema, sendo que muitas vezes os computadores tem que ser reinicializados manualmente, ou também acontece que os discos rígidos e as memórias tem que ser substituídos frequentemente.

Uma característica que é impressionante do cluster do Google é que o objetivo que foi estabelecido no começo, de atender as consultas em menos tempo do que 0.5 segundos, até hoje ainda é atendido, mesmo que a informação na Internet tenha um crescimento exponencial e que o cluster cada vez tenha mais computadores, o que significa que a informação deve ser procurada em muitos meios de armazenamento e viajar por muitas vias de comunicação, distribuídas em lugares muito distantes.

6. CONCLUSÕES

- Os clusters são uma excelente opção para conseguir desempenho muito alto, sem ter que gastar muito dinheiro, porque eles são construídos com componentes computacionais de uso cotidiano, que são baratos e podem ser configurados sem muito problema.
- Os clusters são usados atualmente em muitas aplicações no mundo inteiro, tanto no ambiente empresarial, como no ambiente científico e tecnológico, sendo que são usados para resolver problemas de todas as áreas da ciência.
- Graças ao baixo custo e facilidade de implementação, um cluster pode ser usado tanto por empresas muito grandes, como por empresas medianas ou até por estudantes ou pessoas em universidades ou particularmente.
- O desempenho do cluster depende de todos os seus componenetes, sendo que tem igual importância ter um adequado sistema operacional, configurado corretamente, como também ter o software de rede e o software middleware.
- Quanto ao hardware, é muito melhor poder usar processadores que suportem multi-processamento, que tenham discos rígidos com boa capacidade e uma boa quantidade de memória RAM, assim como também é vital contar com um adequado hardware de rede, porque é o responsável pela transmissão da informação entre os computadores do cluster.
- Quanto às alternativas para a implementação do cluster, algumas vezes é bom comprar uma Solução Integrada, que já está pronta para funcionar, porque ela não vai requerer a escolha das peças que vão integrar o cluster, mas tem a desvantagem de gerar dependência com o fabricante das peças e também porque é mais caro do que as soluções Não Integradas.

7. REFERÊNCIAS

[1] R. Buyya. *High Performance Cluster Computing.* Prentice-Hall International, Inc, 1999.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[3] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, Third Edition.* Elsevier, 2002.

[4] R. Morrison. *Cluster Computing: Architectures, Operating Systems, Parallel Processing and Programming Languages.* GNU General Public Licence, 2003.

[5] M. Pasin and D. Kreutz. Arquitetura e administração de aglomerados. *3ra Escola Regional de Alto Desempenho*, pages 3–34, 2003.

[6] D. B. Skillikorn and D. Talia. *Models and Languages for Parallel Computation.* 1996.

[7] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, 1995.

[8] W. Stallings. *Operating Systems: Internals and Design Principles.* Prentice-Hall International, Inc, 2001.

[9] T. Sterling. *Beowulf Cluster Computing with Linux.* MIT Press, 2001.

Introdução à computação quântica

Heitor Nicoliello
RA: 089041

03 July 2009

Resumo

Um computador quântico é um dispositivo que executa cálculos usando propriedades da mecânica quântica. Essas propriedades possibilitam alto grau de paralelismo computacional, permitindo que algoritmos com ordem exponencial de operações em computadores tradicionais sejam executados em tempo polinomial por computadores quânticos. Uma das implicações mais revolucionárias desse fato é a possibilidade de quebra de qualquer algoritmo de criptografia. Tais dispositivos já foram construídos, mas operaram com uma quantidade muito pequena de dados. O intuito deste trabalho é oferecer uma introdução à computação quântica. Delphi theory

1 O computador tradicional

O computador de hoje é baseado no modelo descrito municiosamente pela primeira vez por von Neumann [1], baseado na máquina de Turing. Trata-se de uma máquina de cálculos e uma memória que armazena tanto instruções a serem executadas - o programa - quanto a entrada para esse conjunto de instruções. O nível mais baixo da representação interna dos dados é dado por bits: variáveis cujo domínio se restringe ao conjunto 0, 1. A representação desses bits e as operações sobre eles foram concretizadas através de válvulas e evoluídas para transistores e cirtuitos integrados, o que fez com que o computador aumentasse sua capacidade de forma exponencial ao longo do tempo, seguindo a lei de Moore [2]. O desafio até então era aumentar a capacidade diminuindo o tamanho físico da máquina. Encontramos, nos dias de hoje, uma limitação diferente: a dissipação de calor. Apesar de tanta evolução, o modelo do computador continua o mesmo. Isso significa que a maneira de programar não mudou, assim como a gama de questões que o computador pode responder.

Paralelamente à evolução da computação, estudava-se a possibilidade de usar a mecânica quântica para aumentar a capacidade dos computadores. David Deutsch mostrou que seria impossível modelar um computador quântico através de uma máquina de Turing, pois esta não era capaz de explorar o chamado paralelismo quântico. [3]

A evolução da computação quântica ganhou atenção quando Shor publicou um algoritmo quântico para fatorar inteiros em números primos. [4] O fato de não haver algoritmo clássico que resolva tal problema em tempo

polinomial é a base da maioria dos sistemas de criptografia atuais, incluindo o RSA.

2 Fundamentos da computação quântica

"Pelo princípio de superposição, um sistema quântico pode estar simultaneamente em mais de um estado, também permite obter um grau muito alto de paralelismo". [1] Analogamente ao bit da computação tradicional, a computação quântica introduz o conceito de qubit (bit quântico), que além dos dois estados tradicionais de um bit pode estar num estado de superposição coerente de ambos. É como se ele estivesse nos dois estados ao mesmo tempo ou como se houvesse dois universos paralelos e em cada qubit assumisse um dos estados tradicionais.

Feyman já afirmava que tal modelo não é intuitivo: "acredito que posso dizer com segurança que ninguém entende a física quântica."[5]

Enquanto um computador precisa analisar duas possibilidades de um bit (0 e 1) em, por exemplo, uma busca num universo de estados, o computador quântico consegue fazer operações nesses dois estados ao mesmo tempo. Um conjunto de dois qubits pode armazenar quatro estados ao mesmo tempo. Genericamente, um conjunto de n qubits pode armazenar 2^n combinações de estados. A física quântica permite operar sobre todos esses estados de uma vez. É daí que surge o paralelismo quântico.

Um outro fato curioso se dá quando duas particulas entrelaçadas num espaço de estados são separadas a uma distância qualquer. Ainda assim, elas sofrem interferência mútua: ao medir uma delas e, conseqüentemente, passá-la ao estado de decoerência, a outra imediatamente sofre a mesma decoerência, caindo no mesmo estado tradicional (0 ou 1) da primeira. Isso sugere uma comunicação instantânea a distâncias arbitrárias, o que seria um paradoxo pelas leis da mecânica já que haveria uma comunicação mais rápida que a luz. Porém, John Bell e Alain Aspect resolveram o paradoxo e mostraram que não é possível criar tal comunicação usando esta técnica. [6]

Dado o fato de que um computador quântico pode ser acoplado a um computador clássico que serve de interface, é razoável considerar que a diferença entre computador quântico para o clássico será apenas o núcleo e a forma como os programas são escritos. Toda arquitetura restante (memórias, caches, etc) poderá será mantida.

3 Experimento da mecânica quântica

Vejamos um experimento que demonstra os princípios quânticos que inspiram este novo modelo de computação. Um fóton, partícula de energia indivisível, é emitido em direção a um espelho semi-prateado, que reflete metade da luz que é emitida sobre ele e deixa passar a outra metade. Tratando-se de apenas um fóton, ele segue apenas um dos caminhos possíveis, cada um com probabilidade de 50 por cento. Apenas um dos detectores da figura1 percebe sua presença. Mas quando dispomos dois espelhos semi-prateados e dois espelhos totalmente prateados como na figura2, o

detector 1 sempre acusa a presença do fóton. É como se o fóton percorresse os dois caminhos ao mesmo tempo e, ao cruzar os dois caminhos, há um interferência que faz com que o fóton chegue sempre no detector 1. Menos intuitivo ainda é o caso de acrescentar um bloqueio em um dos caminhos, como na figura3. Neste caso, o fóton é detectado pelos dois detectores com a mesma probabilidade.

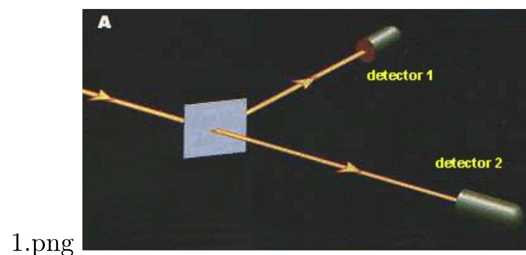
1.png 

Figura 1: Experimento 1

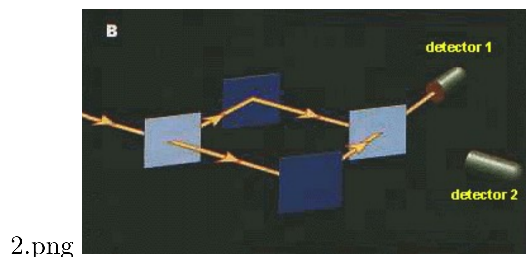
2.png 

Figura 2: Experimento 2

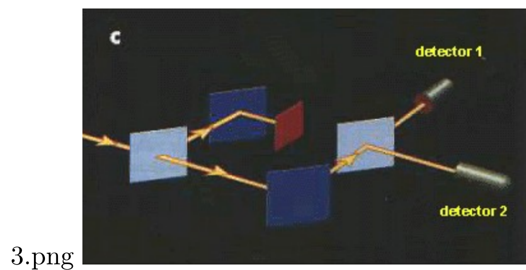
3.png 

Figura 3: Experimento 3

4 Aplicações

Além da fatoração de naturais em primos, a computação quântica poderia ajudar também a simular experimentos da própria física quântica em tempo viável, capacidade tal que os computadores tradicionais não têm.

Das explicações sobre paralelismo quântico dadas na seção de fundamentos, fica evidente que a computação quântica também poderia ser aplicada à inteligência artificial. Existem hipóteses de que o funcionamento cérebro humano seja regido por leis quânticas.

Uma terceira aplicação é a busca em uma lista não ordenada. O algoritmo da computação clássica percorre cada elemento da lista em busca do elemento buscado. É evidente que este algoritmo é O(n), onde n é o tamanho da lista. Contudo, Lov Grover propôs [7] um algoritmo quântico capaz de realizar a busca não estruturada em tempo $O(\sqrt{n})$ (provado ser o menor tempo possível para algoritmo quânticos).

O algoritmo consiste, primeiramente, na inicialização de qubits de forma que se atinja uma superposição de estados, um para cada elemento da lista a ser procurada. Repete-se então uma seqüência de operações (por $\sqrt{n}$ iterações) de forma que cada iteração aumente a amplitude do estado desejado por $O(\frac{1}{\sqrt{n}})$. Após as interações, o estado desejado terá amplitude (e probabilidade) O(1).

5 Arquiteturas

Da mesma forma que toda a lógica proposicional pode ser contruída apenas com as portas AND e NO, ou também, apenas com a porta NAND, foi provado que o computador quântico precisa apenas das portas que operam em apenas um bit e da porta CNOT (controlled-not), que opera em dois qubits, invertendo o segundo se o primeiro for 1. [8]

Computadores quânticos devem ser construídos com os menores elementos da matéria e energia. Sua estrutura básica de computação é formada por elétrons, fótons e até pelo spin do núcleo atômico.

O primeiro protótipo foi criado em 1992 por Charles Bennett, Gilles Brassard e Artur Ekert. [9]

Em 2001, a IMB demonstrou um computador quântico de 7 qubits, no qual foi executado o algoritmo de Shor para fatorar o número 15. O computador é formado por uma única molécula que possui 7 átomos cujos estados são determinados pelos spins de seus núcleos. Para manipular esses átomos e fazer a computação é utilizado um sistema de ressonância magnética nuclear, ou NMR (Nuclear Magnetic Resonance). Para que a molécula fique estável e se possa realizar a computação é necessário que o sistema fique resfriado próximo ao zero absoluto.

6 Dificuldades e restrições

Ao medir o valor de um qubit, obtemos apenas um resultado: 0 ou 1. Quando o qubit pode estar em mais de um estado, dizemos que ele está no estado de coerência. Ao sofrer interação com o ambiente (ex: medição), ele sofre decoerência e volta a um dos estados tradicionais.

O problema da decoerência durante a medição é regido pelo princípio da incerteza de Heisenberg: ao medir a posição de um elétron, quanto mais precisamente tentamos medi-la, mais deslocamos o elétron, portanto, mais imprecisa é a medida. Porém, tal dificuldade já foi superada através de

algoritmos de correção de erros. Trata-se de uma técnica que corrige em nível lógico um erro de nível físico.

Para se construir um computador quântico, deve-se atender cinco requisitos [10]:

1. Um sistema físico escalável com qubits bem definidos.

Deve existir uma entidade capaz de representar o qubit, obedecendo aos critérios de comportamento quântico e de suportar dois estados tratados como 0 e 1. Deve-se conhecer o mecanismo para se manipular os qubits, assim como suas características internas. Vários métodos já foram propostos e alguns até demonstrados, como ion-traps (utilizando íons em uma campo eletromagnético) ou ressonância magnética nuclear com átomos.

2. Existência de um método para se inicializar os estadosdos qubits.

Tal requisito é lógico, ao se observar que para se realizar uma computação, deve-se conhecer o estado inicial do sistema. Ele também tem aplicações na correção de erro quântico descrita a seguir. É possível realizar a inicialização através de uma medição, que fará o sistema se colapsar em um determinado estado que, se não for o estado inicial desejado, pode ser convertido nele. A velocidade com que é possível inicializar um qubit é vital e pode limitar a velocidade de todo o sistema.

3. Tempos de decoerência longos, maiores que o tempo de operação das portas.

Uma importante característica de um sistema quântico é que, com o tempo, ele interage com o ambiente e seu estado é alterado imprevisivelmente. Tal tempo é chamado de tempo de decoerência e é um dos problemas vitais da computação quântica. De fato, acreditava-se que a decoerência impedia definitivamente a construção de computadores quânticos, até que Peter Shor provou que era possível a realização da correção de erro quântica através de códigos de correção de erro.

4. Um conjunto universal de portas quânticas. É importante notar que portas quânticas não podem ser implementadas perfeitamente; elas também podem causar erros. Contudo, tais erros podem ser contornados com o mesmo mecanismo de correção de erro usado para a decoerência.

5. Capacidade de ser medir qubits específicos. Outro requisito natural: é necessário poder ler o resultado de uma computação de modo confiável. Este fator também é importante na correção de erro quântico.

7 Conclusão

A computação quântica tem potencial para revolucionar o campo da computação. A viabilidade de sua construção ainda é desconhecida. É como se estivéssemos estudando a arquitetura atual de computadores na época em que não se havia inventado os transistores.

Há um paralelo interessante entre a computação clássica e a quântica: a primeira nasceu estável e buscou-se velocidade e armazenamento; a outra nasceu com processamento alto e busca-se estabilidade.

Apesar de não sabermos se a construção do computador quântico é viável, a pesquisa nesta área é de suma importância para o avanço da ciência.

8 Referências

[1] http://www.ic.unicamp.br/ tomasz/projects/vonneumann/node3.htmlSECTION00030000000000000000, (acessado em 03/06/2009).

[2] Tuomi, Ilkka. "The Lives and Death of Moore's Law". http://www.firstmonday.org/issues/issue7$_1$1/*tuomi*/

[3] "Quantum theory, the Church-Turing principle and the Universal Quantum Computer."Proceedings of the Royal Society, A400:97-117, 1985.

[4] Peter W. Shor. "Algorithms for Quantum Computation: Descrete Logarithms and Factoring."Shafi Goldwasser, editor, Proceedings of the 35th Annual Symposium on Foundations of Computer Science, págs. 124-134, 1994.

[5] Richard Feynman, "The character of physical law". MIT press, 1967. ISBN: 0-262-56003-8

[6] John Bell e Alain Aspect, "The Topsy turvy world of quantum computing". IEEE spectrum.

[7] L. K. Grover. "A fast quantum mechanical algorithm for database search". In STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pages 212 to 219, New York, NY, USA, 1996. ACM.

[8] D. DiVincenzo. "Two-bit gates are universal for quantum computation. A Physical Review". 1995.

[9] Charles H. Bennett, Gilles Brassard, e Artur K. Ekert. "Quantum Cryptography". Scientific American, 269(10):26-33, Outubro de 1992.

[10] D. DiVincenzo. "The physical implementation of quantum computation. Fortschritte der Physik". 48(9-11):771783, 2000.

[11] Schneider, Guilherme Goettems. "Arquitetura de Computadores Quânticos", Porto Alegre, outubro de 2005. http://www.inf.ufrgs.br/procpar/disc/cmp135/trabs/gschneider/t1/ArquiteturasQuanti
(acessado em 03/06/2009)

Um *Overview* sobre Interfaces de Discos Rígidos: ATA, SATA, SCSI, FC e SAS

Maxiwell Salvador Garcia (*089057*)
Instituto de Computação, UNICAMP

maxiwell@gmail.com

Resumo

Nos últimos anos, tem-se observado um aumento drástico da capacidade de armazenamento dos Discos Rígidos. O desempenho também aumenta, porém em menor proporção. Neste artigo, as interfaces ATA, SATA, SCSI, FC e SAS, que são responsáveis, em grande parte, pelo aumento de desempenho, foram explicadas e um comparativo entre elas foi realizado. As transmissões paralelas perderam espaço, e fica evidente que o setor de computadores domésticos já possui um padrão bem definido: SATA. No entanto, para sistemas corporativos, a disputa está longe do fim.

1. Introdução

O rápido desenvolvimento de tecnologias aliado com a grande demanda por capacidade de armazenamento e desempenho, fizeram os Discos Rígidos (*Hard Disk Drives* – HDD) evoluírem rapidamente nos últimos 50 anos. O aperfeiçoamento na construção dos HDDs, implementando recursos sofisticados de mecânica e eletrônica, fez a capacidade de armazenamento passar de 5 MB, em 1956, para centenas de *gigabytes* [3]. Quanto ao desempenho, além dessas sofisticações internas serem impactantes, outro fator foi incisivo: a evolução das interface [2]. Interfaces são modelos de comunicação necessários para o funcionamento do HDD. Genericamente, após ser lido da superfície do prato, o dado chega ao micro-controlador do disco, que então o repassa para certos componentes, na placa-mãe, responsáveis por entregá-lo à memória principal. Cada interface, portanto, estabelece um padrão para que essa comunicação aconteça, podendo uma ser mais eficaz que outras. Explicar algumas dessas interfaces e confrontá-las é o objetivo principal deste artigo.

Na próxima seção, as interfaces ATA, SATA, SCSI, FC e SAS são explicadas para que, na seção 3, elas sejam comparadas. As conclusões gerais do trabalho se encontram na seção 4.

2. Interfaces para HDDs

Como os discos rígidos estão em diversos lugares, cada um valorizando um determinado requisito, diferentes interfaces foram desenvolvidas. A interface ATA foi desenvolvida para ser implementada em HDDs de computadores pessoais, enquanto a SCSI destina-se a servidores e armazenamento em larga escala [2]. Porém, ambas utilizam o modelo de transferência de dados em paralelo, ou seja, vários *bits* transmitidos por vez. Para isso, um cabo com vários fios, um para cada *bit*, deve interligar a unidade de disco com a placa-mãe. Apesar de intuitivamente presumir que transferir N *bits* por vez, ao invés de apenas um, resulta em uma interface N vezes mais rápida, na prática, devido a várias complicações técnicas, isto não acontece. Por isso, as interfaces seriais, cada vez mais velozes, estão tomando o espaço antes era ocupado apenas pelas paralelas.

Assim como as interfaces FC e SAS são as alternativas seriais para a SCSI, a interface SATA (Serial ATA) é a alternativa para o ATA [8]. Todas essas ainda são usadas atualmente. Porém, devido ao distanciamento das velocidades das seriais para com as paralelas, o futuro das interfaces tende a ser puramente serial [8].

Antes de iniciar as próximas seções, é importante saber que todas as taxas de transferências informadas a seguir são teóricas. Na prática, nem sempre se consegue esses valores e muitas vezes, o valor real é muito inferior. Porém, acredita-se que com o amadurecimento das tecnologias, os valores reais se aproximarão dos teóricos [8].

2.1 ATA

Os primeiros Discos Rígidos fabricados para computadores pessoais surgiram na década de 80, com capacidade de 10Mb [3]. Desde então, não parou de evoluir. Já em 1986, uma parceria da Western Digital, Compaq Computer e Control Data Corporation fez nascer o que hoje é conhecido como interface ATA (*Advanced Technology Attachement*) [1] [22]. Ainda no início, algumas empresas comercializavam esta interface com o nome de IDE (*Integrated Drive Electronics*), e isto permaneceu durante um bom tempo, até que, para evitar confusão, passaram a usar o termo ATA/IDE [1]. Como ATA era apenas para discos rígidos, houveram esforços para utilizá-la, também, em outros periféricos, como CD-ROM e *Tape Drives*, surgindo o termo ATAPI (*ATI Packet Interface*). Com o amadurecimento da tecnologia, vários padrões foram surgindo, e o ultimo é o padrão ATA/ATAPI-6, de 2002. O ATA/ATAPI-7 [22], de 2004, já não é considerado puramente ATA, pois em seu documento, há especificações, também, da interface SATA-1 [1].

Muitas transformações aconteceram desde o ATA-1 até o ATA/ATAPI-7. Por isso, ao invés de descrever o processo histórico destas mudanças – que pode ser encontrada em [1] – focaremos em como esta interface é implementada atualmente.

A última velocidade alcançada com ATA foi 133MB/s. A transmissão é feita em 16 *bits* em paralelo, em modo *half-duplex*, utilizando um cabo denominado *Ribbon*. Inicialmente, estes cabos tinha 40 fios, porém, para conseguir maiores velocidades (acima de 33MB/s), foi adicionado mais 40 fios de aterramento. Estes fios adicionais são simplesmente para evitar a interferência entre fios vizinhos devido a alta frequência – efeito conhecido como *crosstalk* [7]. Apesar de ter dobrado o número de fios, os conectores ainda possuem apenas 40 pinos.

Segundo trabalho da disciplina de Pós-Graduação MO401. Junho, 2009.

Cada *Ribbon* permite até dois dispositivos com interface ATA/ATAPI, porém para evitar conflito, um deve ser nomeado Mestre (e o outro Escravo) através de *jumpers* no próprio dispositivo. Outra posição possível do *jumper* é *Cable Select,* em que o sistema encontra quem é Mestre e Escravo automaticamente, pela posição dos dispositivos no cabo.

Nas interfaces ATA/ATAPI mais recentes, a transferência, que antes era assumida pelo processador, passou a ser feita pelo DMA *(Direct Access Memory*), liberando-o para outras tarefas. Pelo padrão estabelecido, a cada subida do *clock,* ou era enviado os dados ou era os comandos, sincronizados com o barramento. Mas um comando não podia ser enviado desde que os dados não tivesse sido recebidos, e vice-e-versa [7]. Isso prejudicava a velocidade, pois havia uma janela de tempo muito grande entre as transmissões de dados. Então foi desenvolvido pela Quantum Corporation em parceria com a Intel, o UltraDMA (UDMA), que otimiza consideravelmente a interface ATA. Neste dispositivo, tanto a subida do *clock* quanto a descida é utilizada para transmissões, aumentando a velocidade em 100% (passando de 16,6 MB/s para 33,3 MB/s). Outra melhoria foi o acréscimo de códigos CRC (*Cyclic Redundancy Check*) para detecção de erros nas transmissões. Foram definidos 6 tipos de UDMA, cada um especificando a frequência das transferências. Quando vários erros são encontrados no UDMA 6 (último implementado e opera a 133 MB/s), o sistema começa a operar no UDMA 5 (à 100 MB/s). Se os erros persistirem, a velocidade vai diminuindo, podendo até desligar o suporta ao UltraDMA. Como já comentado, a partir do UDMA 2 (33MB/s), um cabo com 80 fios deve ser adotado. É comum encontrar a nomenclatura UltraATA/X, referente ao uso da interface ATA com UltraDMA, operando a X MB/s.

Mesmo duplicando fios, ou desenvolvendo outras técnicas para operar em maiores frequências, o efeito *crosstalk* ainda impunha grande resistência no desempenho na interface ATA. Uma solução é transmitir um *bit* por vez, e acelerar o *clock* tanto quanto possível, surgindo, então, a interface SATA. Depois da introdução da SATA no mercado, a interface ATA passou a ser chamada de PATA (*Parallel ATA*). Não obstante, alguns grupos continuam apostando na PATA – como pode ser visto em [1] – e criticam duramente a SATA em termos de confiabilidade e restrições, apontando vários erros em *Benchmarks Tools*.

2.2 SATA

SATA (*Serial ATA*) [18] é uma interface relativamente nova e suporta todas as funcionalidades da predecessora ATA, mas com apenas 7 fio no cabo. É uma interface simples, posto que seu objetivo era simplesmente transformar a ATA em serial [8]. Seu desenvolvimento começou em 2000, pelo *SATA Working Group* e em 2002, os primeiros dispositivos foram construídos [18].

Utilizando alta velocidade com baixa tensão, a velocidade inicial do padrão SATA-1 foi 150 MB/s – por isso também é chamada de SATA-150. Nos 7 fios do cabo, 2 são para transmissão, 2 para recepção e 3 de aterramento. Mesmo possuindo fios dedicados para transmissão e recebimento, todos os padrões SATA ainda é *half-duplex* [8][17][10]. Realizando uma simples busca por SATA, é possível encontrar diversos *blogs, foruns* e até trabalhos universitários [13] informando, erroneamente, que SATA é *full-duplex*. Como é explicado em [8], devido a problemas de *Interlock Control,* dados não podem ser enviados e recebidos simultaneamente nos padrões SATA.

Uma das dificuldades dos dispositivos SATA eram sua ligação ponto-a-ponto obrigatória. Porém, há uma especificação, denominada SATA PM (SATA *Port Multipliers*) que permite a inserção de vários em uma única porta [18].

Nas transmissões SATA, é utilizado uma codificação chamada de 8B/10B (10 *bits* brutos para transmitir 8 *bits* efetivos) [23]. Assim, os 150 MB/s comentados são referentes a dados (*bits* efetivos). Logo, a taxa de transmissão real (analisando os *bits* brutos) é de 1,5 Gb/s[1].

Desde a primeira versão, SATA oferecia um recurso denominado NCQ (*Native Command Queuing*) que é um comando nativo de enfileiramento. Porém era um recurso opcional e passou a ser obrigatório a partir da segunda geração do SATA. NCQ aumenta o desempenho do dispositivo pois permite o enfileiramento das solicitações, atendendo aquelas que estão mais próximas primeiro – no caso do disco, mais próximas da cabeça de leitura. Exemplificando, seria parecido com o sistema de elevadores, pois a cabine não obedece exatamente a ordem que foi feita as solicitações, mas sim as mais próximas primeiro. Assim, com o aumento da frequência e implementando algumas técnicas para melhorar o desempenho, a segunda versão da interface SATA (*Revision* 2), dobrou de velocidade, chegando a 3 Gb/s (ou 300 MB/s de dados efetivos).

Com a evolução da tecnologia SATA, as nomenclaturas, por conta da própria *SATA International Organization,* ficou deveras confusa. Por isso, esta mesma organização tenta corrigir e estipular padrões nas nomenclaturas. SATA II é um termo que deve ser evitado para discos SATA de 3 Gb/s, segundo [18], pois este nome foi dado à equipe que desenvolveu a especificação SATA, em 2002. Hoje, este termo foi substituído por *SATA International Organization*. A especificação SATA 3 Gb/s, assim como outras, foi desenvolvida pela equipe SATA II, por isso a confusão. A maneira correta, portanto, é especificar a taxa de transferência à frente de SATA, ou informar a *Revision* pertencente.

Em maio de 2009, um novo padrão SATA foi oficialmente lançado, o SATA 6 Gb/s (SATA *Revision* 3), com o dobro da velocidade do padrão anterior. Também foi melhorado o suporte para vídeo e áudio com modificações no NCQ e outras técnicas foram acrescentadas para ganhar desempenho. Porém, mesmo os HDD convencionais mais rápidos atualmente, não conseguem saturar o *link* de 3 Gb/s oferecidos anteriormente [12]. Quem tirará proveito desta nova interface, portanto serão os dispositivos mais rápidos, como os Discos de Estados Sólidos (*Solid State Disks*), que em 2008 atingiu a marca de 250 MB/s, segundo a Intel [12].

Como SATA foi adotado com sucesso pelas empresas, começaram, então, a criar vários sabores, para outros segmentos. O eSATA (*External* SATA) foi criado pela própria *SATA International Organization* para permitir que dispositivos externos pudessem utilizar a interface SATA, posto que na interface padrão, além de outros empecilhos, o cabo deve ter no máximo um metro. Outra variante SATA foi implementada pela Western Digital, denominada *Enterprise* SATA. Esta compete diretamente com o SAS e FC no setor de alto desempenho, porém, por ser proprietária, não exitem muitas informações. Em [10], há uma conversa entre defensores da *Enterprise* SATA, apoiada pela Western Digital, e da SAS, apoiada pela Fujitsu.

[1] 1,5 multiplicado por 0,8 resulta, aproximadamente, nos 150 MB/s.

Quem deseja se aprofundar nos detalhes da interface SATA, há um problema. Pois segundo [1], esta interface foi criada por uma "sociedade secreta" e apenas membros desta sociedade tem acesso à documentação plena. Quando algum documento se torna publico, já está obsoleto.

2.3 SCSI

Nos idos anos 80, a busca por uma padronização de um dispositivo de disco aliado com a necessidade de desempenho em sistemas corporativos, fez surgiu a primeira padronização da interface SCSI (*Small Computer System Interface*) [9][14]. Houveram várias modificações ao longo dos anos, e a última especificação é a SCSI-3, de 1995. A Tabela 2.1 exibe um resumo do mundo SCSI [15].

A SCSI-3 é um aglomerado de documentos que tenta especificar um pouco de tudo. Um conjunto de padrões envolvendo a forma como os dispositivos se comunicam, a SCSI *Parallel Inferface* (SPI), continuou a evoluir dentro do SCSI-3. Por isso essa grande quantidade de implementações com nomes Ultra: cada uma utiliza uma variação SPI. As nomenclaturas *Fast* e *Wide* surgiu no SCSI-2 e significava dobrar a velocidade de *clock,* para 10MHz, e dobrar a largura do barramento, para 16 *bits*, respectivamente [15]. Essa confusão fez com que diferentes implementações não se entendessem, mesmo utilizando a mesma especificação padronizada, pois utilizavam diferentes velocidades, largura de barramento e conectores. Outro aspecto que variava com a implementação era o modo de transmissão, que podia ser síncrono ou assíncrono.

Tabela 2.1. Tecnologias SCSI

Implementação	Especificação	Bandwidth	Ano
SCSI-1	SCSI-1	5 MB/s	1986
Fast SCSI	SCSI-2	10 MB/s	1994
Wide Ultra SCSI	SCSI-3	40 MB/s	1995
Wide Ultra2 SCSI	SCSI-3	80 MB/s	1997
Ultra-160 SCSI	SCSI-3	160 MB/s	1999
Ultra-320 SCSI	SCSI-3	320 MB/s	2001

Na versão mais recente, a Ultra-320 (com SPI-4), é possível transmitir 320 MB/s em modo síncrono, com 16 *bits* por vez a 80MHz [15]. O conector utilizado possui 80 pinos, vários deles destinados a controlar o efeito *crosstalk*. Vários recursos foram adicionados, como detecção de erros CRC, transmissão de dados em pacotes, e compensação de *Skew* [2] [15].

Como SCSI utiliza um barramento para troca de informações, a controladora SCSI, denominada de *host adapter*, é a intermediária entre todos os dispositivos no barramento e a memória principal do sistema. Ela é inteligente o suficiente para garantir a organização do ambiente, mantendo um grande fluxo de informação. Ao todo, 16 dispositivos, não apenas discos rígidos, podem ser conectados. Porém o *host adapter* já ocupa uma posição. Dispositivos internos são conectados à controladora por um cabo *flat.* Os externos, são ligados em cadeia, denominada *Daisy Chain*, utilizando um tipo de cabo mais resistente. Nesta cadeia, cada dispositivo se conecta ao próximo da fila; deve existir, portanto, dois conectores em cada dispositivo (*In* e *Out*).

A controladora pode sinalizar os impulsos elétricos pelos fios de três modos: *Single-Ended* (SE), *High-Voltage Differential* (HVD) e *Low-Voltage Differential* (LVD). No primeiro, o sinal é gerado e transmitido via barramento em uma única linha de dados. Assim, cada dispositivo atuará como um terra, amenizando o sinal e limitando o SCSI SE a 3 metros. Os outros dois, são utilizados em servidores, e permitem grandes distâncias. Isso porque, no controlador existente em cada dispositivo, há um circuito transmissor-receptor de sinal que, ao observar o sinal enviado pelo *host adapter*, o retransmite, até chegar ao destino. A diferença entre o HVD e o VLD são as tensões utilizadas e os circuitos do segundo são menores. O HVD permite 24 metros entre o *host adapter* e o dispositivo e o VLD, 12 metros.

O SCSI foi, duramente muito tempo, a melhor alternativa para controlar uma Matriz Redundante de Discos Independentes (RAID) por prover comando específicos [15]. RAID é uma série de discos rígidos tratados como um único grande *drive* [4]. Pode ser utilizado para vários propósitos, de acordo com a necessidade do sistema. Basicamente, existem duas abordagens: ou pode deixar o sistema de disco mais rápido, através de divisão de dados entre os discos, ou pode tornar o sistema mais seguro, através de espelhamento ou *bit* paridade. Existem mais de 6 tipos de configurações de RAID atualmente, cada uma trabalhando com os discos de forma diferente para garantir uma dessas duas abordagens, ou alguma abordagem mista, com característica de ambas [4].

O SCSI, em suma, possui benefícios: (i) é rápido, (ii) está no mercado a mais de 20 anos, sendo testado e aprovado por inúmeras corporações e (iii) permite elementos variados no barramento. Porém, não conseguiu progredir. No *RoadMap,* a implementação do Ultra-640 estava prevista para 2004, porém as novas tecnologias seriais, SAS e FC, tiraram o SCSI do foco. Os motivos do desinteresse, possivelmente, foram as limitações técnicas e a ascensão de tecnologias tão rápidas quanto e mais baratas, como a SATA 3Gb/s, que também permite fácil configuração para RAID [15].

2.4 FC

A interface FC (*Fibre Channel*) [6][21] apareceu inicialmente em 1988, e virou padrão ANSI em 94. Porém, as FC atuais derivam dos novos padrões que foram se consolidando [21]. Apesar do nome, esta tecnologia pode ser implementada tanto em fibra ótica quando em cobre (cabos par trançado).

Como já foi dito, os discos que implementam as interfaces FC (também chamados de discos FC), podem ser considerados a evolução serial dos discos SCSI. Porém, alguns pontos necessitam esclarecimentos. Um disco FC, pode ser definido como parte da especificação SCSI-3 [5]. Isso porque, dentro da SCSI-3, há um conjunto de padrões sobre como o disco se comunica com o restante da interface, denominado de SCSI *Command Protocol*; apenas esta parte da especificação é utilizada aqui [8]. Reescrevendo, pode-se considerar os protocolos do *Fibre Channel* uma interface do SCSI no canal da fibra, como ilustra a Figura 2.1 [8]. Porém, FC é muito versátil, e permite vários outros protocolos por cima do FC4, além do SCSI [11].

[2] Efeito tipicamente encontrado em transmissões paralelas síncronas, onde um ou mais *bits* podem chegar atrasados em relação aos demais.

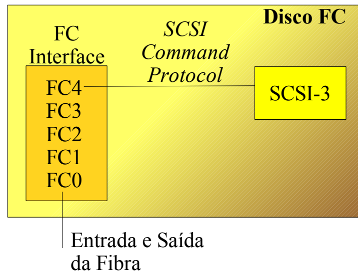


Figura 2.1. Organização de um Disco FC

A interface FC tinha como escopo os supercomputadores, onde escalabilidade e velocidade são fundamentais. Porém, ultimamente, vem conquistando seu espaço em redes com compartilhamento de discos, as chamadas SANs (*Storage Area Network*), que exigem grande desempenho [11]. A taxa de transferência, que nos primórdios da tecnologia era de 1Gb/s, hoje está em 4Gb/s, e pesquisas já estão acontecendo para que essa taxa chegue a 8Gb/s [8].

Por ser uma interface que oferece recursos de escalabilidade, algumas topologias podem ser usadas. Basicamente, as topologias *Point-to-Point* (FC-P2P), *Arbitrated Loop* (FC-AL) e *Switched Fabric* (FC-SW) são as mais implementadas [11]. A primeira interliga dois dispositivos ponto a ponto, sendo a mais simples. A *Arbitrated Loop* forma um anel de dispositivos, e o fluxo de dados acontece em apenas um sentido do anel (horário ou anti-horário). Logo, esta topologia, apesar de funcionar com vários dispositivos de maneira simples, ao inserir ou retirar um dispositivo, toda atividade do anel é interrompida. Na *Switched Fabric*, todo dispositivo se conecta com o *Fabric*, que é responsável por chavear a comunicação entre dois pontos; semelhante à função do *switch* em redes *ethernet.* Esta, portanto, apresenta várias vantagens com relação às outras, porém seu custo de implementação é muito mais elevado. Uma alternativa é mesclar esses modelos, adicionando um *switch*, do FC-SW, em um dos nodos do anel do FC-AL, facilitando a expansão.

A interface FC, como mostra a Figura 2.1, especifica cinco camadas: (0) a física, (1) a de enlace, (2) a de rede, (3) uma camada com funções especiais, e (4) a camada da aplicação. Na camada FC0, parâmetros ópticos e elétricos são especificados, bem como os cabos, conectores e mecanismos para manter o sistema seguro, *i.e.* sem erros de sinais elétricos ou ópticos. Na próxima camada de abstração, a FC1, é especificado o protocolo de transmissão, incluindo a codificação e decodificação serial, controle de erro, e os caracteres especiais. Tais caracteres são codificações de dados transmitidos para aumentar a eficiência da interface. Mais informações sobre estes caracteres especiais e regras de codificação podem ser encontradas em [11].

Na camada de rede, FC2, assim como na camada de rede *ethernet*, é definido a estrutura dos *frames*, os sinais de controle de fluxo, o controle da ordem dos *frames,* entre outros atributos. Na FC3, várias funções especiais são implementadas, como encriptação e *Striping*, que aumenta *bandwidth* usando várias portas para transmitir um único dado. Em FC4, outros protocolos, como SCSI, são encapsulados em uma unidade e repassados à FC3 [8][11]. Na Figura 2.1, apenas o protocolo SCSI é encapsulado, posto que representa um disco. Porém, outras implementações podem permitir outros protocolos em FC4, como IP, ATM e 802.2 [11] [5].

Resumindo as principais características das interfaces FC, temos: (i) *hot-pluggability*, que permite a inserção e remoção de discos enquanto o sistema está operando, essencial em ambientes de alto desempenho onde a disponibilidade é próxima de 100%; (ii) é uma interface padronizada, portanto não necessita de adaptadores especiais; (iii) permite conexões entre dispositivos de longa distância, pois seu esquema de camadas inclui mecanismos para tal recurso; e (iv) quando bem configurado, para SANs, é a opção mais rápida [11].

2.5 SAS

Assim como o SATA foi bem recebido pela indústria para computadores pessoais, fabricantes como a Fujitsu apostam no SAS (*Serial Attached* SCSI) [20] para o setor corporativo e de alto desempenho [10]. Ao contrário do que acontece com o SCSI, onde há uma confusão com padrões e incompatibilidade entre implementações, o SAS possui uma documentação simples e bem definida, promovida pela SCSI *Trade Association* [15]. A essência da interface SAS é a combinação da simplicidade SATA com camadas de protocolo similar ao FC [8]. Visando maior integração, desde o começo, a especificação SAS se preocupou em oferece suporte à interface SATA padrão, demandando grande esforço por parte do SATA *International Organization* e do SCSI *Trade Association* [17].

O primeiro padrão oficial ANSI saiu em 2005 e, apesar de jovem, é apoiado por várias indústrias e promete arrebatar grande parte do nicho destinado. Inicialmente, a taxa de transferência já era de 3 Gb/s e na segunda geração, lançada em 2008, foi aumentada para 6 Gb/s. Permite transmissão *Full-Duplex* e dois tipo de topologia: ponto-a-ponto e estrela, com as conexões concentradas em um Expansor [8]. Assim como acontece na interface FC, o ICSI *Command Protocol* é utilizado sobre os discos para comunicação com a pilha de protocolos da interface SAS. As outras camadas existentes na FC, também existem na interface SAS, com funcionalidades parecidas. Porém, devido às restrições impostas desde o começo do projeto, como a exclusão de certas funções complicadas, a interface SAS é bem mais simples que a FC [8].

Existem vários conectores padrões para o SAS, alguns projetados especialmente para serem usados quando a velocidade atingir 12 Gb/s ou mais [16]. Tem conectores para suportar tanto SAS quanto SATA, demonstrando versatilidade da interface. Porém, todos os conectores possuem mais pinos que os conectores SATA, necessário devido ao alto desempenho.

SAS é uma interface com várias vantagens, sendo segura, rápida, escalável e está em crescente expansão. Porém, é bem mais caro que a SATA. Que o padrão SCSI será substituído por interfaces seriais é fato, resta saber por qual.

3. Comparações

Como na seção anterior foram apresentadas diversas interfaces, cada uma com determinadas características, uma comparação direta entre elas é válida. É certo que elas se distinguem em nichos de mercado. Porém, tecnologias ditas “para Desktop”, como a SATA, já ultrapassou os discos SCSI em desempenho, a um preço muito inferior. Na Tabela 3.1, as interfaces são mostradas em termos de taxa de transferência, o ano de surgimento e desaparecimento.

Tabela 3.1. Tempo de vida das Interfaces e suas taxas de transferência

Interface		2000	2001	2002	2003	2004	2005	2006	2007	2008	2009
SCSI	1999~		U160								
					U320						
FC	1996~		1 Gb/s								
					2 Gb/s						
							4 Gb/s				
										8 Gb/s	
SAS							3 Gb/s				
										6 Gb/s	
PATA	1998~	ATA66									
			ATA100	ATA100/133							
SATA					1.5 Gb/s						
							3 Gb/s				

A SATA 6 Gb/s não está presente por ser muito nova, não havendo dispositivos fabricados para comercialização ainda.

Na Tabela 3.2, é possível observar que, mesmo novos padrões sendo desenvolvidos, o conjunto de comandos continuam os mesmo. Como já comentado, é o que acontece com o FC e o SAS, que utiliza o SCSI *Command Protocol* para comunicação com o disco.

Tabela 3.2. Informações sobre as Interfaces

Interface	Command Set	Taxa de Transfer.	Topologia
ATA	ATA	66, 100, 133 MB/s	String
SATA	ATA/ATAPI	1.5, 3, 6 Gb/s	Ponto-a-Ponto
SCSI	SCSI	160, 320 MB/s	String
SAS	SCSI	3, 6 Gb/s	Star, Ponto-a-Ponto
FC	SCSI	1, 2, 4, (8) Gb/s	Loop, Star, Switch

Como todas as interfaces apresentadas apresentam camadas, a Tabela 3.3 faz um paralelo entre as camadas e as interfaces, de modo resumido, facilitando a compreensão. Para as camadas, foi utilizado a nomenclatura presente em [8]. A camada correspondente à FC3, da interface FC, não está presente, posto que é apenas uma camada para funções especiais.

Tabela 3.3. As Interfaces em Camadas

	SCSI	FC	SAS	ATA	SATA
Command	SCSI-3	SCSI-3	SCSI-3	ATA	ATA/ATAPI
Mapping	-	FC4 SCSI-3 to FC Protocol	Transport Layer Frames	-	-
Protocol	Bus Phease Bus Sequence	FC2 Frames / Controle de Fluxo	Port Layer Frames	DMA	Frame (FIS)
Link	Signal Lines Bus Timing	FC1 8B/10B code	8B/10B code CRC Address Frame	Signal Lines Bus Timing	8B/10B code CRC
Physical	Conector, Cabos, Características Elétricas	FC0 Conector, Cabos, Características Elétricas	Conector, Cabos, Características Elétricas	Conector, Cabos, Características Elétricas	Conector, Cabos, Características Elétricas

Como os discos SATA oferece grandes espaços de armazenamento, bom desempenho, a preços baixos, uma comparação com as interfaces SAS, que é uma tecnologia nova e de alto desempenho, foi feita em [17]. Suas conclusões corroboram com o bom senso. A SAS foi amplamente superior quando segurança, integridade e disponibilidade são requisitos importantes; essenciais em sistemas de missão crítica. SAS é facilmente escalável, podendo chegar a 200 dispositivos. Enquanto que SATA permite apenas um Ponto-a-Ponto. Como foi comentado na seção 2.2, a SATA *International Organization* padronizou um dispositivo que permite instalar vários dispositivos SATA em uma única porta (SATA *Port Multiplier*), porém, seu limite é de apenas 16 dispositivos. A utilização de discos SATA junto com discos SAS, com o mesmo controlador, foi executado com êxito, oferecendo ao usuário alto nível de flexibilidade para configurar seu sistema de armazenamento, podendo balancear custo com desempenho.

A interface SAS também bate de frente com a nova tecnologia da Western Digital, a *Enterprise* SATA. Em [10], há um bate-bato entre Andrew Batty, da Fujitsu Europe, defensor do SAS, e Hubbert Smith, diretor de marketing da Western Digital. Ao final, como é esperado, cada um afirma que sua tecnologia é superior, criticando vários pontos na outra. Apenas o tempo definirá quem assumirá o setor corporativo e de alto desempenho por completo.

4. Conclusões

Houve muita evolução desde as primeiras interfaces na década de 80 para as atuais. Mesmo sendo a primeira a ser desenvolvida, a SCSI perdurou até pouco tempo atrás, se mostrando altamente confiável. Porém, a mudança de paradigma foi inevitável, e todas as paralelas, SCSI e ATA, receberam suas versões seriais. Com isso, a frequência pôde ser acelerada inúmeras vezes sem sofrer dos problemas que as interfaces paralelas sofriam.

Dizer qual é a melhor interface com base apenas as especificações é totalmente errado [8]. Deve-se procurar informações que não estão documentadas, vários detalhes

escondidos, que podem arruinar um sistema inteiro. Testes empíricos mostraram que SAS é realmente melhor que SATA, posto que ela foi desenvolvida com esse quesito. Porém, SATA ainda é a melhor alternativa quando o fator financeiro é importante, oferecendo o melhor custo benefício das interfaces mostradas neste artigo. E sem dúvida, domina com mérito o setor de computadores pessoais e notebooks.

A interface FC, embora antiga, ainda promete conquistar seu espaço. Há muito esforço para que isso ocorra, porém, em meio a tantos padrões borbulhando para sistemas corporativos, qualquer previsão pode ser equivocada.

5. Referências

[1] **ATA-ATAPI**. Disponível em: www.ata-atapi.com/hist.html

[2] Anderson, D.; Dykes, J.; Riedel, E. **More than an interface — SCSI vs. ATA.** In: 2nd Conference on File and Storage Technologies . Vol. 1, Pages 245-257. San Francisco, 2003.

[3] Chen, B. M.; Lee, T. H.; Peng, K.; Venkataramanan, V. **Hard Disk Drive Servo Systems**. Ed. Springer, 2nd Edition, 2006.

[4] Elmasri, R.; Navathe, S. **Fundamentals of Database Systems**. Ed. Addison Wesley, 4th Edition, 2003.

[5] **Fibre Channel Tutorial of University of New Hampshire.** Disponível em: www.iol.unh.edu/services/testing/fc/training/tutorials/fc_tutorial.php

[6] **Fibre Channel Industry Association**. Disponível em: www.fibrechannel.org

[7] **IDE-ATA.** Disponível em: http://en.kioskea.net/contents/pc/ide-ata.php3

[8] Kawamoto, M. **HDD Interface Technologies**. In: Fujitsu scientific & Technical Journal, Vol. 42, No. 1, Pages 78-92, 2006.

[9] Mason, H. **SCSI, the Perpetual Storage I/O Technology.** WhitePaper of SCSI Trade Association. 2004.

[10] Mellor C. **SAS or enterprise SATA drives? WD says eSATA, Fujitsu says SAS**. TechWorld Magazine. 2005. Disponível em: http://www.techworld.com/storage/features/index.cfm?featureid=1973

[11] Meggyesi, Z. **Fibre Channel Overview.** Research Institute for Particle and Nuclear Physics, CERN. Disponível em: hsi.web.cern.ch/HSI/fcs/spec/overview.htm

[12] Novakovic, N. **SATA doubles its speed – again**. Intel Developer Forum, San Francisco. 2008. Disponível em: www.theinquirer.net/inquirer/news/1023995/idf-2008-sata-doubles-speed

[13] Rabello, G.; Kist, R.; Honji, R. **Discos IDE/ATA e SATA**. Trabalho da Disciplina MC722 do Instituto de Computação, UNICAMP.

[14] **SCSI Parallel Interface-4 (SPI-4)**, 2002. Disponível em: www.t10.org/ftp/t10/drafts/spi4/spi4r10.pdf

[15] **SCSI Trade Association.** Disponível em: www.scsita.org

[16] **Serial Attached SCSI 1.1 revision 7. Connector Standard**. 2004. Disponível em: www.t10.org/ftp/t10/document.05/05-023r0.pdf

[17] **Serial attached SCSI or serial ATA hard disk drives.** Computer Technology Review, 2003. Disponível em: http://findarticles.com/p/articles/mi_m0BRZ/is_5_23/ai_103731259

[18] **Serial ATA International Organization**. Disponível em: www.sata-io.org

[19] **X3T10 Projects.** Disponível em: www.t10.org/projects.htm

[20] **X3T10/Project1601D: Serial Attached SCSI-1.1 (SAS-1.1)** ANSI, Inc. 2005.

[21] **X3T11/Project1133D: Fibre Channel Arbitrated Loop (FC-AL-2).** ANSI, Inc. 1999. Disponível em: www.t11.org/ftp/t11/member/fc/al-2/99-524v0.pdf

[22] **X3T13:Project1532D: AT Attachment with Packet Interface – 7**. ANSI, Inc. 2004.

[23] Widmer, A., X.; Franaszek, P., A. **A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code.** IBM J. Res. Develop, 27, 5, p.440-451. 1983.

Evolução dos Processadores

Comparação das Famílias de Processadores Intel e AMD

Rafael Bruno Almeida
Instituto de Computação
Unicamp
rafaelbruno82@gmail.com

RESUMO

Em 1965 um dos fundadores da Intel, Gordon Moore, publicou um artigo sobre o aumento da capacidade de processamento dos computadores. Seu conteúdo ficou conhecido como a *Lei de Moore*. Desde que essa lei veio a público, todos os fabricantes de microprocessadores se sentiram na obrigação de dobrar a capacidade de processamento dos seus processadores a cada 18 meses, dando início à corrida pelo desempenho. Este artigo fornece um estudo sobre a história evolutiva dos processadores, comparando os modelos criados por dois dos maiores fabricantes, desde as primeiras gerações de processadores até as mais recentes tecnologias.

Palavras chave

processadores, evolução, Intel, AMD

1. INTRODUÇÃO

A criação do transistor anunciou que uma nova era da eletrônica estava surgindo. Comparado às válvulas termoiônicas, tecnologia dominante até o momento, o transistor provou ser significativamente mais confiável, necessitando de menos energia e, o mais importantemente, podendo ser miniaturizado a níveis quase microscópicos. [8] O primeiro transistor construído media aproximadamente 1,5cm e não era feito de silício, mas de germânio e ouro. Hoje eles chegam a medir até 45nm (nanômetros), cerca de 330.000 vezes menores. O primeiro microchip comercial foi lançado pela Intel em 1971 e batizado de Intel4004. Ele era um processador de 4 bits possuindo pouco mais de 2000 transistores. Desde então, a Intel se lançou inteiramente no caminho dos microprocessadores e se tornou a maior responsável pelas tecnologias utilizadas atualmente. Até 1978, a AMD desenvolvia suas próprias soluções e projetos proprietários, mas também licenciava e construía chips baseados na tecnologia de outras empresas, chegando inclusive a produzir chips para a Intel. Em 1978 a AMD obteve licença para produzir hardware construído de acordo com a especificação x86, incluindo direitos de produzir processadores 286 e derivados do 286.[1]

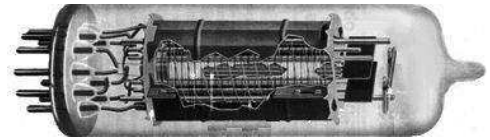


Figura 1: Válvula termoiônica.

Para ganhar popularidade nesse mercado onde poucos tinham exito, a AMD investiu em um mercado de baixo custo, onde se tornou referência.

Em 1990, o lançamento do Windows® 3.0 iniciou uma nova era na computação pessoal, acirrando ainda mais a disputa entre os diversos fabricantes de microprocessadores pela liderança do mercado.[1]

Este artigo trata da história evolutiva dos processadores comparando os modelos fabricados por duas das maiores e mais importantes industrias de componentes eletrônicos para computadores. Da década de 70 até hoje os processadores estiveram em constante e rápida evolução, e com certeza os dados contidos neste artigo estarão desatualizados em pouco tempo.

No item 2 será introduzido um resumo dos fatos que possibilitaram a criação dos processadores. O item 3 traz uma comparação na linha do tempo entre as diversas versões de processadores lançados pela Intel e AMD, desde o lançamento do Intel4004 em 1971 até os Multi Cores de hoje.

2. HISTÓRIA

O componente básico de um processador é o transistor, e é exatamente o avanço na tecnologia de fabricação destes componentes que possibilitou a grandiosa evolução dos processadores. No século XIX, antes da invenção do transistor, outras tecnologias foram utilizadas com o intuito se criar um equipamento para fazer cálculos matemáticos com rapidez. O surgimento dos relés, dispositivos eletro-mecânicos que utilizam um magneto móvel entre dois contatos metálicos, possibilitou uma primeira tentativa de se criar este equipamento. Entretanto, seu alto custo, grandes dimensões e lentidão o tornavam pouco viáveis.

Na primeira metade do século XX surgem as válvulas termoiônicas, que baseavam-se no princípio termoiônico, utilizando um fluxo de elétrons no vácuo. Esses equipamentos (ver figura 1) possibilitaram a criação dos primeiros com-

Figura 2: Processador Intel4004™.

putadores, como o conhecido *ENIAC* (*Electronic Numerical Integrator Analyzer and Computer*), composto por cerca de 18 mil válvulas e capaz de processar 5000 adições, 357 multiplicações e 38 divisões por segundo. Ele foi utilizado para propósitos militares em cálculos de trajetórias de mísseis e codificação de mensagens secretas, porem eram extremamente caros e consumiam muita energia e muito espaço.

O primeiro transistor foi criado em Dezembro de 1947 pelos pesquisadores da empresa *Bell Laboratory* anunciando uma nova era da eletrônica de estado sólido. Esta invenção surgiu para substituir as válvulas, pois consumiam uma ínfima quantidade de energia e realizavam a mesma tarefa em muito menos tempo. [8] Os transistores eram feitos de germânio, um semicondutor metálico, porém na década de 50 se descobriu que o silício oferecia uma série de vantagens sobre o germânio. Em 1955, transistores de silício já eram comercializados.

Pouco tempo depois passaram a utilizar a técnica de litografia óptica, que faz uso da luz, máscaras e alguns produtos químicos para esculpir o silício na fabricação do transistor permitindo alcançar níveis incríveis de miniaturização e possibilitando o surgimento do circuito integrado, que são vários transistores dentro do mesmo encapsulamento formando um sistema lógico complexo. A invenção do circuito integrado se mantém historicamente como uma das inovações mais importantes da humanidade. Quase todos os produtos modernos utilizam essa tecnologia. [8]

3. EVOLUÇÃO

Em 14 de abril de 1965 o fundador da Intel, Gordon Moore, publicou na revista Electronics Magazine um artigo sobre o aumento da capacidade de processamento dos computadores. Moore afirma no artigo [5] que essa capacidade dobraria a cada 18 meses e que o crescimento seria constante. Essa teoria ficou conhecida como a "*Lei de Moore*" e se mantem válida até os dias de hoje.

3.1 1971

O primeiro microchip comercial produzido no mundo foi o Intel 4004™, figura 2, que foi desenvolvido para ser utilizado por uma empresa de calculadoras portáteis, a Japonesa Busicom. Até então, os dispositivos eletrônicos possuíam diversos chips separados para controle de teclado, display, impressora e outras funções, já o Intel 4004™continha todas essas funcionalidades em um único chip. Por esse e outros motivos ele é considerado o primeiro processador do mundo.

Com uma CPU de 4 bits e cerca de 2300 transistores, tinha tanto poder de processamento quanto o *ENIAC*, que ocupava mais de 900m^3 com suas 18.000 válvulas.

3.2 1973

Em 1973 a Intel lança seu novo processador, o Intel 8008™, que possuía uma CPU de 8 bits implementada sobre as tecnologias TTL MSI e com aproximadamente 3.500 transistores. Sua nomenclatura foi definida com base no marketing, por ser o dobro do Intel 4004™. [3]

3.3 1974

Menos de um ano depois do lançamento do Intel 8008™, em 1974, a Intel lança o primeiro processador voltado para computadores pessoais. O Intel 8080™, com 4.800 transistores, herdava várias características do seu predecessor Intel 8008™, possuindo também uma CPU de 8 bits, porém, com uma frequência de operação maior, era capaz de executar 290.000 operações por segundo, oferecendo uma performance cerca de 10 vezes maior que seu predecessor. Ele foi considerado o primeiro processador do mundo verdadeiramente de propósito geral. Enquanto o Intel 4004™e o Intel 8008™usavam a tecnologia P-channel MOS, o Intel 8080™inovou com a utilização de um processo N-channel, resultando em maiores ganhos de velocidade, consumo de energia, densidade do projeto e capacidade de processamento. [3]

3.4 1978

Após o grande sucesso do processador Intel 8080™, que tornou viável a comercialização de computadores pessoais, a Intel investe em pesquisas para produzir o seu primeiro processador com uma CPU de 16 bits. Em 1978, o Intel 8086™é lançado, contendo 29.000 transistores, sua performance era 10 vezes maior que o Intel 8080™, com frequência de 8MHz. [3]

Foi por volta de 1978 que a AMD surge no mercado de microprocessadores, conseguindo a licença para produzir hardware construído de acordo com a especificação dos processadores x86, incluindo direitos de produzir o hardware 286 e derivado do 286. Deste ano em diante a Intel passa a ter que dividir seu mercado com uma grande concorrente. [1]

3.5 1979

O Intel 8086™foi seguido em 1979 pelo Intel 8088™, uma versão do 8086 com barramento de 8 bits. As encomendas para os novos chips crescia constantemente, mas uma poderosa concorrente desenvolveu um processador que possuía vantagens em diversos pontos chave do seu design. Foi então que a Intel lançou uma campanha para fazer da arquitetura 8086/8088 o padrão da indústria de computadores. [3] A escolha do Intel 8088 como a arquitetura do primeiro computador pessoal da IBM foi uma grande ajuda para a Intel. A estratégia da IBM era criar um padrão "aberto"de sistema computacional baseado no modelo de microprocessador da Intel. Esse padrão aberto é capaz de fornecer uma transição fácil à geração seguinte de microprocessadores, existindo assim a compatibilidade de softwares entre as gerações diferentes de microprocessadores. Isso fez com que a Intel conseguisse consolidar a especificação da arquitetura 8086/8088 como o padrão mundial de 16 bits. [3]

3.6 1982

A próxima geração da família Intel 8086™inicia em 1982 com o lançamento do novo processador de 16 bits, o Intel 80286™, mais conhecido como Intel 286™. Ele possuía 134.000 transistores e estava tecnologicamente muito distante dos anteriores, com uma frequência máxima de 12 MHz. Porem, manteve a compatibilidade com os softwares criados para seus predecessores. Visando satisfazer as necessidades do mercado de processadores de 16-bits, que estava mais exigente quanto ao desempenho para gerenciar, desde redes de locais até dispositivos gráficos coloridos, o Intel 286™era multitarefa e possuía uma função de segurança embutida que garantia a proteção dos dados. [3]

Neste mesmo ano a AMD consegue terminar e lançar seu processador baseado no Intel 286™, o Am286®. Como ela implementava microprocessadores baseados em tecnologias criadas pela Intel, a AMD estava sempre um passo atrás de sua concorrente. Porem, o Am286® possuía alguns recursos interessantes que o Intel 286™não era capaz de fazer. Ele tinha um emulador EMS(Expanded Memory Specification) e a capacidade de sair do modo de proteção. Ele era formado por 134.000 transistores e com frequência máxima de 16MHz. [1]

3.7 1985

Em 1985, após uma grande crise mundial da industria de semicondutores e do mercado de eletrônicos, a Intel lança a grande inovação da década, o processador de 32 bits. Com 275.000 transistores, o Intel 386™operava a uma velocidade máxima de 5 milhões de instruções por segundo (MIPS) e frequência de 33MHz. [3]

Em sequência, a AMD lança o Am386®, sua versão do Intel 386™, que possuía 275.000 transistores, frequência máxima de 40 Mhz e uma CPU de 32 bits. [1]

3.8 1988

Apesar do Intel 386™ter sido uma grande revolução na industria de microprocessadores, ele era voltado para usuários comerciais – muito poderoso, e caro, para o usuário comum. Por esse motivo a Intel lança em 1988 o Intel 386SX™, chamado de "386 Lite". Esse processador representa a adição de um novo nível na família Intel 386™, com preço mais competitivo e, ao mesmo tempo, capaz de processar de 2,5 a 3 MIPS, sendo um upgrade natural ao Intel 286™. Ele também possuía uma vantagem distinta, podia rodar softwares de 32 bits. [3] Os processadores de 32 bits colocam a *Lei de Moore* [5] novamente em evidencia, acelerando ainda mais a corrida contra o tempo pela miniaturização dos transistores.

A exemplo da Intel, a AMD lança uma versão mais acessível aos usuários domésticos. [1]

3.9 1989

Em 1989, é lançada uma nova família de processadores. O Intel 486™possuía 1.200.000 transistores e foi o primeiro com um coprocessador matemático integrado e cache L1. Ele trabalhava a uma frequência máxima de 50MHz.

Como o Intel 486™, o Am486® da AMD é construído com um coprocessador matemático integrado. Porém, a frequência do seu barramento interno era de 40MHz, fazendo ele

Figura 3: Processador Intel Pentium®.

ser mais rápido que as primeiras versões do Intel 486™em diversos benchmarks. Ele proporcionou o início da popularidade da AMD.

3.10 1993

O lançamento do grande astro da Intel se deu em 1993. O Pentium® foi um marco na linha do tempo do avanço tecnológico, possuindo cerca de 3.100.000 transistores construídos com a tecnologia CMOS de 0.8μm. Em suas primeiras versões, trabalhava a uma frequência de 66MHz e executava cerca de 112 MIPS, posteriormente chegando aos 233MHz. Este processador incluía duas caches de 8Kb no chip e uma unidade de ponto integrada.

Após assistir ao lançamento do Pentium®, a AMD lança o Am586®, uma versão melhorada do Am486® que mesmo com sua frequência máxima de 150MHz e 1.600.000 transistores não era competitivo ao rival e não foi bem aceito pelos consumidores.

3.11 1995

A Intel investe no mercado de servidores lançando o Pentium® PRO. Ele introduziu a novidade da cache L2, rodava a 200MHz e possuía 5,5 milhões de transistores, sendo o primeiro processador a ser produzido com a tecnologia de 0.35μm. Neste mesmo ano a AMD decide sair da sombra da Intel e introduz o microprocessador AMD-K5®, que foi a primeira arquitetura concebida independentemente, porém com soquete compatível com microprocessador x86.

3.12 1997

A AMD sabia que estava perdendo a batalha contra a gigante Intel e resolveu colocar novas idéias em desenvolvimento, e foi neste momento que souberam que uma empresa de microprocessadores estava vendendo sua tecnologia. Essa empresa possuía um core revolucionário em seu estágio final de desenvolvimento. A conclusão do projeto desse novo core deu origem ao AMD-K6®, que oferecia um desempenho competitivo em aplicativos comerciais e desktop sem perder desempenho com o cálculo de ponto flutuante, que é uma funcionalidade essencial para os jogos e de algumas tarefas de multimídia. Esse processador possuía a tecnologia Intel® MMX™, que amplia a arquitetura do processador para melhorar seu desempenho de processamento multimídia, comunicação, numérico e de outras aplicações. Essa tecnologia usa um SIMD (single-instruction, multiple-data) técnica para explorar o paralelismo possível em muitos algoritmos, produzindo um desempenho total de 1,5 a 2 vezes mais rápido do que as mesmas aplicações executando no

Figura 4: Processador AMD K6-2®.

mesmo processador sem MMX. [7]

O AMD-K6 foi o processador mais rápido por alguns meses, mas a Intel já havia desenvolvido seu novo processador e tinha depósitos cheios dele, prontos para seu lançamento no final de 1997. É claro que, ao saber do sucesso do K6, a Intel decide antecipar o lançamento do Pentium® II.

Seguindo a *Lei de Moore* [5], o crescimento do desempenho dos processadores estava atingindo proporções incríveis, pois o Pentium® II possuía 7,5 milhões de transistores produzidos na tecnologia de 0.25μm e também incorporava a tecnologia Intel® MMX™. Foi introduzido também um chip de memória cache de alta velocidade. [4]

3.13 1998

O Intel® Pentium II Xeon® é concebido para satisfazer os requisitos de desempenho de médios e grandes servidores e estações de trabalho. Consistente na estratégia da Intel para prover, em um único processador, diversos produtos para segmentos de mercados específicos, o Xeon® possui características inovadoras e técnicas especificamente concebidas para estações de trabalho e servidores que utilizam aplicações profissionais exigentes, tais como serviços de Internet, armazenamento de dados corporativos, criação de conteúdos digitais e automação eletrônica e mecânica. Sistemas computacionais baseados nesse processador podem ser configurados para utilizar quatro, oito, ou mais processadores.

Neste ano surge o processador AMD-K6®-2, que acrescentou suporte para instruções SIMD (Single Instruction Multiple Data) e passou a usar uma forma mais avançada do Soquete 7, agora chamada Super Soquete 7. Esse novo formato acrescentava suporte para um barramento externo de 100 MHz. O AMD-K6-2 400 utilizou uma modificação de um multiplicador anterior, permitindo que ele operasse a 400 MHz mesmo em placas-mãe mais antigas. Operava em até 550MHz e foi o primeiro processador a incorporar a inovadora tecnologia AMD 3DNow!™, que proporciona uma excelente combinação de preço e desempenho, juntamente com um poderoso conjunto de instruções para processamento 3D. [1] O 3DNow!™é um conjunto de 21 novas instruções projetadas para acabar com tradicional gargalo no tratamento de ponto flutuante e aplicações multimídia. Ele foi criado quando a AMD decidiu projetar do zero as instruções para processamento de ponto flutuante e instruções do MMX para seu novo processador. [6] Logo depois, a AMD acrescentou ao núcleo do K6-2® 256 KB de cache L2 incorporado ao die, o que resultou em um aumento significativo da performance. Esse novo processador foi chamado de AMD-K6-3®. [1]

3.14 1999

Em 1999 a AMD toma a liderança na corrida pela performance, lançando o AMDK7®, ou AMD Athlon™, o primeiro processador com frequência acima de 1GHz. Com a criação do Atlhon, a AMD rompe de vez com a criação de chips compatíveis com os Intel. Os processadores AMD Athlon™foram projetados especificamente do zero para executar sistemas Windows com performance excepcional. Eles oferecem várias inovações que os destacam em benchmarks com os produtos equivalentes da Intel e representam a primeira grande vitória da AMD sobre a Intel no mercado. Para substituir a única unidade de FPU sem pipeline do AMD-K6, a AMD criou uma FPU com vários pipelines, capaz de executar várias instruções de ponto flutuante em paralelo. As gerações posteriores introduziram o cache L2 incorporado com o mesmo clock do processador. [1]

Estes processadores possuíam excepcionais 37 milhões de transistores, porém ainda utilizavam a tecnologia de fabricação de 0.25 μm e geravam muito calor. Pela primeira vez foi cogitado o fim da *Lei de Moore* [5], pois os processadores esbarraram na barreira térmica e ainda não existia tecnologia suficiente para diminuir ainda mais o tamanho dos transistores e consequentemente diminuir suas temperaturas.

Como resposta, a Intel lança o Pentium® III, que possuia 70 novas instruções, que aumentaram visivelmente o desempenho de gráficos avançados, 3D, streaming de áudio, vídeo e aplicações de reconhecimento de voz. Foi concebido para melhorar significativamente as experiências na Internet, permitindo aos usuários navegar em museus e lojas on-line e fazer download de vídeos de alta qualidade. Suas primeiras versões possuam 9.7 milhões de transistores operando a uma frequência de até 500MHz. Pouco antes, a Intel havia lançado uma nova versão do Pentium® II, que chega a 400MHz e foi o primeiro processador produzido com a nova tecnologia de 0.18μm.

No terceiro trimestre de 2000, o processador AMD Athlon XP incluiu as instruções SSE, e a AMD tornou-se o primeiro fabricante de processadores de uso geral a suportar a memória DDR.

3.15 2000

No ano 2000 as novas tecnologias desenvolvidas, como a fabricação de transistores com fios de 0.18μm, deram novo folego para a *Lei de Moore* [5] e possibilitaram a criação de processadores muito mais rápidos, com menos consumo de energia e dissipação de calor.

Foi baseado nessa tecnologia de fabricação que a Intel lança o Pentium® 4, um dos processadores mais vendidos na história. Com 42 milhões de transistores, suas primeiras versões chegavam a 1,5 Ghz de frequência, possibilitando usar computadores pessoais para edição de vídeos profissionais, assistir filmes pela internet, comunicar-se em tempo real com vídeo e voz, renderizar imagens 3D em tempo real e rodar inúmeras aplicações multimídia simultaneamente, enquanto navega na internet.

3.16 2001 a 2006

No período de 2001 a 2006, surgem diversas novas tecnologias e processadores, sempre confirmando a previsão de Gordon Moore [5]. Contudo, a partir de 2001, após o lançamento de novas versões do Pentium 4, a industria já podia construir processadores com transistores tão pequenos (tecnologia de 0.13μm) que tornou-se muito difícil aumentar o clock por limitações físicas, principalmente porque o calor gerado era tão grande que não podia ser dissipado pelos resfriadores convencionais.

Intel e AMD desenvolveram suas próprias arquiteturas 64 bits, contudo, somente o projeto da AMD (x86-64 AMD64) foi vitorioso. O principal fato para isso ter acontecido foi porque a AMD evoluiu o AMD64 diretamente do x86-32, enquanto que a Intel tentou criar o projeto (Itanium) do zero.

Com o sucesso do Athlon 64, o primeiro processador de 64 bits, as duas empresas criaram um acordo no uso desta arquitetura, onde a AMD licenciou a Intel para o uso do padrão x86-64. Logo, todos os modelos de processadores 64 bits atuais rodam sobre o padrão x86-64 da AMD.

Em 2004 surge a tecnologia de fabricação de 90nm, que possibilitou o lançamento do Intel Pentium M, para maior economia de energia em dispositivos móveis, e novas versões do AMD Athlon 64 mais econômicas e estáveis.

3.17 2006 – Na era do multi core

As barreiras térmicas que atrasavam o avanço dos processadores levaram os fabricantes a criar novas saídas para continuar desenvolvendo novos produtos com maior poder de processamento que os anteriores. Uma das saídas mais palpáveis foi colocar vários núcleos em um mesmo chip. Esses novos processadores ficaram conhecidos como multi core.

O primeiro processador dessa categoria foi o Intel Pentium® D, que nada mais é do que dois núcleos de Pentium 4 em um mesmo chip com adaptações para o compartilhamento do barramento. Suas melhores versões eram produzidas com a tecnologia de 65nm, possuía 2MB de cache de L2 por núcleo e seu barramento tinha frequência de 800MHz.

Porem, mais uma vez a AMD sai ganhando com o lançamento do seu primeiro multi core, o Athlon 64 X2, que tinha muitas vantagens sobre o Pentium D, como o HyperTransport. A tecnologia HyperTransport é uma conexão ponto-a-ponto de alta velocidade e baixa latência, projetada para aumentar a velocidade da comunicação entre os circuitos. Ela ajuda a reduzir a quantidade de barramentos em um sistema, o que pode diminuir os gargalos e possibilitar que os microprocessadores utilizem a memória de forma mais eficiente. [2]

A Intel lança sua nova linha de processadores multi core e deixa o Athlon 64 X2 para traz. Essa nova linha abandona a marca Pentium® e passa a utilizar a Core2®, trazendo também algumas melhorias que tornariam a Intel novamente a líder de mercado.

Com as mais novas tecnologias de fabricação de processadores, agora com transistores de 45nm (e diminuindo), os fabricantes investem em chips com mais e mais cores. Lançamentos recentes para desktops chegam a possuir 4 cores (Intel Core2 Quad e AMD Phenom™X4) e para servidores 6 cores (AMD Opteron™Six-Core), enquanto já existem pesquisas em desenvolvimento na AMD e Intel para produzir processadores com dezenas de cores em um único chip.

4. CONCLUSÃO

A *Lei de Moore* [5] foi a grande responsável pelo vertiginoso crescimento da capacidade de processamento dos processadores. Porem, a competição pela liderança do mercado entre Intel e AMD, as duas maiores empresas do ramo de microprocessadores para computadores pessoais, também contribuiu para essa incrível e rápida evolução. A evolução na tecnologia de fabricação de transistores chegou a níveis de miniaturização tão grandes que, provavelmente, não é mais possível diminui-los, contradizendo a *Lei de Moore*. Porém, a industria encontrou uma saída engenhosa para continuar o crescimento da capacidade de processamento, até que a ciência crie uma nova tecnologia de fabricação de transistores ou um novo paradigma de processadores.

5. REFERÊNCIAS

[1] AMD. A evolução da tecnologia. Acessado em 14 de Junho 2009 de http://www.amd.com.

[2] AMD. Hyper transport. Acessado em 16 de Junho 2009 de http://www.amd.com.

[3] Intel. 20 years - intel: Architect of the microcomputer revolution. Corporate Anniversary Brochures, Acessado em 12 de Junho 2009 de http://www.intel.com/museum.

[4] Intel. Intel museum. Acessado em 16 de Junho 2009 de http://www.intel.com/museum.

[5] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 19 1965.

[6] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! Technology: Architecture and implementations. *IEEE MICRO*, 19(2):37–48, MAR-APR 1999.

[7] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE MICRO*, 16(4):42–50, AUG 1996.

[8] B. Schaller. The origin, nature, and implications of "moore's law"the benchmark of progress in semiconductor electronics, 1996. Acessado em 16 de Junho 2009 de http://research.microsoft.com.

Arquiteturas Superescalares

Ricardo Dutra da Silva (RA 089088)
Instituto de Computação
UNICAMP
Campinas, Brasil
rdutra@ic.unicamp.br

RESUMO

Processadores superescalares exploram paralelismo em nível de instruções de maneira a capacitar a execução de mais de uma instrução por ciclo de clock. Este trabalho discute características de projetos de arquiteturas superescalares para aumentar o paralelismo em programas sequenciais. As principais fases descritas relacionam-se com a busca e processamento de desvios, determinação de dependências, despacho e emissão de instruções para execução paralela, comunicação de dados e commit de estados na ordem correta, evitando interrupções imprecisas. Exemplos de processadores superescalares são usados para ilustrar algumas considerações feitas para essas fases de projeto.

1. INTRODUÇÃO

Em meados da década de 1980, processadores superescalares começaram a aparecer [2, 5, 6, 7, 9, 15]. Os projetistas buscavam romper a barreira de uma única instrução executada por ciclo de clock [13].

Processadores superescalares decodificam múltiplas instruções de uma vez e o resultado de instruções de desvio condicional são geralmente preditas antecipadamente, durante a fase de busca, para assegurar um fluxo ininterrupto. Instruções são analisadas para identificar dependências de dados e, posteriormente, distribuídas para execução em unidades funcionais. Conforme a disponibilidade de operandos e unidades funcionais, as instruções executam em paralelo, possivelmente fora da ordem sequêncial do programa (*dynamic instruction scheduling*). Após as instruções terminarem, os resultados são rearranjados na sequência original do programa, permitindo que o estado do processo seja atualizado na ordem correta do programa.

Pode-se dizer que processadores superescalares procuram remover sequenciamentos de instruções desnecessários, mantendo, aparentemente, o modelo sequêncial de execução [3, 10]. Os elementos essenciais de processadores superescalares são: (1) busca de várias instruções simultaneamente, possivelmente predizendo *branchs*; (2) determinação de dependências verdadeiras envolvendo registradores; (3) despacho de múltipla instruções; (4) execução paralela, incluindo múltiplas unidades funcionais com pipeline e hierarquias de memória capazes de atender múltiplas referências de memória; (5) comunicação de dados através da memória usando loads e stores e interfaces de memória que permitam o comportamento de desempenho dinâmico ou não previsto de hierarquias de memória; (6) métodos para *commit* do estado do processo, em ordem.

Um programa pode ser visto como um conjunto de blocos básicos, com um único ponto de entrada e um único ponto de saída, formados por instruções contíguas [8]. Instruções em um bloco básico serão eventualmente executadas e podem ser iniciadas simultaneamente em uma *janela de execução*. A janela de execução representa um conjunto de instruções que pode ser considerado para execução em paralelo, conforme a dependência de dados. Instruções na janela de execução estão sujeitas a dependências de dados (RAW, WAR, WAW).

Para obter paralelismo maior do que aquele que um bloco básico está sujeito, pode-se usar predição de *branchs* e despacho especulativo. Desta maneira, são superadas dependências de controle. Se a predição é correta, o efeito das instruções sobre o estado do programa sai de especulativo e torna-se efetivo. Se a predição é incorreta, devem ser realizadas ações de recuperação para não alterar incorretamente o estado do processo. Resolvidas dependências de controle e dependências de nome, as instruções são despachadas e iniciam a execução em paralelo.

O hardware rearranja as instruções para execução paralela. Isso é feito considerando-se restrições de dependências verdadeiras e de hardware (unidades funcionais e data paths). Instruções possivelmente completam fora de ordem e, também, podem ser erroneamente especuladas. Para evitar que o estado do processo seja alterado erroneamente, uma estado temporário é mantido para os resultados de instruções. Esse estado temporário é mantido até a identificação de que uma instrução realmente seria executada em um modelo sequencial de processamento. Só então o estado do processo é atualizado através de *committing*.

Processadores superescalares mantém projetos lógicos com-

plexos. Mais de 100 instruções são comumente encontradas em fase de execução, interagindo entre elas e gerando exceções. A combinação de estados que podem ser gerados é enorme. Por esse motivo algumas arquiteturas escolhem mecanismos com técnicas em ordem, mais simples, para diminuir a complexidade e o tempo de lançamento no mercado [12]. Em [11] são avaliadas algumas limitações de processadores superescalares e apontadas características críticas de desempenho. Técnicas para verificação de modelos de processadores superescalares em relação ao seu conjunto de instruções são descritas em [4]

Neste trabalho são apresentada microarquitetura de processadores superescalares. Na seção 2 são discutidos o modelo e as etapas de um pipeline para processadores superescalares. Na seção 3, são descritas algumas considerações para projetos e, na seção 4, são discutidos alguns processadores superescalares. Na seção 5, as considerações finais são apresentadas.

2. ARQUITETURA SUPERESCALAR

A Figura 1 mostra um modelo de execução superescalar. O processo de busca de instruções, com predição de desvio, é usado para formar um conjunto dinâmico de instruções. Dependências são então verificadas e as instruções são despachadas (*dispatch*) para a janela de execução. Nesse ponto, as instruções não são mais sequenciais, mas possuem certa ordenação causada por dependências verdadeiras. As instruções são então emitidas (*issue*) da janela conforme a ordem das dependências verdadeiras e a disponibilidade de recursos de hardware. Após a execução, as instruções são novamente colocadas em ordem sequencial e atualizam o estado do processo.

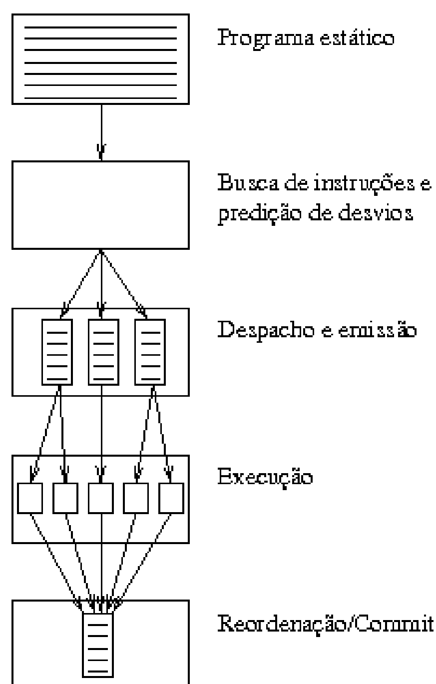


Figure 1: Execução em processadores superescalares.

2.1 Busca de instruções e predição de desvios

Para diminuir a latência de buscas são, em geral, usadas *caches de instruções*. As caches de instruções mantém blocos contendo instruções consecutivas. A arquitetura superescalar deve ser capaz de buscar, a partir da cache, múltiplas instruções em um único ciclo. A separação de caches de dados e de instruções é um ponto quase essencial para habilitar essa característica. Mas há exceções como o PowerPC [12, 13].

O despacho de um número máximo de instruções pode ser impedido por situações como *misses* na cache. Comumente, um buffer de instruções é usado para armazenar instruções buscadas e diminuir o impacto quando a busca está parada ou existem restrições de despacho [13].

O primeiro passo para realizar desvios rápidos envolve o reconhecimento de instruções de desvio. Uma maneira de tornar o processo rápido é armazenar, na cache de instruções, informações de decodificação. Essas informações são obtidas a partir de pré-decodificação das instruções durante a busca. Desvios condicionais costumam depender de dados não disponíveis no momento de decodificação. Os dados podem ainda ser provenientes de instruções anteriores. Nesses casos, podem ser usados métodos de predição.

O reconhecimento de desvios, modificação do PC (*Program counter*) e busca de instruções, a partir do endereço alvo, podem gerar atrasos e resultar em paradas do processador. A solução mais comum para esse tipo de problema envolve o uso do buffer de instruções para mascarar o atraso. Em buffers mais complexos, tanto instruções para o caso de o desvio ser tomado quanto para o caso de não ser tomado são armazenadas. Algumas arquiteturas usavam desvios atrasados (*delayed branchs*).

2.2 Decodificação, renomeação e despacho de instruções

Esta fase envolve a remoção de instruções do buffer de instruções, tratamento de dependências (verdadeiras e de nome) e despacho[1] de instruções para buffers de unidades funcionais. Durante a decodificação são preenchidos campos de: (i) operação; (ii) elementos de armazenamento relacionados com os locais onde operandos residem ou residirão; (iii) locais de armazenamento de resultado.

Os elementos de armazenamento são gerenciados por lógicas de renomeamento que, em geral, podem ser de dois tipos. Na primeira, uma tabela faz o mapeamento de registradores lógicos para registradores físicos. O número de registradores físicos é maior do que o número de registradores lógicos. Os mapeamento lógico-físico é realizado através de uma lista de registradores livres, como mostrado na Figura 2. Caso não haja registradores livres, o despacho é parado momentaneamente até algum registrador seja liberado. A renomeação é feita na ordem do programa [13].

[1]Despacho foi usado como tradução para a palavra *dispatch*, que envolve o envio de instruções para filas de unidades funcionais e estações de reserva. A palavra emissão foi usada para *issue*, que envolve o envio de instruções para unidades funcionais.

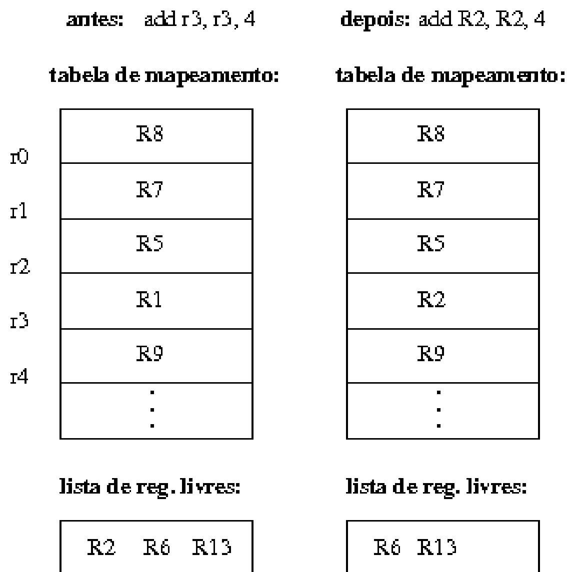


Figure 2: Renomeção de registradores. Registradores lógicos em letras maiúsculas e registradores físicos em letras minúsculas.

Um registrador físico deve ser liberado após a última referência feita a ele. Uma possível solução é incrementar um contador toda vez que uma renomeação de registrador fonte é feita e decrementar o contador sempre que uma instrução for emitida e ler o valor do registrador. Quando o contador chega a zero o registrador é liberado. Outro método, mais simples, espera até que uma instrução que renomeie o registrador receba commit.

O segundo método de renomeação é o *Reorder Buffer* [8]. O número de registradores lógicos e físicos é igual. O *reorder buffer* mantém um entrada para cada instrução despachada mas que ainda não recebeu *commit*. O nome vem do fato que o buffer mantém a ordenação das instruções para interrupções precisas. Pode-se pensar no *reorder buffer* como uma fila circular (Figura 3).

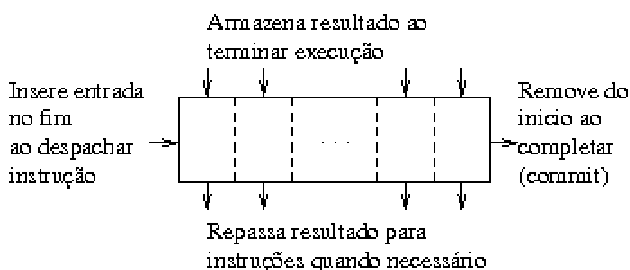


Figure 3: Modelo de *reorder buffer*. Entradas são inseridas e retiradas em ordem de fila. Resultados podem ser armazenados e lidos a qualquer momento.

A medida que instruções são despachadas na ordem sequencial do programa, elas são relacionadas a uma entrada no fim do *reorder buffer*. Os resultados de instruções que completam a execução são enviados para as suas entradas respectivas. Assim que uma instrução chega ao início da fila, seu resultado é escrito em um registrador e a instrução é retirada do *reorder buffer*. O despacho é interrompido se o reorder buffer está cheio. A Figura 4 mostra um exemplo de renomeação.

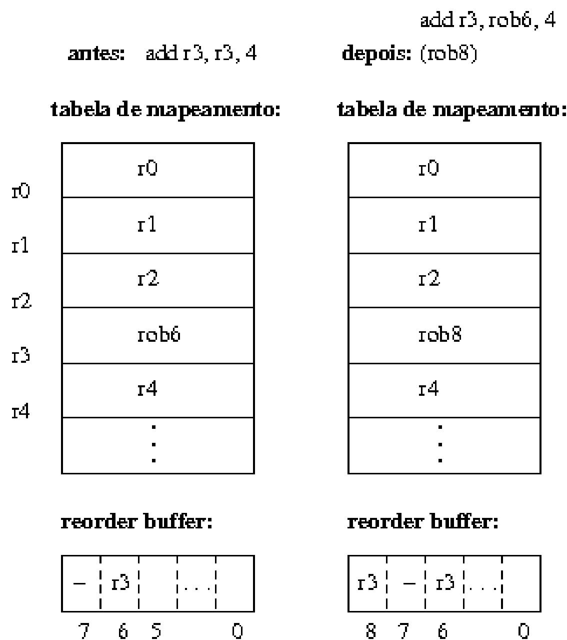


Figure 4: Exemplo de renomeação usando *reorder buffer*.

2.3 Emissão de instrução e execução paralela

Após a decodificação, é preciso identificar quais instruções podem ser executadas. Assim que os operandos estão disponíveis, a instrução está pronta para entrar em execução. No entanto, ainda á preciso verificar a disponibilidade de unidades funcionais, portas do arquivo de registradores ou do *reorder buffer*.

Três modos de organizar buffers para emissão de instruções são: *Método de Fila Única*, *Método de Filas Múltiplas* e *Estações de Reserva* [13]. No Método de Fila Única, apenas uma fila é usada, sem emissão fora de ordem. Não é preciso renomeamento de registradores. A disponibilidade de registradores é gerenciada através de bits de reserva para cada registrador. Um registrador é reservado quando uma instrução que o usa é emitida e liberado quando a instrução completa. Uma instrução pode ser emitida se os registradores de seus operandos não estão reservados.

No Método de Filas Múltiplas, instruções em uma mesma fila são emitidas em ordem, mas as diversas filas (organizadas conforme o tipo de instrução) podem emitir instruções fora de ordem.

Estações de reserva permitem que instruções sejam emitidas fora de ordem. Todas as estações de reserva monitoram simultaneamente a disponibilidade de seus operandos. Quando uma instrução é despachada para uma estação de reserva, os operandos disponíveis são armazenados nela. Caso os operandos não estejam disponíveis, a estação de reserva

espera o operando da operação que irá escrevê-lo. A instrução é emitida quando todos os operandos estão disponíveis. Também podem ser usados apenas apontadores para os locais onde encontram-se os operandos (registradores, reorder buffer) [8]. As estações de reserva podem ser divididas conforme o tipo de instrução ou colocadas juntas em um único bloco.

2.4 Operações de memória

Operandos de instruções de memória, ao contrário do que acontece com operações de ULA, não são conhecidos durante a decodificação. A determinação do local de acesso requer uma adição para formar o endereço, o que é feito na fase de execução. Após o cálculo do endereço, pode ainda ser necessária uma tradução (*translation*) para gerar o endereço físico. Esse processo é comumente acelerado por uma TLB. Uma vez obtido o endereço válido, a operação pode ser submetida para a memória. Existe a possibilidade de traduções e acessos serem realizados simultaneamente.

Para executar operações de memória mais rapidamente, podem ser usadas técnicas como: redução da latência, execução de múltiplas instruções ao mesmo tempo, execução sobreposta de operações de memória e operações sem acesso a memória e, possivelmente, execução de operações de memória fora de ordem [14]. Múltiplas requisições à memória exigem hierarquia com múltiplas portas, possivelmente apenas no primeiro nível, uma vez que acessos não costumam subir na hierarquia de memória [13]. Modos de implementação incluem o uso de células de memória com múltiplas portas, o uso de múltiplos bancos de memória ou a capacidade de realizar múltiplas requisições seriais em um mesmo ciclo.

A sobreposição de operações de memória com outras operações, sejam de memória ou não, exige uma hierarquia de memória não bloqueante. Dessa maneira, se houver *miss* em uma operação, outras operações prosseguem. Além disso, para permitir que instruções de memória sejam sobrepostas ou procedam fora de ordem, deve-se assegurar que *hazards* são tratados e que a semântica sequencial é preservada. *Store address buffers*, que mantêm endereços de todas as operações de store pendentes, são usados para assegurar a submissão de operações à hierarquia de memória sem violações de hazard. Antes de uma instrução load/store ser emitida para a memória, o *store buffer* é verificado em busca de instruções store pendentes para o mesmo endereço.

2.5 Commit

A fase de *commit* permite manter os efeitos das instruções como se a execução fosse sequencial. Duas técnicas são comumente usadas para recuperar estados precisos. Ambas mantêm um estado enquanto a operação executa e outro estado para recuperação.

A primeira técnica usa *checkpoints*. O estado da máquina é salvo em determinados pontos enquanto instruções executam e, também, quando uma estado preciso é necessário. Estados precisos são recuperados de um *history buffer*. Na fase de *commit* são eliminados estados do *history buffer* que não são mais necessários.

A segunda técnica divide em dois o estado da máquina: estado físico e estado lógico. O estado físico é atualizado assim que as instruções completam. O estado lógico é atualizado na ordem sequencial do programa, assim que os estados especulativos são conhecidos. O estado especulativo é mantido em um *reorder buffer* que, após o *commit* de uma instrução, é movido para os registradores ou para a memória. A técnica com *reorder buffer* é mais popular pois, além de proporcionas estados precisos, ajuda a implementar a renomeação de registradores.

2.6 Software

Compiladores podem aumentar a possibilidade de paralelismo em um grupo de instruções, permitindo que sejam emitidas simultaneamente. Outra maneira é manter espaçamento de instruções dependentes para evitar paradas no pipeline. Isso pode ser feito com *scheduling estático* [8].

3. CONSIDERAÇÕES DE PROJETOS SUPERESCALARES

Em [12] é apresentada uma comparação entre duas vertentes de projetos: processadores superescalares desenvolvidos para obter menores tempos de clock e processadores superescalares projetados para maiores taxas de despacho.

Processadores que priorizam projetos de velocidade possuem pipelines profundos. A compensação para maiores frequências de clock é, em geral, obtida com menores taxas de despacho e maiores latências para loads e identificação de predições errôneas (DEC Alpha 21064).

Processadores projetados com taxas de despacho maiores procuram proporcionar um maior número de tarefas feitas por ciclo de clock. Características desses projetos são baixas penalidades para loads e mecanismos que permitem execução de instruções dependentes (IBM POWER).

Seis categorias de sofisticação são apresentadas entre os processadores. (1) Co-processadores de ponto-flutuante que não despacham várias instruções inteiras em um ciclo, nem mesmo uma instrução inteira e uma de desvio. Ao invés disso, são feitas emissões de uma instrução de ponto-flutuante e uma de inteiro. O desempenho é obtido em códigos de ponto-flutuante, permitindo que unidades de inteiros executem loads e stores de ponto-flutuante necessários (MIPS R5000). (2) Processadores que permitem despacho conjunto de instruções de inteiros e desvios, melhorando a performance em códigos de inteiros (HyperSPARC). (3) Múltiplas emissões de inteiros e instruções de memórias são permitidas com a inclusão de múltiplas unidades de inteiros (Intel Pentium). (4) Processadores que usam ULAs de três entradas para permitir despacho múltiplo de instruções inteiras dependentes (Motorola 68060). (5) Processadores que enfatizam exceções precisas. São usados mecanismos de recuperação e múltiplas unidades funcionais, com pouca ou nenhuma restrição de despacho (Motorola 88110). (6) Processadores que permitem despacho fora de ordem para todas as instruções (Pentium Pro).

4. MICROPROCESSADORES SUPERESCALARES

Nesta seção serão descritos alguns processadores superescalares, enfatizando aspectos de projeto característicos de cada processador.

4.1 Motorola 88110

O Motorola 88110 era um processador de emissão dupla com extensões para gráficos [6]. Uma estação de reserva era usada para branchs e três estações para stores. Desta maneira, o processador emitia instruções em ordem, com exceção de branchs e stores. Não era usada renomeação e permitia-se a emissão de instruções com dependências WAR. Stores podiam ser emitidos juntamente com as instruções que calculavam seu resultado. A unidade de store/load continha um buffer de quatro entradas.

Desvios eram preditos usando uma cache de *target instruction*, que matinha um registro dos últimos 32 desvios. Essa cache era indexada por endereços virtuais e precisava ser esvaziada em trocas de contexto. Instruções emitidas especulativamente eram rotuladas e anuladas em caso de predições erradas. Qualquer registrador escrito por instruções preditas erroneamente era restaurado a partir do *history buffer*. Stores condicionais, no entanto, não atualizavam a cache de dados, eram mantidos nas estações de reserva até que o desvio fosse resolvido.

As unidades funcionais eram dez: duas ULAs, somador, multiplicador, divisor, *bit-field*, *instruction/branch*, *data-cache* e duas *graphics*. Para manter exceções precisas e se recuperar de desvios previstos erroneamente, o processador usava *histoy buffers* de instruções. As 10 unidades de funções compartilhavam dois barramentos de 80 bits. Esses barramentos eram disputados pelas instruções por causa da latências diferentes.

4.2 MIPS R8000

O MIPS R8000 tinha o objetivo de ser um processador para computações de ponto-flutuante. Para evitar *misses* em caches, foram separadas referências feitas às instruções de inteiros e de memória das referências às instruções de ponto flutuante. A busca de instruções era feita a partir de uma cache de memória de 16KB, diretamente mapeada e com entradas de 32 bytes. Quatro instruções eram buscadas por ciclo. A cache de instruções, preenchida por uma cache externa em 11 ciclos, usava indexadores e tags virtuais. Um único bit era responsável pela predição de desvios para cada bloco de quatro instruções na cache.

O processador emitia até quatro instruções por ciclo, usando oito unidades funcionais. Havia quatro unidades de inteiros, duas de ponto flutuante e duas para loads e stores. Instruções de ponto flutuante eram armazenadas em uma fila até que pudessem ser despachadas. Desta forma, pipelines de inteiros podiam continuar mesmo quando um load de ponto-flutuante, que demorava cinco ciclos, era emitido com instruções de ponto-flutuante dependentes.

As referências de dados inteiros usavam uma cache interna de 16 KBytes, enquanto instruções de ponto-flutuante usavam uma cache externa de 16 MBytes. Por causa das decisões de projeto, poderiam haver problemas de coerência caso dados de ponto-flutuante e dados inteiros fossem reunidos em uma mesma estrutura de dados, como *union* [9]. A cache interna evitava esse tipo de problema mantendo um bit de validade para cada palavra.

4.3 MIPS 10000

O MIPS 10000 [15] busca até quatro instruções, as quais são pré-decodificadas (quatro bits são usados para identificar o tipo da instrução) antes da inserção na cache de 512 linhas. A cache de instruções, *two-way set associative*, contém uma tag de endereços e um campo de dados. Uma pequena TLB de oito entradas mantém um subconjunto das traduções da TLB principal. Logo após a busca, são calculados os endereços de jumps e branches, que são, então, preditos. A tabela de predição, com 512 entradas de 2 bits, está localizada no mecanismo de busca de instruções. A janela do processador considera até 32 instruções em busca de paralelismo.

Ao tomar um desvio, um ciclo é gasto no redirecionamento da busca de instruções. Durante o ciclo, instruções para um caminho não-tomado do desvio são buscadas e postas em uma *resume cache*, para o caso de uma predição incorreta. A *resume cache* tem espaço para 4 blocos de instruções, o que permite que até 4 desvios sejam considerados em qualquer momento.

Quando um branch é decodificado, o processador salva seu estado numa pilha de branch com 4 entradas. O processador para de decodificar se um branch chega e a pilha está cheia. Se um branch é determinado incorreto, o processador aborta todas as instruções do caminho errado e restabelece o estado a partir da pilha de branch.

Após a busca, as instruções são decodificadas e seus operandos são renomeados. O despacho para a fila apropriada (memória, inteiros ou ponto-flutuante) é feito com base nos bits da pré-decodificação. O despacho é parado se as filas estiverem cheias. No despacho, um *busy-bit* para cada registrador físico de resultado é estabelecido como ocupado. O bit volta ao estado não ocupado quando uma unidade de execução escreve no registrador. Todos registradores lógicos de 32 bits são renomeados para registradores físicos de 64 bits usando *free lists* (Figura 2). As *free lists* de inteiros e ponto-flutuante são quatro listas circulares, paralelas, de profundidade oito. Isso permite até 32 instruções ativas.

Cada instrução, nas filas, monitora os *busy-bits* relacionados com seus operandos até que os registradores não estejam mais ocupados. Filas de inteiros e ponto-flutuante não seguem uma regra de FIFO, funcionam de forma similar a estações de reserva. A fila de endereço é uma FIFO circular que mantém a ordem do programa.

Existem cinco unidades funcionais: um somador de endereços, duas ULAs, uma unidade de ponto flutuante (multiplicação, divisão e raiz quadrada) e um somador de ponto flutuante. Os pipelines de inteiros ocupam um estágio, os de load ocupam dois e os de ponto flutuante ocupam três. O resultado é escrito nos registradores no estágio seguinte. Estados precisos são mantidos no momento de exceções com

um *reorder buffer*. Até quatro instruções por ciclo recebem *commit* na ordem original do programa.

A hierarquia de memória é implementada de modo não bloqueante com dois níveis de cache *set-associative*. Todas as caches usam algoritmo de realocação LRU. Endereços de memória virtual são calculados como a soma de dois registradores de 64 bits ou a soma de um registrador e um campo imediato de 16 bits. A TLB traduz esses endereços virtuais em endereços físicos.

4.4 Alpha 21164

O Alpha 21164 abdica das vantagens de *dynamic scheduling* e favorece taxas de clock altas. Quatro instruções são buscadas de uma vez em uma cache de 8 Kbytes. Desvios são preditos usando uma tabela associada com a cache de instruções. Existe uma entrada de *branch history* com um contador de dois bits para cada instrução na cache. Apenas um branch predito e não resolvido é permitido a cada momento. Se houver um outro desvio sem que um prévio seja resolvido, a emissão é parada.

Instruções são despachadas para dois buffers de instruções, cada um com capacidade para 4 instruções. As instruções são emitidas, a partir dos buffers, na ordem do programa. Um buffer precisa estar totalmente vazio antes do outro começar a emitir instruções, o que restringe a taxa de emissão mas simplifica muito a lógica de controle.

São quatro as unidades funcionais: duas ULAs inteiras, um somador de ponto-flutuante e um multiplicador de ponto-flutuante. As ULAs não são idênticas, apenas uma faz deslocamentos e multiplicações inteiras, a outra é a única que avalia desvios. Resultados que ainda não receberam commit são usados através de *bypassing*. Instruções de ponto-flutuante podem atualizar os registradores fora de ordem, o que permite que aconteçam exceções imprecisas.

Dois níveis de cache são usados no chip. Existe um par de caches primárias de 8 Kbytes, uma para instruções e outra para dados. A cache secundária, *3-way set associative*, de 96 Kbytes, é compartilhada por instruções e dados. A cache primária, diretamente mapeada, permite acesso com taxa de clock muito alto. Existe um *miss address file* (MAF), com seis entradas, que contém o endereço e registrador alvo para loads que têm *miss*. O MAF pode armazenar até 6 misses na cache.

4.5 AMD K5

O AMD K5 usa um conjunto complexo de instruções com tamanho variável, Intel x86 [5]. Por esse motivo, as instruções são determinadas sequencialmente. O processo é feito na pré-decodificação, antes de entrar na cache de instruções. Cinco bits são usados após a pré-decodificação para informar se o byte é o começo ou fim de uma instrução. Também são identificados bytes relacionados com operações e operandos. A taxa de busca de instruções na cache é de 16 bytes por ciclo, que são armazenados em uma fila de 16 elementos para o despacho.

A lógica de predição é integrada com a cache de instruções, com uma entrada por linha da cache. Apenas um bit é usado para indicar a última predição de desvio. A entrada de predição contém um apontador para a instrução alvo, indicando onde esta pode ser encontrada na cache de instruções. Com isso reduz-se o atraso para busca de uma instrução alvo predita.

Dois ciclos são gastos para a decodificação. O primeiro estágio lê os bytes da fila, convertendo-os em instruções simples (ROPs – *RISC-like Operations*). Até quatro ROPs são formadas por vez. Frequentemente, a conversão requer um conjunto pequeno de ROPs por instrução x86 e, neste caso, a conversão é feita em um único ciclo. A conversão de instruções mais complexas é feita através de buscas em uma ROM. Até quatro ROPs são geradas por ciclo através da ROM. Depois da conversão, as instruções são geralmente executadas como operações individuais, ignorando a relação original com as instruções x86.

Após o ciclo de decodificação, as instruções lêem os operandos disponíveis, em registradores ou no reorder buffer, e são despachadas para estações de reserva a uma taxa de até quatro ROPs por ciclo. Se os operandos não estão prontos, a instrução espera na estação de reserva.

Existem 6 unidades funcionais: duas ULAs, uma unidade de ponto-flutuante, duas unidades de load/store e uma unidade de desvio. Uma das ULAs faz deslocamentos e a outra pode fazer divisão de inteiros. As estações de reserva são divididas para cada unidade funcional. Exceto pela unidade de ponto-flutuante, que tem uma estação de reserva, as outras têm duas. Conforme a disponibilidade dos operandos, ROPs são emitidas para a unidade funcional associada. Existem portas de registradores e paths de dados suficientes para que até quatro ROPs sejam emitidas por ciclo.

Existe uma cache de 8 Kbytes com quatro bancos. Store e loads duais são permitidos, desde que sejam para bancos diferentes, com exceção do caso em que a referência é para a mesma linha e duas requisições podem ser servidas.

Para manter estados precisos em casos de interrupção, é usado um *reorder buffer* de 16 entradas. O resultado de uma instrução é mantido no reorder buffer até que possa ser colocado no arquivo de registradores. O reorder buffer possui *bypass* e também é usado para recuperar predições de desvio incorretas.

4.6 AMD Athlon

Usa algumas das abordagens dos processadores K5 e K6, diferenciando-se no uso de um pipeline mais profundo, MacroOps e manipulação especial para instruções de ponto-flutuante e instruções multimídia. A parte inicial do pipeline, relacionada com o despacho contém 6 estágios. A predição de branchs para essa primeira parte do pipeline utiliza uma BHT (*branch history table*) de 2048 entradas, uma BTAC (*Branch Target Address Cache*) de 2028 entradas e um pilha de retorno de 12 entradas [1].

A decodificação é feita por três decodificadores *DirectPath* que podem produzir uma MacroOps cada, ou, para instruções complexas, por um decodificador *VectorPath* que sequencia três MacroOps por ciclo. Bits de pré-decodificação

auxiliam a decodificação.

Uma MacroOp é uma representação de uma instrução IA32 com tamanho fixo e que pode conte uma ou duas operações. Para o pipeline de inteiros as operações podem ser de seis tipos: load, store, load-store combinados, geração de endereços, ULA, e multiplicação. Desta maneira, instruções IA32 de registrador para memória e instruções de memória para registrador pode ser representadas por uma única MacroOp. No pipeline de ponto-flutuante, as operações podem ser: multiplicação, adição ou miscelânea. A vantagem de MacroOps é a redução do número de entradas em buffer necessárias.

Durante o estágio de despacho, MacroOps são alocadas em um *reorder buffer* de 72 entradas chamado *instruction control unit* (ICU). O buffer é organizado em 24 linhas de três entradas cada. O pipeline de inteiros é organizado simetricamente com uma unidade de geração de endereços e uma unidade de funções de inteiros conectados a cada entrada. A Multiplicação de inteiros é a única operação assimétrica, localizada na primeiro entrada. Instruções multimídia e de ponto-flutuante tem maiores restrições para as entradas.

Da ICU, as MacroOps são colocadas em um planejador (*scheduler*) de inteiros, de 18 entradas organizadas em seis linhas de 3 entradas cada, ou no planejador de ponto-flutuante e multimídia, de 36 entradas organizadas em 12 linhas de três entradas cada. Operações de load e store são enviadas para uma fila de 44 entradas para processamento. Inteiros ainda usam uma IFFRF (*future file and register file*) de 24 entradas. Operandos e tags são lidos dessa unidade durante o despacho e os resultados são escritos na unidade e na ICU quando as instruções completam.

Ao invés de ler operandos e tags durante o despacho, referências a registradores de ponto-flutuante e multimídia são renomeadas usando 88 registradores físicos. Como os operandos não são lidos durante o despacho, um estágio extra para leitura de registradores físicos é necessário. A execução dessas instruções não começa até o estágio 12 do Athlon.

O Athlon contém uma L1 integrada de 64 Kbytes e, inicialmente, continha um controlador para uma L2 externa de até 8 MBytes. Posteriormente foram usadas L2 internas. A L1 contém múltiplos bancos, o que permite dois loads ou stores por ciclo.

4.7 Intel P6

A microarquitetura faz a decomposição de instruções IA32 em micro-instruções [7]. Um pipeline de oito estágios para busca e tradução aloca micro-instruções em um *reorder buffer* de 40 entradas e em uma estação de reserva de 20 entradas.

Até três instruções IA32 podem ser decodificadas em paralelo. No entanto, por características de desempenho, as instruções devem ser rearranjadas de maneira que que apenas a primeira gere até quatro micro-instruções e, as outras duas, apenas uma micro-instrução. Instruções IA32 com operadores na memória requerem múltiplas instruções e limitam a taxa de decodificação para uma instrução por ciclo.

O preditor de branchs é adaptativo, de dois níveis, e quando o branch não é encontrado na tabela, um mecanismo é usado para fazer a predição baseando-se no sinal do deslocamento.

A estação de reserva é escaneada em modo de fila a cada ciclo, na tentativa de emitir até quatro micro-instruções para cinco portas. A primeira pota de emissão está ligada a seis unidades funcionais: inteiros, soma de ponto-flutuante, multiplicação de ponto-flutuante, divisão de inteiros, divisão de ponto-flutuante e deslocamento de inteiros. A segunda porta cuida de uma segunda unidade de inteiros e de uma unidade de branch. A terceira porta é dedicada para loads, enquanto as portas quatro e cinco cuidam de stores.

4.8 Pentium 4

O projeto é semelhante ao P6. Uma cache de instruções decodificadas, chamada *trace cache*, foi introduzida. A *trace cache* é organizada em 2048 linhas de seis micro-instruções cada. A largura de banda para busca na *trace cache* é de três micro-instruções por ciclo.

A profundidade do pipeline é maior do que no P6, 30 ou mais estágios. O pipeline para predição errada de desvios contém 20 estágios (o P6 continha 10). Dois preditores de branch são usados, uma para o início do pipeline e outro para a *trace cache*. A primeira BTB (*Branch target buffer*) contém 4096 entradas e usa um esquema híbrido [8].

O Pentium 4, ao contrário do P6, não armazena valores de fonte e resultado nas estações de reserva e no *reorder buffer*. Ao invés disso, são usados 128 registradores físicos para renomeação dos registradores inteiros e mas 128 registradores físicos para renomeação da pilha de pontos-flutuantes.

As micro-instruções são despachadas para duas filas. Uma é usada para operações de memória e a outra para as operações restantes. São usadas quatro portas de emissão, duas para load e store e duas para as outras operações. As duas últimas portas contém planejadores que podem emitir uma instrução por ciclo e outros planejadores que podem emitir duas micro-operações de ULA por ciclo. ULAs de inteiros usam pipelines que operam em três meio-ciclos, com dois meio-ciclos do pipeline de execução para cada atualização. *Offsets* de endereços do ponteiro de pilha são ajustados quando necessário para micro-operações de load e store que referenciam a pilha. Um *history buffer* grava as atualizações especulativas do ponteiro de pilha no caso de um branch predito errado ou no caso de uma exceção.

4.9 Pentium M

O Pentium M usa duas extensões para previsão de desvios. Um detector de loops captura e armazena contagens de loops em um conjunto de contadores em hardware. Isso produz predições precisas para loops *for*. Uma segunda extensão é um esquema adaptativo para desvios indiretos, projetado para desvios dependentes de dados. Para desvios indiretos preditos de forma errada, são alocadas entradas novas correspondentes ao conteúdo corrente de registradores de *global history*. Desta forma o *global histoy* pode ser usado para escolher entre várias instâncias de preditores para desvios indiretos dependentes de dados.

4.10 POWER4

Cada processador possui dois *cores* com oito unidades funcionais e uma cache L1. A cache L2 é compartilhada juntamente com o controlador da cache L3 (*off-ship*). Os cores emitem oito instruções por ciclo e a arquitetura considera maior desempenho através de um pipeline profundo, ao contrário do um pipeline menor com maiores taxas de emissão, usado anteriormente. Até 200 instruções permanecem simultaneamente no fluxo de execução.

A busca de instruções usa um preditor híbrido incomum de 1 bit. Um seletor escolhe entre 16K entradas de um preditor local e um preditor de 16K entradas global. O POWER4 usa grupos de cinco instruções com a quinta instrução sendo de desvio (nops são usados para completar *slots* não usados. Esses grupos são rastreados por uma tabela de 20 entradas, usada para rastrear quando instruções completam.

Apenas um grupo é despachado para as filas de emissão por ciclo. Um vez nas fila, as instruções podem ser emitidas fora de ordem. Onze filas de emissão são usadas, formando um total de 78 entradas. Bastantes registradores físicos são usados: 80 registradores físicos gerais, 72 registradores físicos para ponto-flutuante, 16 registradores físicos para *link* e *count*, 32 registradores físicos para campos de condições de registradores.

Nove estágios de pipeline são usados antes da emissão. Dois estágios são usados para busca das instruções e seis estágios são usados para quebrar instruções e formar grupos. Um estágio faz mapeamento de recursos e despacho. Instruções de inteiros requerem 5 estágios para execução incluindo emissão, leitura de operandos, execução, transferência, e escrita do resultado. Grupos completam em um último estágio. O total é um pipeline de 15 estágios para inteiros. Instruções de ponto-flutuante requerem mais cinco estágios.

5. CONCLUSÕES

Este trabalho apresentou os conceitos principais bem como a organização geral relacionada com processadores superescalares. Uma descrição histórica de alguns microprocessadores ainda foi apresentada, com o intuito de fornecer uma idéia de como as técnicas para processadores superescalares se apresentam em arquiteturas comercializadas.

Considerações de projeto envolvem cada uma das etapas em um pipeline superescalar. Durante a fase de busca usualmente são integradas caches de instruções e mecanismos de predição de desvios. O tamanho da cache e as lógicas de predição permitem ganhos de desempenho ao custo de mecanismos mais complexos para tradução de endereços e buscas antecipadas.

Os mecanismos de despacho apresentam desde restrições para despacho em ordem, que mantém estados mais precisos e simplificam circuitos, até restrições mais livres, que aumentam o paralelismo ao custo de projetos cuidadosos para manter a semântica do programa.

O conjunto de unidades funcionais varia bastante entre projetos diferentes. Pipelines profundos, em geral, são explorada para diminuir o ciclo de clock. Ciclos mais rápidos são comumente obtidos mantendo menores taxas de despacho, enquanto projetos com despachos mais agressivos mantém tempos de ciclos maiores para verificações de paralelismo entre instruções.

As limitações de desempenho das técnicas superescalares levaram à investigação de alternativas como VLIW (*Very Long Instruction Word*), EPIC (*Explicitly Parallel Instruction Computing*), SMT (*simultaneous multithreading*) e processadores multi-core.

6. REFERÊNCIAS

[1] AMD. AMD Athlon Processor – Technical Brief. Technical report, Advanced Micro Devices, Inc, 1999.

[2] P. Bannon and J. Keller. Internal architecture of alpha 21164 microprocessor. In *COMPCON '95: Proceedings of the 40th IEEE Computer Society International Conference*, page 79, Washington, DC, USA, 1995. IEEE Computer Society.

[3] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. *SIGPLAN Not.*, 26(6):241–255, 1991.

[4] J. R. Burch. Techniques for verifying superscalar microprocessors. In *DAC '96: Proceedings of the 33rd annual Design Automation Conference*, pages 552–557, New York, NY, USA, 1996. ACM.

[5] D. Christie. Developing the amd-k5 architecture. *IEEE Micro*, 16(2):16–26, 1996.

[6] K. Diefendorff and M. Allen. Organization of the motorola 88110 superscalar risc microprocessor. *IEEE Micro*, 12(2):40–63, 1992.

[7] L. Gwennap. Intel's p6 uses deeoupled supersealar design. *Microprocessor Report*, 9(2), 1995.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, May 2002.

[9] P. Y.-T. Hsu. Design of the R8000 microprocessor. Technical report, MIPS Technologies, Inc., 1994.

[10] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 272–282, New York, NY, USA, 1989. ACM.

[11] S. Palacharla, N. P. Jouppi, and J. E. Smith. Quantifying the complexity of superscalar processors, 1996.

[12] J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Publishing Company, New Delhi, 2005.

[13] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, December 1995.

[14] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. *SIGPLAN Not.*, 26(4):53–62, 1991.

[15] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

Trace Caches: Uma Abordagem Conceitual

Roberto Pereira, RA:089089
Instituto de Computação – IC
Universidade Estadual de Campinas – Unicamp
Av. Albert Einstein, 1251, Campinas - SP
robertop.ihc@gmail.com

ABSTRACT

Superscalar processors are constantly searching for improvements in order to become more efficient. The fetch engine of instructions is one of the bottlenecks of superscalar processors, because it limits the improvements that can be achieved trough the parallel execution of instructions. The Trace Cache is a solution that aims to achieve a high bandwidth in instruction fetch through the storage and reuse of traces of instructions that are dynamically identified. This paper presents a conceptual approach about trace cache and describes how it can help for improving the provision of instructions for execution on superscalar processors. We also present some related work and some proposals for improving the original proposal, as well as our final considerations on the study.

RESUMO

Os processadores superescalares estão numa busca constante por aperfeiçoamentos de modo a tornarem-se mais eficientes. O mecanismo de busca de instruções é considerado um dos principais gargalos dos processadores superescalares, limitando as melhorias que podem ser obtidas pela execução paralela das instruções. A *Trace Cache* é uma solução proposta para alcançar uma alta largura de banda na busca de instruções por meio do armazenamento e reutilização de *trace*s de instrução que são identificados dinamicamente. Neste artigo, apresentamos uma abordagem conceitual sobre *trace cache* e descrevemos como a mesma pode colaborar para melhorar o fornecimento de instruções para a execução em processadores superescalares. Também apresentamos alguns trabalhos relacionados e algumas propostas de aperfeiçoamento da proposta original, assim como nossas considerações finais sobre o estudo realizado.

Categorias e Termos Descritores

B.3.2 [**Hardware\Memory Structures**]: *Cache* Memories

Termos Gerais

Documentation, Performance, Design, Theory.

Palavras-Chave

Trace Cache, Desempenho, Superescalar, ILP.

1. INTRODUÇÃO

Os processadores superescalares possuem *pipelines* que permitem a execução de mais de uma instrução no mesmo ciclo de *clock*, ou seja, simultaneamente [1][2][3]. Para alcançar um alto desempenho (*performance*), a largura de banda de execução dos processadores modernos tem aumentado progressivamente. Entretanto, para que esse aumento na banda se converta em melhora de desempenho, a unidade de execução precisa ser provida com um número cada vez maior de instruções úteis por ciclo (IPC).

De acordo com Rotemberg *et al.* [1], a organização dos processadores superescalares possui dois mecanismos distintos e claramente divididos chamados de "busca" (*fetch*) e "execução" (*execution*) de instruções, separados por *buffers* (que podem ser filas, estações de reserva, etc.) que atuam como um banco de instruções. Neste contexto, o papel do mecanismo de busca é buscar as instruções, decodificá-las e colocá-las no *buffer*, enquanto o papel do mecanismo de execução é retirar e executar essas instruções levando em conta as suas dependências de dados e as restrições de recursos.

Segundo Rotemberg *et al.* [2], os *buffers* de instrução são coletivamente chamados de "janela de instrução" (*instruction window*) e, é nessa janela, que é possível se aplicar os conceitos do Paralelismo em Nível de Instrução (ILP) nos programas seqüenciais. Naturalmente, quanto maior for essa janela, maiores serão as oportunidades de se encontrar instruções independentes que possam ser executadas paralelamente. Assim, existe uma tendência no projeto de processadores superescalares em se construir janelas de instrução maiores e, ao mesmo tempo, de se prover uma maior emissão e execução de instruções. Além disso, normalmente, o mecanismo de execução de instruções nos processadores superescalares é composto por unidades funcionais paralelas [1], e o mecanismo de busca pode especular múltiplos desvios objetivando fornecer um fluxo de instruções contínuo à janela de instruções de modo a tirar o máximo de proveito do ILP disponível.

Entretanto, como argumentado em Rotemberg *et al.* [1] [2] e em Berndl & Hendren [4], para conseguir manter essa tendência de aumentar o desempenho aumentando a escala na qual as técnicas de ILP são aplicadas, é preciso encontrar um equilíbrio entre todas as partes do processador, uma vez que um gargalo em qualquer uma das partes diminuiria os benefícios obtidos por essas técnicas. Neste contexto, os trabalhos de [1] [2] [3] [4] e [5] citam claramente que ocorre um grande aumento na demanda pelo fornecimento de instruções para serem executadas e que isso, normalmente, se torna um gargalo que limita o ILP. Afinal, de que adianta duplicar recursos e largura de banda se apenas uma quantidade limitada e inferior de instruções pode ser buscada e decodificada simultaneamente? Ou seja, o pico da taxa de busca de instruções deve ser compatível com o pico da taxa de despacho de instruções.

Assim, o fornecimento de instruções se torna um elemento-chave no desempenho dos processadores superescalares [4], pois como mencionam Rotemberg *et al*. [2], as unidades de busca normalmente eram limitadas a uma única previsão de desvio por

ciclo. Conseqüentemente, não era possível buscar mais de um bloco básico[1] a cada ciclo de *clock*. Devido à grande quantidade de desvios existentes em programas típicos e ao tamanho médio pequeno dos blocos básicos[2], Postiff *et al.* [4] afirmam que buscar instruções pertencentes à múltiplos blocos básicos em um único ciclo é algo crítico para evitar gargalos e proporcionar um bom desempenho.

No que diz respeito a busca de instruções, buscar um único bloco básico a cada ciclo é suficiente para implementações que entregam no máximo até quatro instruções por ciclo. Entretanto, de acordo com Rotemberg *et al*. [1] [2], essa quantidade não é suficiente para processadores com taxas mais elevadas. Ao conseguir prever múltiplos desvios, torna-se possível buscar múltiplos blocos básicos contínuos em um mesmo ciclo. Porém, o limite da quantidade de instruções que podem ser buscadas ainda está relacionado à freqüência de desvios que são tomados, pois nesse caso, é preciso buscar as instruções que estão abaixo do desvio tomado no mesmo ciclo no qual o desvio foi buscado. É neste contexto que Rotemberg *et al*. [1] apresenta a proposta de *Trace Cache*.

O mecanismo de *Trace Cache* é uma solução para o problema de se realizar o *fetch* de múltiplos desvios em um único ciclo [4]. Ela armazena o rastreamento do fluxo de execução dinâmico das instruções, de modo que instruções que antes eram não contínuas (separadas por desvios condicionais) passam a aparecer de forma contínua. Segundo Rotemberg *et al.*[1], idealizadores da *trace cache*, a quantidade de desvios condicionais que podem ser previstos em um único ciclo, o alinhamento de instruções não contínuas e a latência da unidade de busca, são questões que devem ser tratadas, e que são consideradas na proposta de *trace cache* para melhorar o desempenho dos processadores superescalares.

Este artigo está organizado da seguinte forma: na seção 2 explica-se o conceito de *trace cache* e demonstra-se sua estrutura e aplicação. Na seção 3 apresenta-se alguns trabalhos relacionados e, também, algumas discussões sobre propostas de aperfeiçoamentos da *trace cache* original proposta por Rotemberg *et al* [1] [2]. Finalmente, na seção 4 são expostas as considerações finais sobre o trabalho.

2. *TRACE CACHE*

A principal função da unidade de busca (*fetch*) é fornecer um fluxo dinâmico de instruções ao decodificador (*decoder*) [5]. Entretanto, Rotenberg *et al.* [1] mencionam que as instruções são colocadas na *cache* de acordo com sua ordem de compilação, o que é favorável para códigos nos quais não há desvios ou que possuem grandes blocos básicos: mas essa não é a realidade comum dos programas normalmente encontrados. Como os autores demonstram em [1] e [2] por meio de estatísticas obtidas da análise de códigos de inteiros (*integer codes*), o tamanho dos blocos básicos é de 4 a 5 instruções, e o percentual de desvios tomados varia de 61 a 86%.

[1] Define-se por bloco básico um trecho de código que não possui desvios, a não ser, talvez, da sua última instrução.

[2] Rotenberg *et al.* [2] sugerem um tamanho médio de 4 a 6 instruções por bloco básico.

Desta forma, Rotenberg *et al.* [1] propõem uma *cache* de instruções especial para capturar a seqüência dinâmica das instruções: a *trace cache*. De acordo com a Figura 1, cada linha da *trace cache* armazena determinado instante (*trace*) do fluxo de instrução dinâmico. Um *trace* é **uma seqüência de no máximo "n" instruções** (determinado pelo tamanho da linha da *cache*) **e de no máximo "m" blocos básicos** iniciando de qualquer ponto do fluxo dinâmico de instruções (determinado pelo *throughput* do previsor de desvios). Assim, um *trace* é especificado por um endereço inicial e por uma seqüência de até m-1 resultados de desvios que descrevem o caminho seguido.

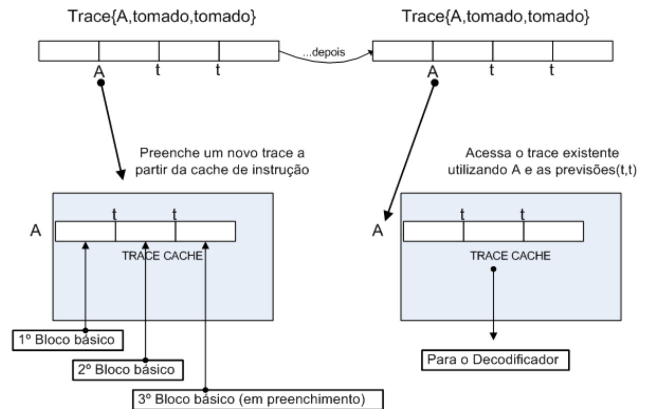


Figura 1 – Visão em alto nível da *trace cache* (adaptado de [1]).

Na primeira vez que um *trace* é encontrado, aloca-se uma linha na *trace cache* e, de acordo com a busca das instruções da *cache* de instruções, preenche-se a linha da *trace cache*. Se no tempo de execução uma mesma linha for encontrada novamente (verifica-se isso comparando o endereço inicial e as previsões para os desvios), ela já estará disponível na *trace cache* e poderá ser fornecida diretamente ao decodificador. Caso contrário, a busca ocorrerá normalmente da *cache* de instruções. Deste modo, é possível observar que a abordagem utilizada pela *trace cache* **fundamenta-se em seqüências dinâmicas de código sendo reutilizadas**. Isso está relacionado ao princípio da **localidade temporal** [1] (instruções utilizadas recentemente tendem a ser utilizadas novamente em um futuro próximo) **e ao comportamento dos desvios** (boa parte dos desvios tende a possuir um viés para alguma direção, devido a isso é que a exatidão da previsão de desvio normalmente é alta).

Para ilustrar, considere o exemplo de uma seqüência dinâmica de blocos básicos ilustrado pela Figura 2(a) — as setas indicam os desvios tomados. Mesmo com múltiplas previsões de desvios por ciclo, para buscar as instruções dos blocos básicos "ABCDE" seriam necessários 4 ciclos. Isso se deve ao fato de que as instruções se encontram armazenadas em lugares não contínuos. Agora considerando o exemplo da Figura 2(b), a mesma seqüência de blocos que aparecia de forma não contínua na *cache* de instrução, aparece agora de forma contínua na *trace cache*.

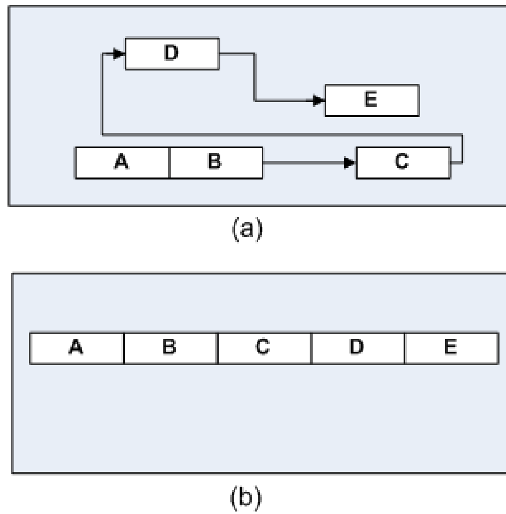


Figura 2 – Armazenamento de seqüências não contínuas de instruções. (a) *Cache* de Instrução. (b) *Trace Cache*. (adaptado de [2]).

2.1 Um exemplo Prático

Durante a execução de um programa, a *trace cache* verificará o resultado das previsões de desvios e, caso eles sejam tomados, buscará as instruções necessárias antecipadamente, obtendo as instruções corretas que realmente serão executadas. Se houver erro na previsão do desvio, então será necessário voltar e buscar as instruções corretas na memória. Para ilustrar o funcionamento da *trace cache*, considere o código do exemplo abaixo, extraído de [13]. Este código corresponde a um laço de repetição no qual existe um desvio que sempre é tomado.

Endereço	**Instrução**	
000F:0001	AND	EBX, ECX
000F:0002	DEC	EAX
000F:0003	CMP	EAX, EBX
000F:0004	JL	0007 *//salta para a instrução 7*
000F:0005	PUSH	EAX
000F:0006	JMP	0001
000F:0007	PUSH	EBX *//continua a execução*
000F:0008	XOR	EAX, EBX
000F:0009	ADD	EAX, 1
000F:000A	PUSH	EAX
000F:000B	MOV	EAX, ESI
000F:000C	JMP	0001 *//retorna para o topo*

Quadro 1 – Código-Fonte para Exemplificação. (adaptado de [13]).

O fluxo seqüencial de código, seguindo-se pelo endereço, seria: 1, 2, 3, 4, 7, 8, 9, A, B, C, devido ao desvio tomado na instrução 4 que ocasiona o salto para a instrução 7. Uma cache convencional capturaria as instruções em ordem seqüencial (1, 2, 3, 4, 5,...). Ao identificar o desvio tomado em 4, a *trace cache* é capaz de carregar as instruções dos dois blocos básicos (1, 2, 3, 4) e (7, 8, 9, A, B, C) numa forma contínua, de modo que o resultado seja: (1, 2, 3, 4, 7, 8, 9, A, B, C). Deste modo, a *trace cache* evitou que instruções desnecessárias ocupassem espaço e fossem carregadas. Dependendo da quantidade de instruções que podem ser carregadas simultaneamente, algumas instruções úteis poderiam ter ficado de fora, as instruções estariam em uma ordem errada e, portanto, isso geraria um *miss* na *chace*, deixando o processo mais lento.

2.2 A Arquitetura da *Trace Cache*

Em Rotemberg *et al.* [1] e [2] é possível encontrar a arquitetura proposta originalmente para a *trace cache* de forma detalhada. Nesta subseção, apresenta-se os seus principais componentes e conceitos de forma simplificada.

A arquitetura da *trace cache*, ilustrada pela Figura 3, foi concebida com o objetivo de prover uma alta largura de banda na busca de instruções com uma baixa latência [2].

O previsor do próximo *trace* trata os *traces* como unidades básicas predizendo explicitamente seqüência de *traces*. Jacobson *et al.* [6] argumentam que essa predição explícita, não somente remove restrições relacionadas ao número de desvios em um *trace*, como também colabora para o alcance de uma taxa média de exatidão de previsão mais alta do que seria obtida por meio de previsores simples. A saída produzia pelo previsor é o "identificador de *trace*" (*trace identifier*), o qual permite identificar um determinado *trace* de forma única pelo seu PC (*Program Count*) e pelas saídas de todos os desvios condicionais que compõem o *trace*[3]. Deste modo, um *trace cache* hit ocorre quando algum *trace* existente corresponde exatamente com o identificador de *trace* previsto.

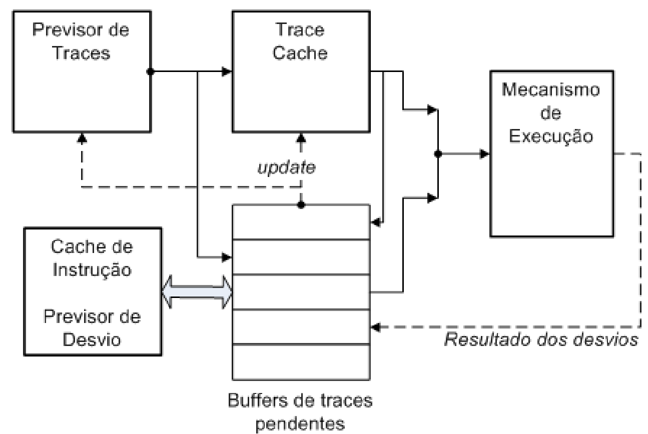


Figura 3 – Micro-arquitetura (adaptado de [2]).

De acordo com Rotemberg *et al.* [2], o previsor de *trace* e a *trace cache* juntos, proporcionam um rápido seqüenciamento em nível de *trace*. Porém, isso nem sempre proporciona o *trace* necessário, principalmente quando se está iniciando um programa ou quando se alcança uma região nova no código que ainda é desconhecida tanto para o previsor quanto para a própria *trace cache*. Para compensar essa limitação, faz-se necessário utilizar, também, o seqüenciamento em nível de instrução.

O *buffer* de *traces* pendentes (*outstanding trace buffers*) é utilizado tanto para construir novos *traces* que não estão na *trace cache*, quanto para rastrear as saídas dos desvios tão logo elas estejam disponíveis no mecanismo de execução, permitindo assim, a detecção de previsões erradas e a correção dos *traces* que

[3] Essa decisão de projeto também permite a existência de diferentes *traces* que começam com um mesmo endereço (redundância).

a compõem [2]. Ainda considerando a Figura 3, cada *trace* buscado é despachado simultaneamente para o mecanismo de execução e para o *buffer*. Assim, no caso de um *miss* (de não encontrar o *trace* desejado na *trace cache*), apenas a previsão do *trace* é recebida pelo *buffer* alocado. A própria previsão fornece informações suficientes para se construir o *trace* a partir da *cache* de instruções[4].

Com relação ao *trace cache hit* e *trace cache miss*, podemos simplificar a explicação da seguinte forma: o mecanismo de busca apresenta simultaneamente um endereço para a *trace cache*, para a *cache* convencional e para a unidade de previsão de desvios. **Se** a *trace cache* possuir um *trace* que se inicia com o mesmo endereço e que também é compatível com a informação de previsão de desvios, **então** ocorre um *hit* e o respectivo *trace* é retornado. **Se** a *trace cache* possui um endereço compatível, mas as informações de previsão não conferem completamente, **então** ocorre um *hit* parcial. Nesse caso, a *cache* de instrução é acessada simultaneamente com a *trace cache*[5]. Nos casos em que a *trace cache* não contém um *trace* começando com o endereço especificado, então ocorre um *trace cache miss*. A *cache* de instrução fornece então a linha contendo o endereço requerido para a unidade de execução e um novo *trace* vai sendo construído.

A arquitetura proposta para a *trace cache* foi validada pelos seus idealizadores em [1] [2] e [6], e também foi discutida por uma série de outros trabalhos, alguns dos quais serão brevemente citados na próxima seção. Os principais resultados obtidos por meio de simulações comparativas permitiram afirmar que a *trace cache* melhora o desempenho numa faixa de 15 a 35% quando comparada a outros mecanismos de busca de múltiplos blocos igualmente sofisticados (mas contínuos).

Também foi possível observar que *trace*s maiores melhoram a exatidão do previsor de *trace*s, enquanto *trace*s menores proporcionam uma taxa de exatidão inferior. Em [1] [2] e [6] os autores concluíram, também, que o desempenho global não é tão afetado pelo tamanho e pela associatividade da *trace cache* do modo como seria o esperado. Isso se deve parcialmente ao robusto seqüenciamento em nível de instrução. De acordo com os resultados obtidos, o IPC não variou mais que 10% em uma ampla gama de configurações.

Outro ponto que deve ser destacado é que a vantagem em se utilizar uma *trace cache* é obtida ao preço da redundância no armazenamento de instruções (vários *trace*s com diferentes previsões de desvios — Ramirez *et al.* [12] apresentam críticas e alternativas para essa questão). Isso pode causar efeitos colaterais na eficiência da *trace cache* devido ao seu limite de tamanho. Finalmente, Rotemberg *et al*. [1][2] e Jacobson *et al.* também concluíram que uma *cache* de instrução combinada com um previsor "agressivo" pode buscar qualquer número de blocos básicos por ciclo, rendendo um aumento de 5 a 25% sobre buscas que retornam um único bloco.

[4] Esse procedimento normalmente leva vários ciclos devido aos desvios previstos como tomados.

[5] A menos que seja necessário economizar energia e, portanto, o acesso não deva ser realizado em paralelo.

2.3 Aplicação Comercial

De acordo com as informações de [13], a *trace cache* é utilizada em processadores modernos como o *Pentium IV*, o *Sparc* e o *Athlon*. O *Pentium IV* é um processador superescalar que se utiliza de paralelismo em nível de instrução. De acordo com Hinton *et al.* [14], a *trace cache* é o nível primário de memória *cache* no *Pentium IV* (L1), conseguindo armazenar até doze mil microoperações, e podendo entregar até três instruções em um único ciclo de *clock*. Ainda, a *trace cache* do *Pentium IV* possui uma taxa de *hit* comparável com a de uma *cache* de instruções de 8 a 16 KB, e a sua aplicação visa, principalmente, minimizar o gargalo existente entre a decodificação de instruções e a sua respectiva execução.

O gargalo entre as unidades de decodificação e de execução de instruções ocorre porque o *Pentium IV* possui um conjunto de instruções complexo. Isso implica na necessidade de haver um decodificador para transformar as instruções complexas em micro-instruções simplificadas para então enviá-las ao *pipeline*. A *trace cache* colabora para que instruções que foram usadas recentemente não precisem ser decodificadas novamente e, conseqüentemente, colabora para uma redução no gargalo [13], além de prover os benefícios já mencionados nas seções anteriores. Em Hennessy & Patterson [5] é possível encontrar várias informações e discussões sobre a utilização da *trace cache* no *Pentium IV*. Provavelmente, esta foi uma das primeiras utilizações da *trace cache* em processadores comerciais.

3. TRABALHOS RELACIONADOS

Nesta seção, apresenta-se brevemente alguns detalhes sobre trabalhos que forneceram bases para o desenvolvimento da proposta de *trace cache*, bem como trabalhos posteriores que propõem modificações e aperfeiçoamentos na proposta original.

3.1 Trabalhos Anteriores

Dentre os trabalhos que geraram idéias para a proposta de trace cache, Rotemberg *et. al* [1] e [2] enfatizam as iniciativas de Yeh *et al.* [7] e Dutta *et al.* [8], descritas abaixo. Segundo Rotenberg *et al.* [2], outras iniciativas de utilização da trace cache foram propostas de forma independente pela comunidade de pesquisa, e maiores comentários sobre as mesmas podem ser encontrados diretamente em [1] e [2].

A pesquisa de Yeh *et al.* [7] considerou um mecanismo de busca que proporcionava uma alta largura de banda ao prever múltiplos endereços de destino dos desvios a cada ciclo — os autores propuseram uma *cache* de endereços de desvios como sendo uma evolução natural do *buffer* de destino de desvios. Assim, um *hit* nessa *cache* combinado com múltiplas previsões de desvios produzia o endereço de início de vários blocos básicos.

A proposta de Franklin *et al.* [8] se baseia em uma abordagem similar à citada anteriormente em Yeh *et al.* [7], com uma diferença na forma de prever múltiplos desvios em um único ciclo: eles incorporaram múltiplas previsões em uma única previsão. Por exemplo: em vez de fazer duas previsões de desvio selecionando um entre dois caminhos, faz-se uma única previsão selecionando um entre quatro caminhos possíveis.

3.2 Propostas e Estudos sobre *Trace Cache*

O trabalho de Postiff *et al.* [4] compara o desempenho da *trace cache* com o limite teórico de um mecanismo de busca de três blocos. Os autores definem várias métricas novas para formalizar a análise da *trace cache* (por exemplo: métricas de fragmentação, duplicação, indexabilidade e eficiência). Neste estudo, os autores demonstram que o desempenho utilizando a *trace cache* é mais limitado por erros na previsão dos desvios do que pela capacidade de se buscar vários blocos por ciclo. Segundo eles, na medida em que se melhora a previsão de desvios, a alta duplicação e a conseqüente baixa eficiência são percebidas como uma das razões pelas quais a *trace cache* não atinge o seu limite máximo de desempenho.

Em [9], os autores exploram a utilização do compilador para otimizar a disposição das instruções na memória. Com isso, possibilita-se permitir que o código faça uma melhor utilização dos recursos de *hardware*, independentemente de detalhes específicos do processador ou da arquitetura, a fim de aumentar o desempenho na busca de instruções. O *Software Trace Cache* (STC) é um algoritmo de *layout* de código que visa não apenas melhorara taxa de *hit* na *cache* de instruções, mas também, um aumento na largura da busca efetiva do mecanismo de *fetch* (quantidade de instruções que podem ser buscadas simultaneamente). O STC organiza os blocos em cadeias tentando fazer com que blocos básicos executados seqüencialmente residam em posições contínuas na memória. O algoritmo mapeia a cadeia de blocos básicos na memória para minimizar os conflitos de *miss* em seções críticas do programa. O trabalho de Ramirez *et al.* [9] apresenta uma análise e avaliação detalhada do impacto do STC, e das otimizações de código de uma forma geral, em três aspectos principais do desempenho de busca de instruções: a largura efetiva da busca, a taxa de *hit* na *cache* de instrução, e a exatidão da previsão de desvios. Os resultados demonstram que os códigos otimizados possuem características especiais que os tornam mais propícios para um alto desempenho na busca de instruções: possuem uma taxa muito elevada de desvios não-tomados e executam longas cadeias seqüenciais de instruções, além de fazerem um uso eficaz das linhas da *cache* de instruções, mapeando apenas instruções úteis que serão executadas em um curto intervalo de tempo, aumentando, assim, tanto a localidade espacial quanto a localidade temporal.

Em [10], os autores apresentam uma nova *trace cache* baseada em blocos e que, segundo simulações, poderia alcançar um maior desempenho de IPC com um armazenamento mais eficiente dos *trace*s. Em vez de armazenar explicitamente as instruções de um *trace*, os ponteiros para blocos que constituem o *trace* são armazenados em uma tabela de *trace*s muito menor do que a utilizada pela *trace cache* original. A *trace cache* baseada em blocos proposta pelos autores renomeia os endereços de busca em nível de bloco básico e armazena os blocos alinhados em uma *cache* de blocos. Os blocos são construídos pelo acesso à *cache* de blocos utilizando os ponteiros de bloco da tabela de *trace*. Os autores compararam o desempenho potencial da proposta com o desempenho da *trace cache* convencional. De acordo com os resultados demonstrados em [10], a nova proposta pode alcançar um IPC maior com um impacto menor no tempo de ciclo.

Hossain *et al.* [11] apresentam parâmetros e expressões analíticas que descrevem o desempenho da busca de instrução. Os autores implementaram expressões analíticas em um programa utilizado para explorar os parâmetros e suas respectivas influências no desempenho do mecanismo de *fetch* de instruções (o programa é denominado de *Tulip*). As taxas de busca de instrução previstas pelas expressões propostas pelos autores apresentaram uma diferença de 7% comparadas com as taxas apresentadas pelos programas do *benchmark SPEC2000*. Além disso, o programa também foi utilizado para tentar identificar e compreender tendências de desempenho da *trace cache*.

Em [12] os autores argumentam que os recursos de *hardware* dos mecanismos da *trace cache* podem ter seu custo de implementação reduzido sem ocasionar perda de desempenho se a replicação de *trace*s entre a *cache* de instrução e a *trace cache* for eliminada. Os autores demonstram que a *trace cache* gera um alto grau de redundância entre os *trace*s armazenados na *trace cache* e os *trace*s gerados pelo compilador e que já estão presentes na *cache* de instrução. Além disso, os autores também abordam que algumas técnicas de reorganização de código, como a STC apresentada em [9], adotam estratégias que colaboram para aumentar ainda mais a redundância. Deste modo, a proposta do trabalho de [12] é efetuar um armazenamento seletivo dos *trace*s de modo a evitar a redundância entre a *cache* de instruções e a *trace cache*. Segundo os autores, isso pode ser obtido modificando-se a unidade que preenche os novos *trace*s para que ela armazene apenas os *trace*s de desvios tomados (uma vez que eles não podem ser obtidos em um único ciclo). Os resultados exibidos demonstram que, com as modificações propostas, uma *trace cache* de 2KB com 32 entradas apresenta um desempenho tão bom quanto uma *trace cache* de 128 KB com 2048 entradas (sem as modificações). Isso enfatiza, segundo [12], que a cooperação entre *software* e *hardware* é fundamental para aumentar o desempenho e reduzir os requisitos de *hardware* necessários para o mecanismo de *fetch* de instruções.

4. CONSIDERAÇÕES FINAIS

A abordagem de *trace cache* apresentada neste artigo se constitui como uma forma efetiva de aumentar a capacidade de fornecimento de instruções para os processadores superescalares. Como mencionado nas seções anteriores, para que seja possível aproveitar os benefícios do paralelismo em nível de instrução, os processadores superescalares precisam que um grande número de instruções seja decodificado e esteja pronto para ser executado a cada ciclo. Deste modo, ao utilizar uma abordagem dinâmica para identificar o fluxo de execução e, assim, conseguir prever desvios e organizar os blocos básicos de forma contínua, a *trace cache* colabora para que mais instruções possam ser buscadas em um único ciclo de *clock*.

Deste modo, o mecanismo de *trace cache* pode ser compreendido como uma solução para o problema de se realizar o *fetch* de múltiplos desvios em um único ciclo. Isso porque a *trace cache* colabora para aumentar a quantidade de desvios condicionais que podem ser previstos em um único ciclo, além de melhorar o alinhamento de instruções não contínuas e de reduzir a latência da unidade de busca.

O fato da *trace cache* ser utilizada em processadores modernos, tais como o *Pentium IV*, o *Athlon* e o *Sparc*, demonstra que as contribuições da mesma são realmente efetivas. No caso do *Pentium IV*, pode-se verificar também o benefício da redução na quantidade de decodificação de instruções, devido ao fato da *trace cache* evitar que instruções decodificadas recentemente,

precisem ser re-decodificadas novamente. Essa vantagem fica clara em programas que utilizam laços de repetição (*for*, *while*, *repeat*), nos quais sem a *trace cache*, as instruções executadas precisariam ser decodificadas a cada nova iteração.

Entretanto, apesar dos benefícios e da viabilidade da abordagem de *trace cache*, é preciso levar em conta que a adição de funcionalidades extras sempre torna o processador maior e mais complexo. Logo, a *trace cache* pode adicionar um grande número de transistores a alguma parte do processador que já esteja muito grande. Considerando que o tamanho de um processador impacta diretamente no seu custo de fabricação, a *trace cache* pode torná-lo mais rápido, mas também o tornará mais caro.

Outro ponto que merece ser mencionado é que a colaboração da *trace cache* para a melhoria do desempenho deve-se, em grande parte, à eficiência do previsor de desvios. Um previsor com uma baixa taxa de acertos certamente comprometerá a colaboração da *trace cache* para a melhora do desempenho geral. Apesar dessa dependência, isto não vem sendo apresentado como um fator limitante, talvez, devido a pesquisas e resultados eficientes em abordagens para a previsão de desvios.

5. REFERÊNCIAS

[1] Rotenberg, E., Benett, S. and Smith, J. E. 1996. *Trace Cache*: a Low Latency Approach to High Bandwidth Instruction Fetching. *In the Proceedings of 29th Annual International Symposium on Microarchitecture*. France.

[2] Rotenberg, E., Benett, S. and Smith, J. E. 1999. A *Trace Cache* Microarchitecture and Evaluation. *IEEE Transactions on Computers*. Vol. 48. Nº 2. Pages 111-120.

[3] Berndl, M. and Hendren, L. Dynamic Profiling and *Trace Cache* Generation. 2003. *In Proceedings of the International Symposium on Code Generation and Optimization (CGO'03).*

[4] Postiff, M., Tyson, G. and Mudge, T. Performance Limits of *Trace Cache*s. 1999. *Journal of Instruction-Level Parallelism*. Pages 1-17.

[5] Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. 2006. 4ª Ed. Elsevier.

[6] Jacobson, Q., Rotenberg, E. and Smith, J. Path-Based Next *Trace* Prediction. 1997. *In Proceedings of 30th Int. Symp. Microarchitecture*. PAFES 14-23.

[7] Yeh, T.-Y., Marr, D. T. and Patt, Y. N. Increasing the instruction fetch rate via multiple branch prediction and a branch address *cache*. 1993. *In Proceedings of 7th Intl. Conf. on Supercomputing*, pp. 67–76.

[8] Dutta, S. and Franklin, M. Control flow prediction with treelike subgraphs for superscalar processors. 1995. *In Proceedings of 28th Intl. Symp. on Microarchitecture*, pp. 258–263.

[9] Ramirez, A., Larriba-Pey, J. and Valero, M. Software *Trace Cache*. 2005. *IEEE Transactions on Computers*, Vol. 54, Nº. 1. Pages 22-35.

[10] Black, B., Rychlik, B. and Shen, J. P. The Block-based *Trace Cache*. 1999. *in Proceedings of the 26th Annual International Symposium on Computer Architecture.*

[11] Hossain, A., Pease, J. D., Burns, J. S. and Parveen, N. *Trace Cache* Performance Parameters. 2002. *In Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02).*

[12] Ramirez, A., Larriba-Pey, J. L., and Valero, M. 2000. *Trace cache* redundancy: Red and blue *traces*. *In Proceedings of the 6th International Symposium on High-Performance Computer Architecture*. pp. 325-333.

[13] Everything2. Trace Cache. 2002. Disponível em: *http://everything2.com/title/Trace%2520cache*. Acesso em 10 de Junho de 2009.

[14] Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., and Roussel, P. The microarchitecture of the Pentium 4 processor. 2001. *Intel Technology Journal Q1*.

Arquitetura IBM Blue Gene para Supercomputadores

Carlos Alberto Petry
Unicamp – IC – PPGCC
MO801 Arquitetura
de Computadores I
Campinas, SP – Brasil
ra095345@students.ic.unicamp.br

Resumo

O presente artigo apresenta uma visão geral da arquitetura IBM Blue Gene/L para supercomputadores. São abordados dois aspectos: a arquitetura do hardware do sistema e da arquitetura de software, esta em nível mais introdutório. O sistema Blue Gene/L corresponde a um supercomputador destinado a executar aplicações altamente paralelizáveis, sobre tudo aplicações científicas, podendo conter mais de 65 mil nodos de computação. O objetivo principal do projeto foi atingir a marca de 360 TeraFLOPS em desempenho utilizando uma relação custo x benefício e consumo x desempenho não obtidas em arquiteturas tradicionalmente usadas em supercomputadores construídos anteriormente ao projeto Blue Gene.

Categoria e descrição do assunto

C.5.1 [**Computer System Implementation**]: Large and Medium ("Mainframe") Computers – Super (verylarge) computers.

Termos Gerais

Desempenho, Projeto de hardware, Experimentos.

Palavras Chaves

Arquitetura de Computadores, Supercomputadores, Hardware, Alto Desempenho.

1. INTRODUÇÃO

Blue Gene constitui um projeto de pesquisa da IBM com o objetivo de explorar os limite da supercomputação em termos de: arquitetura de computadores, requisitos de software para programar e controlar sistemas massivamente paralelos e uso de computação avançada para ampliar o entendimento de processos biológicos como desdobramento de proteínas. Novas aplicações estão sendo exploradas com os sistemas Blue Gene como: hidrodinâmica, química quântica, dinâmica molecular, modelagem climatológica e financeira [2].

Blue Gene/L é um sistema baseado no processador embarcado, PowerPC modificado e acrescido em suas funcionalidades em relação ao original, suportando um grande espaço de endereçamento de memória, fazendo uso de compiladores padrão e de um ambiente de passagem de mensagens para a comunicação [1], além de ser um supercomputador cujo sistema pode chegar a ter até 65536 nodos de computação utilizando grande escala de paralelismo. Através da integração de sistemas em um único chip (SoC) [6] [7] juntamente com uma arquitetura celular escalável, o sistema Blue Gene/L foi projetado para atingir um desempenho de pico de 180 ou 360 TeraFLOPS [1] [2], dependendo de como o sistema for configurado.

BlueGene/L foi projetado e construído em colaboração com o Departamento de Energia norte-americano o NNSA/Lawrence Livermore National Laboratory (LLNL), devido a uma parceria estabelecida em Novembro de 2001, o qual atingiu o pico de desempenho de 596,38 TeraFLOPS na unidade localizada no LLNL [2], alcançado em 2007 utilizando 212992 núcleos de computação [4].

A versão Blue Gene/L Beta System DD2 chegou ao 1º lugar entre os supercomputadores em Novembro de 2004, atingindo desempenho de pico de 91,75 TeraFLOPS conforme publicado em Top500 [3]. Atualmente ainda ocupa o 5º lugar na classificação deste mesmo site [4], conforme levantamento anunciado em Junho de 2009. O objetivo do projeto foi disponibilizar um sistema cuja relação custo x desempenho fosse a mais alta possível, além de atingir altas taxas de desempenho em relação ao consumo de potência e área ocupada pelo sistema. Este objetivo foi, em grande parte, viabilizados pelo uso dos processadores embarcados IBM PowerPC (PPC) de baixo consumo e baixa freqüência, tendo sido um dos primeiros projetos de supercomputadores a usarem o paradigma *low power*. O sucessor da versão /L, o modelo Blue Gene /P, instalado no Argonne National Laboratory, atingiu o 4º lugar entre os computadores mais eficientes em relação ao desempenho por watt confirme publicado em Green500 [5] (371,75MFlops/Watt). A medição para o modelo /L não está disponível uma vez que esta avaliação iniciou apenas em Novembro de 2007.

O restante deste trabalho está organizado da seguinte forma: a seção 2 apresenta as características da arquitetura de hardware do sistema Blue Gene/L; a seção 3 apresenta as características básicas da arquitetura de software utilizada neste sistema; por fim a seção 4 apresenta a conclusão e considerações finais.

2. ARQUITETURA DE HARDWARE

Esta seção apresenta um estudo da arquitetura de hardware do supercomputador Blue Gene/L.

A abordagem de construção do Blue Gene/L (BG/L) representou uma inovação para a arquitetura de supercomputadores devido ao uso em grande escala dos chamados nodos de computação, detalhados posteriormente, operando a uma freqüência de clock de 700MHz, baixa para os padrões da época.

O componente básico da arquitetura BG/L se constitui de um SoC baseado em processadores embarcados PowerPC 440 CMOS incorporando todas as funcionalidades necessárias, tais como: processador para computação, processador para comunicação e controle, três níveis de memória cache, memória DRam embarcada (EDRam) e múltiplas interconexões de rede de alta velocidade usando um sofisticado roteamento de dados, sobre um único ASIC (Application Specific Integrated Circuit) [1]. Devido a freqüência da memória Ram ser próxima a do processador, foi possível construir nodos de computação com baixo consumo de energia e alta densidade de agrupamento, podendo chegar a conter 1024 nodos de computação inseridos em um único *cabinet* (detalhado da próxima seção). A comunicação entre nodos está embutida na lógica do ASIC dispensando o uso de Switches externos.

2.1 Estrutura modular do Hardware BG/L

O sistema BG/L é escalável, podendo atingir um máximo de 2^{16} (65536) nodos de computação, configurados como uma rede *Torus 3D* com dimensões de 64x32x32. Sua arquitetura física é caracterizado por uma estrutura modular composta de 5 níveis, conforme apresenta a Figura 1. O componente básico é um chip ASIC contendo dois processadores e denominado nodo de computação; um compute card que é composto de dois chips ASIC; um node board que contém 16 compute card; 32 node board que formam um cabinet ou rack, o qual é composto por dois midplanes com 16 node board cada um; e por fim, podendo existir até 64 cabinets agrupados formando um System. Na especificação completa de um *System*, cada componente modular possuem as seguintes quantidade de nodos de computação: *compute card* → 2, *node board* → 32, *cabinet/rack* → 1024 e *system* → 65536.

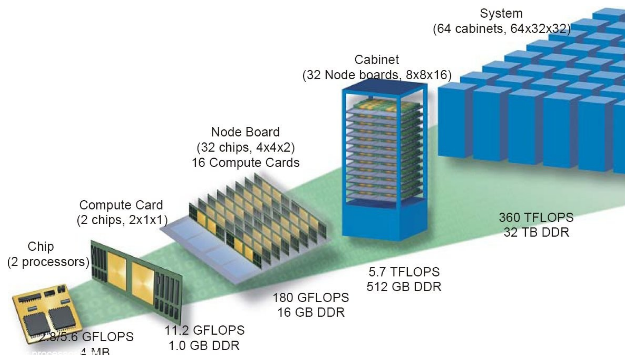


Figura 1 – BG/L: estrutura modular do hardware.

Cada processador do nodo de computação pode realizar quatro operações de ponto flutuante na forma de duas adições /multiplicações de 64 bits por ciclo, o que permite atingir um desempenho máximo (pico) de 1.4 GigaFLOPS, considerando uma freqüência de operação de 700MHz. Os nodos de computação do BG/L podem operar em dois modos distintos [1] [7]: (i) os dois processadores são usadas para computação, operações sobre a aplicação; (ii) um processador é usado para computação e outro para controle da comunicação MPI e controle, sendo que o modo de operação é definido em função das características da aplicação a ser executada. O nodo de computação atinge um pico de desempenho de 5,6 ou 2,8 GFLOPS dependendo do modo de operação configurado. Como existem 1024 nodos de computação em um *cabinet*, o seu desempenho máximo atinge 5,6 ou 2,8 TeraFLOPS. Por fim na configuração completa, 64 *cabinets*, um *System* atinge o desempenho de pico aproximado de 360 ou 180 TeraFLOPS, conforme o modo de operação em uso.

Os nodos são interconectados através de 5 redes [1]: (i) rede torus 3D usada para comunicação ponto-a-ponto de mensagens entre nodos de computação; (ii) árvore global combining /broadcast para operações coletivas através de toda a aplicação, como por exemplo MPI_Allreduce; (iii) rede global barrier e interrupt; (iv) duas redes Ethernet gigabit, uma para operações de controle via conexão JTAG e outra para conexão com outros sistemas como hosts e file system. Por motivos de custo e eficiência, nodos de computação não são diretamente conectados à rede Ethernet, usando a árvore global para realizar a comunicação com nodos de I/O. Além dos 65536 possíveis nodos de computação, o BG/L contém 1024 nodos de I/O que são idênticos aos nodos de computação porém dedicados a operações de comunicação com os dispositivos externos como *hosts* e *file system*.

2.2 Estrutura do nodo de computação

O componente básico de computação, nodo de computação, consiste de dois processadores, três níveis de memória cache, dispositivos de suporte à conexões externas e memória Ram DDR ECC cuja capacidade máxima é de 2GB, entretanto são usados apenas 256MB devido a definição de projeto. A Figura 2 apresenta o diagrama de blocos simplificado do SoC contido no chip ASIC, exceto a memória Ram DDR que está contida no *compute card*, comunicando com o ASIC via barramento local.

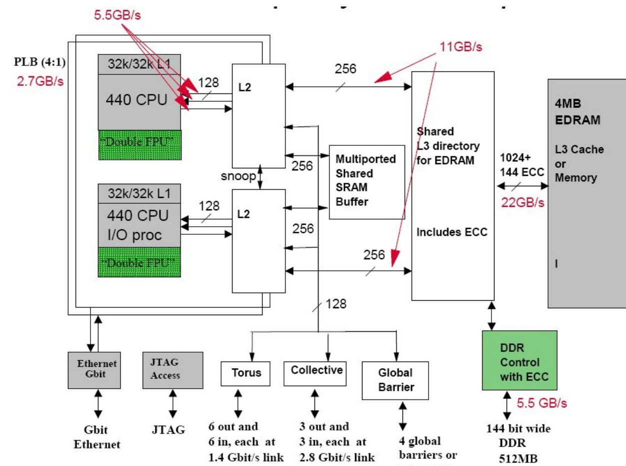


Figura 2 –Diagrama de blocos simplificado do SoC ASIC BG/L.

O nodo de computação é composto por dois processadores PPC 440, cada um dispondo de uma unidade de ponto flutuante (FPU2) dupla que corresponde a uma unidade de 128 bits melhorada. O processador PPC 440 se caracteriza por uma arquitetura superescalar de 32 bits, usado em sistemas embarcados e produzido pela IBM, não dispondo de hardware para suporte a multiprocessamento simétrico (SMP). A FPU2 é

composta por duas FPUs convencionais de 64 bits unidas, entretanto as unidades foram estendidas para suportar instruções do PPC estilo SIMD, além do conjunto de instruções original [7].

Existem três níveis de memória cache: (i) L1, uma cache para cada núcleo, não coerente e não dispondo de operações atômicas em memória, porém estas deficiências foram contornadas pelo uso de dispositivos de sincronização no chip, como mecanismo *lockbox, L3*-scratchpad, *blind device* e SRam compartilhada; (ii) um segundo nível de cache (L2) é implementado para cada processador, possui capacidade de 2KB e é controlada por um mecanismo *data pre-fetch*, que corresponde a um *SRam array* para acesso a comunicação entre os dois núcleos; (iii) o projeto disponibiliza ainda um terceiro nível de cache (L3) produzidas com memórias *Embedded DRam* (ERam) com capacidade de 4MB. Complementando a arquitetura do ASIC, existem ainda interfaces usadas para comunicação com dispositivos externos: Ethernet, JTAG, operações de roteamento e controladora de memória Ram DDR externa. Os níveis de cache L2 e L3 são coerentes entre os dois processadores PPC440.

Em modo de operação normal, um nodo de computação é usado para realizar o processamento da aplicação, enquanto o outro serve às funções de comunicação e controle de rede fazendo uso de troca de mensagens MPI. Entretanto, não existe impedimento do ponto de visto do hardware para usar os dois nodos para computação. Este uso é determinado por aplicações que se caracterizam por possuir alta taxa de computação se comparado ao processamento de mensagens.

Como o caminho de dados entre a cache de dados e a FPU possui 128 bits, é possível armazenar ou recuperar dois elementos de dados, usando precisão simples ou dupla, a cada ciclo de clock, implementado também um mecanismo para transferências de dados entre posições de memória de alta velocidade, útil no processamento de mensagens. Todas as instruções de computação, exceto divisão e operações sobre valores não normalizados, são executadas em cinco ciclos de clock, mantendo uma vazão de de um ciclo por instrução. Divisões são executadas de forma iterativa produzindo dois bits a cada ciclo. A FPU2 também pode ser usada pelo processador para otimizar operações de comunicação, uma vez que ela permite ampla largura de banda para acesso ao hardware da rede [1].

A característica de baixo consumo do nodo de computação é obtida pela implementação de técnicas como uso de transistores com baixa corrente de fuga, *clock gate* local e habilidade de desativar dispositivos como CPU/FPU, além do uso de métodos que permitem reduzir o chaveamento do circuito.

O sistema de memória foi projetado de forma a oferecer alta largura de banda e baixa latência tanto para memória como cache, por exemplos *hits* em cache consomem 6 a 10 ciclos de CPU para L2 e 25 ciclos para L3 e um *misses* sobre a cache L3 consome cerca de 75 ciclos.

2.3 Hardware de rede

Toda a comunicação entre nodos de computação é realizada através da troca de mensagens MPI. Entre as cinco redes do BG/L (torus, árvore, Ethernet, JTAG e interrupção global) a principal comunicação ponto-a-ponto na rede para mensagens é realizada pela rede torus 3D [7]. Cabe salientar que o nodo de I/O não faz parte da rede torus 3D.

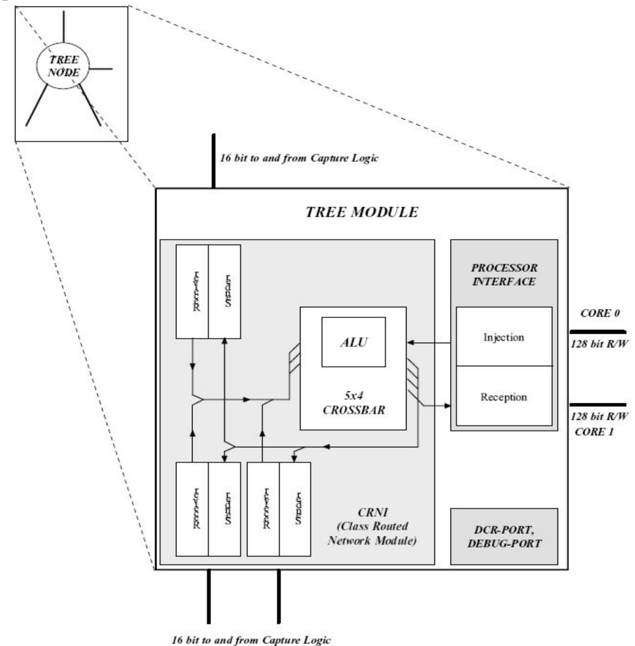


Figura 3 – Módulo de controle da rede em árvore.

A rede em árvore suporta comunicação ponto-a-ponto usando mensagens de tamanho fixo de 256 bytes, bem como *broadcast*, operando com uma latência de 1,5ms para um sistema composto por 65536 nodos de computação. Tanto nodos de computação como de I/O compartilham a rede em árvore, constituindo esta rede o principal mecanismo para comunicação entre as duas classes de nodos (computação e I/O). Para suportar a rede em árvore existe um módulo que opera a funcionalidade necessária, o módulo árvore, apresentado na figura 3. Nele, existe uma ALU de inteiros e *bitwise que* combina pacotes que chegam e encaminha os pacotes resultantes ao longo da árvore. Além da ALU, podem ser visto à esquerda (acima e abaixo) as três conexões bidirecionais (send, receive) com a rede torus 3D e à direita aparece a interface local com os processadores PPC. A rede em árvore possui 2 canais virtuais (*virtual channels*–VC) e faz uso de pacotes (componente atômico que trafegam na rede) para viabilizar o fluxo de comunicação, sendo o controle realizado pelo uso de *tokens*. O fluxo de dados, que flui através dos VCs, é realizado de forma não bloqueante. Através de registradores, existentes no hardware da rede em árvore, é possível selecionar um modo de comunicação. entre duas formas existentes: *combining* e *point-to-point*. O modo *point-to-point* é usado para nodos de computação que precisam se comunicar com os nodos de I/O. O modo *combining* opera agrupando pacotes a serem transmitidos, sendo usado para suportar operações MPI coletivas, como por exemplo MPI_Allreduced, através de todos os nodos conectados na rede em árvore.

Tanto a rede em árvore como a torus são acessadas através de endereços mapeados em memória, ou seja, o processador envia uma mensagem para um endereço especial que é redirecionado pelo hardware para a rede e implementado como uma FIFO usando registradores SIMD de 128 bits.

O BG/L possui agrupamentos computacionais denominados *partições* que são conjuntos de nodos de computação eletricamente isoladas constituindo subconjuntos auto-contidos dentro de um *System*. Cada *partição* é dedicada a executar uma aplicação ou conjunto comum de aplicações que podem ser distribuídas em tarefa,. O menor dimensionamento configurável corresponde a um *midplane* (meio *cabinet*), que constitui uma rede torus com 512 nodos de computação. É possível criar redes menores, até o limite inferior de 128 nodo de computação. Isto é feito desativando-se as FIFOs nos chips usados para controle de limites do *midplane*.

Cada SoC ASIC no BG/L possui ainda uma rede Ethernet utilizada para conectividade externa e uma rede serial JTAG para operações de boot, controle e monitoração do sistema.

Cada nodo da rede torus 3D possui seis conexões bidirecionais com os nodos vizinhos (adjacentes). Os 65536 nodos de computação, previstos no projeto, são organizados numa distribuição de rede cuja dimensão é 64x32x32, conforme já mencionado. Os dispositivos de rede disponíveis no ASIC garantem a entrega de pacotes contendo até 256 bytes de forma confiável, não ordenada e livre de *dead-lock*, usando algoritmo de roteamento adaptativo mínimo. O algoritmo de roteamento selecionado para uso no BG/L foi o *virtual cut-through* (VCT) [8] de forma a aperfeiçoar a capacidade vazão e latência da rede. As mensagem podem ser compostas por mais de um pacote, podendo ser enviadas numa ordem e recebidas em ordem diferente, variando seu tamanho entre 32 e 256 bytes com granularidade de 32 bytes, portanto foram criados mecanismos para recompor a ordem original da mensagem. A implementação de canais virtuais ajuda a melhorar a vazão, existindo na configuração original do sistema 4 CVs, dois dedicados a roteamento adaptativo e dois para roteamento determinístico. O controle de fluxo entre roteadores é realizado pelo uso de *tokens*. A rede torus é usada para operações de propósito geral, passagem de mensagem ponto-a-ponto e operações de *multicast* dentro de uma *partição*.

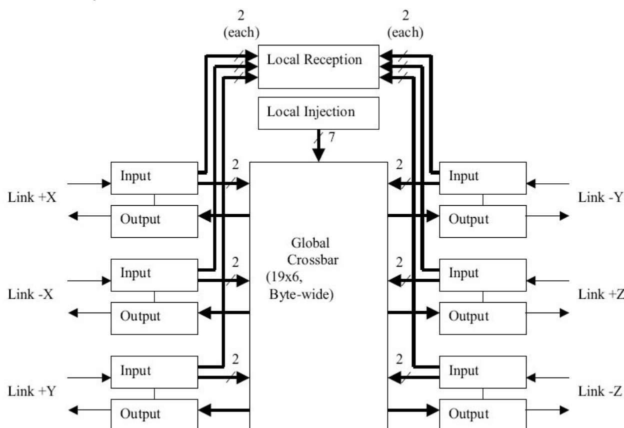


Figura 4 – Estrutura básica da rede torus inserida nos nodos.

A estrutura básica da rede torus inserida em cada nodo de computação pode ser vista na figura 4, onde podem ser observados as seis conexões bidirecionais (x, y e z), a conexão local (reception e injection) e o *crossbar* que realiza a interconexão entre as portas de duas conexões. Cada conexão bidirecional possui duas FIFOs usadas para recepção de dados direcionados ao nodo e para repasse de dados destinados a outros nodos. Existe um coprocessador de mensagens que é responsável pelo controle de recebimento e envio de mensagens contidas nas duas FIFOs. Pacotes recebidos de outros nodos são imediatamente repassados caso não haja contenção na rede, caso exista contenção eles são armazenados na FIFOs de entrada e aguardam até que possam ser enviados. Existe capacidade suficiente nos dispositivos FIFO de forma a manter o fluxo de dados na rede, desde que não inexista contenção [1]. O controle de arbitragem é altamente *pipelinizado* e distribuído, existindo um árbitro em cada unidade de entrada e saída.

Baseado em experimentos e simuladores da rede, os projetistas determinaram os valores de dimensionamento dos dispositivos existentes no módulo (VC, FIFO size, etc), para otimizar e atender o fluxo de rede de foram adequada. Experimentos realizados em um S*ystem* contendo 32768 nodos de computação, usando comunicação intensiva através da chamada MPI_Alltoall, demonstraram que o uso de roteamento dinâmico demonstrou ser mais eficaz que o estático devido à característica *pipelinizada* da rede, além de demonstrar que usando apenas dois VC dinâmicos as conexões se mantém ocupadas (utilizadas) praticamente todo o tempo, conforme observado na tabela 1 [1].

Tabela 1 – Experimento de comunicação contendo 32768 nodos de computação utilizando roteamento estático x dinâmico.

	Média da utilização da conexão (Pacote)	Média da utilização da conexão (Payload)
Roteamento estático	76%	66%
Roteamento dinâmico 1 VC	95%	83%
Roteamento dinâmico 2 VC	99%	87%

3. ARQUITETURA DE SOTEWARE

De forma a atender os requisitos de escalabilidade e complexidade e disponibilizar um ambiente de desenvolvimento confiável, mantendo os padrões de software o máximo possível, os projetistas de software do BG/L implementaram a arquitetura apresentada na figura 5.

Aplicações destinadas a executarem no BG/L são organizadas como uma coleção de processos computacionais[1], cada um sendo executado em seu próprio nodo de computação a partir de uma *partição* no sistema, auxiliados pelos nodos de I/O que oferecem serviços de suporte à execução das aplicações.

A preocupação principal durante o projeto do sistema de software foi disponibilizar um ambiente com alto nível de desempenho, oferecendo facilidades de desenvolvimento familiar, onde as aplicações executassem sob um sistema operacional (SO) unix-

like, porém mantendo capacidade suficiente para atender aos requisitos do projeto. A abordagem adotada foi dividir as funcionalidades do SO em dois grupos: computação e I/O. Nodos de computação executam exclusivamente aplicações de usuário, enquanto nodos de I/O executam operações de sistema, proporcionando acesso ao sistema de arquivos e *host*, além de dispor de facilidades como operações de controle, disparo de tarefas, boot e depuração do software. Esta abordagem manteve a arquitetura do hardware e software o mais simples possível [1].

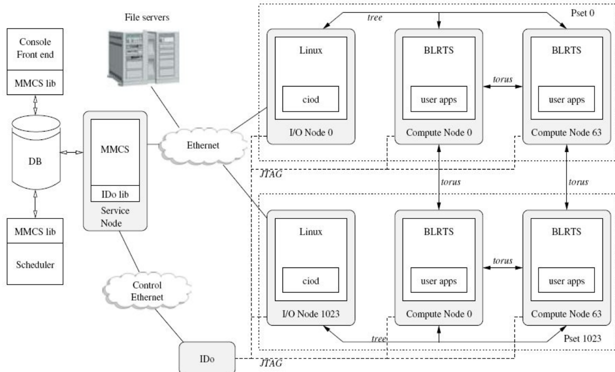


Figura 5 – Estrutura básica do sistema de software BG/L.

3.1 Software para nodos de computação

O BG/L executa uma SO personalizado nos nodos de computação denominado BLRTS (BG/L Run Time Supervisor). Este SO oferece um espaço de endereçamento contínuo de 256MB sem suporte a paginação, compartilhando este endereçamento entre o kernel e a aplicação sendo executada. O kernel fica localizado na parte inicial da memória (a partir do endereço 0), esta área está protegida através da programação da MMU realizada durante o processo de boot. Acima do SO fica localizada a aplicação a ser executada pelo processador seguida pelas suas áreas de pilha e *heap*.

Os recursos físicos, como rede em árvore e torus, são particionados entre aplicação e kernel, sendo que toda a rede torus é mapeada dentro do espaço de usuário, de forma a se obter melhor desempenho da aplicação, além de ser disponibilizado para kernel e aplicações de usuário um dos dois canais da rede em árvore.

O BLRTS segue o padrão Posix dispondo da biblioteca GNU/Glibc que foi portada para este SO, de forma a prover o suporte básico para operações de I/O sobre o sistema de arquivos. O suporte a multitarefa é desnecessário uma vez que o SO é monousuário executando uma única aplicação com suporte *dual-thread*, assim não foi incluída na biblioteca o suporte multiusuário. Operações de carga, finalização e acesso à arquivos para aplicações são realizadas via passagem de mensagens entre os nodos de computação e de I/O, através da rede em árvore. Sobre o nodo de I/O é executado o *deamon* CIOD (Console I/O deamon) que provê as tarefas de controle e gerenciamento de entrada e saída de dados para todos os nodos da *partição*. Kernel e biblioteca implementam a comunicação entre nodos de computação através da rede torus. Devido à natureza monousuária o SO não necessita realizar trocas de contexto e operações de paginação.

3.2 Software para nodos de I/O

Sobre os nodos de I/O é executado o SO GNU/Linux versão 2.4.19 contendo suporte para execução multitarefa. Para suportar os requisitos de projeto foram necessárias alterações na seqüencia de boot, gerenciamento de interrupções, layout de memória, suporte à FPU e drivers de dispositivos, em relação ao kernel original. Interrupções e exceções forma adequados ao BIC (BG/L custom interrupt controller); a implementação do kernel para MMU remapeia as redes em árvore e torus para o espaço de usuário; foi inserido suporte para controlador Ethernet Gigabit; também foi disponibilizado suporte para operações *save/restore* sobre registradores de ponto flutuante de precisão dupla para a FPU2 e suporte a trocas de contexto, uma vez que neste nodo o SO é multitarefa. No nodo de I/O é executado apenas o sistema operacional, ou seja, nenhuma aplicação esta presente ou em execução. O objetivo do processamento realizado neste nodo é dispor de serviços não suportados pelo SO do nodo de computação como acesso a sistemas de arquivos e *sokets* de conexões com processos de outros sistemas. Quando uma operação é requisitada pela aplicação a rede em árvore transporta esta solicitação para ser processada no nodo de I/O. Entre as operações suportadas estão: autenticação, contabilização e autorizações em nome de seus nodos de computação. Ainda está disponível no nodo de I/O facilidades para depuração de aplicações de usuário executadas nos nodos de computação. O fluxo de dados relativo à depuração também faz uso da rede em árvore.

Como os nodos do BG/L são *diskless* o sistema de arquivos raiz é fornecido por um *ramdisk* inserido no kernel do linux. O *ramdisk* do nodo de I/O contém interfaces de shell, utilitários básicos, bibliotecas compartilhamento e clientes de rede como ftp e nfs.

3.3 Gerenciamento e controle do sistema

A infra-estrutura de controle provê uma separação entre o mecanismo de execução da aplicação e as políticas de decisões em nodos externos à partição. O SO dos nodos locais são responsáveis pelas decisão locais. O SO “global” [7] toma todas as decisões coletivas e atua como interface com os módulos externos, realizando diversos serviços globais (boot, monitoramento, etc). O SO “global” executa em nodos de serviços externos (*hosts*), sendo que cada *midplane* é controlado por um MMCS (Midplane Management and Control System).

O processo de boot realiza a carga de um pequeno código (boot loader) inserido na memória do nodo (de computação e I/O) fazendo uso dos nodos de serviço através da rede JTAG. Este pequeno código carrega na seqüência a imagem do boot principal que então passa a controlar o processamento. Como os SOs dos nodos de computação e I/O são diferentes cada nodo recebe sua imagem correspondente. A imagem para computação possui cerca de 64KB e a I/O aproximadamente 2MB, já incluso o sistema de arquivos e *ramdisk*. Uma vez que existem diversos nodos é necessário carregar também configurações adicionais

que individualizam os nodos, este procedimento foi denominado de *personalidade* do nodo [7].

O sistema é monitorado pelos nodos de serviço externos ao sistema principal. Estes serviços são implementados através do software contido nestes nodos. Informações tais como velocidade dos ventiladores e voltagem da fonte de energia são obtidos através da rede de controle, sendo registradas (log) pelos serviços destes nodos.

A execução de tarefas é viabilizada conjuntamente pelos nodos de computação e de I/O. Quando uma aplicação é submetida a execução o usuário especifica a forma e tamanho da *partição* desejada, estalecida no sistema pelo escalonador que é uma facilidade fornecido pelos nodos de serviço.

A arquitetura de comunicação no BG/L é implementada por três camadas. A primeira corresponde a camada de pacotes (*packet layer*) constituída de uma biblioteca que permite acesso ao hardware da rede; a segunda é a camada de mensagem (*message layer*) que provê um sistema para entrega de mensagens de baixa latência e alta largura de banda, cuja comunicação ocorre ponto-a -ponto; a última camada é a MPI que corresponde a uma biblioteca de comunicação em nível de usuário (aplicação), designada basicamente para executar as facilidades da especificação MPI [9].

3.4 Modelo de programação

A passagem de mensagens foi elegido como o modelo de programação adotado como principal abordagem para uso na implementação da programação paralela existente no BG/L Este modelo é suportado através da implementação da biblioteca MPI message-passing, sobre a qual os projetistas tiveram especial atenção quanto a eficiência no mapeamento de operações para as redes em árvore e torus.

4. CONCLUSÃO

O projeto IBM Blue Gene introduziu novos paradigmas para a construção de supercomputadores, sobre tudo o uso de componentes de baixo consumo de energia como processadores embarcados além do emprego de SoCs. Atingiu um baixo consumo de energia em relação ao alto desempenho disponibilizado pelo sistema.

O projeto foi construído e testado fazendo uso de aplicações altamente exigentes em termos de recursos computacionais, caracterizando uma abordagem massivamente paralelizável.

A arquitetura IBM Blue Gene introduziu novos paradigmas no projeto e construção de supercomputadores, levando ao desenvolvimento de sistemas de alta capacidade computacional, cujo custo, alto desempenho e baixo consumo forneceram referenciais atualmente utilizados em outros projetos de supercomputadores não só da IBM mas também de outros fabricantes.

5. REFERÊNCIAS

[1] Adiga, N. R.; Almasi, G.; et. al. "An Overview of the BlueGene/L Supercomputer". Proceedings of the IEEE/ACM SC2002, pp. 60-75, Nov 2002.

[2] IBM – Blue Gene [online]. http://domino.research.ibm.com/comm/research_projects.nsf/pages/bluegene.index.html, acessado em Junho de 2009.

[3] TOP500 Supercomputer Sites – Top List November 2004 [online].http://www.top500.org/list/2004/11/100, acessado em Junho de 2009.

[4] TOP500 Supercomputer Sites – Top List June 2009 [online]. http://www.top500.org/list/2009/06/100, acessado em Junho de 2009.

[5] The GREEN500 List – June 2008 [online]. http://www.green500.org/lists/2008/06/list.php, acessado em Junho de 2009.

[6] Almasi, G.; Almasi, G.S.; et. al. "Cellular supercomputing with system-on-a-chip". Proceedings of the IEEE International ISSCC 2002, vol1, pp. 196-197, Feb 2002.

[7] Almási, G.; Bellofatto, R.; et. al. "An Overview of the Blue Gene/L System Software Organization". Lecture Notes in Computer Science, Euro-Par 2003, vol 2790, pp. 543-555, Jun 2004.

[8] Kermani, P.; Kleinrock, L. "Virtual Cut-Through: A New Computer Communication Switching Technique" Computer Networks, vol. 3, pp. 267-286, 1979.

[9] Snir, M.; Otto, S.; et. al. "MPI – The Complete Reference. Volume 1 – The MPI Core, 2nd edition". The MIT Press, 448 pp, Sep 1998.

Verificação Formal de Blocos Complexos em Arquiteturas

Guilherme Henrique R. Jorge
IC / Unicamp RA: 096231
guijorge@gmail.com

Resumo

O crescimento exponencial de complexidade dos dispositivos de integração em escala muito grande (do inglês VLSI) vem acontecendo continuamente atraves das ultimas décadas e não mostra sinais de desaceleração. Esse crescimento tem levado ao limite nossa capacidade para projetar e em particular para verificar dispositivos VLSI. A industria de projeto VLSI está em constrante inovação, introduzindo novas metodologias e novas tecnologias para lidar com esse crescimento de complexidade[1]. Uma tecnologia promissora que vem sendo desenvolvida recentemente é a verificação baseada em asserções (Assertion-Based Verification ou ABV) com o uso de ferramentas de análise baseadas em simulação e métodos formais. A proposta desse trabalho é prover uma visão sobre ABV e analise formal e suas metodologias.

Palavras-Chave

Verificação Formal, VLSI, Asserções, Analise Formal, Complexidade.

1. Verificação Baseada em Asserções

Fundamentalmente, ABV envolve o uso de asserções como forma de validar parte ou toda a lógica comportamental funcional de um projeto. Asserções são afirmações formais totalmente não ambíguas sobre um comportamento requerido ou esperado. Asserções podem ser criadas e aplicadas em qualquer lugar na hierarquia de projeto. No nível mais alto, elas essencialmente provêem especificações de comportamento geral. Em níveis mais baixos, podem ser usadas para expressar o desejo do projetista. Um simulador pode checkar automaticamente as asserções durante o curso normal em que se roda um testbench enquanto uma ferramenta de analise formal pode checkar totalmente as asserções sem o uso de um testbench[2].

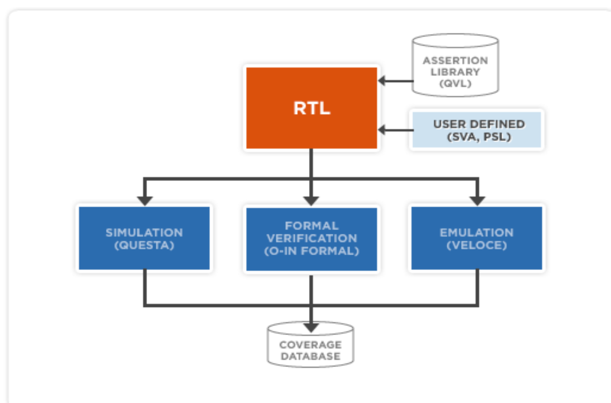


Figura 1. Fluxo de Verificação contendo ABV

O grande beneficio do uso de ABV é o aumento de produtividade no processo de design e especialmente na tarefa de verificação. Existem muitas razões para isso:

- Asserções permitem uma visibilidade melhor do design.

Com o uso de técnicas de verificação tradicionais erros comumente passam desapercebidos. Isso acontece porque normalmente os erros não se propagam até o ponto de serem visíveis ao ambiente de verificação. Uma vez que asserções normalmente monitoram sinais internos, elas provem uma visão melhor do design. O resultado, quando usado em ambientes baseados em simulação, é que elas aumentam a chance de se encontrar um bug no design.

- Asserções provem um mecanismo adicional de provar que o design esta correto.

Os tipos de erros feitos quando se escreve asserções são bem diferentes dos de quando se desenvolve código RTL (Register Transfer Level, ou Verilog/VHDL sintetizável) e de quando se escreve testbenchs.

Por causa disso, erros em RTL e testbenchs serão freqüentemente pegos por asserções, enquanto erros em asserções serão pegos pelo correto exercício funcional do RTL.

- Asserções ajudam na documentação do código.

Normalmente, um projetista escreverá uma asserção que indica uma presunção feita durante o projeto do bloco. Futuramente, o mesmo projetista, ou outra pessoa, pode ler essas mesmas presunções para se familiarizar com o design.

Alem do mais, presunções em um bloco podem ser comparadas com outros blocos afim de assegurar que esses outros blocos não estimulam ou controlam esse bloco de forma diferente da que foi assumida.

- Asserções permitem um debug mais rápido.

Devido ao fato de as asserções estarem associadas a partes bem específicas do hardware que elas verificam, a falha de uma assertiva normalmente indica uma falha num ponto bem específico do RTL. Isso facilita a identificação da causa raiz da falha. O uso de ABV em sistemas baseados em simulação e com o uso de analise formal são complementarias. Asserções escritas para uso em analise formal também podem ser usadas em simulações. Reciprocamente, asserções escritas para uso em sistemas baseados em simulação também podem ser usadas em analise formal. Outras ferramentas como emuladores também podem ser usados para provar as mesmas asserções. Essa

interoperabilidade a nível de asserções possibilita que os esforços feitos para uma forma de ABV sejam reutilizados em outras formas.

2.Analise Formal

Analise formal é o processo de verificação funcional (ao contrário de elétrica, de tempo, etc) de asserções, ou a procura de bugs funcionais em um design através do uso de ferramentas de análise formal[3]. Através do uso de algorítimos sofisticados, ferramentas de analise formal são capazes de checar completamente todos os possíveis estados operacionais e todas as combinações de entrada de um determinado design, como ilustrado na Figura 2.

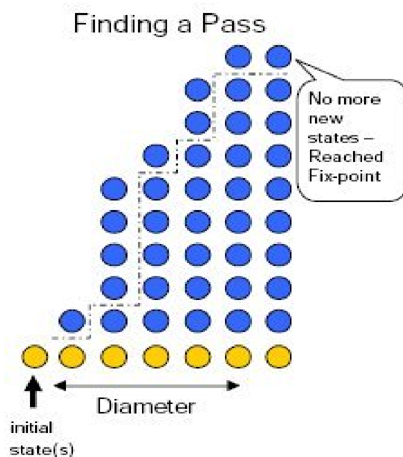


Figura 2. Exploração de asserções.

Por esse motivo, elas pode conclusivamente comprovar que uma asserção é verdadeira em todos os seus estados possíveis.

Analise formal tem ganhado atenção recentemente pelo seu potencial em aumentar a qualidade e a produtividade além do que pode ser alcançado por métodos de verificação tradicionais, ou através do uso de ABV por simulações somente.A qualidade aumenta porque a analise formal encontra bugs que a verificação baseada em simulação pode não encontrar. A produtividade aumenta porque:

- Os bugs são isolados o que torna o debug e correção de problemas mais fácil e rápido.
- Os bugs são identificados mais facilmente e mais cedo no ciclo de projeto.
- Quanto antes se acha um bug, menos pessoas são envolvidas.
- Eliminar os bugs mais cedo significa menos pessoas impactadas e o fluxo de projeto flui melhor.

A combinação de melhor qualidade e produtividade significa melhor time-to-market e reduz custos (pela necessidade de menos re-spins).

2.1 Modelos de Analise formal baseado em asserções

Como mostrado na Figura 3, asserções documentam a proposta do design para as entradas e saídas de um bloco assim como para o comportamento interno do bloco. Asserções que fazem referencia apenas a E/S do bloco são chamadas “black-box” uma vez que elas não dependem da implementação interna do bloco. Asserções “End-to-end” especificam o comportamento das entradas para as saídas “end to end”. Elas então relacionam o resultado esperado nas saídas com os valores das entradas. Asserções que fazem referencia as estruturas internas do design são chamadas “White-box”. Elas ficam “amarradas” a implementação do RTL[4].

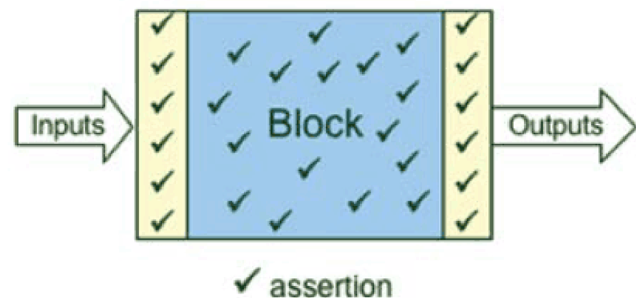


Figura 3. Asserções dentro do design

Asserções que especificam protocolos de entrada e saída refletem não apenas a um único bloco, mas a interação entre o bloco e seus vizinhos. Asserções de saída checam que os resultados do bloco estão de acordo com o que é esperado pelos outros blocos que recebem os seus sinais de saída. De modo inverso, asserções de entrada representam a gama de entradas para a qual o bloco foi projetado. Elas averiguam entradas ilegais vindas de outros blocos.

Asserções de entrada são especialmente importantes para a análise formal, pois podem ser aplicadas ao bloco antes de qualquer testbench ou teste serem escritos. Como mostrado na Figura 4, asserções de entrada são tratadas como constantes (constraints). Isso assegura que a análise formal considera apenas combinações válidas de entrada. Um problema que venha a ser detectado pelo uso de entradas ilegais geralmente não é de interesse do projetista. Quando se está preocupado em verificar o comportamento de um bloco, normalmente, isso não é um bug real uma vez que a especificação do sistema não deve permitir que os outros blocos tenham tal comportamento.

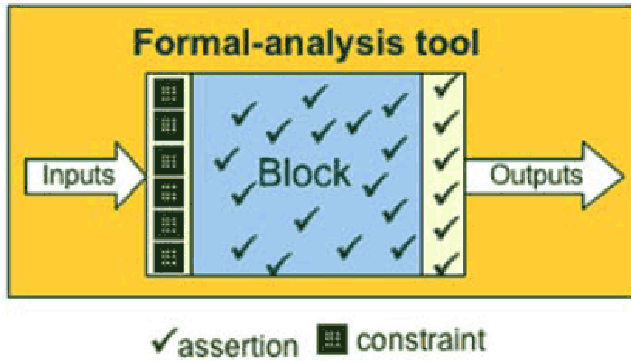


Figura 4. Asserções em analise formal

Para cada asserção que não é convertida em constante, a analise formal tenta provar que a assertiva não será nunca violada ou gerará um "contra-exemplo" mostrando como/quando uma violação pode ocorrer. Algumas ferramentas conseguem gerar o contra-exemplo na forma de caso de teste. Esse caso de teste pode ser rodado em uma simulação. O bug que está causando a falha da asserção pode ser então diagnosticado em um ambiente mais familiar.

A sintaxe de uma asserção simples parece muito com o Verilog tradicional. Uma asserção que garante que uma FIFO não deve sofrer um "underflow" deve ser escrita da seguinte forma:

```
assert_no_underflow:   assert   property
(@(posedge                          clk)
   !(fifo_read && fifo_empty && rst_n));
```

Essa asserção, nomeada de assert_no_underflow pelo projetista, reporta uma violação ou falha quando o sinal de escrita da FIFO e o sinal de full (cheio) da FIFO estão ambos ativos. Esse teste é feito apenas quando o sinal de reset não está ativo e apenas na borda de subida do clock. Sinais em transição não são considerados. Esse é um bom exemplo de uma asserção de grande valor em uma simulação chip-level.

3.Complexidade em Analise Formal

3.1O que define complexidade em Analise Formal?

Em geral, o que define a complexidade em Analise Formal é uma analise que demore um tempo longo para apresentar resultados ou que consuma uma quantidade muito grande memória.

3.2Impacto da complexidade nos resultados

Muitos fatores contribuem para o problema da complexidade em Analise Formal. Além dos fatores de primeira ordem (que incluem características como tamanho e diâmetro do design), existem vários componentes adicionais em um problema complexo (fatores de complexidade de segunda e terceira ordem como a estrutura do design e transições de maquinas de estado). Esses componentes adicionam várias dimensões adicionais ao problema. Para encontrar maneiras de superar a complexidade nós devemos nos concentrar em fatores que podem ser facilmente entendidos e que podem assim nos ajudar a superar a complexidade com relativa facilidade.

Muitas ferramentas de analise formal executam suas analises com base em um tempo limite de time-out. Esse tempo de time-out é geralmente controlado pelo usuário. Quando a complexidade de um problema é muito grande para ser superado em uma dada quantidade de tempo, as ferramentas não irão produzir um resultado de PASS ou FAIL. Nesse caso, a maior parte das ferramentas prove algum tipo de métrica que informa o quanto da análise foi feita até que o tempo limite fosse atingido. No IFV (Incisive Formal Verifier) da Cadence, esse tipo de resultado é chamado de "explored" e a métrica associada é chamada de "depth". Portanto, um resultado "explored" é uma função de tempo em relação a complexidade.

Quando se fala em superar complexidade, quer-se dizer que é desejado diminuir o tempo de execução, aumentar o "depth" de um resultado "explored" ou, mais ainda, transformar um "explored" em um resultado PASS ou FAIL. No final, quer-se ser capaz de conseguir uma melhor performance da ferramenta através da redução da complexidade[5].

3.3Precisão das medidas de complexidade

Conforme os fatores de complexidade são encontrados, tenha em mente que esses fatores variam muito de caso para caso. Por exemplo, de um lado é possível ver casos aonde as medidas de complexidade são altas, mas as ferramentas ainda sim são capazes de convergir a um resultado em um tempo razoavelmente curto e usando uma quantidade de memória pequena. Isso é possível porque as ferramentas usam meios automáticos de superar as complexidades.[6]

De outro lado, é possível encontrar casos aonde as medidas de complexidade são baixas e ainda sim as ferramentas não chegam a resultados conclusivos.

A complexidade nesses casos é causada por fatores de segunda e terceira ordem cuja discussão e analise estão além do escopo desse trabalho.

Em outras palavras, não é sempre possível de se fazer uma estimativa direta do desempenho a ser esperado das ferramentas baseado apenas nos fatores aqui apresentados. Existe sempre a possibilidade de se encontrar os dois tipos de caso listados: altas medidas de complexidade com bons resultados e baixas medidas de complexidade com resultados "explored".

4.Conclusão

Com todos os benefícios que o uso da análise formal prove, pode parecer que tudo o que precisa ser feito para verificar um projeto é:

- Escrever asserções que especifiquem como o design deve se comportar.
- Colocar o design e as asserções em uma ferramenta formal e analisar se as asserções provam o comportamento correto ou não do design.

Há se fosse tão fácil! Na realidade existem outras coisas que precisam ser levadas em consideração quando do uso de analise formal em designs na vida real.

As ferramentas de analise formal tem mais limitações que as simulações em termos de complexidade e tamanho do design com que elas podem trabalhar pelo simples motivo que o numero de estados que precisam ser analisados cresce de forma exponencial de acordo com o numero de elementos em cada design.

Isso significa que poucos chips hoje podem ser analisados por métodos formais em apenas uma grande rodada. E em outros casos, blocos de complexidade relativamente alta podem ser analisados separadamente.

Isso não significa que a analise formal não dê bons retornos ao seu investimento em designs reais, mas sim que para conseguir bons resultados você precisa de bons modelos, metodologias, e o treinamento necessário para usar correta e eficientemente as ferramentas.

5.REFERENCIAS

[1] Maniliac, David. 2002 Assertion-Based Verification Smoothes The Road to IP Reuse. Electronic Design Online ID #2748

[2] Lee, James M. Assertion Based Verification. http://www.jmlzone.com/

[3] Bjesse, Per. 2005 What is formal verification? ACM, New York, NY, USA.

[4] Anderson, Tom. 2007 SystemVerilog Assertions and Functional Coverage Support Advanced Verification. Chip Design Magazine

[5] Cadence web site. http://www.cadence.com

[6] Mentor web site. http://www.mentor.com

Execução Especulativa

Washington Luís Pereira Barbosa – RA971766
MO401 – 1º semestre de 2009

RESUMO

Para que o potencial de desempenho oferecido pelos processadores com pipeline seja explorado, é necessário que não haja paralisação do processador por conta de dependências no fluxo de execução.

A execução especulativa é uma técnica que tem como objetivo reduzir impactos de latência, executando instruções antes que suas dependências sejam totalmente resolvidas. Esse trabalho examina de forma geral os principais conceitos e taxonomia da técnica, relaciona suas principais variações e faz uma descrição mais cuidadosa das técnicas aplicadas às dependências de dados.

Categoria e Descrição de Assunto

Arquitetura de Computadores

Termos Gerais

Execução Especulativa

Palavras chave

Predição de endereço, predição de valor, Renomeação de memória

INTRODUÇÃO

O paralelismo em nível de instrução, possibilitado pelo recurso de *pipeline* nos processadores, representa a execução de diferentes ciclos de instruções distintas, ou mesmo a execução simultânea de múltiplas instruções, distribuídas entre diferentes unidades funcionais. Todavia, pode haver restrições que impeçam instruções distintas de serem realizadas paralelamente ou condicionem o início de execução de uma instrução ao término de outra. A existência ou não dessas dependências determina o grau do paralelismo em nível de instrução, ao qual está condicionado também o desempenho que pode ser alcançado pelo processamento de forma geral.

Existem diversos tipos de restrições, ou dependências. Uma delas é a concorrência pela utilização da mesma unidade funcional. Uma arquitetura que dispõe de apenas uma unidade funcional não pode, por exemplo, ter duas operações aritméticas escalonadas ao mesmo tempo. Esse tipo de restrição é usualmente denominado como *hazard* funcional. Alguns algoritmos são capazes de realizar um escalonamento adequado para instruções em teoria incompatíveis, minimizando dessa forma os inconvenientes desse tipo de dependência restrição.

Existem também outros tipos de restrições que, em teoria, ditam a ordem que as instruções devem ser executadas: as dependências de dados e de controle. Freqüentemente essa ordem limita o paralelismo que pode ser extraído de programas seqüenciais, reduzindo o desempenho. As dependências de dados ocorrem quando uma instrução necessita de um valor que ainda não foi calculado. As dependências de controle, por sua vez ocorrem quando há paralisação do *pipeline*, em virtude da espera pela decisão do fluxo de execução. Ambos os tipos de dependências podem ter seus efeitos reduzidos ou eliminados através de análise e **execução especulativa**.

Execução especulativa consiste em um conjunto de técnicas para antecipar a execução de instruções antes que todas as dependências tenham sido resolvidas. A não resolução de todas as dependências pode significar que instruções são executadas desnecessariamente ou erroneamente. Todavia, o ganho representado pela não paralisação do *pipeline* é comumente maior que o custo de possíveis execuções equivocadas.

A antecipação proporcionada pela execução especulativa, na maioria dos casos, é baseada em análise estatística dos resultados previamente obtidos. Essa análise é realizada explorando-se a localidade de valor, espaço e tempo, que é observada na imensa maioria dos softwares desenvolvidos.

PREDIÇÃO DE ENDEREÇO

Todas as instruções *load* em um programa necessitam que o endereço efetivo do dado desejado seja calculado antes da leitura ser executada. Se uma dessas instruções encontra-se no caminho crítico de execução, seria benéfico para o desempenho se o endereço da instrução pudesse ser previsto, e assim o dado alvo carregado tão breve quanto possível.

Dessa forma, as técnicas de predição de endereços procuram basear-se no conceito de localidade temporal, ou seja, posições de memória uma vez acessadas tendem a ser repetidas no futuro, ou mesmo endereços visinhos à instrução executada

tendem a ser executados na seqüência. Desta forma tentamos predizer os endereços e aumentar o desempenho na execução do programa.

Em geral, endereços utilizados por mesmas instruções de *load* ou *store* diferem do endereço utilizado na execução anterior apenas por uma constante. Vale salientar que esse fato é bastante típico, principalmente no que diz respeito ao acesso de vetores na memória e ressaltando o conceito de localidade espacial.

Baseado nesse resultado é proposta uma estratégia simples e eficaz para predição de endereços que consiste em determinar os endereços durante a decodificação das instruções através de uma tabela. Essa tabela é indexada através dos últimos bits da instrução e contém três campos: o endereço anterior, o valor do passo e um contador, cujo bit mais significativo indica se a instrução pode ser predita ou não.

Instruções executadas especulativamente devem ser verificadas. Em caso de acertos, o contador é incrementado e o endereço atualizado. Em caso de erro, o contador é decrementado e endereço e passo são modificados para que possam refletir a nova realidade.

PREDIÇÃO DE VALOR

Técnicas de predição de valores são baseadas no conceito de localidade de valor, ou seja, a probabilidade de um mesmo valor ser encontrado em sucessivos acessos ao conteúdo de um registrador ou endereço de memória.

Abaixo são listadas algumas motivações que procuram justificar a existência de localidade de valor em programas reais, como por exemplo:

• Redundância de dados, como ocorre em matrizes esparsas.

• Constantes do programa.

• Spill de registradores.

Foi realizada uma série de experimentos para mensurar a localidade de valor em leituras da memória e acesso a registradores utilizando um benchmark de 17 programas, utilizando o SPEC95.

Na figura 1 é mostrada graficamente uma avaliação de localidade de valor para instruções load realizada por esses pesquisadores. A localidade apresentada por cada um dos programas listados no eixo horizontal foi estimada contando o número de instruções nas quais o valor lido corresponde a um valor obtido em uma execução anterior da mesma instrução e dividindo pelo total de instruções de leitura.

São apresentadas duas estimativas. Na primeira delas, representada pelas barras claras, é verificado se o valor lido corresponde ao obtido na execução imediatamente anterior.

Na segunda, representada pelas barras escuras, é considerado um histórico de seis execuções.

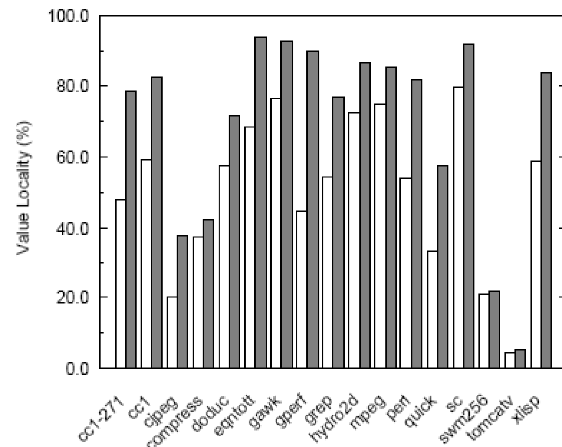


Figura 1: Localidade de valor em acessos à memória

A grande maioria dos programas apresenta correspondência entre 40 e 60% para o primeiro caso e uma significativa melhora desse valor quando um histórico maior é utilizado, alcançando índices superiores a 80%. Apenas quatro programas (cjpeg, compress, swm256 e tomcatv) apresentam pouca localidade. Como pode ser observado, a grande maioria dos programas apresenta correspondência entre 40 e 60% para o primeiro caso e uma significativa melhora desse valor quando um histórico maior é utilizado, alcançando índices superiores a 80%.

Apenas três programas (cjpeg, swm256 e tomcatv) apresentam pouca localidade.

A figura 2 representa a aplicação da mesma metodologia para estimativa de localidade de valor em registradores, ou seja, foi contado o número de vezes em que é escrito em um registrador um valor previamente armazenado nele, dividindo-o pelo número total de escritas em registradores. Quando apenas o histórico mais recente é utilizado, o índice médio de localidade apresentado pela maior parte dos programas avaliados fica em torno de 50%, para um histórico de quatro valores, essa taxa é de 60% aproximadamente.

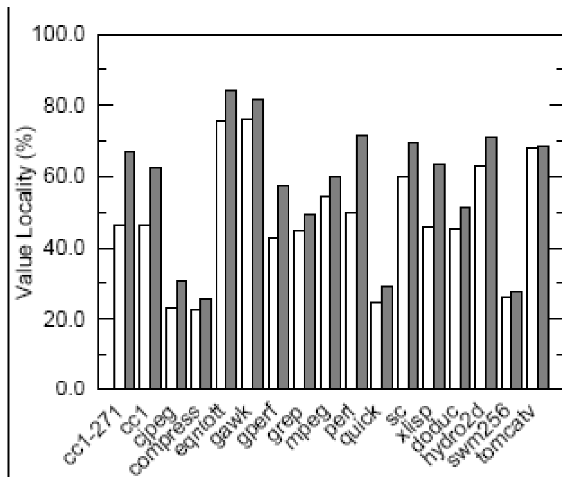


Figura 2. Localidade de valor em escritas a registradores

Para explorar a localidade de valor apresentado pela maioria dos programas, adicionando um grau maior de liberdade para o escalonamento de instruções, uma melhor utilização dos recursos disponíveis e, possivelmente, uma redução do tempo de execução, o trabalho apresenta uma implementação de um mecanismo de predição de valores baseados em duas unidades: uma de predição de valores e outra de verificação.

A unidade de predição de valores é composta por duas tabelas indexadas através dos últimos bits de endereço da respectiva instrução, como mostrado na figura 3.

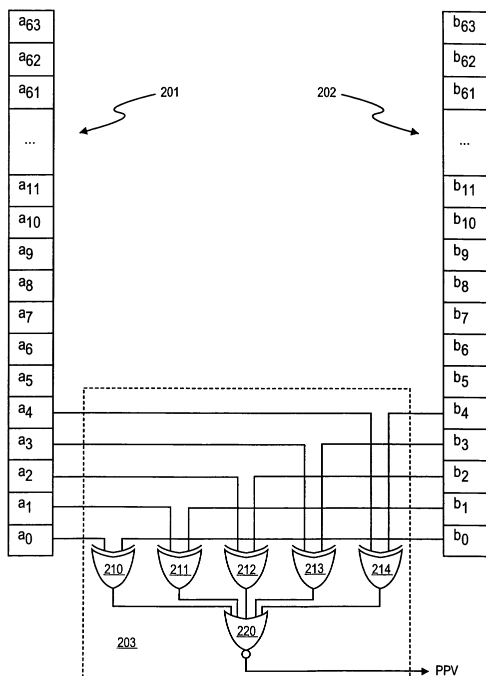


Figure 3: Unidade de Predição de Valores

A tabela de classificação armazena um contador que é incrementado ou decrementado de acordo com acertos ou erros de predição e é utilizado para classificar a instrução como previsível ou não. A tabela de predição contém o valor esperado. Ambas possuem um campo de válido, que pode ser um bit indicando se a entrada é válida ou não ou um campo que deve ser comparado aos bits mais significativos do contador de instruções para verificar a validade da respectiva entrada.

RENOMEAÇÃO DE MEMÓRIA

A técnica de Renomeação de Memória consiste basicamente no processo de localização das dependências entre instruções *store* e *load*. Uma vez que essas dependências são identificadas é possível realizar a previsão, comunicando o dado armazenado pela instrução *store* para a instrução *load* na seqüência.

A figura 4 ilustra uma técnica para comunicação efetiva de memória. A arquitetura é constituída por uma Store Cache para armazenar instruções store recentes (4K entradas diretamente mapeadas), uma Store/Load Cache para armazenar dependências localizadas (4K entradas diretamente mapeadas) e uma Value File para predição de valores. A técnica conta também com um mecanismo responsável por determinar quando usar a predição.

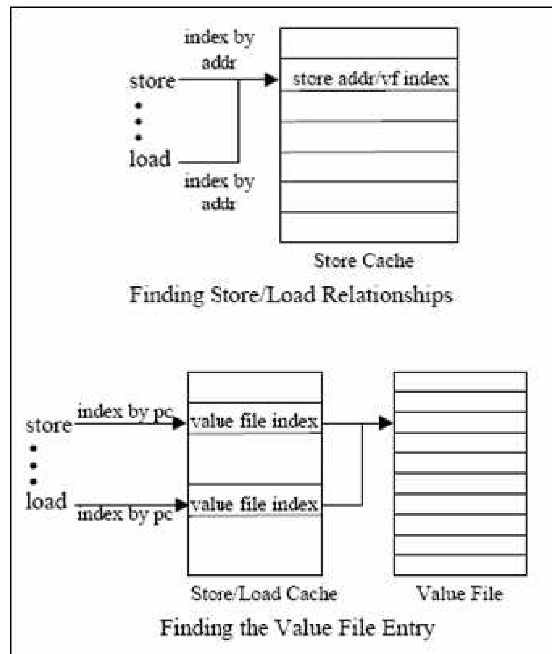


Figura 4. Técnica de predição de valores

Quando uma instrução *store* é decodificada, esta é indexada na Store/Load Cache com o store PC

para localizar a entrada Value File. Se não localizada, a instrução *store* é alocada como entrada recente na Value File e é atualizada uma nova entrada na Store/Load Cache.

CONCLUSÕES

Apresentamos de forma geral as principais restrições de desempenho nos processadores com pipeline, e como as técnicas de Execução Especulativa podem auxiliar a reduzir o efeito dessas limitações. Uma classificação e taxonomia das técnicas foram apresentadas, em relação ao tipo de dependência, de controle ou de dados, que esta procura solucionar.

Após essa conceituação geral, apresentamos ao longo do texto algumas abordagens de execução especulativa de dados, assim como algumas respectivas análises de desempenho. Por fim citou-se a importância do tratamento exceções e de predições incorretas.

Ao longo do trabalho podemos observar a relevância da aplicação da técnica, propiciando o aumento do grau de paralelismo em nível de instrução e evitando paralisações do processador. Todavia, o aumento de desempenho obtido é dependente da técnica utilizada e das características do programa em execução. Isso significa que técnicas de execução especulativa, especialmente as que realizam predições, terão efeitos mais significativos em programas que apresentam alta localidade.

BIBLIOGRAFIA

[1] J. L. Hennessy and D. A. Patterson. Arquitetura de Computadores Uma Abordagem Quantitativa. Campus, Rio de Janeiro. Pages 126-210, 2003

[2] J. Gonzalez and A. Gonzalez. Memory address prediction for data speculation. Technical report, Universitat Politecnica de Catalunya, 1996.

[3] G. Tyson and T. M. Austin. Improving the accuracy and performance of memory communication through renaming. In 30th Annual International Symposium on Microarchitecture, pages 218–227, December 1997.

[4] M. H. Lipasti and J. P. Shen. Exploiting value locality to exceed dataflow limit. International Journal of Parallel Pogramming, 26(4):505–538, 1998.