Computação de Alto desempenho: Clusters

Leandro Rodrigues Magalhães de Marco - RA009089
Instituto de Computação - UNICAMP
Avenida Albert Einstein, 1251
Campinas, SP - Brasil
ra009089@students.ic.unicamp.br

ABSTRACT

Existem limitações para o poder de processamento de máquinas individuais. Por mais que se procurem soluções em máquinas multiprocessadas, essas limitações sempre existirão.

O emprego de cluster, que são diversas máquinas conectadas em rede e que podem ser vistas como apenas um supercomputador de fora, ajuda a resolver o problema de processamento que não poderia ser realizado em tempo hábil por uma máquina isolada.

Clusters também são aplicados para aumentar, através de redundância, a confiabilidade de sistemas que devem ser tolerantes a falhas.

Com a popularização dos PCs, tornou-se possível criar-se clusters de baixo-custo e que podem superar o poder de processamento do mais caro supercomputador.

Keywords

Computação de alta performance, Arquitetura de Computadores, Clusters

1. INTRODUÇÃO

A partir do momento em que usuários de Computadores notaram que todo o trabalho computacional não poderia ser realizado em tempo hábil ou de forma confiável por um simples computador, surge a idéia de dividir o trabalho em diversas máquinas.

Um Cluster Computacional é um conjunto de computadores que estão ligados em rede e que trabalham em conjunto. Cada computador de um cluster é denominado nó. Dependendo-se do objetivo do Cluster, esses nós podem processar pequenas partes de um problema maior que foi dividido ou podem servir como nós de reserva para garantir um sistema toletrante a falhas.

O objetivo de se construir um cluster é em geral, ganho de desempenho e/ou confiabilidade quando comparado ao que seria obtido em uma única máquina. Um cluster também possui um custo mais baixo do que um único computador com velocidade e confiabilidae comparáveis.

2. A HISTÓRIA DOS CLUSTERS

Uma citação de *In Search of Clusters de Greg Pfister*[4], ajuda a definir a origem dos Clusters: "Os clientes inventaram os clusters a partir do momento em que não conseguiam colocar todo o trabalho em um simples computador, ou precisavam de um backup. A data do prmeiro Cluster é desconhecida, mas eu me surpreenderia se não fosse nos anos 60 ou no fim dos anos 50".

Entretanto, a base formal do Cluster computacional como meio de executar tarefas em paralelo foi criada por *Gene Amdahl* da IBM, que em 1967 publicou o artigo *"Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities"*[1]. Neste, Amdahl apresenta a conhecida *Lei de Amdahl* que descreve matematicamente o ganho de velocidade resultante da paralelização de uma tarefa serial. Este artigo contém a base para para multiprocessadores e clusters. A diferença entre paralelizar utilizando-se multiprocessamento e cluster é que nos primeiros a comunicação é feita dentro do computador através de barramentos, etc. enquanto nos últimos a comunicação é feita fora, através de rede.

Devido a isso, a história dos clusters está intimamente ligada a históroa das redes de computadores. Uma das motivações do desenvolvimento de redes, era ligar computadores para poder-se somar recursos, construindo-se assim clusters. A internet pode ser encarada como um imenso cluster. Iniciativas como o projeto *SETI@home*, se valem desse imenso poder de processamento que a internet possui para executar tarefas e cálculos complicados com maior eficiência.

Outro aspecto importante que possibilita a existencia de Clusters é a definição de protocolos padrão para comunicação (TCP/IP).

O primeiro cluster comercial foi o *ARCNET* desenvolvido pela Datapoint em 1977. O ARCNET não foi um grande sucesso comercial e a idéia de Cluster só obteve sucesso comercial em 1984 quando a DEC lançou *VAXcluster* para o sistema operacional *VAX/VMS*. Este produto suportava além da execução de tarefas em paralelo, sistemas de ar-

quivos e dispositivos compartilhados. Este sistema proporcionava a vantagem da computação paralela mantendo a confiabilidade da integridade dos dados. Outros dois dos primeiros clusters comerciais foram o *Tandem Himalaya* e o *IBM S/390 Parallel Sysplex*.

Com o invento do *PVM: Parallel Virtual Machine*, e com a popularização dos PCS, foi possível construir supercomputadores a partir de PCs normais. Foram criados Clusters de baixo custo com poder de processamento maior que o dos melhores e mais caros supercomputadores. Em 1993 a NASA iniciou um projeto para a criação de supercomputadores a partir de PCs comuns.

3. TIPOS DE CLUSTERS

- *High-Availability (HA) Clusters*

  Clusters de alta disponibilidade são construidos para aumentar a disponibilidade de serviços que os mesmos provém. Isto é feito através de nós redundantes que são usados em caso de falha de componentes. Comumente estes Clusters possuem dois nós, que é o mínimo necessário para prover redundância.

  Entretanto existem clusters com até dezenas de nós, dependendo do nível de confiabilidade desejada.

  Algumas configurações deste tipo de Cluster estão descritas em *"Blueprints for High Availability: Designing Resilient Distributed Systems"*[3] Podemos categorizar as possíveis configurações destes cluster da seguinte maneira:

  - Active/Active: o tráfego que se destinaria a um nó que falhou é ou passado a um nó existente ou balanceado entre os nós remanescentes. Usualmente exige que os nós possuam uma configuração de software homogênea.
  - Active/Passive: Cada nó contém uma instância redudante que é ativiada apenas em caso de falha. Esta configuração é a que requer a maior quantidade de hardware.
  - N+1: Existe um nó extra que é ativado para substituir um nó que venha a falhar. Em caso de configurações heterogêneas de software, o nó extra deve ser capaz de assumir qualquer uma das tarefas dos nós principais. Esta nomenclatura se aplica para clusters com múltiplos serviços. No caso de um serviço só, este modelo é equivalente ao Active/Passive
  - N+M: Em clusters que executam muitos seviços, um nó redundante apenas pode não ser suficiente. Neste caso M nós redundantes são colocados, dependendo dos requisitos de custo e confiabilidade.

- *Load Balancing Clusters*

  Estes clusters operam tendo toda carga de trabalho chegando a um ou mais "front-ends" que distribuem a carga entre os demais nós (denominados "back-end servers"). Este tipo de Cluster é utilizado para prover alta-performance, entretanto ele comumente possui recursos que adicionam características de Clusters de alta-disponibilidade. São conhecidos também como "Server Farm"

- *High Performance (HPC) Clusters*

  Clusters de alta-performance são construidos com o objetivo de aumentar a velocidade de tarefas dividindo-as entre inúmeros nós.

  Este tipo de cluster é mais comumente utilizado para aplicações científicas. Muitas aplicações que rodam em HPC foram desenvolvidas especialmente para tirar proveito do paralelismo oferecido pelo cluster.

  Clusters HPC são otimizados para rodar tarefas que executam em nós separados e comunicam-se entre si ativamente. Isso inclui problemas nos quais resultados intermediários de um nó afetam resultados de computações futuras em outros nós.

  Beowulf é uma das mais populares implementações de HPC. O Beowulf é um modelo criado para obter um sistema de alta performance com baixo custo. Roda em Sistema Operacional Linux e utiliza Software livre para explorar paralelismo.

- *Grid Computing*

  Grid[2] é uma tecnologia que é parecida com a tecnologia de Cluster. A principal diferença é que um Grid conecta computadores que não confiam totalmennte uns nos outros. Grids suportam ambientes mais heterogêneos do que clusters.

  O grid é otimizado para cargas de trabalho que consistem de muitas tarefas independetes entre si e que portanto não tem que se comunicar durante o processamento.

  Assim, grids administram distribuição de tarefas para computadores que trabalharão independentemente. Alguns recursos como os de armazenamente podem ser compartilhados, mas resultados intermediários de uma tarefa em um nó não alteram a computação de tarefas em outros nós.

4. TECNOLOGIAS DE CLUSTERS

Alguns exemplos de tecnologias que estão relacinadas a Clusters

- distcc: permite a execução de compilação paralela quando se utiliza o GCC
- MPI: biblioteca que implementa um protocolo de comunicação entre processos rodando em diferentes nós de um Cluster.
- Linux Virtual Server, Linux-HA: clusters que implementam soluções de balanceamento de carga entre nós
- MOSIX, openMosix, Kerrighed, OpenSSI: cluster integrados ao Kernel do Sistema Operacional e permitem migração automática de processos entre nós homogêneos.
- MSCS: Solução da Microsoft para clusters de alta-disponibilidade em ambiente Windows.

5. CONCLUSÕES

A idéia de dividir tarefas ou mesmo recursos em computação não é nova. Muito provavelmente clusters surgiram como uma solução natural para problemas que ocorriam devido a limitações das máquinas. Limitações estas que existem até hoje e muito provavelmente irão existir no futuro. A evolução dos Clusters anda de mãos dadas com a evolução das redes de computadores e da criação de padrões de comunicação. Talvez a maior revolução trazida pela tecnologia de Cluster, seja a possibilidade de montar supercomputadores com máquinas comuns. Possuimos por exemplo na internet, um imenso poder computacional que é em sua maioria inexplorado.

6. REFERENCES

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings*, pages 745–770, 1967.

[2] I. Foster. What is the grid? a three point checklist.

[3] E. Marcus and H. Stern. *Blueprints for High Availability: Designing Resilient Distributed Systems.* John Wiley Sons.

[4] G. Pfister. *In Search of Clusters.* Prentice Hall.

Arquiteturas VLIW*

Uma Alternativa Para a Exploração de ILP

Rafael Augusto Scaraficci (RA: 009649)
Instituto de Computação - UNICAMP
Caixa Postal 6176
Campinas - SP, Brasil
ra009649@students.ic.unicamp.br

RESUMO

Os processadores VLIW possuem múltiplas unidades funcionais, permitindo-lhes executarem várias operações por ciclo de clock. No entanto, diferentemente dos processadores superescalares, o hardware do VLIW é bem simples e o escalonamento das instruções é de total responsabilidade do compilador. Neste trabalho é descrito os conceitos fundamentais da arquitetura VLIW assim como as suas vantagens e desvantagens. Também focamos nas principais técnicas de exploração estática de ILP (*Instruction Level Paralellism*), uma vez que o desempenho da arquitetura é totalmente depende do compilador. As técnicas abordadas são: *Loop Unrolling*, *Software Pipelining*, *Trace Scheduling*, *SuperBlock Scheduling* e *HyperBlock Scheduling*. O trabalho é finalizado com uma perspectiva futura sobre o desenvolvimento e a comercialização dos processadores VLIW.

Termos Gerais

Arquitetura de Computadores

PALAVRAS CHAVES

VLIW, Paralelismo em Nível de Instrução, Escalonamento Estático

1. INTRODUÇÃO

Very Long Instruction Word, mais conhecida como VLIW é um paradigma que propõe uma implementação em hardware simples e uma alta capacidade de processamento. As arquiteturas VLIW tentam alcançar um alto nível de ILP (*Instruction Level Parallelism* [1]) através da execução de instruções longas, que são compostas de múltiplas operações independentes pré-escalonadas pelo compilador. A CPU VLIW, diferentemente da superescalar, não contém nenhum hardware especializado para escalonar ou verificar dependências entre as instruções, toda essa complexidade lógica é transferida ao compilador.

A arquitetura VLIW surgiu no final da década de 70 como uma evolução do microcódigo horizontal. Mas foi o trabalho de Joseph Fisher sobre *trace scheduling* [2] que impulsionou o desenvolvimento deste novo conceito de arquitetura.

Motivados com os resultados, Fisher e alguns colegas de Yale fundaram em 1984 a Multiflow, com o objetivo de criar supercomputadores VLIW. No mesmo ano, Bob Rau fundou a Cydrome que também tinha o mesmo objetivo. Apesar de ambas empresas lançarem produtos finais, o mercado ainda não estava pronto para absorvê-la, além disso a tecnologia proposta era muito prematura na época. Isto fez com que as empresas entrassem em crise financeira e encerrassem as suas atividades anos mais tardes.

Com o fechamento dessas empresas, o desenvolvimento da tecnologia VLIW procedeu-se de maneira lenta, até que em meados da década de 90, descobriu-se que a arquiteturad VLIW eram ideais para o processamento de algoritmos complexos e repetitivos. Isto re-impulsionou a produção de processadores VLIW, principalmente voltados para o processamento digital de sinais e multimídia. O primeiro grande sucesso de um processador VLIW, foi a série C6X da Texas Instruments.

O grande sucesso dos DSPs da Texas Instruments, alavancou o projeto de uma nova geração de processadores VLIW. Em 2000, a Transmeta lançou os processadores Crusoe, sendo o foco deste projeto o mercado de processadores para dispositivo embarcados. Ademais, a Intel manteve a mesma perpectiva, dando continuidade a linha Intanium.

O ressurgimento da tecnologia VLIW é uma conseqüência da tecnologia certa disponível no tempo certo. Em meados da década de 70, a arquitetura VLIW era praticamente implementável, devido ao preço da memória naquela época. Enquanto uma instrução VLIW tinha centenas de bits, uma instrução CISC (tecnologia adotada na época) podia ser muito pequena. No entanto, nos dias de hoje, a memória deixou de ser um problema e a arquitetura VLIW passou novamente a ser uma solução viável.

*Trabalho da disciplina MO401 - Arquiteturas de Computadores I, ministrada pelo professor Paulo Cesar Centoducatte, 1^o semestre de 2006

[1]Uma tradução deste termo é Paralelismo em Nível de Instrução. Todavia, optou-se por manter o termo em inglês para se fazer uso da sigla ILP que é amplamente conhecida.

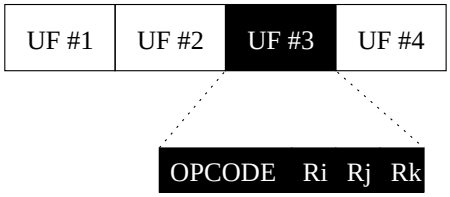


Figure 2: Esquema de uma Instrução VLIW

CARACTERÍSTICAS	VLIW
Tamanho da Instrução	Tamanho único
Formato da Instrução	Regular, posição consistente dos campos
Semântica da Instrução	Várias operações simples e independentes
Registradores	Vários, propósito geral
Acesso à Memória	Arquitetura do tipo *load-store*
Foco do Projeto de Hardware	Simples, explora múltiplas unidades funcionais, lógica de despacho de baixa complexidade

Table 1: Características da Arquitetura VLIW

O objetivo deste trabalho é introduzir os conceitos básicos das arquiteturas VLIW juntamente com as principais técnicas de compilação para a exploração de ILP. Lista-se também as vantagens e desvantagens desta arquitetura e a sua tendência futura. Este trabalho está organizado da seguinte forma. Na seção 2 é apresentada as características fundamentais das máquinas VLIW. Na seção seguinte é introduzido várias técnicas para a exploração estática de ILP. Na seção 4 são listadas as principais vantagens e desvantagens dos processadores VLIW. Na seção 5 é apresentado a perpectiva futura para a arquitetura. E por fim, a seção 6 encerra o trabalho com uma breve conclusão.

2. ARQUITETURA VLIW

Uma máquina VLIW genérica [4] pode ser esquematizada como uma máquina do tipo *load-store* [7] com múltiplas unidades funcionais que são conectadas a um banco central de registradores de propósito geral (vide Figura 1). Todas as operações a serem executadas simultaneamente nas diversas unidades funcionais são encapsuladas em uma única instrução, que pode ser imaginada como um conjunto de instruções RISC (vide Figura 2). Este escalonamento de várias operações em uma única instrução é feita estaticamente pelo compilador o que simplifica o hardware, que não precisa escalonar as instruções para explorar ILP como nos casos das máquinas superescalares.

As principais características da arquitetura VLIW estão sintetizadas na Tabela 1. Observando-a, podemos notar que a arquitetura VLIW tende a ser parecida com a arquitetura RISC [7], diferindo apenas pelo fato das instruções conterem múltiplas operações pré-escalonadas e o hardware explorar várias unidades funcionais utilizando uma lógica simples.

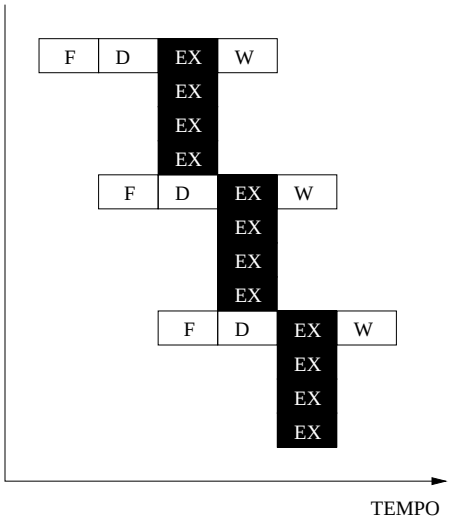


Figure 3: Modelo de Pipeline - VLIW

Um exemplo de pipeline com quatro unidades funcionais e quatro estágios: *fetch* (F), decodificação (D), execução (EX) e escrita (W), para uma máquina VLIW genérica é apresentado na Figura 3. Neste pipeline, faz-se o *fetch* de uma única instrução por vez, porém esta instrução longa especifica várias operações a serem executadas simultaneamente. Como a instrução VLIW é regular e contém os *opcodes* de todas as operações encapsuladas, o processo de decodificação fica simples. No estágio de execução, as operações são despachadas para as unidades funcionais, não havendo nenhum hardware extra para a detecção e tratamento de dependências, sendo responsabilidade do processador gerenciar tudo isso.

A eficiência deste tipo de pipeline fica limitada a capacidade do compilador em encapsular as operações em instruções longas, pois uma baixa densidade de operações por instrução irá resultar em unidades funcionais ociosas e conseqüentemente num baixo número de operações completadas por vez. No entanto, apesar das diversas técnicas aplicadas em compiladores, encapsular às instruções com operações que preencham todas as unidades funcionais é uma tarefa difícil e às vezes impossível. Por exemplo, numa máquina VLIW com unidades funcionais de inteiros e ponto flutuantes, não é possível gerar trabalho para as unidades de ponto flutuantes durante a execução de uma aplicação de inteiros.

Outros problemas inerentes a essas instruções esparsas são: o desperdício de espaço na memória e na cache de instruções, além de banda de transmissão. Uma das soluções para esse problema é o uso de técnicas de compressão de dados. No entanto, o fato de haver ou não compressão tem impacto direto na complexidade da unidade de *fetch*. Segundo [4] pode-se classificar a arquitetura VLIW em duas categorias de acordo com o tipo de codificação da instrução, conforme descrito a seguir:

- *Uncompressed encoding* - formato de instrução com número fixo de bits, os NOPs [2] são codificados ex-

[2]Cosidera-se como NOP qualquer operação que não faz nada

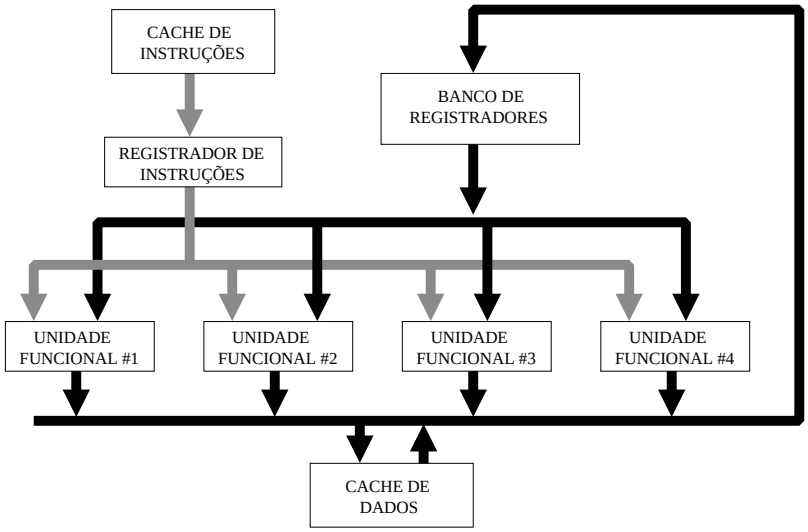


Figure 1: Esquema de uma Arquitetura VLIW Genérica

plicitamente quando uma operação não pode ser escalonada para a instrução.

- *Compressed encoding* - formato de instrução com um número variável de bits, os NOPs não são codificados.

3. ESCALONAMENTO ESTÁTICO

Embora as arquiteturas VLIW reduzam a complexidade do seu hardware com relação a outras arquiteturas como as superescalares, elaa necessitam de um compilador muito mais complexo, que seja capaz de explorar um alto grau de paralelismo em nível de instrução a fim de garantir um bom desempenho do processador. Como o compilador é responsável pelo escalonamento das operações a serem executadas em paralelo (encapsulamento das operações nas instruções longas) e pelo tratamento das dependências, a arquitetura é exposta ao compilador, permitindo-o um alto grau de liberdade para explorar os recursos da arquitetura.

Para garantir um bom desempenho do processador VLIW, várias técnicas de otimização e escalonamento de código são utilizadas. Para o escalonamento de operações tem-se basicamente dois métodos: *basic block scheduling* e *extended basic block scheduling* [1]. O primeiro método explorar ILP dentro de um bloco básico. Esta técnica acaba sendo muito limitada uma vez que um bloco básico tem em média 4–5 operações interdependentes. O segundo método estende os blocos básicos de forma a aumentar o número de operações interdependentes. A seguir são descritas algumas destas técnicas.

3.1 Loop Unrolling

O *loop unrolling* [3, 8, 5] é uma técnica para se aumentar o tamanho do corpo do loop através da diminuição do seu número de iterações, permitindo desta forma uma exploração de paralelismo entre instruções que encontram-se em diferentes iterações. Esta técnica basicamente reorganiza o corpo do loop, seguindo os seguintes passos:

de útil à execução do programa.

1. Replica-se o corpo do loop n vezes.
2. Exceto para o último bloco, remove-se as instruções de incremento/decremento do contador do loop.
3. Multiplica-se o incremento/decremento por n e ajusta os *offsets*.
4. Exceto para o último bloco, remove-se as instruções de teste/desvio do loop.

Na Figura 4(b) é apresentado o código para o desenrolamento de uma iteração do loop da Figura 4(a). Como pode ser observado, após ser aplicado o *loop unrolling* o corpo do loop passa a ser composto por instruções de duas iterações adjacentes, além disso o número de iterações foi reduzido pela metade. Conforme mostrado em [9], esta técnica provê os seguintes benefícios:

1. Elimina um número significante de instruções de desvios.
2. Reduz o número total de instruções despachadas, pois elimina-se instruções de incremento/decremento que são desnecessárias entre blocos adjacentes.
3. Permite um melhor escalonamento das instruções, pois aumenta o tamanho do bloco básico.

3.2 Sotware Pipelining

O *software pipelining* [3, 5] é uma técnica que reorganiza o loop de forma que cada iteração no loop reestruturado é formado por instruções escolhidas de diferentes interações do loop original (vide Figura 4(c)). O objetivo desta técnica é eliminar as dependências que ocorrem entre as instruções de uma iteração do loop. O *software pipelining* também é conhecido como *symbolic loop unrolling*, devido a sua capacidade de intercalar instruções de diferentes iterações, sem explicitamente desenrolar o loop.

for($i = 1, i \leq 1000, i = i + 1$){
$I_1(i)$
$I_2(i)$
$I_3(i)$
}

(a) Loop Original

for($i = 1, i \leq 999, i = i + 1$){
$I_1(i)$
$I_2(i)$
$I_3(i)$
$I_1(i+1)$
$I_2(i+1)$
$I_3(i+1)$
}

(b) ***Loop Unrolling***

$I_1(1)$
$I_2(1)$
$I_1(2)$
for($i = 1, i \leq 1000, i = i + 1$){
$I_1(i)$
$I_2(i-1)$
$I_3(i-2)$
}
$I_3(999)$
$I_2(1000)$
$I_3(1000)$

(c)*Software Pipelining*

Figure 4: *Loop Unrolling* e *Sotware Pipelining*

No exemplo apresentado na Figura 4(c), observa-se que a aplicação do *software pipelining* resultou na intercalação de instruções de três iterações diferentes do loop orignal (vide Figura 4(a)). O código adicionado no início e no fim do loop, faz-se necessário para manter a semântica do loop original.

Como as técnicas de *loop unrolling* e *software pipelining* eliminam diferentes tipos de *overhead*. Sendo que a primeira elimina o overhead das instruções de controle do loop (instruções de desvio e incremento/decremento do contador do loop) e a segunda reduz o número de vezes em que o loop não está com o overlap máximo [3]. Elas podem ser combinadas para produzir um melhor escalonamento do código.

3.3 Trace Scheduling

Apesar do *trace scheduling* [3] ter sido inicialmente proposto como um algoritmo para compactação de microcódigo [2], esta técnica se tornou um dos principais métodos para escalonamento de código para máquinas VLIW. O que motiva a sua utilização é a sua capacidade de otimização além das fronteiras de um bloco básico.

A idéia desta técnica é transformar o caminho mais comum de execução em um conjunto de instruções sequenciais, livres de instruções de desvios. O algoritmo de *trace scheduling* funciona da seguinte maneira. Inicialmente, seleciona-se um *trace* [3] com alta probabilidade de ser executado. Em seguida, compacta-se o *trace*, ou seja, tenta-se encapsular as suas operações no menor número de palavras longas (instruções VLIW) possíveis. Durante esta fase, pode ser necessário a geração de código de compensação, uma vez que outros *traces* de execução podem entrar ou sair do meio do *trace* que está sendo compactado.

3.4 Superblock Scheduling

O *superblock scheduling* [3, 1] é uma técnica similar ao *trace scheduling*, exceto pelo fato de não permitir pontos de entrada no meio do *trace*. A compactação é feita em cima de um *trace* com apenas uma entrada e uma ou mais saídas. Esta estrutura simplifica o processo de compactação das operações em instruções longas, pois a ausência de pontos de entrada elimina a movimentação de código para *traces* entrantes.

Para se formar os superblocos, inicialmente encontra-se os *traces* através de informações de *profiling*. Em seguida, elimina-se os pontos de entrada que estão no meio do *trace* através da utilização de um método denominado de *tail duplicatication*, que move os pontos de entrada para blocos básicos duplicados.

Uma desvantagem do *superblock scheduling* e do *trace scheduling* é que ambos os esquemas de escalonamento consideram um único caminho de execução. Logo a escolha de um caminho errado de execução baseado em informações de *profiling* levará a um desperdício de ciclos de execução do processador.

3.5 Hyperblock Scheduling

No *Hyperblock Scheduling* [1], múltiplos caminhos de execução são escalonados em uma única unidade. Esta técnica

[3]Um *trace* é uma seqüência linear de blocos básicos

utiliza predicação para formar escopos de escalonamento. A predicação envolve a execução de instruções baseadas num predicado (valor booleano de um operador). Um *hyperblock* pode conter vários caminhos de execução que são combinados por *if-conversion* e *tail duplication*. Blocos básico que contenham chamadas de procedimento e acessos a memória não resolvidos, não são incluídos no *hyperblock*.

O *hyperblock* corresponde a uma estrutura com uma única entrada e múltiplas saídas laterais. O método *if-conversion* transforma dependência de controle em dependência de dado e consequentemente novas otimizações podem ser aplicadas.

4. VANTAGENS E DESVANTAGENS

Assim como as arquiteturas escalares e superescalares, as arquiteturas VLIW também tem os seus prós e contras. Algumas das vantagens dos VLIW são:

- Hardware de controle simples, uma vez que o escalonamento das operações e a verificação das dependências é feita pelo compilador.
- O fato do hardware ser simples implica num menor consumo de energia.
- Capacidade de despacho de múltiplas operações através do uso de instruções longas.
- O fato da arquitetura ser exposta ao compilador, permite a aplicação de uma vasta gama de otimizações.
- O acesso ao código fonte permite ao compilador analisar janelas maiores de código, aumentando o nível de otimização sem aumentar a complexidade do hardware.

Alguns dos pontos negativos que podem desistimular a produção comercial destes processadores são:

- A construção de um compilador VLIW não é uma tarefa nada trivial, uma vez que o desempenho desta arquitetura depende da capacidade do compilador em explorar o paralelismo entre as instruções.
- Não tem vantagem de rodar o mesmo código que outras plataformas, como, por exemplo, os superescalares que rodam código de escalares.
- Existe uma dependência evidente da arquitetura, sendo díficil existir compatibilidade entre diferentes máquinas VLIW.
- Se a taxa de operações por intrução for baixa, ocorrerá um mal uso da memória (desperdício de banda de transmissão e espaço na cache), uma vez que as instruções longas serão completadas com Nops.

5. PERSPECTIVAS FUTURAS

De maneira geral, pode-se dizer que o succeso comercial das arquiteturas VLIW ainda é moderado, como é exemplificado pelos processadores Philips Trimedia, C6X da Texas Instruments, Intel Intanium e Transmeta Crusoe [6]. No entanto o papel desta arquitetura vem mudando ao longo dos anos, deixando de ser usada para o desenvolvimento de processadores destinados a supercomputadores científicos e passando a ser utilizada em processadores destinados ao processamento digital de sinais, imagem, multimídia e dispositivos móveis, onde a relação desempenho/energia é importante.

A crescente demanda por aplicações multimídias provavelmente irá continuar incentivando o desenvolvimento da tecnologia VLIW, pois para estas aplicações o comportamento das instruções de desvios são bem regulares, ou seja, os *traces* de execução são bem definidos, além disso, a quantidade de instruções paralelizáveis é muito grande, o que faz destas aplicações ideais ao estilo VLIW de execução. Todavia, a curto prazo, os processadores superescalares provavelmente irão continuar dominando o mercado de processadores de propósito geral.

Atualmente, a pesquisa relacionada a tecnologia VLIW concentra-se nas seguintes áreas:

- Compiladores - desenvolvimento de novas técnicas de otimização e escalonamento que aumentem a capacidade de exploração de ILP.
- Arquitetura - desenvolvimento de uma arquitetura mais portável.

6. CONCLUSÕES

Este trabalho apresentou os principais conceitos relacionados as arquiteturas VLIW, focando principalmente na questão de exploração de paralelismo em nível de instrução. Apesar do desenvolvimento desta tecnologia ter se sucedido de maneira lenta ao longo da última década, nos últimos anos o crescente mercado de DSPs e sistemas dedicados, vem alavancando novamente um novo interesse por essa arquitetura.

7. REFERÊNCIAS

[1] M. Agarwal. Speculative trace scheduling in vliw processors, 2002.

[2] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Computers*, 30(7):478–490, 1981.

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[4] S. A. Ito. Uma alternativa para a exploração de paralelismo a nível de instrução. Relatório técnico, Universidade Federal do Rio Grande do Sul, 2000.

[5] N. Kondo, A. Koseki, H. Komatsu, and Y. Fukazawa. A method for applying loop unrolling and software pipelining to instruction-level parallel architectures. *Systems and Computers in Japan*, 29(9):62–73, 1998.

[6] B. Mathew. *The Computer Engineering Handbook*, chapter Very Large Instruction Word Architectures. CRC Press, Dec. 2001.

[7] P. W. Papers. An introduction to very-long instruction word (vliw) computer architecture. Technical Report 9397-750-01759, Philips Groups, 2002.

[8] S. Sair and D. Kaeli. A study of loop unrolling for vliw based dsp processor, 1998.

[9] S. Weiss and J. E. Smith. A study of scalar compilation techniques for pipelined supercomputers. In *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pages 105–109, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

A Arquitetura Cell

Douglas José S. Rodrigues
RA - 011104
IC - Unicamp
Avenida Albert Einstein, 1251
Caixa Postal 6176
+55 (19) 3788-5842
douglasjose@gmail.com

ABSTRACT

Neste artigo serão apresentadas as principais características e funcionalidades do processador Cell. Serão apresentados detalhes do funcionamento de cada uma das estruturas que compõe a arquitetura Cell, e sua organização será comparada com a de outros processadores, dentre eles o Emotion Engine, core do console PlayStation 2.

Categories and Subject Descriptors

C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: *Array and vector processors*

General Terms

Design.

Keywords

Cell CellBE PlayStation PPE SPE.

1. INTRODUÇÃO

Cell Broadband Engine Architecture, ou também CellBE, é o nome da nova arquitetura de microprocessadores em desenvolvimento pela parceria entre Sony, Toshiba e IBM.

A maior aplicação comercial da arquitetura Cell é o novo console da Sony, o PlayStation 3, que deverá ser lançado em novembro de 2006. A Toshiba anunciou também que pretende usar a arquitetura Cell aos seus novos aparelhos televisores de alta definição.

Nas próximas seções deste artigo serão apresentados os detalhes da arquitetura Cell, bem como a sua comparação com outras arquiteturas existentes.

2. ORGANIZAÇÃO

O chip do processador Cell pode ser organizado em várias configurações diferentes. A mais comum é a utilização de uma unidade PPE, o *Power Processor Element*, e múltiplas SPEs, os *Synergistic Processing Elements*. A PPE e as SPEs são ligadas por um barramento interno de alta velocidade, o EIB (*Element Interconnect Bus)*.

2.1 *Power Processor Element*

O PPE é uma unidade baseada na arquitetura POWER, que é a base das linhas POWER e PowerPc [1]. O papel do PPE é servir como um controlador das várias SPEs, que serão responsáveis pela execução da maior parte da carga de trabalho.

Como o PPE é muito similar a outros processadores PowerPc de 64 bits, será possível utiliza-lo com sistemas operacionais convencionais, já disponíveis para esta arquitetura.

O PPE é um processador superescalar de 64 bits que não permite execução de instruções out-of-order, e é capaz de despachar até duas instruções por ciclo. É composto também por uma cache L2 de 512MB, bem como um cache L1 *two-way* de 32KB de instruções e um cache *four-way set-associative* de dados.

2.2 Synergistic Processing Elements

Os SPEs são as unidades responsáveis pela execução das tarefas que exigem maior processamento [1]. Em um ambiente padrão, o sistema operacional seria executado no PPE, que designaria a cada uma das SPEs uma tarefa específica e isolada.

Cada SPE é um processador RISC SIMD (*single instruction, multiple data*) [2], que contém 128 registradores de 128 bits, quatro unidades de ponto flutuante capazes de realizar um total de 32 bilhões de operações por segundo, e mais quatro unidades de operações sobre inteiros, capazes também de 32 bilhões de operações por segundo, trabalhando com um clock de 4GHz de freqüência. É importante observar que este cálculo refere-se à utilização de instruções do tipo “multiply-add”, que recebe três operandos de entrada, e é contabilizada como duas operações (em apenas uma instrução). As instruções de um SPE podem receber até três operandos fonte, e um operando destino, conforme exemplifica a figura 2.

Um dos resultados mais vistosos da implementação desta arquitetura é o rompimento da barreira de 4GHz para o clock de um processador. É publico que a Intel tem vários problemas tentando alcançar 4.0GHz de clock para o Pentium 4. Entretanto, o processador Cell é capaz de operar em até 4.6 GHz de freqüência, embora este valor deva ser ligeiramente reduzido em produtos onde a energia elétrica consumida seja um fator a ser levado em consideração.

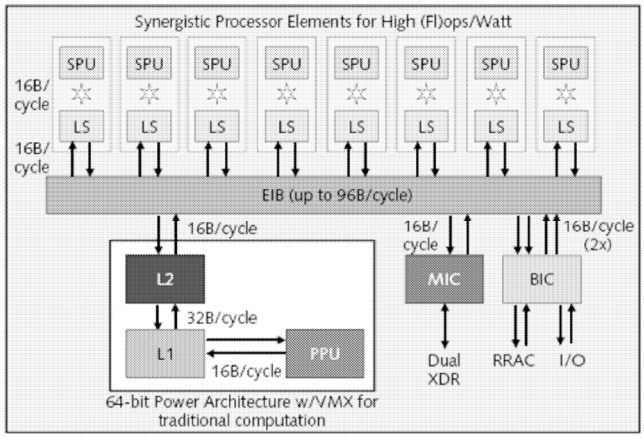


Figura 1. Organização da Arquitetura Cell.

Os SPEs diferem-se dos processadores convencionais também pela falta de uma memória cache. Ao invés de uma cache, cada SPE dispõe de uma região de memória chamada de *Local Store* (LS). Cada *Local Store* é uma memória SRAM de 256KB, e cada SPE possui a sua LS privada. O endereçamento na LS é mapeado diretamente na memória principal, e nenhum protocolo de coerência de cache é utilizado.

É responsabilidade do software gerenciar o movimento de dados entre a memória principal e a LS. Esta abordagem foi escolhida devido à natureza dos programas que se espera executar no processador, e à economia de hardware necessário para a implementação de um protocolo de coerência.

Figura 2. Formato de uma instrução do SPE

2.3 Element Interconnect Bus

A transferência de dados entre o PPE, os SPEs, e as interfaces de entrada e saída do processador é responsabilidade de uma estrutura chamada *Element Interconnect Bus* (EIB).

O EIB é implementado como quatro "anéis" que circulam em sentidos contrários em pares. Ele trabalha em metade do ciclo de clock do processador, e é capaz de transmitir 16 bytes a cada dois clocks do sistema. Devido a esta característica, o EIB é comumente definido como um barramento de oito bytes por ciclo.

2.4 Memory Interface Controller

A memória e os canais de entrada e saída são baseados em tecnologia Rambus. Estão presentes dois controladores independentes XDR, que provêem mais flexibilidade do que apenas um. A interface de memória pode trabalhar à incrível taxa de 25.6GB/s.

3. TECNOLOGIA

O processador Cell será fabricado usando a tecnologia SOI (*Silicon-On-Insulator)* de 90nm com oito camadas de metal. A área do *die* da versão de produção do Cell será de 235mm^2. Testes mostram que o processador consumindo 1,2 volts e trabalhando em uma freqüência de 4 GHz dissipa 6 watts de potência [3].

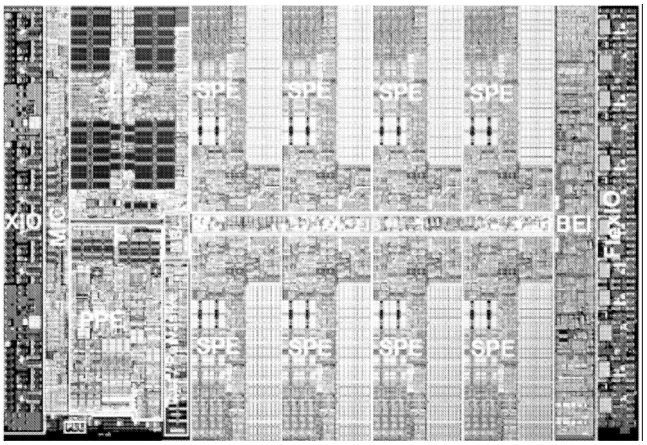


Figura 3. Foto do *die* do Cell

4. CONJUNTO DE INSTRUÇÕES

O Cell possui um pipeline de 21 níveis. É capaz de explorar especulação com a *branch prediction*, onde em um SPE, a penalidade de um erro na especulação é de 18 ciclos.

Cada instrução tem tamanho fixo, de 32 bits. E a arquitetura Cell é *big-endian*. Vale observar aqui que

a arquitetura POWER suporta tanto o *little* quanto *big-endian*, mas no caso da arquitetura Cell, apenas um modo de execução é suportado.

5. SOFTWARE

As ferramentas para desenvolvimento para o Cell estão definidas em cima do Linux para PowerPc. A programação para as SPEs é baseada em C, com suporte parcial a C++. O suporte a outras linguagens, por exemplo, Fortran, está sendo ainda desenvolvido. Estão disponíveis também ferramentas de depuração, como por exemplo, o GDB.

O objetivo principal da pesquisa de ferramentas e frameworks para o desenvolvimento de aplicações para o Cell é prover uma camada de abstração acima do hardware, que permita que as aplicações sejam escaláveis na presença de outros processadores.

Este processo demanda novas ferramentas, já que o paradigma de programação para o Cell envolve programas para cada um dos SPEs, que contém todas as suas instruções e seus dados. Este paradigma é diferente do utilizado por linguagens que definem estruturas em classes, como por exemplo, Java.

6. EMOTION ENGINE

A arquitetura *Emotion Engine* é a utilizada no processador central do console PlayStation 2. Não é possível afirmar que a arquitetura Cell seja uma evolução da arquitetura *Emotion Engine*, pois elas compartilham apenas a sua aplicação na família de videogames PlayStation.

Enquanto o Cell é baseado na arquitetura POWER, o *Emotion Engine* é baseado na arquitetura MIPS64. Um aspecto bastante interessante de se observar na comparação das duas arquiteturas é a área do *die* de cada processador. Mesmo o desempenho do Cell sendo ordens de magnitude maior que o do *Emotion Engine*, a área ocupada por ele é menor. Mais dados comparativos estão disponíveis na Tabela 1.

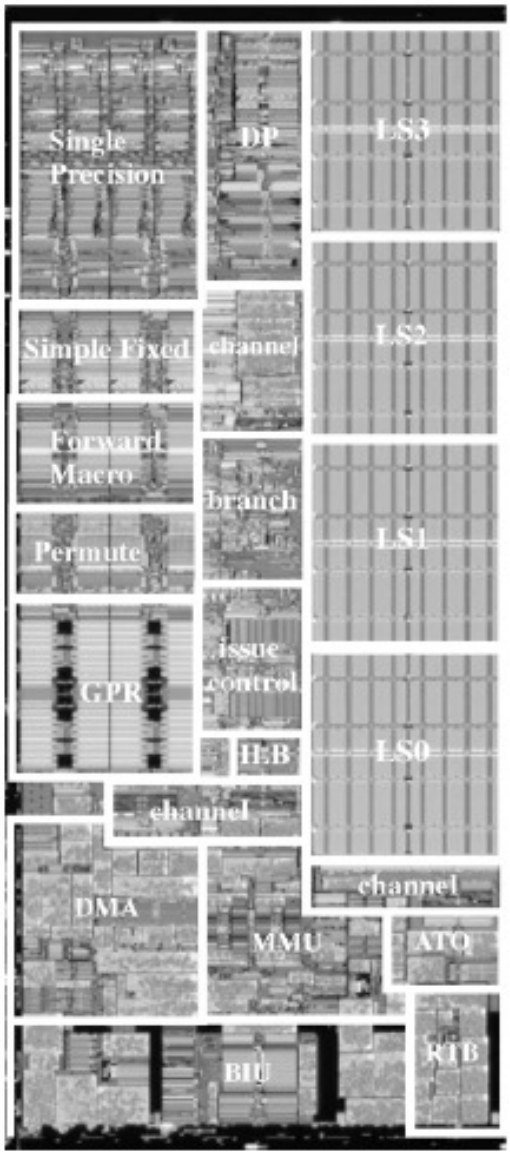


Figura 4. Foto do *die* de um SPE.

Tabela 1. Comparação entre os processadores *Emotion Engine* e Cell.

	Sony Emotion Engine	Cell Processor
CPU Core ISA	MIP64	64-bit Power Architecture
Core Issue Rate	Dual	Dual
Core Frequency	300MHz	~4GHz (est.)
Core Pipeline	6 stages	21 stages
Core L1 Cache	16KB I-Cache + 8KB D-Cache	32KB I-Cache + 32KB D-Cache
Core Additional Memory	16KB scratch	512KB L2
Vector Units	2	8
Vector Registers (#, width)	32, 128-bit + 16, 16-bit	128, 128-bit
Vector Local Memory	4K/16KB I-Cache + 4K/16KB D-Cache	256KB unified
Memory Bandwidth	3.2GB/s peak	25.6GB/s peak (est.)
Total Chip Peak FLOPS	6.2GFLOPS	256GFLOPS
Transistor Count	10.5 million	235 million
Power	15W @ 1.8V	~80W (est.)
Die Size	240mm^2	235mm^2
Process	250nm, 4LM	90nm, 8LM + LI

7. CONCLUSÃO

A arquitetura Cell mostra-se extremamente revolucionária na abordagem de alguns conceitos que são trabalhados de maneira diferente em outras arquiteturas de renome e já consolidadas. O consórcio entre Sony, Toshiba e IBM planeja estender a utilização de sua arquitetura para outros tipos de dispositivos, além de sua principal aplicação, o console PlayStation 3, mesmo não sendo seu alvo o mercado de processadores para computadores de propósito geral.

8. REFERENCES

[1] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa. The Design and Implementation of a First-Generation CELL Processor. *ISSCC 2005 MICROPROCESSORS AND SIGNAL PROCESSING,* (Feb. 2005), 591-593.

[2] H. Peter Hofstee. *Power Efficient Processor Architecture and The Cell Processor.* 11th Int'l Symposium on High-Performance Computer Architecture, 2005.

[3] Flachs, B.; Asano, S.; Dhong, S.H.; Hofstee, H.P.; Gervais, G.; Roy Kim; Le, T.; Peichun Liu; Leenstra, J.; Liberty, J.; Michael, B.; Hwa-Joon Oh; Mueller, S.M.; Takahashi, O.; Hatakeyama, A.; Watanabe, Y.; Yano, N.; Brokenshire, D.A.; Peyravian, M.; Vandung To; Iwata, E. The microarchitecture of the synergistic processor for a cell processor. *IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 41, NO. 1, JANUARY 2006*, 63-70.

A Arquitetura x86-64 e o Processador Athlon 64

João Paulo Porto
IC - Unicamp
Rua Garaúna, 173
Alphaville Campinas
+55 19 3262 1323
jpporto@gmail.com

ABSTRACT
Neste paper é descrita a arquitetura x86-64 (previamente conhecida como AMD64) bem como o processador Athlon 64, que é uma implementação desta arquitetura.

Categoria
C.1.1 [**Single Data Stream Architectures**]: Arquiteturas RISC/CISC e VLIW. Classificação obtida 22/06/2006 em: http://www.acm.org/class/1998/C.1.1.html.

Keywords
CISC, x86-64, AMD64, EM64T, AMD, Intel, Athlon 64, IA-32e, x64.

1. INTRODUÇÃO
A arquitetura x86-64 é uma evolução natural do que é conhecido como arquitetura x86

Esta arquitetura nasceu em 1978 (portanto, há quase 30 anos) com o processador 8086. Todos os processadores desta família são *backwards compatibles* em nível binário. I.e., código gerado para a geração *n* do processador será executado corretamente em todas as gerações subseqüentes.

Este paper está organizado da seguinte forma. Na seção 2 é mostrada um pouco da história da família x86, partindo 8086. Na seção 3 as modificações feitas pela AMD para criação da arquitetura x86-64 são descritas. Na seção 4, é detalhado o processador Athlon 64, que é uma implementação desta arquitetura. Na seção 5 é feita uma breve conclusão sobre a nova arquitetura.

2. UM POUCO DE HISTÓRIA
O primeiro processador da família x86 é o 8086, lançado em 1978. Este processador é uma evolução em relação ao seu predecessor, o 8085, sendo que eles são compatíveis em nível de código *assembly* (note que isso não os torna compatíveis em nível binário).

O 8086 é um processador de 16 bits, com barramento de memória de 20 bits (ou seja, ele endereçava até 1 MB), quatro registradores de 16 bits (os conhecidos AX, BX, CX e DX) além de 4 registradores de indexação de 16 bits (SI, DI, BP e SP).

Este processador CISC (*Complex Instruction Set Computer*) foi evoluindo e, desta evolução, temos como resultado os processadores Pentium, Athlon e similares. Apesar de toda evolução, estes processadores recentes, quando inicializados, são nada além de um 8086, sendo necessários alguns passos para habilitar o processamento de 64 bits (chamdo pela AMD de *long mode*. Veja seção 3.3 Modos de Operação). Isso é necessário para garantir compatibilidade retrógrada em nível binário.

Os novos processadores são implementados de maneira completamente diferente dos processadores originais, sendo que a compatibilidade é mantida com o uso de uma camada responsável por traduzir o código CISC em algo que o seja executável internamente. Em particular, o Athlon 64 tem um *datapath* superescalar (veja a seção 4.1.4).

Apesar de muito criticada, a decisão da Intel de manter compatibilidade binária entre gerações dos processadores é a grande responsável pelo sucesso da arquitetura x86. É sabido que esta arquitetura tem graves problemas estruturais, como a codificação das instruções ter formato e tamanho variável; a quantidade reduzida de registradores, cada qual com possíveis usos especiais (e.g., multiplicação usa implicitamente ax, a quantidade a ser deslocada em um *shift* deve ser colocada no registrador cl); inúmeros modos de endereçamento.

3. A ARQUITETURA x86-64
3.1 Visão Geral
As aplicações atuais estão trabalhando com uma quantidade de dados crescente. Este crescimento tem tornado insuficiente os 32 bits de espaço de endereçamento fornecido pelo x86. Na verdade, antes do AMD64, já havia uma extensão chamada PAE (*Physical Address Extension*) que aumentava o espaço de endereçamento físico dos processadores x86 para 48 bits.

O Athlon 64 operando em *long mode* é um processador de 64 bits que é capaz de executar código legado. A seguir há uma descrição dos aspectos mais marcantes desta arquitetura

3.2 Registradores
Conforme mencionado, a arquitetura do x86 possuia apenas 8 registradores, o que acabava gerando muitos *spills* (armazenamento de variáveis) nos programas para a arquitetura.

Para resolver este problema, a AMD inseriu outros 8 registradores. Estes novos registradores estão disponíveis somente

quando o *long mode* está ativo. Apesar de ser de grande valia estes novos registradores, há um problema para acessá-los.

Na arquitetura x86, o acesso ao banco de registradores é feito de algumas maneiras: (1) com o uso de um byte especial chamado *ModRM*, cujo formato é muito bem definido (veja a Figura 1), e alterá-lo quebraria a compatibilidade com o código legado; (2)byte *ModRM* em conjunto com o byte *SIB* (*Scale, index,* base, detalhado também na Figura 1); ou ainda pelo campo *reg* de algumas instruções.

Como a arquitetura possuía apenas 8 registradores, apenas 3 bits são reservados para indexação do banco de registradores.

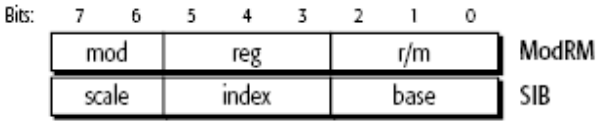


Figura 1: Bytes ModRM e SIB ([6])

Este problema foi solucionado de uma forma elegante.

A AMD introduziu na arquitetura x86-64 um novo tipo de byte de prefixo, chamado de REX. Este byte de prefixo indica para o processador que a instrução seguinte deve ser interpretada como uma instrução de 64 bits. Este byte de prefixo tem os valores na faixa 0x40 a 0x4F. A Tabela 1 a interpretação deste byte.

Tabela 1: O Prefixo REX

Mnemônico	Bit	Descrição
-	7-4	0100
REX.W	3	0 = operandos de 32 bits 1 = operados de 64 bits
REX.R	2	Extende o campo *reg* do byte ModRM
REX.S	1	Extende o campo *index* do byte SIB
REX.B	0	Extende o campo *r/m* do byte ModRM, campo *base* do byte SIB ou o campo *reg* do opcode da instrução

Repare, na Figura 2 como os campos REX.R, REX.S e REX.B são utilizados.

Desta forma, a AMD conseguiu ao mesmo tempo manter a compatibilidade sem alterar nenhuma estrutura legada e aumentar o número de registradores. A codificação do prefixo REX colide com os *opcodes* das instruções INC e DEC de 1 byte. Como conseqüência, estas instruções não estão disponíveis quando o processador está operando em *long mode*. Note que a codificação das mesmas que utiliza 2 bytes é válida.

Além da extensão dos GPR (*General Purpose Registers*) para 64 bits e da adição de 8 novos registradores, a AMD adicionou ainda 8 novos registradores MMX (XMM8 até XMM15) de 128 bits, utilizados pelas instruções SSE e SSE2. Vale lembrar os registradores adicionais somente estão disponíveis com o processador operando em *long mode.*

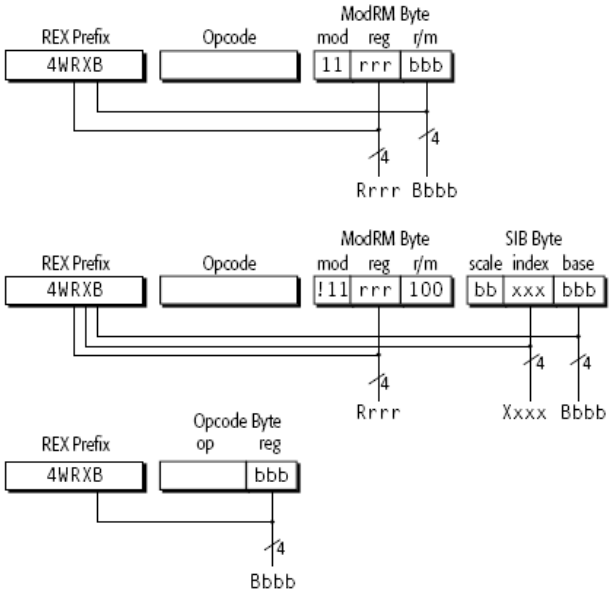


Figura 2: Uso dos Campos R, S e B do Prefixo REX ([6])

3.3 Modos de Operação

Os processadores candidatos a implementar a arquitetura x86-64 devem suportar uma grande quantidade de modos de operação, como pode ser visto na Tabela 2.

Repare, na mesma tabela, que há dois grandes modos de operação: *Long mode* (64 bits) e *Legacy mode* (16 e 32 bits). Os modos legados são os mesmos encontrados nos processadores que implementam a arquitetura IA-32 (outro nome para a arquitetura x86). Neste modo de operação é possível executar todos os programas existentes atualmente sem a necessidade de haver recompilação.

Quando o *long mode* está ativo, ainda é possível executar código legado utilizando o modo de compatibilidade (*compatibility mode*). Neste modo de operação, as aplicações percebem o sistema como um processador operando em *legacy mode* (i.e., apenas 8 registradores, com operandos e registradores de 32 bits), enquanto o sistema operacional utiliza todas as características do modo de 64 bits para tradução de endereçoes, tratamento de interrupções e exceções.

Para que seja utilizado o modo de 64 bits é necessário que tanto as aplicações como o sitema operacional sejam compilados especificamente para a arquitetura x86-64.

3.3.1 Habilitando o Modo de 64 bits

Uma série de passos devem ser tomados antes que o processador possa ser colocado em *long mode*.

Assim que é inicializado, o processador é colocado em *real mode*, que é o modo de operação do 8086. Neste modo de execução é necessário:

- Inicializar o vetor de interrupções
- Carregar as rotinas de tratamento de interrupções e exceções antes de habilitar as interrupções externas

- Carregar o par SS:SP com um endereço válido para o caso de alguma interrupção ocorrer
- Carregar um ou mais registradores de segmento com valores válidos para armazenamento das estruturas de modo protegido que são criadas ainda enquanto o processador está em modo real.

Depois que todos esses passos foram cumpridos, é possível colocar o computador em modo protegido. Para tanto, os seguintes passos devem ser tomados:

- Inidicalizar o vetor de interrupção do modo protegido.
- As rotinas de tratamento de interrupção de modo protegido devem ser carregadas.
- Deve ser inicializado o GDT (*Global Descriptor Table*) composto de um descritor do segmento de código que deve ser executado em modo protegido e um descritor de um segmento que possa ser lido e escrito para ser utilizado como segmento de pilha.

Com esses passos tomados, pode-se habilitar o modo protegido. Se for utilizado a paginação legada, ao menos um diretório de páginas e uma tabela de páginas devem estar disponíveis para que a tradução dos endereços possa ocorrer.

Com o ambiente de modo protegido inicializado, é possível inicializar o modo de 64 bits. Para tanto:

- Inicialize o vetor de interrupções, bem como carregue as rotinas de tratamento de exceções e interrupções que serão utilizadas em modo 64 bits.
- Inicialize também a GDT que será utilizada em modo 64 bits
- Inicialize ao menos um *task state segment* (TSS) para os dados que serão acessados em modo privilegiado.
- Crie a tabela de tradução de endereços de 4 níveis. Isso é necessário para o *long mode*. Também deve ser habilitado as extensões de endereçamento físico (PAE, *physical address extensions*).

Um ponto importante: se paginação estiver sendo utilizada ela *deve* ser desabilitada para que seja possível habilitar e ativar o modo de execução de 64 bits.

Para habilitar o *long mode*, é necessário definir um registrador de controle da arquitetura. Entretanto, após habilitado é necessário *ativar* o modo de 64 bits. Para tanto, basta ativar a paginação.

Após todos estes passos o sistema estará executando no novo modo de operação.

3.4 Gerenciamento de Memória

Este é outro ponto em que a arquitetura x86 pecava. A Intel decidiu adotar para o 8086 o modelo de gerenciamento de memória com uso de segmentação. Para tanto, haviam vários registradores de segmento disponíveis. Entretanto, os sistemas operacionais modernos utiliza o conceito de memória virtual.

Tendo isso em vista, os projetistas do x86-64 decidiram acabar com a segmentação em modo 64 bits. Assim, neste modo, a memória é toda vista como um vetor de bytes, e os registradores de segmento (e os prefixos de seleção de segmentos) são ignorados. É claro que os registradores de segmento estão presentes na arquitetura e a segmentação é utilizada

Tabela 2: Modos de Operação

<table>
<tr><th colspan="2" rowspan="2">Operating Mode</th><th rowspan="2">Operating System Required</th><th rowspan="2">Application Recompile Required</th><th colspan="2">Defaults</th><th rowspan="2">Register Extensions</th><th rowspan="2">Typical GPR Width (bits)</th></tr>
<tr><th>Address Size (bits)</th><th>Operand Size (bits)</th></tr>
<tr><td rowspan="3">Long Mode</td><td>64-Bit Mode</td><td rowspan="3">New 64-bit OS</td><td>yes</td><td>64</td><td rowspan="2">32</td><td>yes</td><td>64</td></tr>
<tr><td rowspan="2">Compatibility Mode</td><td rowspan="2">no</td><td>32</td><td rowspan="2">no</td><td>32</td></tr>
<tr><td>16</td><td>16</td><td>16</td></tr>
<tr><td rowspan="4">Legacy Mode</td><td>Protected Mode</td><td rowspan="2">Legacy 32-bit OS</td><td rowspan="4">no</td><td>32</td><td>32</td><td rowspan="4">no</td><td rowspan="2">32</td></tr>
<tr><td>16</td><td>16</td></tr>
<tr><td>Virtual-8086 Mode</td><td rowspan="2">16</td><td rowspan="2">16</td><td rowspan="2">16</td></tr>
<tr><td>Real Mode</td><td>Legacy 16-bit OS</td></tr>
</table>

para execução de código legado (tudo em nome da compatibilidade). O esquema de decodificação do endereço virtual na arquitetura x64-64 pode ser visto na Figura 3.

Outra alteração significativa foi a inclusão do bit Nx (*No Execute*) nos descritores das páginas de memória. Quando este bit está sinalizado para uma página, qualquer tentativa de execução a partir dela gera uma falha de página (*page fault*). Esta restrição é aplicada independente do privilégio do código atualmente em execução. O sistema operacional é responsável por gerenciar a definição deste bit.

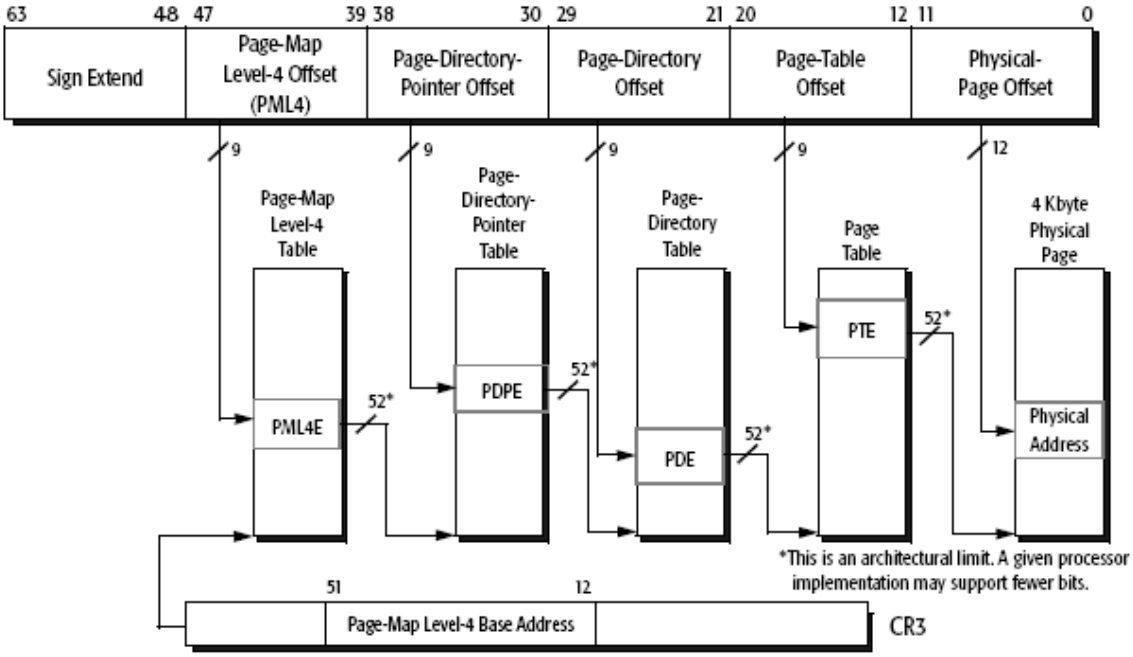


Figura 3: Tradução de Endereço Virtuais para Páginas de 4 KB([5])

Uma característica que os projetistas deixaram em aberto é a existência ou não de *cache* na implementação da arquitetura. Entretanto, eles definiram alguns tipos de memória, conforme pode ser visto na Tabela 3, que mostra também os tipos de acesso permitidos para cada tipo de memória

A seguir, uma breve descrição de cada tipo.

- UC (*Unchacheable*): memória não cacheável. Útil para endereçoes de E/S mapeados em memória.
- CD (*Cache disabled*): quando o processador é instruído a desabilitar a cache, esta memória comporta-se como a memória do tipo UC
- WC (*Write combining*): as escritas a este tipo de memória *podem* ser combinadas internamente dentro do processador para evitar acessos desnecessários à memória. Por exemplo, é possível combinar 4 escritas de 16 bits a endereços consecutivos em uma única *quadword* e escrevê-la de uma única vez. Particularmente útil para *buffers* de dispositivos gráficos.
- WP (*Write protect*): neste tipo de memória, as leituras são cacheáveis, porém as escritas escrevem diretamente na memória e invalidam qualquer linha de cache que reporte *hit* na escrita.
- WT (*Write through*): semelhante ao WP, memórias do tipo *write through* que venham a reportar um *hit* durante uma escrita tem a linha de cache correspondente atualizada.
- WB (*Write back*): este tipo de memória implementa, como o nome sugere, a política de escrita de *write back*. A política de coerência para este tipo de memória é a MOESI (*Modified, Owned, Exclusive, Shared, Invalid*).

4. O Processador Athlon 64

Além de especificar uma nova arquitetura, os engenheiros da AMD implementaram um processador com as especificações dela.

O nome escolhido (Athlon 64) remete aos processadores Athlon e Athlon XP, desenvolvidos pela AMD e que têm uma excelente aceitação no mercado. A inclusão do 64 serviu para diferenciar as novas implementações das antigas.

Mantendo a tradição da família x86, este processador mantém compatibilidade binária com o código gerado para o "pai de todos" (8086) e para todos os seus descendentes.

Tabela 3: Tipos de Acesso por Tipo de Memória

Tipo de Acesso à Memória		Tipo de Memória				
		UC/CD	WC	WP	WT	WB
Leitura	Fora de ordem	N	S	S	S	S
	Especulativo	N	S	S	S	S
	Reordena Antes de Escrita	N	S	S	S	S
Escrita	Fora de Ordem	N	S	N	N	N
	Especulativo	N	N	N	N	N
	Buffering	N	S	S	S	S
	Combinar[1]	N	S	N	S	S

4.1 Detalhes da Implementação

Apesar de executar um conjunto de instruções CISC, o Athlon 64 é, internamente, um computador RISC superescalar, com execução fora de ordem e algum tipo de execução especulativa.

Conforme pode ser observado na Figura 6, a AMD criou um processador com *hardware* suficiente para obter um desenpenho muito bom.

A seguir, serão detalhadas algumas das características mais marcantes do processador.

4.1.1 HyperTransport

Uma característica marcante do processador Athlon é a habilidade de utilizar esta tecnologia.

Desenvolvido pela própria AMD, o *HyperTransport* é um barramento de alta velocidade que é utilizado pelo processador para comunicação entre o processador e o *chipset* da placa mãe. Também é utilizado para comunicação entre dispositivos de E/S e, no caso de sistemas multi processados, na comunicação entre outros processadores.

Este barramento tem largura de 16 bits e funciona com um *clock* de 800 MHz utilizando tecnologia DDR, que permite transferências de até 3,2 GB/s. Este barramento foi desenvolvido tendo em mente os dispositivos PCI-X

4.1.2 Controlador de Memória Integrado

Os processadores da família x86 não eram capazes de, diretamente, fazer requisições à memória principal do sistema, fazendo, para isso, uso de um controlador de memória, localizado, em geral, na ponte norte (*north bridge*) do *chipset*. A Figura 4 ilustra essa separação.

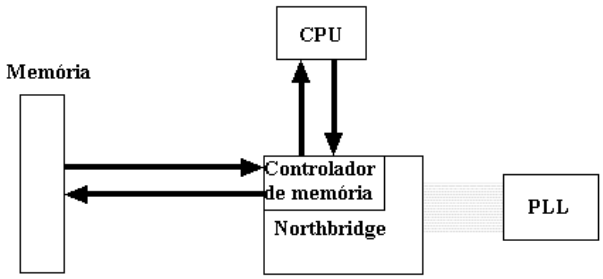


Figura 4: Estrutura Tradicional de um Processador x86

Sistemas implementados desta forma têm a vantagem de poderem ser conectados a diferentes tipos de memória bastando, para tanto, que seja feita uma nova versão do *north bridge*.

Repare que, nesta configuração, para fazer um acesso à memória é necessário que a CPU se comunique com o controlador de memória para que este faça a requisição. Isso acaba gerando atrasos desnecessários e indesejáveis.

Para solucionar este problema, a AMD decidiu integrar o controlador de memória ao *die* do processador, como pode ser visto na Figura 5.

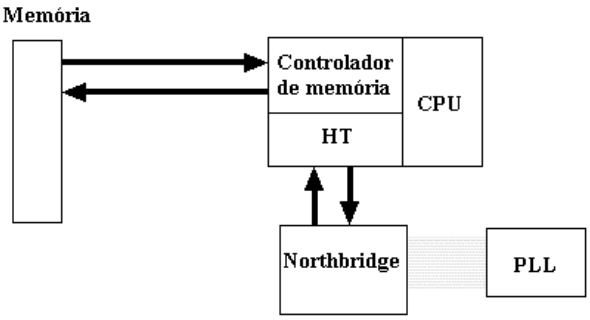


Figura 5: Estrutura do Athlon 64

Desta forma, a própria CPU interage com a memória, evitando perda de tempo. Repare também o controlador HT integrado. Apesar da perda da capacidade de interfacear com diferentes tipos de memória, esta otimização eleva o desempenho do processador. Além do mais, a AMD trabalha com o mesmo tipo de memória (*double data* rate, as DDR) desde as primeiras versões do Athlon. Parece, portanto, improvável uma mudança na tecnologia de memória adotada.

4.1.3 Tecnologia Cool'n'Quiet

A tecnologia *Cool'n'Quiet* é a versão para desktop da tecnologia *PowerNow!* utilizada em processadores para notebooks. Ela é responsável por alterar a freqüência de operação do processador, bem como a voltagem do *core* do sistema dependendo do uso do processador. Essas alterações ocorrem muito rapidamente, sendo que nenhuma perda de performance sensível ocorre.

Para utilização do *Cool'n'Quiet* é necessário habilitá-lo na placa-mãe do sistema e instalar um *driver* no sistema operacional.

[1] Combinar (de *write combining*) é essencialmente diferente de *buffering*. O primeiro é utilizado para escrever montar pacotes que serão tratados como uma única escrita; o segundo armazena as solicitações de escrita para poder liberar o processador.

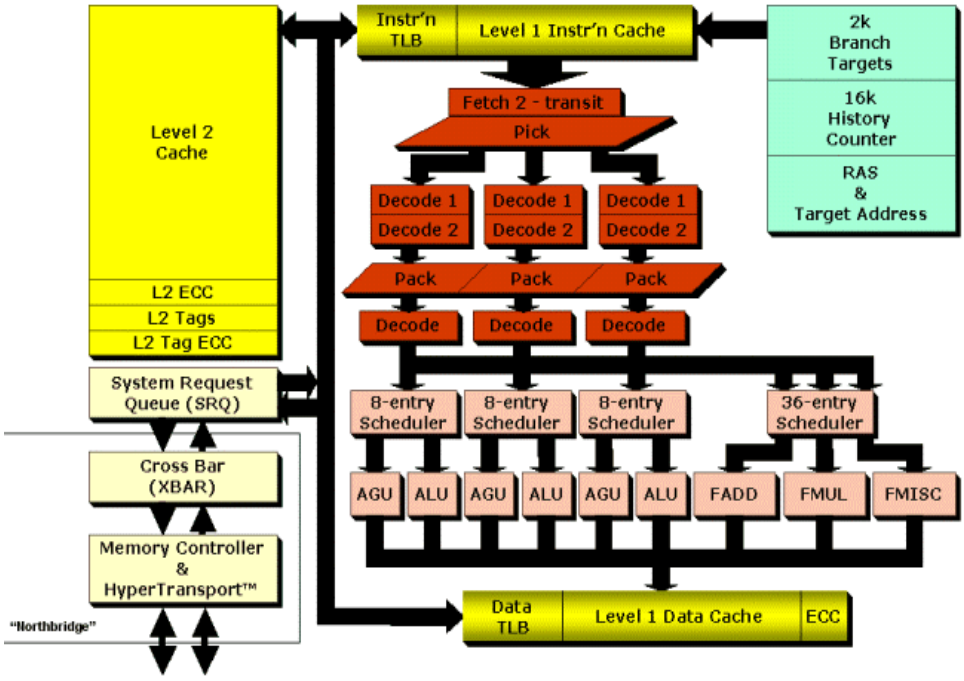


Figura 6: Datapath do Athlon 64

4.1.4 Pipeline

Tabela 4: Pipeline do Athlon 64

ULA	FPU
Fetch 1	
Fetch 2	
Pick	
Decode 1	
Decode 2	
Pack	
Pack / Decode	
Dispatch	Dispatch
Schedule	Stack Rename
Exec	Register Rename
Data Cache 1	Write Schedule
Data Cache 2	Schedule
	Register Read
	FX0
	FX1
	FX2
	FX3

A Figura 6 ilustra o *datapath* do Athlon 64. Mas como ele é utilizado? Como a AMD implementou a execução das instruções?

A Tabela 4 mostra os estágios do *pipeline* do Athlon 64. Repare que há dois caminhos possíveis para execução de uma instrução: ou ela será executada pelo *pipeline* tradicional (operações sobre números inteiros) ou pelo *pipeline* para dados de ponto flutuante. Isso não penaliza tanto os estágios de aritmética inteira, já que as operações de ponto flutuante (notoriamente mais lentas) são executadas em 4 ciclos (uma operação sobre dados inteiros leva 1 ciclo para ser executada).

4.1.4.1 Estágios de Fetch/Decode

Estes 7 estágios são comuns tanto para instruções de aritmética inteira como para operações de ponto flutuante.

O estágio de *fetch* é capaz de fornecer até 16 bytes de instrução por ciclo de *clock* para as três unidades de decodificação.

Os decodificadores são responsáveis por traduzir as intruções CISC em uma ou mais operações RISC. Estas operações RISC são chamadas de µOps (µ *operations*).

As instruções que são mapeadas em 1 ou 2 µOps são enviadas diretamente para execução (este caminho de execução é conhecido como FastPath). Instruções mais complexas (que precisam de mais que 3 µOps) são executadas por microcódigo.

4.1.4.2 Estágio de Pack

Alguns padrões de µOps ocorrem tão freqüentemente que os engenheiros da AMD criaram um estágio especial responsável por detectar a ocorrência deles. Assim que um padrão é detectado, ele é substituído por uma µOp especial, que

representa todas as μOps do padrão. Isso diminui o overhead de execução destes padrões.

4.1.4.3 Estágios de execução

Nestes estágios é onde a operação que a μOp representa é executado.

Para instruções que operam sobre dados inteiros, este ramo do *pipeline* contém somente 5 estágios e, destes 5 estágios, apenas 1 é utilizado para fazer o processamento da instrução.

As instruções da FPU utilizam um ramo do *pipeline* composto por 10 estágios, sendo que, destes 10 estágios, 4 são utilizados para executar as instruções.

4.1.4.4 Estágios de Load/Store

São os últimos estágios do *pipeline* do Athlon 64. São responsáveis por escrever/ler dados da Cache de dados L1. Esta cache é *dual port*, ou seja: ela é capaz de atender a duas operações de *load* ou *store* a cada ciclo de clock.

4.1.5 Hierarquia de Memória

Para aumentar ainda mais o desempenho do Athlon 64, os engenheiros responsáveis pelo projeto implementaram um sistema de memória hieráquico, composto de dois níveis de cache (L1 e L2) antes da memória principal.

Tabela 5: Cache no Athlon 64

	L1	L2
Tamanho	Código: 64KB Dados: 64KB	512KB(NewCastle) 1024KB(Hammer)
Associatividade	Código: 2 way Dados: 2 way	16 way
Tamanho da Linha	Código: 64 bytes Dados: 64 bytes	64 bytes
Política de Escrita	*Write Back*	N/D
Latência	3 ciclos	N/D
Largura do Barramento	N/D	128 bits
Relação com L1		Exclusiva

Para minimizar a diferença de performance entre as caches L1 e L2, a AMD faz uso de um *victim buffer* sempre que um dado precisa ser levado de L1 para L2. Este buffer contém, tipicamente, 8 ou 16 entradas.

4.1.5.1 O Que é a Relação Exclusiva

A relação de exclusividade entre as caches L1 e L2 dos processadores Athlon 64 indicam que os dados nunca estarão, ao mesmo tempo, em ambas as caches. Isso faz com que o tamanho real da cache seja a soma dos tamanhos de L1 e L2.

5. CONCLUSÃO

A arquitetura x86 alcançou tanto sucesso devido à sua capacidade de executar código legado. Apesar dos novos processadores ficarem "amarrados" com os problemas das antigas versões, a retrocompabilidade garante uma transição tranqüila entre diferentes versões da arquitetura.

A arquitetura x86-64 é uma evolução natural da arquitetura x86. Como seu predecessores, ela contém problemas de *design* por estar dos seus antecessores. Apesar disso, as extensões de 64 bits propostas pela AMD são elegantes e garantem uma transição tranqüila para o mundo dos 64 bits.

Para tirar proveito dos novos recursos, é necessária a recompilação das aplicações, que podem ser executadas em modo legado nativamente no Athlon 64. Isso garante que não há perda de performance nas aplicações legadas.

6. ACKNOWLEDGMENTS

Meu sinceros agradecimentos aos engenheiros e projetistas da AMD, que documentaram muito bem a nova arquitetura.

7. REFERÊNCIAS

[1] Ramos, F., Sogumo F., e Silva, F., Arquitetura AMD 64. Em *Trabalho de Graduação da Disciplina mc722*, obtido 22/06/2006 em http://www.ic.unicamp.br/~rodolfo/Cursos/mc722/2s2004/g07-amd64-texto.pdf

[2] Vários. Intel 8086. Em *http://en.wikipedia.org/wiki/8086*, acessado em 22/06/2006

[3] Delattre, F.,. CPUID.COM – AMD K8 Architecture. Em *http://www.cpuid.com/reviews/K8/index.php* acessado em 22/06/2006

[4] Vários. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*.

[5] Vários. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*.

[6] Vários. *AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions*

Introdução à Computação Quântica

Tony Minoru Tamura Lopes
Instituto de Computação - UNICAMP
Avenida Albert Einstein, 1251
Campinas, Brasil
ra017502@students.ic.unicamp.br

RESUMO

A incessante busca por aumento na eficiência de processamento chegará a seu limite quando poucos átomos forem necessários para representar um bit. Quando isso acontecer, a Computação Quântica entrará em voga. Ela, além de permitir a construção de chips minúsculos, fornecerá novas bases para a computação, permitindo a realização de tarefas impossíveis na computação utilizada atualmente. Para compreender isso, serão introduzidos neste trabalho, a unidade básica de informação quântica, o Qubit, e os diversos fenômenos quânticos necessários na obtenção de novos limitantes inferiores para problemas conhecidos. Será detalhada a aplicação mais notável da Computação Quântica, que é a fatoração de números inteiros em tempo polinomial, a qual por sua vez têm implicações diretas na segurança em comunicações. Em contrapartida a isso, um modo totalmente novo e seguro de comunicar-se será descrito utilizando de canais quânticos de informação. Todas essas aplicações já possuem implementações reais, mostrando que a Computação Quântica tem, além de modelos matemáticos concisos, possíveis vantagens práticas sobre a Computação Clássica.

Categorias e Assuntos

A.1 [**INTRODUCTORY AND SURVEY**]: Computação Quântica

Termos Gerais

Teoria

Palavras Chaves

Computador Quântico, Algoritmos Quânticos, Informação Quântica, Criptografia Quântica, Qubit

1. INTRODUÇÃO

"A informação, a transmissão de informação e o processamento de informações são governadas por fenômenos físicos". Como veremos, essa simples afirmação possui implicações nada triviais para a computação.

Em toda sua história, o computador foi implementado de diversas maneiras, utilizando desde engrenagens, relês e válvulas, chegando mais atualmente aos transistores, circuitos integrados e chips. Todas essas maneiras de construir um computador baseiam-se em leis da física e, portanto, são nada mais do que experimentos físicos em princípio.

Nos últimos 40 anos ocorreram reduções dramáticas nas dimensões dos chips em busca de mais eficiência no armazenamento de memória e velocidade de processamento. Ao passo do desenvolvimento tecnológico atual, chegaremos rapidamente à necessidade de poucos átomos para representar-se um bit. Quando isso acontecer, as leis que governarão os fenomenos físicos serão as da Quântica.

A física Quântica, por outro lado, fornecerá muito mais do que a possibilidade de miniaturização dos chips, ela fornecerá uma revolução nas bases da computação, entendendo a classe de problemas tratáveis e permitindo novos limitantes inferiores para algoritmos. Hoje, alguns resultados já demonstram que computadores quânticos podem realizar a fatoração de inteiros em tempo polinomial. Essa habilidade é de extremo impacto, já que muito da segurança em comunicações baseia-se na intratabilidade deste problema.

Em contrapartida, ao fato de tornar os sistemas de comunicação inseguros, a Quântica fornece um meio totalmente seguro de comunicar-se. É possível, através de canais quânticos, criar uma chave privada compartilhada, com garantias de que ela só é conhecida pelas partes comunicantes.

Para compreender como esses avanços serão possíveis, a seção 2 fará uma análise das bases da computação, enquanto a seção 3 mostrará onde a Computação Quântica realizará suas modificações. As seções 4 e 5 discorrerão sobre os principais fenômenos quânticos e quais objetos úteis à computação, eles fornecerão. Além disso, serão detalhados os modelos matemáticos que explicam os fenômenos e alguns experimentos utilizados para identificá-los. Em seguida, a seção 6 apresentará o modelo de um computador quântico completo. Com este modelo, poderão ser introduzidos alguns algoritmos quânticos e a Comunicação Quântica segura, na seção 7. Por fim, as seções 8 e 9 mostrarão respectivamente, o estado atual da implementação de computadores quânticos reais e uma conclusão final.

2. INICIO DA COMPUTAÇÃO

Em 1900, o matemático alemão David Hilbert propôs 100 problemas, os quais sobre sua perspectiva seriam os grandes desafios daquele século. Um deles era o *Entscheidungspro-*

blem, ou "Problema de Decisão", que questionava se existiria um procedimento mecânico para determinar a veracidade de qualquer conjectura ou questão matemática. Para responder a esse problema, Alan Turing descreveu o que conhecemos como "Máquina de Turing" ou "Máquina Determinística de Turing".

A "Máquina de Turing" consistia de uma fita unidimensional de dados e uma cabeça deslizante de leitura e escrita (Figura 1). A cabeça possuía um estado interno e um conjunto de instruções, as quais diziam, sobre o estado interno da cabeça, qual direção ela devia deslizar, qual o seu próximo estado e o que devia ser escrito na fita.

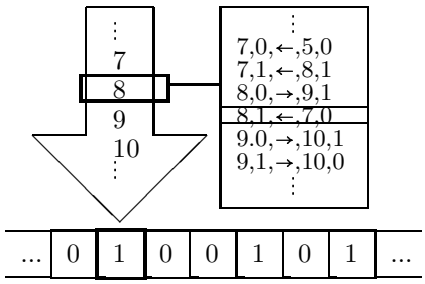


Figura 1: Uma Máquina Determinística de Turing. Nela, a flecha representa a cabeça de escrita e leitura com seu estado atual assinalado. Na caixa ao lado estão as instruções.

Esse mecanismo resolvia o *Entscheidungsproblem*, demonstrando não haver um procedimento mecânico capaz de decidir a veracidade de todos as conjecturas ou questões matemáticas. Um exemplo clássico de problema indecidível está em determinar se a máquina termina de executar para um conjunto de estados iniciais qualquer. Por outro lado, essa máquina possuía o mesmo poder de fornecer provas da própria matemática e, portanto, a matemática também era incompleta[5].

Não levando-se em conta alguns problemas ditos *indecidíveis*, a máquina proposta ainda representava um grande avanço, pois solidificava o conceito da computação. De fato, como modelo matemático, essa máquina fundamentou os computadores atuais. A máquina em si, nunca foi implementada comercialmente, mas sua simplicidade foi útil para o desenvolvimento da Teoria da Computação, já que qualquer teorema provado sobre ela é válido para todos as implementações de computadores.

No entanto, na época do desenvolvimento da Máquina de Turing, não estavam claras quais eram as suposições físicas intrínsecas no modelo matemático e quais seriam as implicações disso. Hoje, sabe-se que a Máquina de Turing sustenta-se, dentre outras, na suposição, de que os símbolos na cabeça de leitura e na fita existem unicamente em um dado momento e unidade de armazenamento, ou seja, uma região da fita pode conter 0 ou 1, em cada instante.

Essas suposições levantam a questão: A máquina de Turing é uma boa abstração matemática, mas ela é consistente com a física conhecida? A resposta seria afirmativa na época de Turing, pois a compreensão da física até então não poderia implementar o armazenamento de informação de outra maneira. No entanto, a resposta torna-se negativa quando consideramos pequenas dimensões e os fenômenos são explicados pela Quântica.

3. COMPUTAÇÃO QUÂNTICA

A crescente miniaturização dos computadores, como forma primária para obtenção de mais eficiência computacional, irá alcançar seu limite por volta de 2020, conforme a lei de Moore, com os processadores operando a cerca 40GHz. Neste ponto, serão necessários poucos átomos para representar um bit e as leis da Quântica serão as válidas.

É notável que os ganhos em utilizar efeitos quânticos em um computador não serão somente para viabilizar chips ainda mais minúsculos. Os fenômenos quânticos permitirão uma revolução no paradigma primordial da computação e proverão ganhos, em certas aplicações, nunca alcançáveis por computadores clássicos mesmo em paralelo.

Nesse contexto, David Deutsch em 1985, apresentou a "Máquina Quântica de Turing" em continuidade ao trabalhos de Richard Feynman. Essa máquina diferenciava-se inicialmente por ser reversível, ou seja, é possível da entrada obter a saída e vice-versa. Um exemplo de computação irreversível é a porta lógica AND, na qual a saída 0 não determina sua entrada. Essa reversibilidade é necessária em sistemas quânticos. No entanto, os dois pontos mais contrastantes nesse novo modelo, eram a possibilidade de armazenamento de 0, 1, ou ambos ao mesmo tempo, e o processamento de todos os caminhos possíveis em paralelo pela cabeça da máquina. Apesar desses fenômenos fugirem ao senso comum, e de fato não são explicáveis pela física clássica, eles são perfeitamente descritos pela física Quântica.

O primeiro ganho da "Máquina Quântica de Turing" sobre a "Máquina Determinística de Turing" é a possibilidade de simular sistemas quânticos de maneira eficiente. Atualmente, há argumentos fortes mostrando a necessidade de uma carga exponencial para "Máquina Determinística de Turing" simular esses sistemas, e conseqüentemente, os computadores atuais herdam essa ineficiência.

Nas próximas seções serão descritos os componentes necessários à construção de um computador Quântico que seja equivalente a uma "Máquina Quântica de Turing".

4. SUPERPOSIÇÃO E INTERFERÊNCIA

A melhor maneira de explicar a *superposição* e a *interferência*, os dois fenômenos principais para a Computação Quântica, é visualizar o experimento das duas fendas[1]. Nesse experimento, um canhão de elétrons é posicionado à frente de um anteparo com duas fendas e atrás do anteparo é colocado um detector de elétrons, conforme a Figura 2.

Caso uma das fendas seja deixada fechada, encontraremos no dectetor os padrões da Figura 2.a e 2.b. A explicação para esses padrões vêm do comportamento dual, onde partículas atômicas comportam-se ou como ondas ou como partículas em diferentes condições. Outra configuração para o experimento é quando as duas fendas estão abertas apresentando o padrão da Figura 2.c. Esse padrão é descrito pela *interferência* entre ondas.

Até aqui, existem explicações plausíveis pela física clássica para todos os resultados obtidos. O problema começa quando realizamos a emissão de um elétron por vez. Como o elétron pode passar por somente uma das fendas, não seria possível a interferência. No entanto, o resultado do experimento diz o contrário, como se o elétron interferisse consigo mesmo. O único modo de explicar esse comportamento é através da

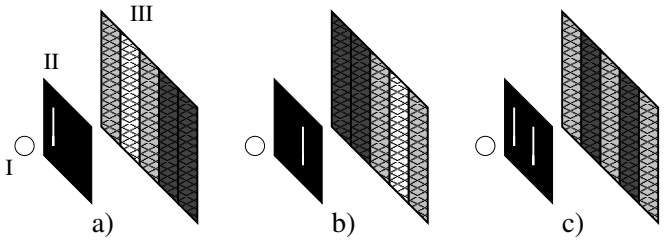


Figura 2: O ítem I é o emissor de elétrons. II é o anteparo com fendas e III o anteparo detector. Em a) e b) só uma das fendas estão abertas. Com as duas fendas abertas em c) percebe-se a *interferência*

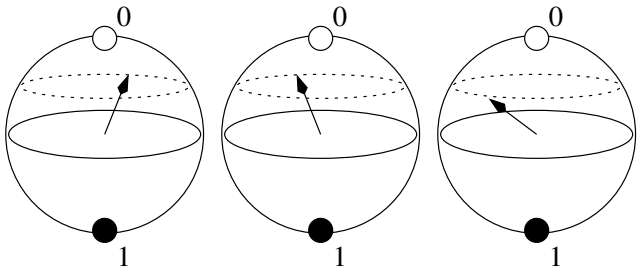


Figura 3: Três Qubits com mesmas probabilidades de obtenção de 0's ou 1's, porém, com fases diferentes

física Quântica.

4.1 Estado Quântico

Uma pergunta natural, sobre o experimento das duas fendas com uma partícula por vez, é "qual das duas fendas o elétron passou?". A explicação quântica para o experimento impede que responda-se a essa pergunta e ainda obtenha-se o padrão de *interferência*. Na verdade, a passagem do elétron por uma das fendas configura um estado quântico condicionado ao princípio da incerteza de Heisenberg[3]. Ou seja, para determinar a fenda devemos entrar em contato com a partícula e o efeito quântico não ocorrerá, pois houve decoerência.

Para compreender o que ocorre com o elétron, não devemos interagir com o sistema, mas podemos prever seu comportamento através de probabilidades. É nesse momento que introduziremos a notação **BraKet** ($\langle . || . \rangle$) de Dirac[3], originalmente utilizada em probabilidade condicional. Utilizaremos somente o **Ket** ($|.\rangle$) , que representa os eventos finais ou estados quânticos.

No caso do experimento, os eventos finais ou estados quânticos são "passou pela fenda 1", "passou pela fenda 2" ou uma *sobreposição* de ambos com diferentes probabilidades. Para poder representar todas as possibilidades de *sobreposição* devemos utilizar espaços de Hilbert ou, mais especificamente, *espaços vetoriais complexos de dimensão finita*.

O estado quântico $|\psi\rangle$, dizendo qual fenda o elétron passou, é representado neste caso como:

$$|\psi\rangle = c_0|\text{passa pela fenda 1}\rangle + c_1|\text{passa pela fenda 2}\rangle \qquad (1)$$

onde $c_0 = c_1 = \frac{1}{\sqrt{2}}$, e $|c_0|^2 + |c_1|^2 = 1$, pois $|c_0|^2$ e $|c_1|^2$ são as probabilidades de passar em cada uma das fendas. Como deve-se passar por alguma delas, o somatório das probabilidades é 1 (100%). Os termos $|\text{passa pela fenda 1}\rangle$ e $|\text{passa pela fenda 2}\rangle$ são bases ortogonais do espaço vetorial e os dois únicos valores possíveis de obter-se em uma medição.

Sendo assim, somos tentados a dizer que, portanto, a partícula passa pelas duas fendas em diferentes proporções e essas diferentes proporções interferem entre si. Na realidade, a partícula é uma entidade localizada e só podemos dizer sobre a probabilidade dela passar em cada uma das fendas. Entretanto, esse último argumento não explica a interferência. Na realidade, o comportamento resultante da *interferência* é condicionado pela simples probabilidade do elétron passar por cada uma das fendas e, portanto, o elétron realmente interfere consigo mesmo.

A utilização de elétron nesse experimento não é determinante para o comportamento obtido. Outras partículas atômicas, como prótons, nêutrons e até os próprios atomos já foram utilizados e obtiveram o mesmo resultado[2]. O estado quântico neste experimento também não é o único existente. O spin de um núcleo, o nível energético de um elétron e até mesmo a polarização do fóton são estados quânticos e obedecem as mesmas leis quânticas apresentadas.

4.2 Qubit

A unidade fundamental de informação clássica é o bit. Ele é caracterizado por uma chave com dois estados "desligado" ou 0 e "ligado" ou 1. A implementação física de um bit requer um dispositivo com dois estados possíveis, onde exista uma barreira forte o suficiente para impedir uma transição de estados, a não ser quando desejado.

O análogo quântico para o bit é o Qubit ou Quantum Bit, este também possui dois estados, porém quânticos, designados por $|0\rangle$ e $|1\rangle$. Teoricamente, qualquer sistema quântico que possua dois estados possíveis pode implementar um Qubit. Assim, denotamos o estado genérico de um Qubit como:

$$|Q\rangle = \alpha|0\rangle + \beta|1\rangle \qquad (2)$$

onde $\alpha, \beta \in \mathbb{C}$ e $|\alpha|^2 + |\beta|^2 = 1$. O que significa que o Qubit pode ter qualquer valor entre 0 e 1, e se realizarmos uma medição, ele tem probabilidade $|\alpha|^2$ de ficar em 0 e $|\beta|^2$ de ficar em 1. A Figura 3 mostra a representação gráfica de um Qubit. Os valores possíveis são todos aqueles na superfície da esfera apontados pela flecha, denotando a *fase* do Qubit. Essa forma é obtida por que os coeficientes são números complexos e a soma dos quadrados de suas normas resulta em 1.

A Figura 3 também ilustra três Qubits com mesma probabilidade de medir-se 0 e 1, porém com *fases* diferentes. Essa diferenças de fase acontecem, pois $|\alpha|^2 = |-\alpha|^2$. Apesar dessa propriedade ser inócua na perspectiva de medição, onde só importa a probabilidade, em computadores quânticos, os algoritmos funcionam através de transformações em Qubits, alterando os coeficientes complexos controladamente. Ou seja, trabalhamos com um vetor de entrada $In = (\alpha\,\beta)^T$ e temos $Out = XIn = (\alpha'\,\beta')^T$ após a aplicação de uma transformação X unitária, ou seja, uma bijeção do espaço original a ele mesmo.

5. MÚLTIPLOS QUBITS E PARALELISMO

Ao construir um computador quântico ou mesmo para a "Máquina Quântica de Turing" precisamos lidar com mais

de um Qubit. Análogamente à computação clássica, podemos criar um *registrador quântico*. No entanto, enquanto que em um registrador clássico de n bits podemos ter qualquer número de 0 até 2^n, mas somente um desses números por vez, em um *registrador quântico* podemos ter uma *superposição* de todos eles.

Um exemplo de estado quântico para um registrador de 3 Qubits seria:

$$|\psi\rangle = c_0|0\rangle|0\rangle|0\rangle + c_1|0\rangle|0\rangle|1\rangle + c_2|0\rangle|1\rangle|0\rangle + \cdots + c_7|1\rangle|1\rangle|1\rangle \quad (3)$$

ou ainda, na base decimal:

$$|\psi\rangle = c_0|0\rangle + c_1|1\rangle + c_2|2\rangle + c_3|3\rangle + \cdots + c_6|6\rangle + c_7|7\rangle \quad (4)$$

onde $|c_0|^2 + |c_1|^2 + \cdots + |c_7|^2 = 1$. Isso significa que podemos ter todos esses números ao mesmo tempo dentro de um registrador. Dessa forma, a capacidade de lidar com informação é exponencialmente maior se comparado ao modo clássico. É claro que, ao realizarmos uma medição somente um dos 2^n números será visto. Isto aparentemente nos leva a concluir que afinal não existem ganhos reais em um *registrador quântico*. Entretanto, é nesse momento que introduzimos o *paralelismo quântico*, a habilidade especial que faltava para concluir a "Máquina Quântica de Turing".

O *paralelismo quântico* acontece quando realizam-se operações em um *registrador quântico* em estado de *superposição*. Quando isso ocorre, a operação é realizada sobre todos os valores superpostos, conseqüência direta da *interferência*, e obtém-se com isso um ganho exponencial de tempo. Já que não podemos obter todos os resultados dessa operação ao realizar uma medição, devemos aplicar as operações de forma que somente as respostas válidas possuam probabilidades altas e seja possível recuperá-las após um tempo previsto.

5.1 Qubits emaranhados e não-Localidade

Uma outra forma mais interessante de apresentação de múltiplos Qubits é quando estes estão *emaranhados*. Para compreender esse fenômeno, imagine um par de partículas emitidos de uma fonte em sentidos opostos do mesmo eixo (Figura 4). Caso a partícula 1 esteja no raio apontando para cima, a partícula 2 estará no raio apontando para abaixo e vice-versa. Sendo assim, após a emissão das partículas, o estado quântico do sistema é:

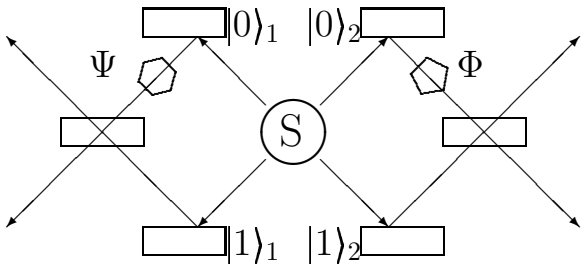


Figura 4: A fonte de S emite duas partículas em sentidos opostos de um mesmo eixo. Assim, ao medir-se o estado quântico de uma partícula saberemos exatamente o estado da outra, pois elas estão *emaranhadas*.

$$|Q\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle_1|1\rangle_2 + |1\rangle_1|0\rangle_2\right) \quad (5)$$

Veja que neste estado quântico não possuímos todas bases possíveis com dois Qubits normais. Isso acontece por que quando os Qubits estão *emaranhados*, o estado quântico de um Qubit implica no estado quântico do outro. No caso desse experimento, o fato de serem emitidos em sentidos opostos, implica que ao medir-se o primeiro Qubit já concluiremos com certeza a condição do outro sem necessidade de medição.

Por outro lado, o resultado mais interessante desse fenômeno acontece após a aplicação de diferentes rotações Ψ e Φ nas fases das partículas. Feito isso, ao realizar certas medições, não é possível explicar os parâmetros obtidos em uma partícula sem considerar a transformação realizada na outra partícula!

Esse fenômeno é conhecido como *não-localidade* e a influência é detectada independentemente da distância entre as partículas, sugerindo uma interação não-local entre elas. Apesar de cientistas como Einstein, Podolsky e Rosen[6] argumentarem filosoficamente que existiria uma "variável escondida" ligando as partículas e realizando a interação local, Jonh Bell demonstrou matematicamente a inconsistência desse argumento, evidenciando a *não-localidade*.

5.2 O ciclo Preparar-Evoluir-Medir

Um computador clássico opera em um ciclo de **Carregar-Executar-Gravar**, já um computador quântico opera no ciclo **Preparar-Evoluir-Medir**. A etapa de preparação consiste em inicializar o *registrador quântico* em um estado qualquer, como por exemplo uma superposição de todos os números com igual probabilidade de medição. Após isso, é necessário aplicar um conjunto de operações reversíveis, que como dito anteriormente, são transformações nos coeficientes complexos dos Qubits. Por fim, deve-se medir o resultado obtido após um tempo determinado.

Em uma "Máquina Quântica de Turing" este ciclo faria com que todas as decisões fossem tomadas entre o estado inicial até o momento da medição, através do *paralelismo quântico*. Deve-se, portanto, implementar essas operações de forma que todos os caminhos levem no momento da medição às respostas do problema. Outra implicação desse ciclo, é incapacidade de uma "Máquina Determinística de Turing" eficientemente simular a fase de evolução com todo seu paralelismo ou armazenar em sua memória um estado de superposição preparado inicialmente.

Não há, entretanto, ganhos em computabilidade. A classe dos problemas *indecidíveis* é inalterada sobre a perspectiva Quântica em relação à clássica. Os ganhos estão em complexidade, sendo possíveis novos limitantes inferiores inalcançáveis anteriormente.

Da mesma forma que a versão clássica determinística, a "Máquina Quântica de Turing" é um modelo matemático útil para provar teoremas sobre Computação Quântica. Há também a implementação lógica deste conceito matemático, a qual é base para a construção de um computador quântico real. Nas próximas seções, serão discutidos os componentes básicos para implementá-lo e com isso será possível definir algoritmos quânticos.

6. PORTAS QUÂNTICAS E CIRCUITOS QUÂNTICOS

Para a construção de computador quântico, ou mesmo para a descrição de um algoritmo, é necessária a definição de elementos básicos sobre os quais seja possível criar qualquer outra operação. Em um computador clássico, as portas lógicas AND, NOT e a ligação delas através de suas entradas e saídas são suficientes para executar qualquer algoritmo ou mesmo construir um computador genérico nos moldes de Von Neunman. Para os computadores quânticos, precisamos das portas lógicas quânticas de Rotação de 1-Qubit (um conjunto infinito de portas) e NOT-Controlado, detalhadas a seguir, e que cada uma dessas portas sejam reversíveis. Além disso, a ligação entre as portas deve transferir estados quânticos em *superposição*.

6.1 Rotação de 1-Qubit

Como foi dito anteriormente, transformações sobre Qubits são alterações nos coeficientes complexos de suas bases. Além disso, no caso de um Qubit essa alteração representa uma rotação da fase, levando na representação em esfera (Figura 3) de um ponto da superfície a outro. Uma porta lógica quântica de rotação de 1-Qubit pode ser construída para realizar uma rotação arbitrária. Como existe um número infinito de rotações possíveis, existem infinitas portas de rotação de 1-Qubit que podem ser definidas.

Um exemplo é a porta "Raiz-Quadrada-do-NOT" ($\sqrt{\text{NOT}}$), a qual recebe uma entrada e fornece uma saída. Se colocadas duas dessas portas encadeadas temos a operação NOT. Como $\sqrt{\text{NOT}}$ trata-se de uma transformação, só é necessário defini-la sobre as bases do espaço vetorial para ter o resultado sobre qualquer outro ponto do espaço:

$$U_{\sqrt{\text{NOT}}}|0\rangle = \left(\frac{1}{2} + \frac{i}{2}\right)|0\rangle + \left(\frac{1}{2} - \frac{i}{2}\right)|1\rangle$$
$$U_{\sqrt{\text{NOT}}}|1\rangle = \left(\frac{1}{2} - \frac{i}{2}\right)|0\rangle + \left(\frac{1}{2} + \frac{i}{2}\right)|1\rangle$$

E para a operação NOT:

$$\text{NOT}|0\rangle = U_{\sqrt{\text{NOT}}}U_{\sqrt{\text{NOT}}}|0\rangle = |1\rangle$$
$$\text{NOT}|1\rangle = U_{\sqrt{\text{NOT}}}U_{\sqrt{\text{NOT}}}|1\rangle = |0\rangle$$

Uma outra porta de rotação 1-Qubit muito importante é a de Walsh-Hadamard, definida como:

$$U_{\text{WH}}|0\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)$$
$$U_{\text{WH}}|1\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right)$$

Essa porta, se aplicada a um Qubit com $|0\rangle$, leva ao meio do caminho até $|1\rangle$ e vice-versa. Nestes casos, a probabilidade de obter 0 ou 1 em uma medição será de 50% para cada.

A porta de Walsh-Hadamard é muito utilizada em algoritmos quânticos para, a partir de um registrador quântico com n Qubits, deixá-los todos em estado de superposição, com o registrador possuindo os 2^n valores possíveis com igual probabilidade de serem medidos.

6.2 NOT-Controlado

Para conseguir alcançar qualquer tipo de Computação Quântica, só é necessário mais uma porta lógica em conjunto às de rotação de 1-Qubit, a porta NOT-Controlado ou CN, definida como:

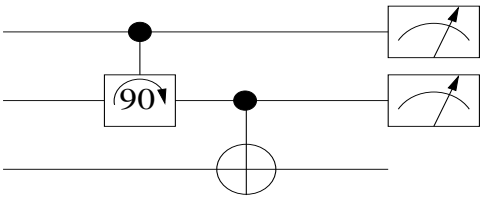


Figura 5: Exemplo de um circuito quântico com 3 Qubits de entrada, duas medições, uma porta de rotação e uma NOT-Controlado.

$$U_{\text{CN}}|00\rangle = |00\rangle$$
$$U_{\text{CN}}|01\rangle = |01\rangle$$
$$U_{\text{CN}}|10\rangle = |11\rangle$$
$$U_{\text{CN}}|11\rangle = |10\rangle$$

CN é uma porta 2-Qubit que, basicamente, inverte o segundo Qubit caso o primeiro seja $|1\rangle$ e não altera-o, caso contrário. Deve-se notar que esta lógica condicional não envolve medições em momento algum.

A influência de um Qubit no outro, apresentado por esta porta, realiza a ligação necessária para criar circuitos quânticos e ainda pode ser utilizada para produzir estados *emaranhados*, extremamente úteis em algoritmos quânticos.

6.3 Circuitos Quânticos

A Figura 5 demonstra o esquema básico de um circuito quântico. Essa representação é independente de implementação física. Nela, as linhas horizontais representam a transferência de um estado quântico da esquerda para a direita. As linhas verticais, interligando duas horizontais, sincronizam as duas transferências.

A presença de um símbolo •, ligando um estado quântico a uma porta lógica, designa um controle desse estado quântico sobre a operação da porta lógica, similarmente ao CN. Durante as linhas horizontais, mas mais comumente ao fim delas, pode haver uma medição representada pelo símbolo da fase.

É comum a utilização de diagramas simplificados, onde há linhas horizontais com múltiplos Qubits e "caixa-pretas", que não possuem seu circuito interno mostrado.

7. ALGORITMOS QUÂNTICOS

A Computação Quântica fornece um novo paradigma para o processamento de informação e seu armazenamento. A grande questão é: Quais aplicações poderão tirar proveito de todo o seu potencial? A primeira grande resposta para essa pergunta, e que trouxe atenção e investimentos à Computação Quântica, foi a fatoração de inteiros. Enquanto, outras aplicações já demonstravam a obtenção de novos limitantes inferiores para algoritmos[5] ou mesmo a viabilidade de simulações quânticas, a fatoração de inteiros possuía implicações de proporções gigantescas.

7.1 Fatoração de Inteiros

Grande parte da segurança em comunicações baseia-se na utilização de chaves para encriptação e decriptação. O RSA

é um esquema muito famoso baseado em tal conceito. Nele, uma pessoa faz a distribuição em um canal inseguro de uma chave pública. Caso outra pessoa desejar comunicar-se, ela deve encriptar a mensagem com aquela chave. A impossibilidade de decriptação dos dados, senão por uma chave privada, cria um canal seguro de comunicação.

Para possibilitar a encriptação e decriptação, a construção dessas chaves utiliza, por exemplo, dois números primos grandes como fatores. Mesmo sendo possível, teoricamente, a partir da chave pública descobrir a chave privada, seria necessário realizar a fatoração de inteiros. A suposta intratabilidade desta tarefa garante a segurança das comunicações.

Não há provas de que a fatoração de inteiros seja um problema intratável classicamente. Mas, atualmente, o melhor algoritmo para a resolução do problema, o "Number Field Sieve", possui tempo de execução proporcional a $exp(cL^{1/3}log(L)^{2/3})$ onde L é a quantidade de bits necessários para representar o número. A última conquista desse algoritmo foi a resolução do desafio RSA-640, onde o número original possuía 193 dígitos decimais. Para conseguir isso, foram necessários 80 processadores Opteron de 2.2Ghz e 3 meses de processamento.

Aparentemente, o paralelismo clássico pode ser usado intensamente para fatorar números mais desafiadores. Entretanto, pode-se afimar que: dado um número com 2000 dígitos para o algoritmo "Number Field Sieve", mesmo se cada átomo do universo fosse um computador clássico processando à máxima velocidade por toda a vida do universo, o esforço não seria suficiente para fatorar o número.

Essa incapacidade dos computadores clássicos pode ser resolvida caso seja encontrado um algoritmo polinomial para o problema, o que ainda não aconteceu. Neste caso, a Computação Quântica poderia fornecer algum ganho?

7.1.1 O Algoritmo de Shor

Em 1994, Peter Shor[4] descobriu um modo de fatorar inteiros utilizando um computador quântico em tempo polinomial. Para conseguir esse feito, ele necessitou principalmente de resultados anteriores sobre Transformadas Rápidas de Fourier, por Simon[1], e da relação entre o período de uma função periódica e os fatores de um número.

Devemos detalhar cada um desses resultados para compreender o Algoritmo de Shor.

7.1.1.1 Teoria dos Números e Fatoração de Inteiros.

Tendo um número n (o número a ser fatorado), definimos a função $f_n(a) = x^a \bmod n$, onde x é relativamente primo a n, pois $mdc(x,n) = 1$. Sendo assim, essa função é periódica, ou seja, existe um $f_n(x+r) = f(x)$ e r é o período da função. A teoria do números, diz que há grande probabilidade do $\mathrm{mdc}(x^{r/2}-1,n)$ ser um fator não-trivial de n, ou seja, diferente de 1 e do próprio n.

Para exemplificar, digamos que $n = 15$ e $x = 8$ escolhido aleatoriamente. Calculando $f_{15}(a) = 8^a \ mod\ 15$ para $a = \{0,1,2\dots\}$, temos a seqüencia $\{1,8,4,2,1,8,4,2,1,\dots\}$. Percebe-se rapidamente que $f_{15}(a+4) = f(a)$, portanto, $r = 4$. Finalmente, computando $\mathrm{mdc}(x^{r/2}-1,n)$, temos $\mathrm{mdc}(8^{4/2}-1,15) = \mathrm{mdc}(63,15) = 3$. De fato, 3 é um fator não-trivial de 15 e tendo este fator, é fácil descobrir o outro fator com $15/3 = 5$. Os dois fatores são, então, 3 e 5.

7.1.1.2 Transformada Rápida de Fourier e Periodicidade.

A partir dos resultados anteriores, poderíamos chegar facilmente a um algoritmo probabilístico para fatorar inteiros. O único problema está em achar $f_n(x+r) = f(x)$, o qual necessitaria $O(2^L)$ tentativas em um computador clássico, onde L é a quantidade de bits necessários para representar o número n. Usando quântica conseguimos calcular r com somente $O(L^2)$ passos, representando uma aceleração exponencial neste algoritmo.

Para tal, calculamos todos os valores de f_n usando *paralelismo quântico* sobre um estado $|x\rangle$ inicial em superposição:

$$|f_n\rangle = \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} |x\rangle |f_n(x)\rangle \tag{6}$$

O estado $|f_n(x)\rangle$ resultante da aplicação função f_n possui sua periodicidade, mas não está claro ainda como extraí-la. Um fato interessante é que a aplicação de f_n em $|x\rangle$ torna-o *emaranhado* a $|f_n(x)\rangle$. Portanto, ao medirmos $|f_n(x)\rangle$ teremos um valor y_0 e o estado $|x\rangle$ tornará-se uma superposição de todos valores x tais que $f_n(x) = y_0$. Se x_0 é o menor desses valores e k é o menor número tal que $n \geq k.r$, podemos representar esse novo estado da entrada como:

$$|\psi\rangle = \frac{1}{\sqrt{k}} \sum_{p=0}^{k-1} |x_0 + p.r\rangle \tag{7}$$

Se fosse realizada uma medição em $|\psi\rangle$ não conseguiríamos extrair r, pois a presença do x_0 torna o resultado inútil. É nesse momento que a Transformada Discreta de Fourier (**DFT**) possui utilidade. A DFT consegue representar uma função de domínio finito em uma base periódica e insensível a deslocamentos. Como essa transformação é unitária, podemos aplicá-la a um estado quântico e se fizermos isso a $|\psi\rangle$ temos:

$$\mathcal{F}|\psi\rangle = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} e^{2\pi i \frac{x_0 j}{r}} |j\frac{n}{r}\rangle \tag{8}$$

Note agora, que x_0 não aparece mais no estado quântico, portanto, ao medirmos $\mathcal{F}|\psi\rangle$ obteremos $c = j\frac{n}{r}$, para algum j aleatório. O importante disso é que sabemos c pela medição e n é nosso número sendo fatorado. Como $0 \leq j \leq r-1$, há uma probabilidade de $\frac{1}{\log r}$ de que ele seja relativamente primo a r, e caso isso aconteça, conseguimos descobrir r a partir de $c = j\frac{n}{r}$, cancelando $\frac{c}{n}$ até uma fração irredutível. Não é difícil provar que, ao realizar $O(L)$ medições de $\mathcal{F}|\psi\rangle$, é ínfima a possibilidade de não acharmos r.

Para garantirmos o tempo $O(L^2)$ devemos utilizar a implementação quântica da Transformada Rápida de Fourier[3].

7.1.1.3 O Algoritmo.

De forma sucinta o Algoritmo de Shor utiliza os resultados da Teoria dos Números, junto a um cálculo quântico eficiente do período de uma função periódica, para fatorar números inteiros. Essa união produz um algoritmo quântico polinomial para a tarefa, como foi mostrado.

7.2 Comunicação Segura

Devido aos resultados do Algoritmo de Shor, a viabilidade de um computador quântico tornaria inseguro os esquemas atuais de comunicação encriptada. Em contrapartida, é

possível através da Quântica criar um canal de comunicação 100% seguro.

Existem diversas formas de implementação deste canal seguro, mas todas baseiam-se no fenômeno quântico da *não-clonabilidade*. Este fenômeno representa a impossibilidade de copiar um estado quântico superposto sem realizar um número infinito de medições. Sendo assim, se alguma pessoa espionar uma transmissão de estados quânticos estará invariavelmente alterando o estado quântico, o qual poderá ser detectado por um protocolo especial de comunicação.

A comunicação quântica segura mais conhecida é realizada através de fótons polarizados. Nela, é possível a criação de uma chave secreta compartilhada através do uso de dois tipos de medidores de polarização.

7.2.1 Polarização do Fóton e Bases Não-Ortogonais

A polarização de um fóton quando medida apresenta dois resultados possíveis. Já esses resultados podem ser vistos em duas bases diferentes: a base horizontal-vertical $\oplus$, com os resultados $|\updownarrow\rangle$ e $|\leftrightarrow\rangle$; e a base diagonal $\otimes$, com os resultados $|\nwarrow\rangle$ e $|\nearrow\rangle$. A existência dessas bases fornece dois padrões diferentes para representar os estados $|0\rangle$ e $|1\rangle$ de um Qubit.

Inicializando um Qubit em $|0\rangle$ ou $|1\rangle$ em alguma das bases citadas, temos um estado superposto em relação à outra. Isso porque, essas bases são não-ortogonais. Por exemplo, caso enviemos um $|0\rangle$ na base $\oplus$ para alguém e a outra pessoa utilizar a base $\otimes$, ela terá 50% de ler ou $|0\rangle$ ou $|1\rangle$.

7.2.2 Protocolo de Comunicação

Duas pessoas, Alice e Bob, desejam comunicar-se de forma segura. Primeiro, elas irão definir uma chave privada compartilhada só conhecida por elas através de um canal quântico. Depois disso, irão comunicar-se de forma segura com dados encriptados.[1]

Para conseguir definir a chave compartilhada, Alice e Bob seguirão os seguintes passos:

1. Alice define um conjunto de Qubits e aleatoriamente escolhe entre as bases $\oplus$ e $\otimes$ para representar cada um deles.
2. Alice envie os fótons polarizados para Bob.
3. Bob escolhe aleatoriamente entre os medidores nas bases $\oplus$ e $\otimes$ para medir cada Qubit enviado por Alice.
4. Alice e Bob divulgam entre si as bases utilizadas.
5. Alice e Bob verificam quais Qubits tiveram as mesmas bases escolhidas e enviam suas medições nessas bases.
6. Se tudo ocorreu bem, Alice e Bob não reclamarão de divergências entre suas medições. A chave então possui o número de Qubits aceitos e se for necessário mais Qubits, reinicia-se o processo.

Uma terceira pessoa, chamada Eve, deseja saber o que Alice está enviando para Bob. Para não ser descoberta, Eve intercepta o fóton de Alice, utiliza um medidor sobre alguma das bases $\oplus$ e $\otimes$ e reenvia o fóton para Bob no valor medido. Felizmente, quando Alice e Bob verificam os Qubits onde utilizaram as mesmas bases, há uma probabilidade de $\frac{1}{4}$ do valor não ser o mesmo, expondo Eve. Isso acontece quando Eve escolhe a base errada e Bob a correta, levando o Qubit reenviado a Bob a ficar em superposição em relação a sua base.

Neste protocolo, temos uma garantia de $1 - (\frac{3}{4})^N$ de detectar um espião, onde N é o número de Qubits verificados. Essa probabilidade aproxima-se rapidamente de 1, sendo ínfimamente possível não detectar o espião.

7.2.3 Outras implementações

A implementação através de fótons polarizados é útil para a compreensão e verificação da validade da comunicação segura, mas não é a mais eficiente. As distâncias máximas permitidas de comunicação não passam de 1 metro. Já existem outras implementações[2], beirando a comercialização, que ultrapassam a marca de kilômetros.

7.3 Outros algoritmos quânticos

Enquanto na Computação Quântica, as aplicações mais reais da são na área de criptografia, outros algoritmos quânticos foram propostos para demonstrar seus poderes teóricos. Um dos primeiros deles, foi o Algoritmo de Deutsch-Jozsa[5], que demonstrava como os estados *emaranhados* são determinantes para o *paralelismo quântico*. Mas o mais famoso entre os teóricos é o Algoritmo de Grover para realizar buscas sublineares em conjuntos de números não-ordenados. Esse algoritmo mostra como a Computação Quântica pode redefinir os limitantes inferiores de alguns problemas.

8. COMPUTADORES QUÂNTICOS REAIS

A construção de um computador clássico sofreu grandes evoluções antes de chegar ao estado atual. Nessa evolução um dos desafios foi respeitar um conjunto de restrições de estabilidade, sem as quais não seria possível construir circuitos. Entretanto, o maior desafio foi, e continua sendo, a maximização de certos objetivos, como eficiência e armazenamento.

O computador quântico possue exatamente esses desafios, porém em condições opostas. Enquanto um computador quântico já nascerá em proporções mínimas, fornecendo uma capacidade de processamento jamais visto, é muito difícil construí-lo de forma estável.

Atualmente, os esforços na construção de computadores quânticos reais estão em respeitar as seguintes condições de estabilidade:

1. Representar de forma robusta a informação quântica.
2. Realizar uma família universal de transformações unitárias (portas lógicas).
3. Preparar fielmente um estado inicial.
4. Medir o resultado da saída.

Existem muitas tecnologias que tratam essas restrições, mas duas vem apresentado resultados práticos notáveis: os ópticos e os de ressonância magnética nuclear.

Computadores Quânticos Ópticos

[1] Note que, um canal quântico só é necessário durante a definição da chave compartilhada. Após isso, ela será um conjunto de bits clássicos, podendo ser utilizada em um canal de comunicação clássico

[2] www.infoworld.com/article/04/09/16/HNneccryptography_1.html

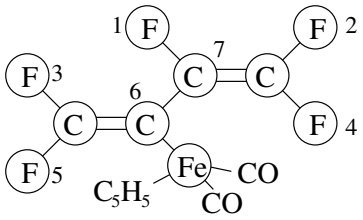


Figura 6: A molécula utilizada para implementar um Algoritmo de Shor com 7 Qubits em um Computador Quântico de Ressonância Magnética Nuclear. Os números ao lado de alguns átomos enumeram seus respectivos Qubits.

Nessa classe de implementações, utiliza-se fótons para representar Qubits. As transformações e medições dos estados quânticos dos fótons são feitas principalmente por aparelhos óticos. Nesse âmbito, o grande problema desse tipo de implementação está em realizar a interação entre fótons, necessária para produção de estados *emaranhados* e construção da porta NOT-Controlado. Existem duas alternativas, ou utiliza-se um meio especial (nonlinear Kerr media), atualmente não eficiente, ou cavidades eletromagnéticas em átomos. A última tem obtido melhores resultados, mas ainda está longe de viabilizar grandes interações entre fótons.

Esta dificuldade em construir as portas NOT-Controlado impede um uso amplo em Computação Quântica, mas há grandes ganhos na área de Criptografia Quântica, onde não é necessária a interação entre fótons. Além disso, fótons são fáceis de emitir e trafegam por distâncias consideráveis utilizando pouca energia, viabilizando as comunicações.

Computadores Quânticos de Ressonância Magnética Nuclear

Atualmente, a implementação mais efetiva em Computadores Quânticos é através de Ressonância Magnética Nuclear. Nela um Qubit é representado pelo spin do núcleo atômico e, geralmente, as transformações são realizadas através de fortes campos magnéticos. O spin é medido por uma voltagem induzida pela processamento do momento magnético do núcleo.

Para implementar um registrador quântico é utilizada uma molécula, a qual é escolhida especialmente de forma que os Qubits a interagir possuam ligações químicas entre seus átomos representantes.

Sendo assim, diferentes tipos de moléculas implementam as arquiteturas necessárias para cada tipo de algoritmo quântico. Existe uma implementação do Algoritmo de Shor com 7 Qubits através da molécula da Figura 6, e do algoritmo de Grover e Deutsch-Jozsa com moléculas de Clorofórmio[2]. Invariavelmente, esses computadores perdem sua eficiência exponencialmente com o aumento de Qubits.

9. CONCLUSÃO

A Computação Quântica modifica os pilares da Teoria da Computação e seus ganhos são notáveis ao fornecer soluções para problemas tidos como intratáveis pela Computação Clássica, como a fatoração de inteiros.

Além disso, é possível com a Quântica criar canais de comunicação seguros que dispensam esquemas de criptografia baseados em conjecturas matemáticas.

Entretanto, a implementação real desses mecanismos ainda está em um nível muito inicial, sendo impossível dizer quando será possível utilizar desses ganhos na prática.

Existem, também, muitos outros assuntos em Quântica não tratados aqui, como: Teoria da Informação Quântica, Novas Classes de Complexidade, Correção de Códigos, Teletransporte, entre outros.

De fato, a Quântica ainda possui grande atividade no desenvolvimento de suas bases. Isso porque, mesmo sendo umas das físicas mais precisas até hoje, seus experimentos são muito controlados e facilmente perde-se a capacidade de previsão quando há uma interação com o meio. É necessário, portanto, mais conhecimento desses efeitos para viabilizar a construção de Computadores Quânticos.

10. REFERÊNCIAS

[1] Dirk Bouwmeester, Artur Ekert, and Anton Zeilinger. *The Physics of Quantum Information*. Springer-Verlag, Berlin Heidelberg New York, 2001.

[2] C. Macchiavello, G. M. Palma, and A. Zeilinger. *Quantum Computation and Quantum Information Theory*. World Scientific Publishing, Singapore, 1999.

[3] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, UK, 2000.

[4] Peter W. Shor. Algorithms for quantum computation: Discrete log and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. Institute of Electrical and Electronic Engineers Computer Society Press, 1994.

[5] Colin P. Williams and Scott H. Clearwater. *Ultimate Zero And One*. Copernicus Springer-Verlag, New York, 2000.

[6] H. M. Wiseman. From einstein's theorem to bell's theorem: A history of quantum nonlocality. *CONTEMPORARY PHYSICS*, 47:79, 2006.

Trace Scheduling

Márcio Paixão Dantas
Universidade Estadual de Campinas
Instituto de Computação
Campinas, Brasil
ra049174@students.ic.unicamp.br

RESUMO

Trace Scheduling é uma técnica de otimização (ou compactação) global de microcódigo criada no início da década de 80. Este trabalho tem como objetivos: explicar o que é *Trace Scheduling*, mostrar algumas variações presentes na literatura e fazer uma comparação entre elas.

Termos Gerais

Dependência de dados, otimização global de microcódigo, compactação de microcódigo, escalonamento de instruções paralelas, processamento paralelo, conflito de recursos.

Palavras-Chave

Otimização global de microcódigo, *trace scheduling*.

1. INTRODUÇÃO

Técnicas de otimização (ou compactação) de microcódigo são importantes porque permitem o desenvolvimento de compiladores eficientes para linguagens de microprogramação de alto nível. Outro fator que impulsiona o estudo de técnicas deste tipo é que à medida em que os microprogramas aumentam de tamanho, torna-se humanamente impossível identificar todas as oportunidades de otimização no código [4]. O problema consiste basicamente em converter microcódigo sequencial (ou vertical) em eficiente microcódigo paralelo (ou horizontal). Historicamente, há duas abordagens de otimização para este problema:

1. local - compactação de blocos básicos[1], separadamente;
2. global - compactação do microprograma, como um todo.

Ao se estudar o problema de otimização local, descobriu-se que o problema é NP-Completo e, então, várias heurísticas foram propostas para resolvê-lo. Desde a década de 80 considera-se o problema de otimização local bem resolvido. [1] e [2] recomendam alguns *surveys* sobre otimização local.

[1]Sequência de instruções sem desvios, exceto no seu começo e fim.

Inicialmente, as técnicas de compactação global consistiam no uso de otimização local nos blocos básicos, separadamente, e, em sequência, na busca por movimentações de operações entre os blocos. [1] mostrou através da técnica de *Trace Scheduling* que compactar blocos separadamente, sem levar em conta as necessidades e capacidades dos blocos vizinhos, leva a escolhas arbitrárias que prejudicam o processo de otimização.

Este trabalho tem como objetivo apresentar os conceitos básicos de *Trace Scheduling* e algumas importantes variações desta técnica. A Seção 2 explora o algoritmo original de *Trace Scheduling*. Nas Seções 3, 4, 5 são apresentadas, respectivamente, as seguintes técnicas: *Singly Rooted Directed Acyclic Graph* (SRDAG), *Tree Compaction* e *Improved Trace Scheduling Compaction* (ITSC). Depois, na Seção 6, é feita uma comparação dos algoritmos apresentados baseando-se no trabalho de [4]. Finalmente, o trabalho é concluído na Seção 7.

2. TRACE SCHEDULING

Ao invés de trabalhar sobre blocos básicos, *Trace Scheduling* trabalha sobre *traces*. Um *trace* ou caminho de execução é uma sequência de instruções livre de ciclos que pode ser executada contiguamente para alguma escolha de dados (entrada). Dado um *trace*, $T = (m_1, m_2, \cdots, m_t)$, um grafo direcionado acíclico (DAG) é construído a partir da seguinte ordenação parcial em T:

- se m_i escreve em um registrador e m_{i+n}, $n > 1$, lê este valor, então $m_i \ll m_{i+n}$ de forma que m_{i+n} não tentará ler o dado até que esteja lá;
- se m_i lê um registrador e m_k é o próximo a escrever neste registrador, então $m_i \ll m_k$. Assim, m_k não sobrescreverá o registrador até que este seja totalmente lido.

O DAG formado pela ordenação parcial $\ll$ é denominado DAG de precedência de dados. Os nós são as instruções de T e arestas de m_i para m_j são estabelecidas se $m_i \ll m_j$.

Dado T, com um DAG de precedência de dados associado, uma compactação ou escalonamento é definido como qualquer particionamento de T em uma sequência exaustiva de subconjuntos disjuntos, $S = (S_1, S_2, \cdots, S_h)$, que satisfaça as seguintes propriedades:

- em cada elemento de S, não pode haver instruções com conflitos estruturais (*hardware*);
- se $mi \ll m_j$, com m_i em S_k e m_j em S_h, então $k < h$. Isto é, o escalonamento preserva a dependência de dados.

O DAG construído para um *trace* abstrai todas as restrições necessárias sobre a movimentação de instruções entre blocos. Para isto, é preciso fazer algumas modificações no DAG de dependência de dados, tendo como base algumas regras básicas de movimentação de microoperações (MOPS's) entre blocos, explicitadas na Tabela 1.

Regra	**De**	**Para**	**Condições**
R1	B2	B1 e B4	a MOP está livre no topo de B2
R2	B1 e B4	B2	cópias idênticas da MOP estão livres no final de ambos B1 e B4
R3	B2	B3 e B5	a MOP está livre no final de B2
R4	B3 e B5	B2	cópias idênticas da MOP estão livres nos topos de B3 e B5
R5	B2	B3 (ou B5)	a MOP está livre no final de B2 e todos os registradores escritos pela MOP estão mortos em B5 (ou B3)
R6	B3 (ou B5)	B2	a MPO está livre no topo de B3 (ou B5) e todos os registradores escritos pela MOP estão mortos em B5 (ou B3).

Table 1: Regras de movimentação de MOP entre blocos básicos da Figura 1.

Uma MOP é *livre no topo* de seu bloco se ela tem nenhum predecessor no DAG de precedência de dados do bloco e, *livre no fim*, se não possui sucessores.

Um *registrador está* vivo em algum ponto do grafo de fluxo se o valor armazenado nele pode ser referenciado em algum bloco depois daquele ponto, mas antes que ele seja sobrescrito. Se o registrador não está vivo em um ponto, então está *morto*.

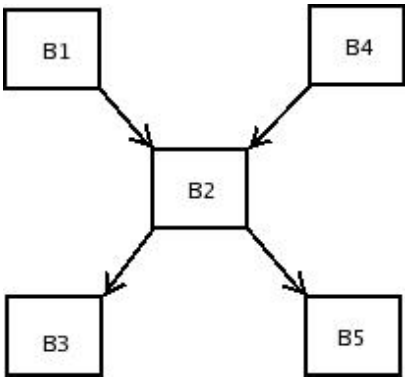


Figure 1: Grafo de fluxo entre blocos básicos.

O escalonador compacta o *trace* produzindo o menor escalonamento possível. O algoritmo usado para fazer isto é o *List Scheduling*, o qual é explicado resumidamente no trabalho de [1].

Uma vez que o escalonador trabalha sobre um *trace* e não sobre o código-fonte do microprograma, é preciso depois do escalonamento de T duplicar algumas MOP's em locais fora do *trace*. Isto mantém o programa correto ao percorrer outros caminhos de execução. Esta etapa chama-se *bookkeeping* e é bastante complexa. Uma explicação conceitual é dada em [1].

Seja, P, uma sequência de instruções (código vertical) que representa um microprograma. De forma resumida, o algoritmo de *trace scheduling* funciona da seguinte forma:

- Para escalonar P, repetidamente é escolhido a partir das MOP's ainda não compactadas o *trace*, T, mais provável de se concretizar. Um DAG com base em T é construído e compactado. O escalonamento é integrado a P fazendo-se as devidas correções (*bookkeeping*). O processo se repete até que todas as MOP's estejam compactadas.

A escolha de qual caminho de execução é o mais provável geralmente baseia-se em simulações ou experimentos reais com o microprograma P.

3. SINGLY ROOTED DIRECTED ACYCLIC GRAPH (SRDAG)

Trace Scheduling compacta caminhos ou *traces* do programa essencialmente como se fosse blocos básicos. As interações entre blocos do caminho são tratadas na construção do DAG de caminhos do programa. Entretanto, interações com blocos fora do caminho não são levadas em consideração. A idéia que suporta esta estratégia é que apenas os caminhos com maior probabilidade de execução devem ser compactados. Porém, segundo [2], compactar blocos de um caminho pode alterar a probabilidade de execução do mesmo. Assim, após compactar um bloco do caminho, deveríamos selecionar novamente o caminho mais provável.

Para aumentar as possibilidades de compactação e escolher corretamente os blocos com maior probabilidade de serem executados foi proposto o *Singly Rooted Directed Acyclic Graph* (SRDAG) [2] ou DAG singularmente enraizado. A idéia básica deste algoritmo é usar um SRDAG para compactar cada bloco do microprograma ao invés de usar e compactar caminhos inteiros. O bloco compactado em cada passo é a raiz do SRDAG. A raiz escolhida em cada iteração é o bloco não-compactado o qual possui a maior probabilidade de execução e, ao mesmo tempo, não possui predecessores não-compactados. Especificada a raiz, o SRDAG selecionado é simplesmente o SRDAG maximal de todos os blocos não-compactados.

A seguir, apresentamos um esquema simples do algoritmo:

- Enquanto houver bloco não-compactado: seleciona SRDAG, compacta raiz de SRDAG e atualiza probabilidades e DAG original.

No SRDAG, é priorizada a movimentação de operações que estão livres no topo de diversos blocos diretamente ligados ao bloco raiz. Assim, diversos caminhos de execução se beneficiam com a compactação. Muitas vezes, a melhor compactação é obtida com a criação de novos blocos.

Outra característica positiva do SRDAG é que a fase de *bookkeeping*, realizado no *Trace Scheduling*, não é necessária nesta abordagem (por construção). Isso facilita bastante a sua implementação.

SRDAG é uma generalização de *Trace Scheduling* que visa ao aumento de desempenho através do uso de um contexto maior de informação. Os resultados obtidos mostram que é possível melhorar o desempenho obtido com a técnica anterior em até 100%. Quando o SRDAG escolhido é um caminho e a compactação de blocos em ordem não afeta as probabilidades de execução, SRDAG se degenera em *Trace Scheduling*. Se um programa compactado com base no *trace* ocupa $O(n)$ de espaço, o mesmo compactado com SRDAG pode chegar a ocupar $O(sqrt(n))$.

4. TREE COMPACTION

Um dos problemas de *Trace Scheduling* é que o crescimento do consumo de memória devido a cópia extensiva de blocos durante o processo de *bookkeeping* pode ser enorme. Em casos específicos pode exponencial [3]. *Tree Compaction* ou Compactação de Árvore foi uma técnica proposta para aliviar este problema por [3]. A idéia básica desta técnica é realizar todas as compactações realizadas por *Trace Scheduling*, exceto àquelas que implicam cópia de blocos. Assim, pretende-se atingir quase o mesmo nível de compactação, porém com uma redução drástica no tamanho de micromemória necessária.

[3] introduz as seguintes definições de árvores:

- *top tree* - subgrafo conexo (do grafo de programa) o qual é em si uma árvore com um único nó com grau de **entrada** zero, a raiz da árvore.
- *bottom tree* - subgrafo conexo (do grafo de programa) o qual é em si uma árvore com um único nó com grau de **saída** zero, a raiz da árvore.

Segundo esta definição, um caminho é simultaneamente uma árvore *top* e *bottom*. O mesmo vale para um nó isolado.

Uma descrição básica do algoritmo:

1. blocos são particionados em subconjuntos de *top trees*;
2. *Trace Scheduling* é aplicado a cada árvore separadamente. Isto é, em cada árvore, um caminho é repetidamente selecionado e compactado usando *List Scheduling* com cópia de MOP's quando necessário;
3. blocos são particionados em *bottom trees*;
4. fazer 2 até que todos os blocos estejam compactados.

No pior caso, *Tree Compaction* apresenta crescimento de memória na ordem de n^2, onde n é o número de desvios condicionais. Lembrando que no pior caso de *Trace Scheduling* o crescimento é exponencial, este é um excelente resultado.

O desempenho em tempo de execução é quase igual (um pouco pior) ao obtido com *Trace Scheduling*. Entretanto, o consumo de memória é drasticamente menor e, se estas duas variáveis tiverem mesma importância, as vantagens de *Tree Compaction* tornam-se bastante evidentes. Outras vantagem é que esta técnica não afeta a estrutura topológica dos blocos. Isto é importante porque facilita a análise de erros (*debug*) no código otimizado. Normalmente, o código produzido por *Trace Scheduling* possui estrutura completamente diferente do programa original, dificultando bastante a análise de erros [3].

5. IMPROVED TRACE SCHEDULING COMPACTION (ITSC)

O ITSC [5] foi desenvolvido para reduzir os problemas de espaço requerido e tempo de execução de *Trace Scheduling* na presença de ciclos nos caminhos. O algoritmo ITSC é baseado em: um conjunto modificado de regras de movimentação de microinstruções entre blocos; um algoritmo de *trace scheduling* melhorado e um algoritmo especial de URCR (*unrolling*, *compacting*, e *rerolling*) para compactação de ciclos. Adotando as novas regras de movimentação é possível aumentar as chances de movimentar MOP's através das fronteiras de blocos básicos, simplificar o procedimento de *bookkeeping* e reduzir o número de cópias desnecessárias.

As regras modificadas de movimentação de MOP's entre blocos básicos são mostradas na Tabela 2.

O algoritmo melhorado de *Trace Scheduling* (ITS) pode ser divido em:

1. partição das MOP's de um caminho em dois conjuntos de acordo com a possibilidade de cada MOP gerar cópias desnecessárias ou não. Parte *MAIN* consiste de MOP's no caminho crítico do DAG e, a outra parte, é formada por desvios que não estão contidos no caminho crítico do DAG;

2. compactação com *List Scheduling* da parte *MAIN*;

3. arranjo de cada MOP remanescente de acordo com seu tipo e características no grafo de fluxo para prevenir movimentos ou cópias desnecessárias.

E o algoritmo URCR para compactação de ciclos tem como etapas: (a) desenrolamento do corpo do ciclo e criação de dois corpos; (b) compactação do ciclo desenrolado com *List Scheduling*; e (c) busca por um novo corpo de ciclo, exclusão de MOP's redundantes e enrolamento do ciclo.

Os resultados obtidos com a compactação de microcódigo usando ITSC mostram que: o tempo de execução do código otimizado é muito parecido com o de *Trace Scheduling*; a melhora percentual no uso de espaço varia de 9% a 24%; e a compactação de ciclos realizada pelo algoritmo URCR consegue fazer com que o código compactado seja comparável ao código compactado manualmente.

6. COMPARAÇÃO ENTRE OS ALGORITMOS

[4] fez experimentos para comparar algoritmos de compactação global de microcódigo. Todos os algoritmos estudados até aqui fazem parte do estudo. Os testes foram realizados em duas arquiteturas diferentes (PUMA e VAX) e os resultados comparados entre si e com as compactações local e manual (tida como ótima).

Os algoritmos apresentados apresentam relações de compromisso diferentes entre vantagens e custos quando comparados com *Trace Scheduling* (TS). Os dados obtidos não permitem conclusões definitivas sobre os métodos, uma vez que só duas arquiteturas foram testadas e os resultados são bastantes próximos.

Em média, *Tree Compaction* sempre apresentou tempo de execução pior e uso de memória melhor que TS. Por sua vez, dentre os algoritmos estudados é o que possui menor complexidade computacional. SRDAG consegue resultados, em média, sempre melhores que TS, tanto para tempo de execução quanto para uso de memória. ITSC apresenta o mesmo rendimento médio. Como desvantagem, ambos os métodos possuem complexidade maior que TS, sendo que o que mais se aproxima de TS é ITSC. Nos testes realizados, no máximo, os algoritmos igualaram o desempenho de otimizações manuais. Comparando os resultados dos algoritmos apresentados com compactação local, fica clara a superioridade de otimização global em relação à primeira.

7. CONCLUSÃO

Este trabalho inicialmente apresentou o método de *Trace Scheduling* para otimização global de microcódigo. Em seguida, variações de otimização global baseadas em TS foram abordadas, entre elas: SRDAG, *Tree Compaction* e ITSC. Uma comparação entre os algoritmos pôde ser estabelecida através do trabalho de [4]. Ficou evidente a superioridade de métodos de otimização global em relação a métodos que realizam otimização local de microprogramas. Uma sugestão dada por [4] na escolha do método apropriado de otimização global para uma arquitetura é comparar os resultados disponíveis de otimização local com os de otimização manual. Quanto maior for a diferença entre os resultados, maior é a chance que métodos globais mais complexos ofereçam resultados melhores que métodos globais mais simples.

8. REFERÊNCIAS

[1] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *Instruction-level parallel processors*, pages 478–490, 1981.

[2] J. Lah and D. E. Atkins. Tree compaction of microprograms. *SIGMICRO Newsletters*, 14(4):23–33, 1983.

[3] J. Linn. Srdag compaction – a generalization of trace scheduling to increase the use of global context information. In *The Proc. of 16th Annu. Workshop on Microprogramming*, pages 11–22. ACM Press, 1983.

[4] B. Su and S. Ding. Some experiments in global microcode compaction. In *MICRO 18: Proceedings of the 18th annual workshop on Microprogramming*, pages 175–180. ACM Press, 1985.

[5] B. Su, S. Ding, and L. Jin. An improvement of trace scheduling for global microcode compaction. In *Proc. of the 17th annual workshop on Microprogramming*, pages 78–85. ACM Press, 1984.

Regra	De	Para	Condições
R1	B2	B1 e B4	a MOP está livre no topo de B2
R2	B1 e B4	B2	cópias idênticas da MOP estão livres no final de ambos B1 e B4
R3	B2	B3 e B5	a MOP está livre no final de B2
R4	B3 e B5	B2	cópias idênticas da MOP estão livres nos topos de B3 e B5
R5	B2	B3 (ou B5)	para MOP's exceto de desvio: a MOP está livre no final de B2 e todos os registradores escritos pela MOP estão mortos em B5 (ou B3). Condições adicionais para MOP's de desvio: sua saída é a mesma do desvio B no fim de B2 ou Condição(MOP) E Condição(MOP B) = falso
R6	B3 (ou B5)	B2	a MOP está livre no topo de B3 (ou B5) e todos os registradores escritos pela MOP estão mortos em B5 (ou B3)
R7	B2	B1 (ou B4)	a MOP, exceto de desvio, está livre no topo de B2 e todos os registradores escritos pela MOP estão mortos abaixo da MOP B em B1 (ou B4)
R8	B1 (ou B4)	B2	a MOP, exceto de desvio, está livre no fim de B1 (ou B4) e todos os registradores escritos pela MOP estão mortos no topo de B2

Table 2: ITSC: Regras de movimentação de MOP entre blocos básicos referentes à Figura 1.

Execução Especulativa

Conceitos Gerais e Técnicas

Daniel Carlos Guimarães Pedronette – RA050269
Instituto de Computação, Universidade de Campinas
Cidade Universitária Zeferino Vaz
Campinas, São Paulo, Brasil
daniel.pedronette@students.ic.unicamp.br

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
Arquitetura de Computadores

Termos Gerais
Execução Especulativa

Palavras chave
Pipeline ,Depenências de Controle, Dependência de Dados

1. INTRODUÇÃO

O paralelismo a 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 a nível de instrução, ao qual está condicionado também o desempenho que pode ser alcançado pelo processamento de forma geral.

Pode haver diversos tipos de restrições, ou dependências. Uma das possíveis restrições 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 denominada como *functional hazard*. 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 pela qual instruções devem ser executadas: as dependências de dados e de controle. Frequentemente essa ordem limitam 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 ser 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 inclusive que instruções são executadas desnecessaria 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. A figura 1 ilustra essa propriedade comum dos softwares como três dimensões, aplicadas em técnicas importantes de execução especulativa.

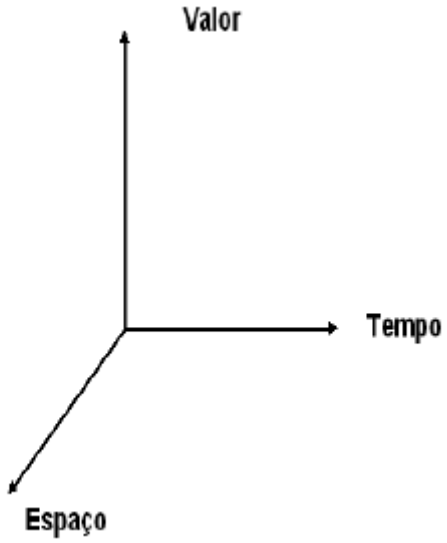


Figura 1. Três dimensões exploradas pela execução especulativa

2. CLASSIFICAÇÃO DAS TÉCNICAS

As técnicas de execução especulativa são comumente agrupadas e classificadas na literatura de acordo com o tipo de dependência que pretende solucionar. São listadas abaixo, e ilustradas na figura 2, as principais técnicas e as dependências relacionadas.

As Dependências de Controle consistem naquelas relacionadas ao fluxo de execução do programa, ou seja, na relação entre duas instruções de maneira que a execução de uma delas determina se a outra deverá ou não ser executada. Pode-se citar algumas técnicas de especulação de controle classificadas da seguinte forma:

- Branch Prediction: Técnicas que tentam predizer o fluxo de execução do programa após uma instrução de desvio com base na probabilidade desse ocorrer.
- Eager Execution: Nessa técnica todos os possíveis caminhos de execução são testados.
- Disjoint Eager Execution: Variação da técnica Eager Execution em que as restrições de disponibilidade de recursos são levadas em consideração para determinação de quais caminhos, selecionados de acordo com a probabilidade de serem tomados, serão executados.

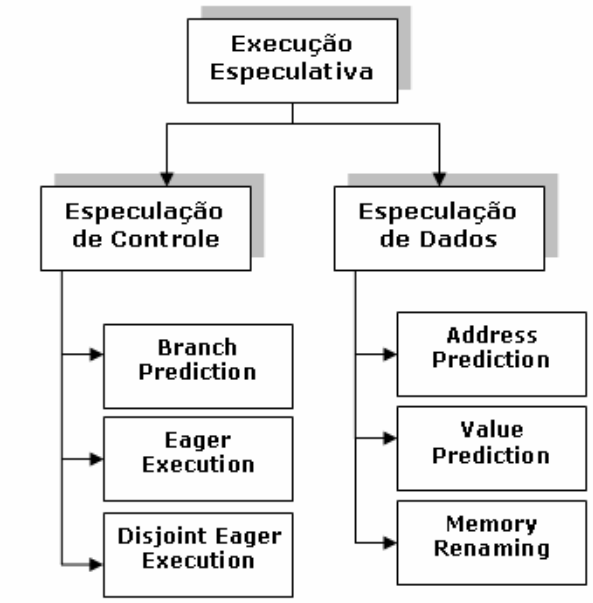


Figura 2. Classificação e taxonomia das técnicas de execução especulativa.

As Dependências de Dados ocorrem quando uma instrução depende de um dado que ainda não foi obtido ou calculado por outra instrução precedente. Principalmente em virtude da latência da instrução *load*, as dependências de dados consistem num importante gargalo para os processadores com

pipeline. As seguintes técnicas de execução especulativa procuram reduzir esse problema:

- Address Prediction: Técnicas que tentam predizer em qual posição de memória dados ou instruções estão armazenados.
- Value Prediction: Técnicas que procuram predizer qual o valor está armazenado em um registrador ou em um endereço de memória.
- Memory Renaming: Técnicas que procuram comunicar valores já armazenados para instruções *loads*.

3. ADDRESS PREDICTION

Todas as instruções *load* necessitam que seu endereço efetivo seja calculado, até que possam ser executadas. 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 sê-lo novamente no futuro, para tentar predizer os endereços e aumentar o desempenho na execução de programas. A figura 3, adaptada de Reiman e Carmen [1], procura ilustrar como a predição de endereços reduz o efeitos das dependências.

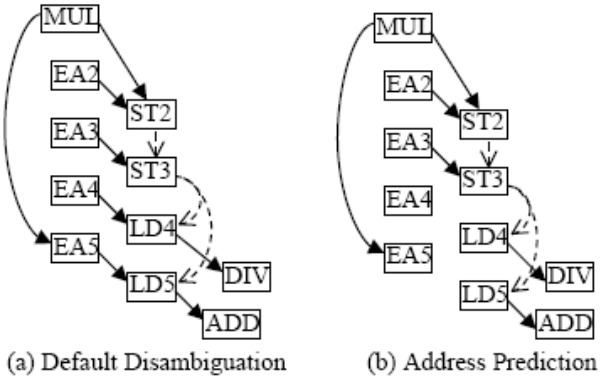


Figura 3. Figura (b) mostra como o LD5 pode ser executado mesmo antes que seu Efective Address tenha sido calculado

Um estudo sobre a possibilidade de predição de instruções *load* é apresentado em Gonzalez e Gonzalez [3] com bons resultados. Em geral, endereços utilizados por mesmas instruções de load ou store seguem uma progressão aritmética ou, em outras palavras, 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 chama da MAP (Memory Address Prediction). Essa estratégia consiste em determinar os endereços durante a decodificação das instruções através de uma tabela chamada Memory History Table (MHT).

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 da MHT 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. O estudo mostra uma avaliação do mecanismo utilizando o SPEC95, mostrando bons ganhos de desempenho.

4. VALUE PREDICTION

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.

No estudo apresentado por Lipasti e Shen [4], são listados 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.

No mesmo estudo, foram realizados realizados 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 4 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.

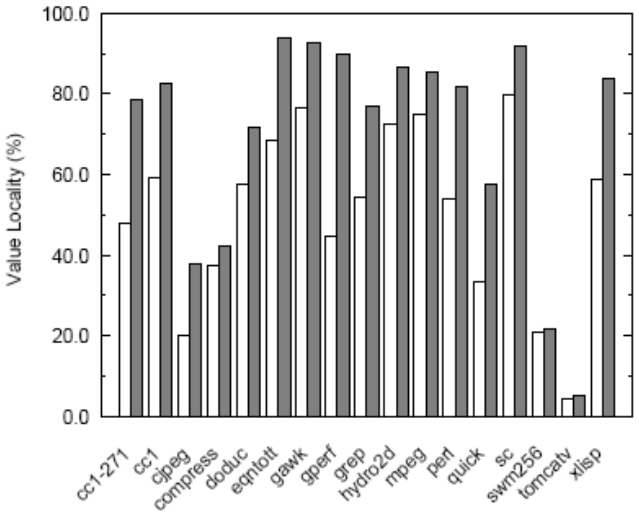


Figura 4: Localidade de valor em acessos à memória

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, swm 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, swm e tomcatv) apresentam pouca localidade.

A figura 5 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.

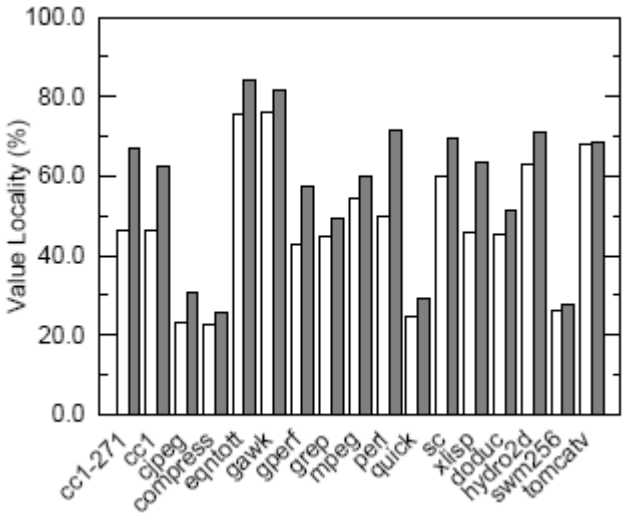


Figura 5. Localidade de valor em escritas a registradores

Para explorar a localidade de valor apresentado pela maioria dos programas, adicionando uma 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 uma mecanismo de predição de valores baseadas 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 6, extraída do citado trabalho.

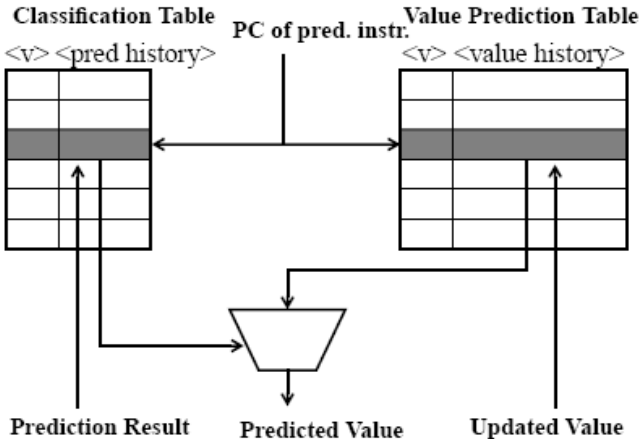


Figure 6: Unidade de Predição de Valores

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 preditível 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.

5. MEMORY RENAMING

A técnica de Memory Renaming consiste basicamente no processo de localização das dependências entre instruções *store* e *load*. Uma vez idenficadas essas dependências é possível realizar a previsão, comunicando o dado armazenado pela instrução *store* para a instrução *load* na sequência.

Em Tyson e Austin [5] é apresentada uma técnica para comunicação efetiva de memória, ilustrada na figura 7. A arquitetura é constituída por uma Store Cache (1) para armazenar instruções store recentes (4K entradas diretamente mapeadas), uma Store/Load Cache (2) para aramazentar dependências localizadas (4K entradas diretamente mapeadas) e uma Value File (3) para predição de valores. A técnica conta tamb´me com um mecanismo responsável por determinar quando usar a predição.

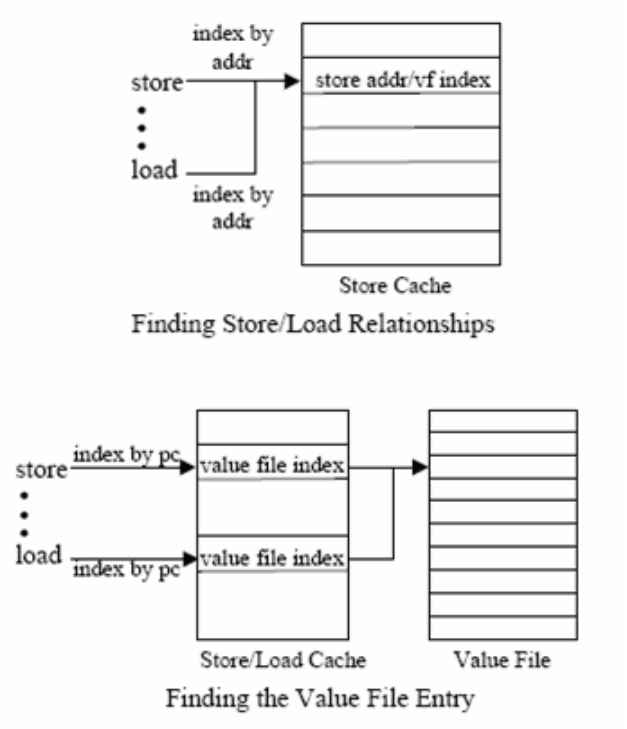


Figura 7. Técnica de predição de valores

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.

6. PREDIÇÕES INCORRETAS E EXECEÇÕES

Visando garantir a correta execução do programa as técnicas de execução especulativa devem funcionar de maneira que o resultado final produzido não seja alterado. Para tanto, deve existir mecanismos eficientes de recuperação para predições incorretas, além do devido tratamento de exceções. Ou seja, a ocorrência de uma exceção durante a execução de

código especulativo não pode alterar o estado do programa até que a verificação da predição seja feita.

Como descrito em [1], as predições incorretas podem ser corretamente são tratadas de duas formas:

- Squash Recovery: quando ocorre um predição incorreta, todas as instruções no reorder buffer são invalidadas
- Reexecução: realiza novamente a execução das instruções diretamente ou indiretamente dependentes da predição incorreta.

Há diversas soluções para o problema do tratamento de exceções durante execuções especulativas. A abordagem mais simples, como descrito em [2], consiste em não especular em instruções possíveis de gerarem exceções, conhecido como modelo de especulação restritivo.

7. 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 a 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.

Um trabalho futuro e mais extensor seria uma análise comparativa entre melhorias em desempenho obtida com aplicação de diferentes técnicas, avaliadas individualmente e através de possíveis combinações.

8. REFERÊNCIAS

[1] B. Calder and G. Reinman, "A Comparative Survey of Load Speculation Architectures," Journal of Instruction-Level Parallelism 1, 2000.

[2] A. R. Ganesh Lakshminarayana and N. K. Jha. Incorporating speculative execution into scheduling of control-flow-intensive designs. IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, pages 308–324, March 2000.

[3] J. Gonzalez and A. Gonzalez. Memory address prediction for data speculation. Technical report, Universitat Politecnica de Catalunya, 1996.

[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.

[5] 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.

Previsão de Desvios

Daniel Nicácio
Universidade Estadual de Campinas
E-mail: dnicacios@yahoo.com.br
RA: 057612

Resumo

Pipelines profundos e altas velocidades do clock exigem o desenvolvimento de previsores de desvios altamente precisos. Este artigo detalha a necessidade de previsores de desvios, apresenta seus princípios básicos e então reúne as principais técnicas desenvolvidas nos últimos anos. Estas técnicas englobam tanto esquemas estáticos quanto dinâmicos. Em cada uma delas é feita uma comparação de seu desempenho com as demais. Ao final, concluímos em que estágio de desenvolvimento os previsores de desvio estão e quais são os possíveis caminhos para sua evolução.

1. Introducão

Os *pipelines* dos processadores estão cada vez mais profundos e as taxas de *clock* cada vez maiores. Portanto, as penalidades devido a erros de previsão de desvios se tornam cada vez mais altas e mais prejudiciais ao sistema, se tornando o principal limitador para a evolução dos sistemas computacionais.

Desenvolver um processador é um exercício de compromisso entre desempenho, custo e uso de energia. Uma técnica que não requer este compromisso, ou seja, que provê um ganho para todas as três métricas é muito desejada. Uma dessas técnicas é previsão de desvios.

Com esses fatos em mente, é certo que para o desenvolvimento de um bom processador, e de uma arquitetura como um todo, a escolha de um bom método para a previsão de desvios é fundamental.

Este artigo começa explicando os principais fundamentos previsores de desvios e quais foram as primeiras técnicas utilizadas e como elas evoluíram até chegarem a previsores de desvios com taxas de erros menores do que 5%.

Em seguida é mostrado diversas pesquisas atuais e o que tem sido feito para que os previsores de desvios deixem d ser um fator limitante para a evolução da computação.

Este artigo está organizado da seguinte forma: seção 2 apresenta a motivação para o estudo de previsores. As seções 3,4 e 5 apresentam os tipos de técnicas existentes pra prever os resultados dos desvios. A Seção 6 contém diversos tópicos de pesquisa atuais e por fim, a seção 7 apresenta as conclusões e por fim, a seção 8 contém as referências bibliográficas.

2. Necessidade da previsão de desvios

Segundo [1], existem três tipos de perigos: estrutural, dados e de controle. Este último é gerado devido às dependências de controle que são freqüentemente encontradas nos programas. Em média, a cada cinco instruções, uma é operação de desvio. Dessa forma, quando queremos explorar o paralelismo do código, ou seja, emitir e executar múltiplas instruções por ciclo de *clock,* as operações de desvio rapidamente se tornam um fator limitante. De forma geral, não seria possível emitir mais de cinco instruções por ciclo, pois teríamos um desvio impedindo a emissão da sexta instrução. Portanto, eliminar as dependências de controle é crucial para emissão múltipla de instruções.

Com o intuito de evitar paradas devido às dependências de controle, existem diversas técnicas para a previsão de desvios, ou seja, técnicas para definir prematuramente o resultado de um desvio, evitando uma parada na execução do programa. A eficácia de um esquema de previsão é dada pela exatidão da previsão, o custo quando a previsão é correta e quando é incorreta. As penalidades de desvio dependem da estrutura do *pipeline,* do tipo de previsão e da estratégia usada para recuperar uma previsão incorreta.

As previsões podem ser estáticas ou dinâmicas. As previsões dinâmicas são realizadas pelo hardware durante a execução do programa, enquanto que a previsão estática de desvios permite otimizar o escalonamento das instruções. Portanto, os desvios devem ser previstos estaticamente quando o programa for compilado.

Na próxima seção serão apresentadas algumas técnicas básicas de previsão e a evolução das mesmas.

3. Técnicas de previsão dinâmica

3.1 Previsão de desvio básico e *buffers* de previsão de desvios.

O esquema mais simples é utilizar um buffer de previsão de desvios ou uma tabela de histórico de desvios. O buffer é uma pequena memória indexada pela porção mais baixa do endereço da instrução de desvio. Esta memória contém um bit informando se o desvio deve ser tomado ou não. O programa toma a direção indicada pelo bit, se mais tarde for detectado que a decisão estava errada o bit na memória é invertido.

Esta técnica possui o inconveniente de que dois desvios podem acessar o mesmo bit de previsão, ou seja, o resultado de um desvio pode influenciar na previsão de outro desvio, sendo que estes dois desvios deveriam ser independentes um do outro.

Visando evitar esta perda de exatidão, a memória passa a usar dois bits para predizer o resultado de um desvio. Dessa forma são necessárias duas previsões incorretas para que o resultado da previsão seja invertido.

Estudos mostram que utilizar mais de dois bits para a previsão não resulta em uma exatidão maior do que a apresentada com 2 bits.

3.2 Previsores de desvio com correlacionamento

Existem instruções de desvio com comportamento correlacionado a outras instruções de desvio. Dessa forma, é recomendado que um previsor de desvio analise as relações entre os desvios para um resultado mais exato. Esses previsores são chamados de previsores com correlacionamento.

De forma geral, é utilizado um previsor (m,n), ou seja, o previsor utiliza o comportamento dos últimos m desvios para escolher entre 2^m previsores de desvio, cada um dos quais é um previsor com n bits para um único desvio. Como já foi dito, n geralmente possui o valor igual a 2.

3.3 Previsores por torneio – combinação adaptativa de previsores locais e globais

A previsão por torneio considera o resultado dado por diferentes técnicas e determina qual destes é o mais apropriado para o desvio em questão. Isto é útil, pois alguns desvios dependem de informações locais, enquanto que outros são influenciados por informações globais.

3.4 Exemplo – Alpha 21264

Esta arquitetura utiliza 2 bits (indexados pelo endereço da instrução de desvio em questão) para selecionar entre o previsor global e o previsor local. O previsor global é indexado pelos últimos 12 desvios, e cada entrada deste previsor é um previsor padrão de 2 bits. O previsor local consiste em um previsor de dois níveis. O nível superior é composto por uma tabela de histórico local com 1024 entradas de 10 bits. Cada entrada corresponde ao resultado dos 10 desvios mais recentes. Esse método possibilita identificar padrões de até 10 desvios. A entrada selecionada na tabela de histórico local é utilizada para indexar um segundo nível formado por contadores de saturação de 3 bits, os quais fornecem a previsão local. No *benchmark* SPECfp95 a taxa de previsão incorreta chegou a ser menor do que 0,1%. Já no SPECint95, esta taxa ficou menor que 1,2%, ambas nos melhores casos.

4 Técnicas de previsão estática

Existem diversas técnicas para prever estaticamente o comportamento de um desvio. A mais simples delas é prever um desvio como seguido. Porém, a taxa de previsões incorretas deste método varia muito, entre 9% e 59%. Uma maneira de melhorar esta técnica é prever os desvios com base na orientação do desvio, onde os desvios com sentido inverso são definidos como seguidos e os com sentido direto como não-seguidos. Mas esta técnica ainda apresenta alta taxa de previsões incorretas, em torno de 30% a 40%.

Uma estratégia mais precisa é prever desvios de acordo com informações de perfil coletadas de execuções anteriores. Esta abordagem é vantajosa porque um desvio individual tem forte tendência a assumir um mesmo valor: seguido ou não-seguido. Com este tipo de previsão é possível manter a taxa de erros entre 5% e 22%

5 Uma técnica alternativa - instruções predicadas

Outra forma de evitar o perigo de controle causado pelos desvios é fazer com que a dependência de desvios não exista mais. Para isso, todas as instruções são executadas normalmente, inclusive os desvios, independentemente de quaisquer desvios. As instruções de desvios armazenam seus resultados em uma memória específica e as demais instruções armazenam seus resultados em registradores provisórios, os quais dependem de um predicado. Este predicado aponta para o endereço de memória no qual a instrução de desvio guarda seu resultado, quando este resultado estiver pronto, o registrador provisório saberá se seu resultado deve passar para o

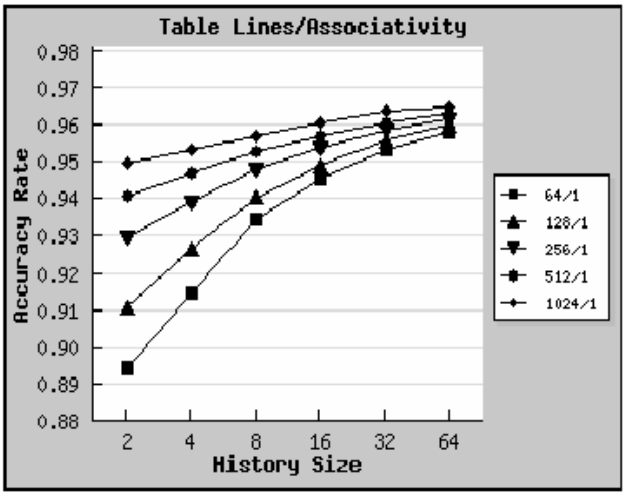


Figura 1 - TLPT Global: tabela X tamanho do histórico de desvios

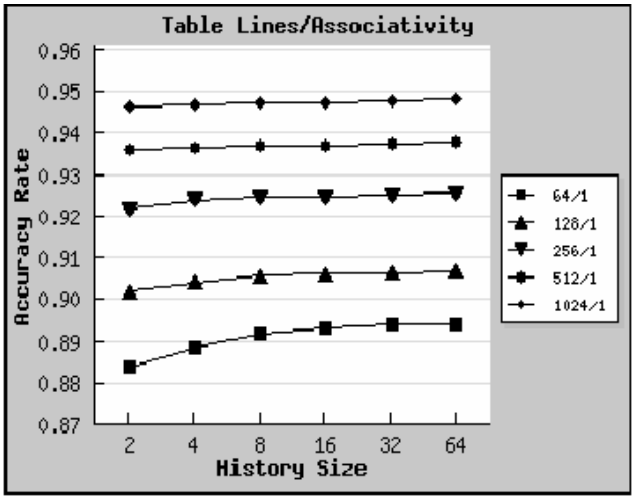


Figura 2 - TLPT Local: tabela X tamanho do histórico de desvios

registrador definitivo ou não.

Dessa forma, a dependência de controle passa a ser uma dependência de dados, a qual é mais fácil de ser evitada, tanto pelo hardware quanto pelo compilador.

6 Pesquisas recentes

6.1 O previsor 2bc-gskew

[9] relata que em uma tabela de histórico de desvios, uma interferência destrutiva ocorre quando dois desvios com resultados distintos apontam para a mesma entrada da tabela de histórico. Isto resulta no aumento da taxa de erros de previsão. Dessa forma, uma maneira de melhorar a previsão de desvios é diminuir as interferências destrutivas.

Em 1997, [2] propôs o uso de 3 bancos de historicos de desvios destintos uns dos outros, ou seja, cada um com a sua própria função de hash para indexar suas entradas. Um desse bancos é um previsor bimodal, sua função é alocar os desvio seguidos em uma tabela e os desvios não seguidos em outra tabela. Dessa forma, as interferências destrutivas diminuem drasticamente. Quando um desvio está sendo previsto, os três bancos são verificados e um "juiz" determina qual dos resultados deve ser utilizado. Esta técnica foi batizada de e-gskew.

Em 1999, [8] aprimorou seu trabalho e propôs o previsor hibrido 2bc-gskew, a idéia central deste previsor é combinar a o e-gskew com um previsor de dois bits correlacionados. O 2bc-gskew consiste de quatro bancos de contadores de 2 bits cada, sendo que um deles é o previsor bimodal.

6.2 Métodos neurais para previsão de desvios

Em Novembro de 2002, [3] apresentou um método bastante preciso para a previsão de desvios. A idéia central é utilizar o mais simples dos métodos neurais, o perceptron, como uma alternativa ao comumente usado contador de dois bits. O segredo da precisão deste método é o uso de longos históricos de desvios, isso é possível porque a quantidade de hardware necessário cresce de forma linear, e não exponencialmente, em relação ao tamanho do histórico.

Um perceptron é um dispositivo capaz de aprender e dado a ele um conjunto de valores e um conjunto de pesos (adquiridos através do aprendizado) ele é capaz de produzir um valor de saída. Neste método, cada peso representa o grau de correlacionamento entre o comportamento de um desvio passado e do desvio que está sendo previsto. Um peso positivo representa uma correlação positiva e um peso negativo apresenta uma correlação negativa.

A previsão do desvio acontece da seguinte forma: cada peso contribui na proporção de sua magnitude da seguinte maneira: se o desvio correspondente foi tomado, o peso é adicionado, senão, o peso é subtraído. Se o resultado da soma for positivo, o desvio é tomado, se for negativo, ele não é tomado. Para fazer esta solução funcionar, o histórico dos desvios usa 1 para desvios tomados e -1 para desvios não tomados. O perceptron é treinado por um algorítmo que incrementa o peso quando o resultado de um desvio concorda com o correlacionamento do peso, e decrementa o peso caso contrário.

Esse método apresentou uma taxa de erros de apenas 4,6%, o que significa uma melhora de 26%

em relação ao gshare e de 14% em relação ao previsor híbrido de McFarling (utilizado no Alpha 21264). A melhora obtida no IPC foi de 15,8% em relação ao gshare e de 5,7% em relação ao McFarling.

6.3 O uso de perceptrons em dois níveis

[4] propõem um modelo de predição de desvios utilizando um conjunto de perceptrons organizados em dois níveis. Este modelo, embasado em conceitos de redes neurais, é chamado de TLPT (Two-Level Perceptron Table). A primeira tabela do modelo possui a história dos desvios e é indexada pelo endereço do desvio. Nesta tabela, é possível identificar alguns padrões de desvios do programa. Cada entrada desta primeira tabela endereça uma entrada da segunda tabela, a qual possui os pesos sinápticos para o perceptron associado àquela historia de desvios. O modelo pode considerar a história global de desvios ou a história local.

As figuras 1 e 2 ilustram os resultados dos testes realizados.

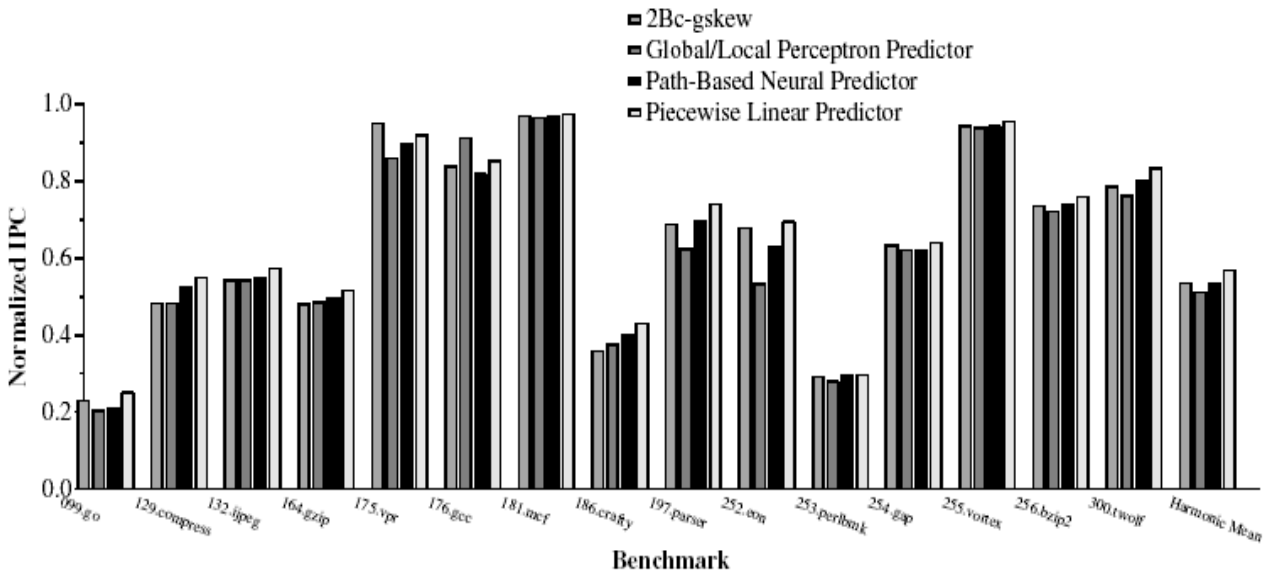


Figura 4 - IPC utilizando 256K

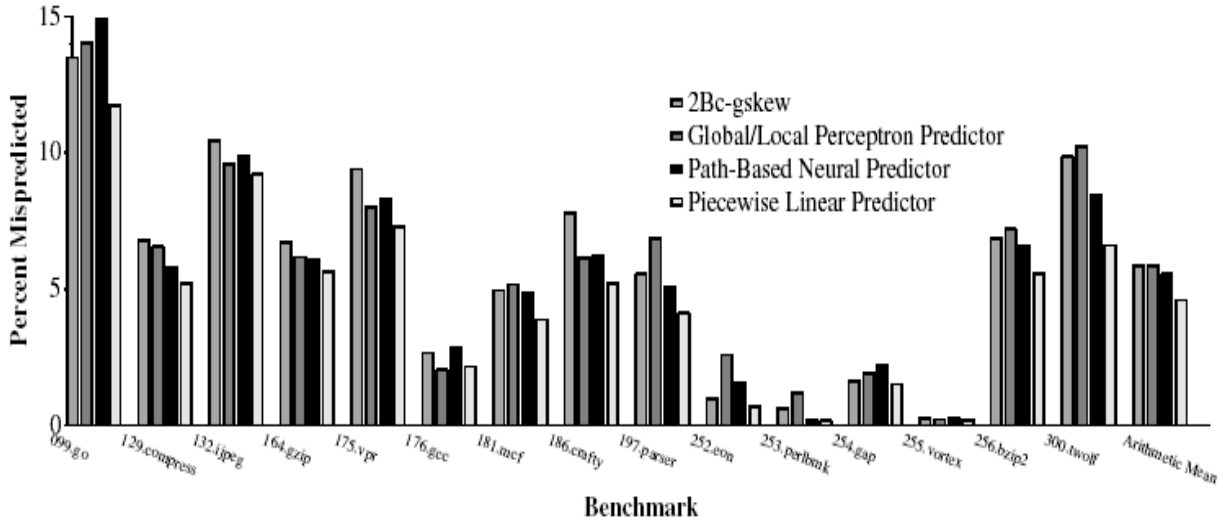


Figura 5 - Média da taxa de erros utilizando 256K

6.4 O previsor Piecewise

[5] apresentou um previsor de desvios que se baseia no aprendizado de um conjunto de funções lineares para cada desvio. Juntas, estas funções formam uma superfície plana com padrões de histórias de desvios bem definidas. Esta superfície permite identificar padrões de desvios que outras técnicas não conseguem. Por exemplo a função XOR, cujos resultados de ambas as técnicas podem ser vistos na figura 3.

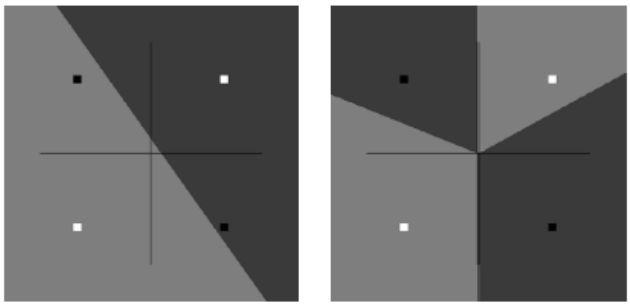


Figura 3 - A função XOR não é aprendida por um perceptron (a), mas pode ser aprendida utilizando uma superfície de decisão linear (b)

Para formar a superfície é preciso analisar todos os caminhos do programa que terminam no desvio B. Cada análise é uma função linear, e juntas elas formam a superfície. Portanto, para analisar um desvio especifico, basta ver qual o caminho utilizado para chegar neste desvio e em qual área da superfície a função deste caminho está.

Comparações entre este previsor, o 2Bc-gskew, Global/Local Perceptron e Path-Based Neural mostraram que o previsor piecewise possui a menor taxa de erros e também o maior numero de instruções executadas por ciclo. Estes resultados são vistos nas figuras 4 e 5. Trabalhos futuros visam diminuir a quantidade de hardware extra necessário para a implementação deste previsor.

6.5 Identicando desvios correlacionados através de um longo histórico global

Segundo [7], os previsores atuais possuem alguns estágios para chegarem a um resultado promissor: em um primeiro estagio, é feita a predição em apenas um ciclo, em seguida um previsor global apresenta um resultado mais eficaz e por fim um previsor ainda mais acurado corrige seletivamente o resultado dos últimos previsores. A estrutura do *pipeline* de um previsor corretivo é mostrada na figura 6.

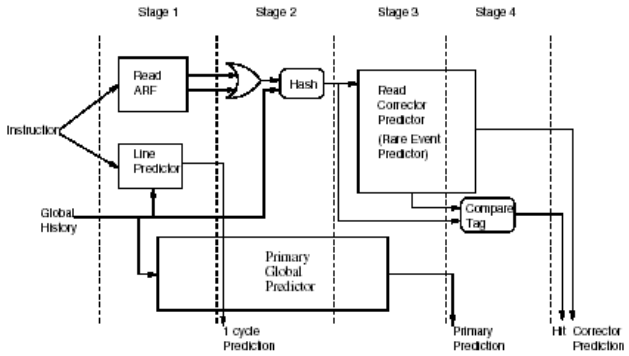


Figura 6 - Tempo de pipeline de um previsor corretor

Portanto, o último previsor deve possuir uma taxa de erros extremamente baixa. Baseando-se neste fato, foi proposta uma técnica baseada em longos históricos globais de desvios e em identificando desvios correlacionados neste histórico. Cada desvio que afeta o resultado de outro desvio é um *affector*.

Os *affectors* de cada desvio são identificados em tempo de execução, rastreando o fluxo de dados do programa no *front-end* do *pipeline*. O hardware necessário para identificar os *affectors* de 64 desvios requer apenas 312 bytes.

Dois esquemas são propostos:

Zeroing – Neste esquema, os desvios que não são *affectors* são mascarados, se tornam zeros, e portanto, não importam para a decisão do desvio. Isto pode ser feito utilizando a função AND tendo como operandos o *bitmap* dos *affectors* do desvio e o histórico do desvio. O resultado é então padronizado através de uma função de *hash* para que o previsor de desvios possa indexá-lo.

Packing – Neste segundo esquema os zeros que não afetam o resultado do previsor são retirados do resultado antes que a função de *hash* seja aplicada. Em outra palavras, os *affectors* são agrupados, e esse grupo é então formatado de forma que também possa ser indexado. A figura 7 ilustra os dois métodos.

Testes comparativos mostram que estas técnicas conseguem proporcionar uma boa melhora em previsores primários existentes (*Perceptron* e YAGS).

A figura 8 mostra o desempenho obtido por vários esquemas de previsores corretivos, e mostra também a taxa de erros obtida por eles. É importante ressaltar que com a adição de apenas 8K a um *perceptron* de 16K (total de 24K) é possível ter um desempenho comparável apenas a um *perceptron* de 256K o qual leva o dobro de ciclos para alcançar o resultado.

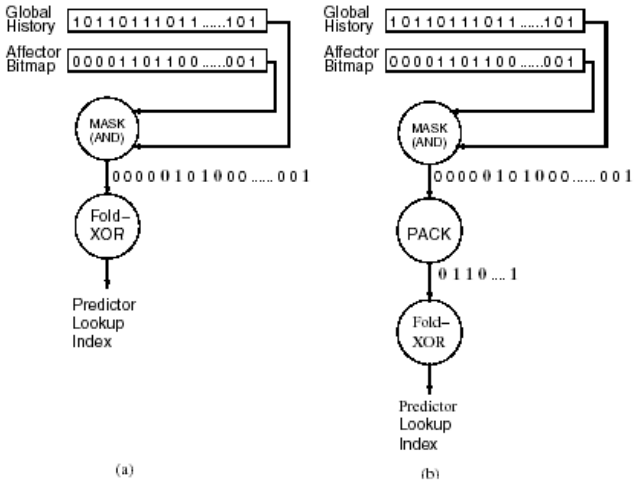


Figura 7 - Ilustração dos esquemas para utilizar informação dos "affectors" na previsão de desvios: (a) Zeroing; (b) Packing.

6.6 O previsor Profeta/Crítico (*Prophet/Critic*)

[6] introduziu uma técnica híbrida para melhorar a predição de desvios chamada de profeta/crítico (*prophet/critic*). Este esquema híbrido é composto por dois componentes, os quais desempenham o papel de profeta ou crítico. O profeta é um previsor tradicional que utiliza o histórico do desvio para predizer seu resultado. Desvios posteriores a este são considerados o futuro do desvio. O componente crítico utiliza tanto o histórico do desvio quanto o futuro do desvio para criticar se a decisão do profeta foi correta ou não.

Sendo assim, durante a execução do programa, o profeta provê um resultado para o desvio em questão; o programa segue sua execução normalmente, gerando outros resultados para novos desvios. Baseados nestes novos resultados (chamados de futuro do primeiro desvio) o crítico determina se o resultado dado ao primeiro desvio foi correto ou não. Se sim, o programa continua normalmente e o crítico continua criticando os resultados seguintes do profeta; se não, são executadas as devidas instruções para que o programa siga pelo caminho correto.

Resultados mostraram que um previsor 8K + 8K byte profeta/crítico possui taxa de erros 39% menor do que um previsor 2bc-gskew de 16K.

6 Adição de código para aumentar a precisão dos previsores de desvio

Como já foi dito, uma maneira de melhorar a previsão de desvios é diminuir as interferências destrutivas em uma tabela de histórico de desvios. Uma forma de alcançar este objetivo é particionar a tabela de histórico em duas regiões: uma com desvios que normalmente são seguidos e outra com desvios que normalmente não são seguidos. [9] propõem uma técnica em software (estática) para implementar este esquema.

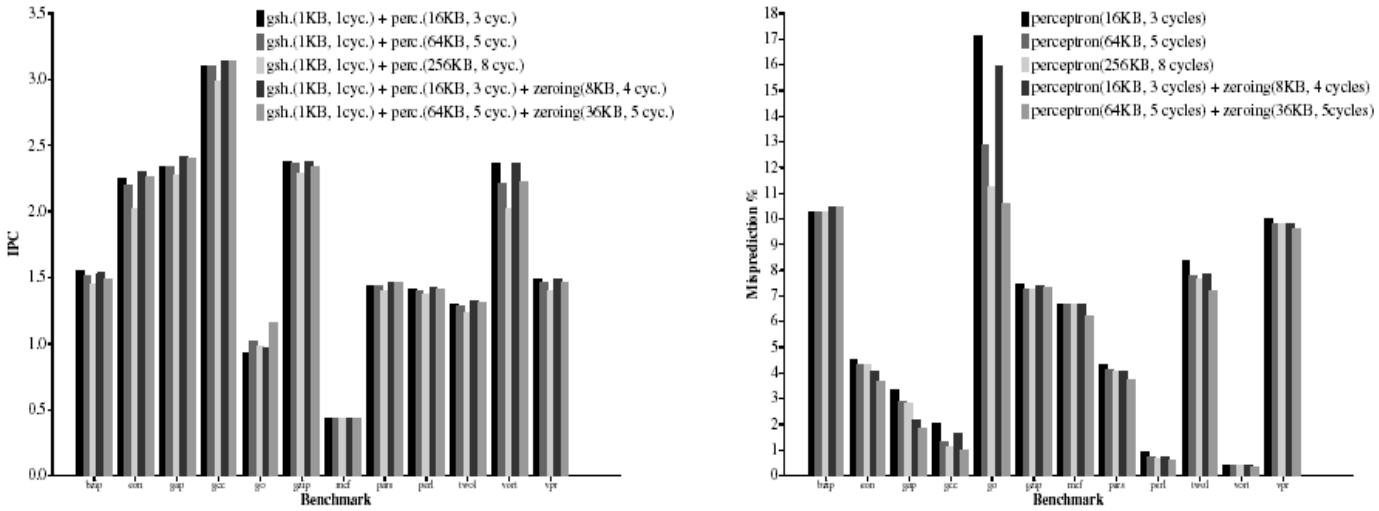


Figura 8

(i) performance de diversos previsores com correção; (ii) taxa de erros de diversos previsores com correção.

A idéia central é de que todo desvio utiliza, pelo menos, o seu bit menos significativo para indexar as entradas da tabela de histórico. Portanto, pode-se definir que o valor deste bit será sempre 0 para desvios não seguidos e 1 para desvios seguidos; dessa forma, a tabela fica particionada em duas regiões bem definidas: desvios seguidos e desvios não seguidos. Para que esses bits tenham o valor desejado, é preciso adicionar instruções não operacionais (no-ops) no código fonte, de maneira que os desvios adquiram endereços conforme a técnica enunciada anteriormente.

Para que não sejam incluídos muitos no-ops, e inevitavelmente diminuir a desempenho da máquina, é utilizada uma heurística que determina em quais regiões do código a inclusão de no-ops não afetará o fluxo de instruções da máquina.

Como geralmente é utilizado mais de um bit (pelo menos dois) para indexar a tabela de histórico de desvios, uma otimização natural desta técnica é dividir a tabela em quatro regiões ao invés de duas. Estas regiões são: desvios fortemente seguidos, desvios fortemente não seguidos, desvios fracamente seguidos e desvios fracamente não seguidos. O funcionamento da técnica é análogo ao uso de apenas dois bits.

Resultados mostram que esta técnica apresenta melhoria em relação a técnicas semelhantes. O particionamento da tabela de histórico em quatro regiões obteve, em média, uma melhora de 4,5% no *speedup* e uma taxa de erros 3,5% menor. Todos estes resultados são comparados ao uso da tabela de histórico de desvios simples.

7 Conclusões

Com o aumento da profundidade dos *pipelines* e da velocidade do clock das máquinas, fica clara a necessidade de se evitar ao máximo a penalidade gerada por previsões de desvios incorretas. Portanto, nos últimos dez anos muitas pesquisas se voltaram para esta área e muitos progressos foram obtidos. Hoje, existem previsores de desvios com taxas de erro menores do que 5%.

Diversas técnicas, tanto estáticas quanto dinâmicas, foram propostas e a maioria delas ainda apresentam boa perspectiva para sua evolução. È certo que, em um futuro próximo, teremos acesso a técnicas capazes de praticamente anular as penalidades causadas por desvios.

8 Referências

[1] Hennessy, Patterson. Arquitetura de Computadores – Uma abordagem quantitativa. 3ª Edição. Editora Campus. 2003

[2] P. Michaud, A. Seznec, e R. Uhlig. Trading conict and capacity aliasing in conditional branch predictors. In Proceedings of the 24th Annual International Symposium on Computer Architecture, June 1997. pages 292-303.

[3] D. A. Jiménez e C. Lin. Neural methods for dynamic branch prediction. ACM Transactions on Computer Systems, Volume 20 Issue 4. 2002. pages 369-397

[4] L. Ribas e R. Gonçalves. Evaluating branch prediction using two-level perceptron table. 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. Page(s):4 pp.

[5] D. A. Jiménez. Piecewise Linear Branch Prediction. Proceedings of the 32nd International Symposium on Computer Architecture. 2005. pages 12.

[6] A. Falcón, J. Stark, A. Ramirez, K. Lai e M. Valero. Prophet/Critic hybrid branch prediction. In Proceedings of the 31st annual international symposium on computer architecture. 2004. page 250.

[7] R. Thomas, M. Franklin, C. Wilkerson e J. Stark. Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History. Proceedings of the 30th annual international symposium on Computer architecture. Volume 31 Issue 2. 2003.

[8] A. Seznec e P. Michaud. Dealised hybrid branch predictors. Technical Report PI-1229, IRISA. 1999.

[9] D. A. Jiménez. Code Placement for Improving Dynamic Branch Prediction Accuracy. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation PLDI '05, Volume 40 Issue 6. 2005.

BlueGene/L

Douglas Gielo Quinellato
Unicamp
RA: 057615
ra057615@students.ic.unicamp.br

ABSTRACT
Este artigo descreve a arquitetura do supercomputador BlueGene/L. São descritos a sua estrutura de máquinas, sua rede de interconexão e o modelo de programação utilizado para o desevolvimento de aplicativos.

Categories and Subject Descriptors
B.0 [**Hardware**]: general

General Terms
Design

Keywords
BlueGene/L, supercomputer, MPI, torus network

1. INTRODUÇÃO
BlueGene/L é um supercomputador desenvolvido por uma parceria entre a IBM e o Lawrence Livermore National Laboratory. Ele é um supercomputador massivamente paralelo, composto de até 65.536 nós de processamento, com pico de performance de 360TeraFLOPS. Seu desenvolvimento foi feito tendo como meta um bom relacionamento cuto/benefício, e visando uma redução no consumo de energia. Para atingir estes objetivos, foi utilizado a tecnologia *System on a chip*, que integra processador, memória, cache, controlador de comunicações num único ASIC (Application Specific Integrated Circuit), permitindo a utilização de uma quantidade enorme de nós com uma capacidade relativamente modesta. É utilizado o modelo de programação por trocas de mensagens, através do padrão MPI.

Seu projeto tem como filosofia a utilização de processadores de baixo custo e de baixo consumo, com uma freqüência de operação próxima à de acesso à memória. Dessa forma, obtém-se uma economia de energia (aprox. 1MW) e de espaço (menos que 232 m²).

O BlueGene/L ocupa as duas primeiras posições na lista top500[5] de Novembro/2005. A primeira posição foi conseguida com o computador instalado no Lawrence Livermore National Laboratory, com 131072 processadores; a segunda posição foi conquistada com o BlueGene/L instalado no IBM Thomas J. Watson Research Center, com 40960 processadores.

2. ARQUITETURA[1]
O BlueGene/L é formado por vários nós, interconectados por 5 redes de interconexão, uma para comunicação geral em formato de toro e as outras com finalidades específicas. Estes nós são compostos de dois processadores IBM PowerPC, com memória de até 2GB e controladora de rede gigabit ethernet. Utilizando a tecnologia *System on a chip*, os nós possuem um *die size* de 11.1 mm². Cada processador é capaz de realizar 4 operações de ponto-flutuante, na forma de duas operações de multiplicação-adição. Geralmente um dos processadores é dedicado à recepção e envio de mensagens entre os nós. Neste caso, o desempenho máximo é de 180 TeraFLOPS.

O *packing* do sistema é mostrado na Figura 1. Cada *computing card* é composto de dois nós. 16 computing card podem ser colocados numa *board*, 16 boards podem ser colocados num *midplane* e até dois midplanes formam um *rack*, num total de até 1024 nós por rack.

Na figura também é mostrado a quantidade de operações em ponto flutuante (FLOPS) e a capacidade máxima de memória nos diferentes componentes.

Cada nó executa um pequeno kernel, responsável por tarefas básicas de comunicação e por prover funções de alto nível para código científico de alto desempenho. Para a compilação, análise de depuração, é necessário uma máquina externa. A comunicação com esta máquina externa é feita através de nós de I/O, que são nós dedicados a execução de funções de suporte, executadas pelo sistema operacional. Deve-se haver nós de I/O na proporção máxima de 1 para cada 8 de processamento, ficando esta normalmente na razão de 1 para 64.

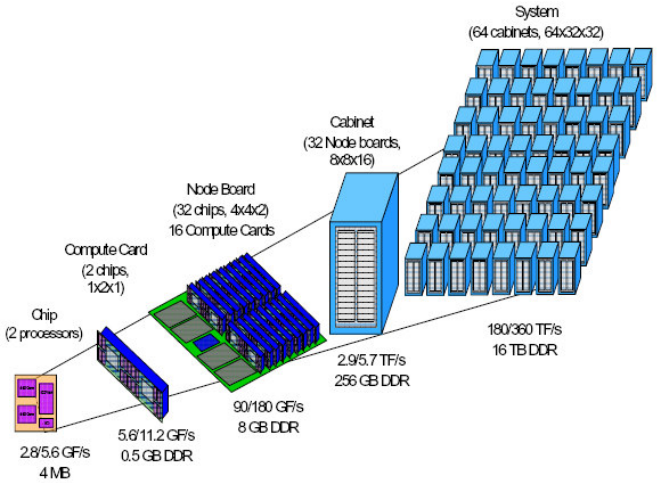


Figura 1. Packing do sistema

2.1 Nós de processamento

Um nó é formado por dois processadores PowerPC 440, com co-processadores PowerPC FP2. Este processador é um processador superscalar de 32 bits padrão. Por este motivo, as chaches L1 dos processadores do mesmo nó não mantém coerência.

Cada nó possui uma chache L2 de 2KB controlada por um mecanismo de prefetch de dados, uma memória SRAM rápida pra comunicação entre os nós, uma *L3 chache directory* e uma cache L3 compartilhada de 4MB. Além disso, também compõe um nó um controlador de ethernet gigabit, uma interface JTAG(Joint Test Action Group), uma interface para acesso a memória DDR e buffers de link de rede. A Figura 2 mostra o diagrama de blocos de um nó. Fecha a formação do nó uma *lockbox*, que provê mecanismos de *test and set* atômicos. As caches do nó são coerentes a partir da L2.

Uma forma comum de uso do nó é utilizar um núcleo para os cálculos do algoritmo e o outro para processamento de mensagens. Porém, caso o programa a ser executado necessite de pouca troca de mensagens, pode-se utilizar ambos para a execução do programa.

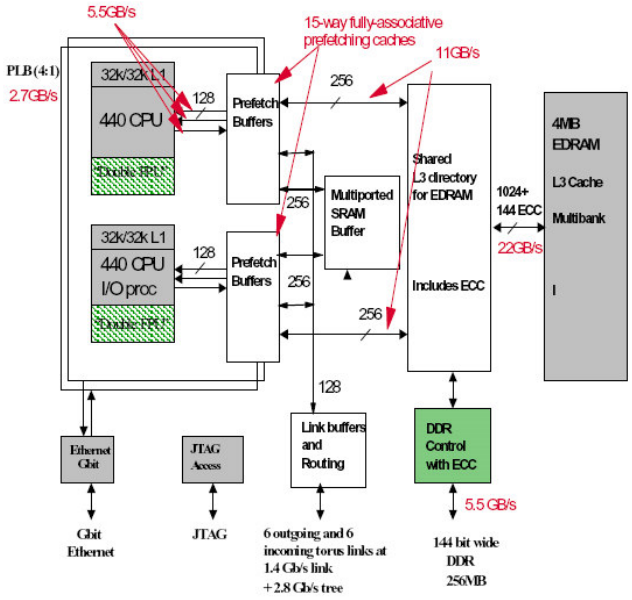


Figura 2. Diagrama de blocos de um nó

O núcleo FP2 é formada por duas unidades de ponto flutuante, cada uma com 32 registradores de 64-bit. As duas unidades são bastantes parecidas, mas há algumas diferenças. A primeria unidade age como uma unidade de ponto flutuante padrão; a segunda possui um conjunto de instruções SIMD. Algumas dessas instruções rodam paralelamente nas duas unidades. Algumas instruções rodam instruções diferentes nas unidades diferentes do FP2. Estes tipos de instruções foram denominadas de SIMOMD (Single Instruction Multiple Operation on Multiple Data).

O pipeline do processador possui latência de 5 ciclos para cada instrução, com exceção das instruçoes de divisão e *denormal*, entregando uma instrução por ciclo. As instruções de divisão

entregam o resultado iterativamente, 2 bits do quociente por ciclo.

O acesso à memória é feito de forma a maximizar a banda. O cache L2 retorna um dado em 6 a 10 ciclos, o L3 em aproximadamente 25 ciclos, e miss do L3 retorna o dado em 75 ciclos. Até 3 loads podem ser feitos em seqüência.

2.2 Redes de interconexão

O BlueGene/L possui 5 redes de interconexão, cada uma otimizada para um determinado tipo de conexão. Os tipos de redes são:

- Uma rede no formato de toro 3D, de dimensões 64x32x32;
- uma árvore de combinação/broadcast, para otimização de operações do tipo AllReduce();
- uma rede para barrier e interrupt;
- uma rede gigabit para conexão JTAG, para transmitir informações de controle e testes;
- uma rede gigabit para conexão com outros hosts.

A rede de toro é utilizada para comunicação geral entre os nós, e para operações de multicast. Ela é formada por links seriais ponto a ponto, entre roteadores embutidos nos nós. Cada roteador possui conexões com 6 vizinhos. A troca de mensagens é feita colocando as mensagens nas filas de saída. A mensagem então é enviada para a fila de entrada do próximo roteador, sendo tarefa de um dos processadores do nó o envio e a recepção destas mensagens. É implementado mecanismos de detecção de falha e de retransmissão, fornecendo guarantia de entrega.

O roteamento é feito de forma dinâmica, com *virtual cut-through* buffering[3]. Cada link possui 4 canais virtuais para troca de mensagens, mais alguns circuitos virtuais dinâmicos para troca de mensagens de roteamento e mensagens para garantir que o canal seja *deadlock free*. O roteamento é feito através do protocolo *Bubble router*[4], desenvolvido para determinar o roteamento em redes toro evitando deadlocks.

A rede em árvore é utilizada para comunicação de broadcast e de reduces. ALU's nesta rede povêem operaçoes aritméticas e bitwise para as operaçòes de reduçào em cada nó. Esta rede também é utilizada para comunicação com os nós de I/O.

3. Arquitetura de software[3]

A arquitetura de software do BG/L é composta port três entidades: um *núcleo computacional*, uma *infraestrutura de controle* e uma *infraestrutura de serviço*. Os nós de I/O rodam um kernel linux, fazendo parte da infraestutura de controle. O código de usuário é rodado nos nós que formam o núcleo computacional. Estes nós rodam um kernel linux miminalista, monoprocessado e monousuário, dedicado totalmente a aplicação sendo executada. A infraestutura de serviços é feita por programas comerciais aparte do núcleo, conectadas ao resto da arquitetura por uma rede ethernet.

O núcleo de computação pode ser particionado em conjuntos de nós isolados e idependentes, cada partição rodando um único processo.

O kernel do linux utilizado nos nós de controle necessitou de algumas modificações, para se adaptar ao mecanismos de acesso à memória, interrupção, boot e drives de dispositivo espec

3.1 Sistema operacional

O sistema operacional do BlueGene/L consiste de um kernel linux distribuído nos nós de I/O, e de pequenos kernels rodando nos nós de processamento. Este último é denominado *Blue Gene/L Run Time Supervisor* (BLRTS), e provê um espaço de endereçamento simples, sem paginaçao. Os recursos físicos são particionados entre o BLRTS e o processo do usuário. O acesso a rede fica no espaço de usuário, para aumentar a eficiência. A aplicação escreve diretamente neste espaço de memória a mensagem a ser enviada, através das funções de envio de pacotes.

O BLRTS implementa a interface POSIX. Foi portado a biblioteca GNU Glibc para prover suporte a chamadas básicas de sistema. As

chamadas de criação de processo e outras chamadas multiprocessos não foram implementadas, pois o kernel dos nós são monoprocessados. As funções de início e término de programas comunicam-se com os seus nós de I/O através da rede em árvore, usando modo de endereçamento ponto a ponto para as mensagens. Este kernel suporta a execução de processos com duas threads, sendo que cada thread é alocada para um processador, rodando exclusivamente nele.

O kernel linux rodando nos nós de I/O é um kernel padrão linux, com suporte a multiplos processos. A função destes processos é prover serviços adicionais aos nós, funcionando como o sistema operacional onde os processos estão rodando, provendo serviços como sockets de comunicação e acesso a sistemas de arquivos.

Além destes dois kernels, responsáveis pela execução das tarefas, existe um sistema operacional "global", responsável pela aplicação das políticas de funcionamento e pelo funcionamento geral do sistema, como inicialização, monitoração e início de processos. Ele roda como um serviço, numa parte em separado do sistema.

O estado do sistema é mantido em um banco de dados. Esta decisão foi feita porque banco de dados provêem confiabilidade, robustez e segurança, entre outros serviços. No banco de dados ão armazenados configurações estáticas, como conexões físicas entre nós, e dinâmicas, como o particionamento dos nós. Estas informações podem ser alteradas por processos externos, através de *stored procedures* e *triggers*, sendo um mecanismo de configuração.

3.1.1 Inicialização do sistema

Os nós não possuem disco rígido. A inicialização, portanto, deve ser feita pela rede. Ela é feita em dois passos: primeiro, um *boot loader* é escrito diretamente na memória através do protocolo JTAG. Este loader então carrega a imagem de boot efetiva do nó através do protocolo de mailbox, também pelo chip do JTAG.

É utilizado uma imagem para os nós de processamento e uma para os nós de I/O. A imagem dos nós de processamento possui tamanho de aproximadamente 64KB. A imagem dos nós de I/O é uma imagem linux de 2MB e um ramdisk com o sistema de arquivos raiz do nó. Sistemas de arquivos adicionais podem ser montados posteriormente. Como as imagens carregadas nos nós são iguais (comparadas com nós do mesmo tipo), é necessário carregar informações específicas para cada nó, como posição nas redes. Estas informações são chamadas de *personalidade(personality)* do nó.

3.1.2 Execução de jobs

A execução dos processos é iniciada pelos serviços externos. O usuário especifica o *job*(processo) a ser executado, e especifica a partição para a execução. O *scheduler* seleciona um grupo de nós e os configura para formar a partição desejada. Uma vez criada a partição, o processo é carregado nos nós de computação através dos seus respectivos nós de I/O.

3.2 Programação

O BlueGene/L utiliza a estrutura de programação do linux, utilizando o conjunto de ferramentas de compilação da GNU(binutils, gcc, gdb, etc). A compilação é feita externamente ao BlueGene/L, fazendo compilação cruzada. Existem dois alvos possíveis para a compilação: Linux para os nós de I/O, ou BLRTS para os nós de processamento.

Os programas feitos para o BlueGene/L utilizam o modelo de passagem de mensagens. Este modelo é implementado utilizando o padrão MPI. Este padrão provê um conjunto de funções para troca de mensagens entre processos.

A arquitetura de comunicação é dividida em três camadas: pacotes, mensagens e MPI.

A camada de pacotes provê mecanismos básicos para envio e recepção de pacotes. Este nível possui apenas três operações, *inicializar*, *enviar* e

receber. A inserção do pacote na fila de envio é abstraída, funcionando para as redes em toro e a árvore. Porém, fica a cargo do programador enviar pacotes com o tamanho máximo de 256 Bytes, e alinhar os dados em 16bytes. O envio e a recepção são não-bloqueantes.

A camada de mensagens provê mecanismos para envio de mensagens de formato arbritário pelo toro. Este nível possui as seguintes características:

- Sem mecanismos de retransmissão, pois a rede já possui mecanismos de retransmissão;
- Empacotamento e alinhamento dos dados, requerido pela camanda inferior. Os dados da mensagem são divididos em pacotes e enviados;
- Ordenação dos pacotes: uma mensagem pode ser dividida em vários pacotes, e estes pacotes podem chegar fora de ordem; esta camada reordena os pacotes antes de repassar à camada superior.

Para garantir uma melhor eficiência neste nível, é necessário utlizar um dos processadores do nó apenas com a função de lidar com as mensagens. Dessa forma obtém-se paralelismo entre a execução do código e tratamento de mensagens.

A camada de MPI é a camada utilizada em nível de usuário. Para suporte a essa camada foi portado uma implementação do MPICH2, com a adição de alguns módulos adicionais. Envio de Mensagens de broadcast é otimizado para envio pelo toro, enquanto mensagens de redução são enviadas pela árvore.

4. Conclusões

O supercomputador BlueGene/L é um computador massivamente paralelo formado por uma quantidade enorme de processadores de baixo custo/consumo, interligados por uma rede de interconexão em formato de toro. Programas desenvolvidos devem utilizar o modelo de passagem de mensagens, atravéz do padrão MPI.

5. Referências

[1] N.R. Adiga et al. An overview of the BlueGene/L supercomputer. *Proceeding of the IEEE/ACM SC2002 Conference* (2002).

[2] Almasi, Geroge et al. An overview of the BlueGene/L System Software Organization. *Proceedings of the Euro-Par – Springer*(2003) , 243-255.

[3] Blumrich, M. et al. Design and Analysis of the BlueGene/L Torus Interconnection Network. *IBM Research Report RC23025*. (Dez. 2003)

[4] V. Puente, C. Izu, R. Beivide, J.A. Gregorio, F. Vallejo et al. Adaptive bubble router. *Journal of Parallel and Distributed Computing* (2001). 58-67.

[5] November 2005 – Top 500 Supercomputing sites - http://www.top500.org/lists/2005/11 acesso em 22 jun. 2006

Comparativo entre Diferentes Estratégias de Multithreading

Fernando A. Kronbauer
Instituto de Computação – Universidade Estadual de Campinas
Av. Albert Einstein, 1251 – 13084-971 Campinas, SP
+55 (19) 3788-5842

fernando.kronbauer@students.ic.unicamp.br

RESUMO

Neste trabalho fazemos uma exposição de estratégias utilizadas na implementação de *multithreading* explícito em processadores atuais. É feita uma análise detalhada de duas tecnologias concorrentes, a tecnologia Hyper-Threading da Intel e o novo processador UltraSPARC T1 desenvolvido pela Sun Microsystems, expondo suas principais características, vantagens e desvantagens.

Categorias e Descritores de Assunto

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) – *Multiple-instruction-stream, multiple-data-stream processors (MIMD)*.

Termos Gerais

Performance, Design.

Palavras-Chave

Multithreading, Hyper-Threading, UltraSPARC T1.

1.INTRODUÇÃO

Este trabalho visa analisar e comparar diferentes estratégias usadas na implementação de *multithreading* explícito em processadores atuais. Para que um processador implemente *multithreading* é necessário que tenha a habilidade de executar duas ou mais *threads* de controle em um mesmo *pipeline*, isto é, o processador deve prover dois ou mais contadores de programa (PC) independentes, um sistema interno de rotulação para distinguir instruções de diferentes *threads* no *pipeline* e um mecanismo capaz de provocar trocas de contexto entre as *threads*.

Por limitação de espaço analisaremos apenas duas arquiteturas atuais de maior importância comercial, e ignoraremos arquiteturas de interesse histórico ou menos utilizadas. O excelente *survey* conduzido por Ungerer *et all* [1] aborda vários processadores *multithreaded* que não pudemos tratar.

Primeiramente faremos uma exposição detalhada da tecnologia Hyper-Threading desenvolvida pela Intel, passando a uma descrição da arquitetura do novo processador UltraSPARC T1 desenvolvido pela Sun Microsystems. A seguir faremos um comparativo das duas tecnologias, mostrando algumas de suas principais vantagens e desvantagens.

2.A TECNOLOGIA HYPER-THREADING DA INTEL

A tecnologia Hyper-Threading desenvolvida pela Intel propõe o uso de *Simultaneous Multithreading* (SMT) na família de processadores Xeon usado em servidores multiprocessados, e nos processadores Pentium 4 para uso em estações de trabalho. Esta tecnologia faz com que um único processador físico pareça como dois processadores lógicos que compartilhan recursos do núcleo de processamento. Cada processador lógico mantém um conjunto completo do estado da arquitetura, que consiste dos registradores de propósito geral, registradores de controle, registradores do controle avançado de interrupções programáveis (APIC), e alguns registradores de controle da máquina. Os processadores lógicos compartilham quase todos os outros recursos do processador físico, como as caches, unidades de execução e barramentos. Cada processador lógico possui seus próprios registradores APIC, e interrupções enviadas a um processador lógico específico são atendidas somente por este [1].

Quando um processador lógico está temporariamente bloqueado o outro pode continuar executando novas instruções. Um processador lógico pode ficar temporariamente bloqueado por várias razões, como faltas (*misses*) na cache, predições incorretas do alvo de instruções de desvio de controle, ou a espera pelo resultado de instruções anteriores. A execução contínua e independente de instruções dos dois processadores lógicos é assegurada através do gerenciamento das filas de espera de recursos compartilhados, de modo que nenhum processador lógico use todas as entradas de uma fila quando duas *threads* de *software* estão ativas no mesmo processador físico.

As filas de espera, ou *buffers*, que separam os principais blocos do *pipeline* são particionadas ou duplicadas para assegurar fluxo independente de cada bloco lógico. Se apenas uma *thread* de *software* está ativa, esta deverá executar no processador com tecnologia Hyper-Threading na mesma velocidade que executaria em um processador sem esta capacidade. Isto significa que recursos compartilhados devem ser recombinados quando apenas uma *thread* de *software* está ativa, aplicando uma política de compartilhamento flexível.

No processador Xeon, bem como no Pentium 4, instruções geralmente vêm da *trace cache* (TC), que substitui a cache de instruções primária (IL1). Dois conjuntos de ponteiros de instrução acompanham o progresso das duas *threads* de *software* independentemente e as entradas da TC são rotuladas com a informação da *thread* a que pertencem. Os dois processadores lógicos disputam o acesso à TC, e se ambos desejam acesso ao mesmo tempo, este é dado de maneira exclusiva a cada um a cada ciclo. Se um processador lógico está bloqueado e

incapacitado a utilizar a TC, o outro pode usar a largura de banda total de busca de instruções durante ciclos consecutivos.

Se houver uma falta (*miss*) na TC é necessário buscar os bytes de instrução e decodificá-los em micro operações (*µ*ops) a serem colocadas na TC. Cada processador lógico possui sua própia tabela de conversão de endereços virtuais de intruções e conjunto de ponteiros para acompanhar a busca das instruções. A lógica responsável bela busca de instruções na cache secundária (L2) arbitra os acessos visando servir as requisições que chegam primeiro, mantendo no entanto a capacidade de cada processador lógico possuir ao menos uma requisição pendente. Desta forma ambos os processadores lógicos pode ter requisições pendentes simultaneamente. Cada processador lógico possui seu próprio conjunto de dois *buffers* de 64 bytes para guardar os bytes de instrução em preparação para o estágio de decodificação.

As estruturas de predição de desvios de controle podem ser duplicadas ou compartilhadas. O *buffer* de histórico de predição de desvios usado para indexar o histórico global é mantido separadamente para cada processador lógico. No entando o vetor histórico global, uma estrutura bem maior que o histórico de predição de desvios, é compartilhado e suas entradas são rotuladas com a informação do processador lógico a que pertencem. A pilha de predição de retorno de função também é duplicada para os dois processadores lógicos.

Quando ambas as *threads* estão decodificando instruções simultaneamente, os *buffers* de entrada do estágio de codificação são alternados entre as *threads* de maneira que ambas compartilham a lógica de decodificação. A lógica de decodificação precisa manter cópias de toda a informação necessária para decodificar instruções IA-32 para os dois processadores lógicos, apesar do fato de decodificar instruções de um processador de cada vez. De maneira geral, várias instruções são decodificadas para um processador lógico antes de ser iniciada a decodificação de instruções para o outro.

A fila de *µ*ops que desacopla a parte inicial do pipeline da parte de execução fora-de-ordem é particionada de forma que cada processador lógico possua metade das entradas. A lógica de alocação de recursos retira as *µ*ops da fila e aloca vários dos *buffers* necessários para executar cada *µ*op, incluindo as 126 entradas do *buffer* de reordenação, os 128 registradores de inteiros e 128 registradores de ponto flutuante físicos, e os 48 *buffers* para operações *load* e 24 *buffers* para operações *store*. Se houver *µ*ops de ambos os processadores lógicos na fila, o alocador irá alternar a seleção de *µ*ops de cada um a cada ciclo e atribuir recursos. Cada processador lógico pode usar até no máximo 63 *buffers* de reordenação, 24 *buffers* de *load* e 12 *buffers* de *store*. Uma vez que cada processador lógico precisa manter informações sobre seu estado arquitetural completo, as tabelas de *alias* de registradores usadas para fazer a renomeação de registradores são duplicadas. A renomeação de registradores é feita em paralelo com a lógica de alocação de recursos, de forma que trabalha sobre as mesmas *µ*ops.

Uma vez que as *µ*ops passaram pelo processo de alocação de recursos e renomeação, são postas na fila de instruções de memória ou na fila de instruções gerais. Os dois conjuntos de filas também são particionados de forma que *µ*ops de cada um dos processadores lógicos possam ocupar apenas metade das entradas. As filas de instruções de memória e instruções gerais enviam *µ*ops para as cinco filas de escalonamento o mais rápido que conseguem, alternando *µ*ops de cada processador lógico a cada ciclo conforme necessário.

Cada escalonador de instruções possui sua própria fila com 8 a 12 entradas a partir da qual envia *µ*ops para as unidades de execução. Os escalonadores selecionam *µ*ops sem considerar a que processador lógico pertencem. As *µ*ops são avaliadas com base apenas nos valores de entrada pendentes e nos recursos de execução disponíveis. Por exemplo, o escalonador poderia despachar duas *µ*ops de um processador lógico e duas *µ*ops do outro no mesmo ciclo. Para evitar *deadlocks* e garantir a distribuição justa de recursos, há um limite para o número de entradas ativas que um processador lógico pode ter na fila de cada escalonador. Após a execução, as *µ*ops são postas no *buffer* de reordenação, que desacopla o estágio de execução do estágio de confirmação das instruções. O *buffer* de reordenação é particionado de maneira que cada processador lógico utilize no máximo metade de suas entradas.

A lógica de confirmação das instruções (*commit*, ou *retirement*) acompanha quando *µ*ops dos dois processadores lógicos estão prontos para serem confirmadas, e o faz na ordem que as instruções aparecem no programa alternando entre os processadores lógicos a cada ciclo. Se um processador lógico não possui nunhuma *µ*op pronta para confirmação, então a largura de banda total é utilizada para confirmar *µ*ops do outro processador lógico.

A implementação da tecnologia Hyper-Threading nos processadores Xeon e Pentium 4 da Intel adiciona menos de 5% ao tamanho do chip e potência consumida máxima, trazendo melhoria de performance de 20% ou mais para alguns tipos de carga de trabalho, comparando com uma versão do mesmo processador sem Hyper-Threading [2]. No entanto, o fato de o nível primário da cache de dados (DL1), a TC e a cache L2 serem compartilhados entre os processadores lógicos pode representar um problema quando *threads* que possuem uma área de trabalho bastante grande interferem umas com as outras ao serem escalonadas no mesmo processador físico, levando a uma alta freqüência de *misses* de conflito e eventual perda de performance. Fornecedores de *software* às vezes recomendam a seus clientes desabilitar a tecnologia quando utilizam determinado produto ou combinação de produtos de *software* em um mesmo servidor [3]. A perda de performance pode ocorrer mesmo quando *threads* não possuem espaço de trabalho grande, porém as pilhas de variáveis das chamadas de função interferem demasiadamente. Uma técnica que pode amenizar esta potencial perda de performance é a realocação das pilhas das *threads* para um *off-set* diferente na memória para que interfiram o mínimo possível [4].

3.NIAGARA

"Niagara" é o nome-código do processador UltraSPARC T1, desenvolvido pela Sun Microsystems, e encontrado em suas linhas de servidores Sun Fire T1000 e Sun Fire T2000 . Niagara é uma aposta arquitetural que visa aumentar o *throughput* de aplicativos de servidores comerciais envolvendo o aumento dramático do número de *threads* suportadas pelo processador, juntamente a um sistema de memória de alta largura de banda.

Niagara possui suporte em *hardware* para a execução de 32 *threads*. A arquitetura organiza cada quatro *threads* em um grupo, que por sua vez compartilha um *pipeline* simples de processamento. O processador Niagara utiliza oito destes grupos, resultando em 32 *threads* em uma CPU. Cada *pipeline* possui caches de nível primário (L1) para dados e instruções. O *hardware* esconde os bloqueios (*stalls)* estruturais e de memória

de uma determinada *thread* escalonando as demais do mesmo grupo para execução sem penalidade de troca de contexto.

As 32 *threads* compartilham uma cache de segundo nível de 3 MB, organizada em 4 bancos com dados intercalados, e os acessos são feitos em *pipelining* para aumentar a largura de banda. A cache de segundo nível possui associatividade 12-*way* para diminuir os *misses* de conflito entre as várias *threads*. A organização da hierarquia de memória do Niagara leva em conta que o código de programas para servidores comerciais possui em geral alto nível compartilhamento de dados, o que pode levar a uma freqüência elevada de *misses* de coerência. Em sistemas multiprocessados convencionais usando processadores discretos, *misses* de coerência seguem para barramentos de baixa freqüência fora dos processadores, e portanto podem apresentar altas latências. A cache compartilhada elimina estes *misses* fora do processador e os substitui com comunicação de baixa latência com a cache de segundo nível compartilhada entre os núcleos de processamento. Esta organização de memória, no entanto, limita o número de processadores a apenas um em sistema multiprocessado de memória compartilhada, pois não existe lógica para garantir a coerência entre caches de diferentes processadores em um mesmo sistema.

O barramento de conexão transversal (*crossbar*) provê a comunicação entre os *pipelines*, bancos de caches L2 e outros recursos compartilhados na CPU, a uma taxa de transmissão superior a 200 GB/s. Uma fila está disponível para cada par de fonte-destino, e é capaz de enfileirar até 96 transações em cada direção no barramento. O *crossbar* também provê uma porta para comunicação com o subsistema de I/O. A arbitragem para as portas de destino utiliza um esquema simples de prioridade baseado em envelhecimento (quanto mais tempo uma requisição aguarda, maior a sua prioridade), assegurando o escalonamento justo a todos os requisitantes. O *crossbar* é o ponto onde os acessos à memória são ordenados. A interface com a memória principal é composta por quatro canais DDR2 DRAM, suportando largura de banda máxima de 25.6 GB/s e capacidade máxima de endereçamento de até 128 GB [5].

A cache primária de instruções (IL1) tem 16 kB de capacidade e possui associatividade 4-*way*, com bloco de 32 bytes. A política de reposição de blocos é randômica por economia de espaço, sem no entanto acarretar perda de performance significativa. A cache de instruções é capaz de fornecer duas instruções a cada ciclo. Se a segunda instrução for útil, a cache possui tempo livre no ciclo seguinte para realizar o preenchimento de uma linha de cache sem bloquear o *pipeline*. A cache primária de dados (DL1) possui 8 kB de capacidade, associatividade 4-*way*, blocos de 16 bytes, e implementa uma política *write-through*. Apesar de as caches L1 serem pequenas, elas reduzem o tempo médio de acesso a memória por *thread* significativamente, com tachas de *miss* de cerca de 10 porcento. Isto se deve ao fato de aplicações comerciais em servidores possuirem em geral um espaço de trabalho bastante grande, e caches L1 precisarem ser muito maiores para atingir taxas de *miss* significativamente menores. Por outro lado, as quatro *threads* no mesmo grupo são capazes de esconder as latências umas das outras resultantes de *misses* nas caches L1 e L2. Portanto, as caches são dimensionadas tendo em vista a economia de espaço e tempo de acesso, tentando equilibrar as taxas de *miss* e a capacidade que as *threads* possuem para esconder as latências umas das outras.

A implementação de cada pipeline do Niagara suporta até quatro *threads*. Cada *thread* possui um conjunto único de registradores, *buffers* de instrução e *buffers* de operações de *store*. O grupo de *threads* compartilha as caches L1, tabelas de conversão de endereços virtuais (*translation look-aside buffer* - TLB), unidades de execução, e a maioria dos registradores do *pipeline*. Apenas uma unidade de cáculo de números de ponto flutuante é compartilhada entre todas as *trheads* do processador, o que significa que programas que fazem uso pesado deste tipo de operação não terão desempenho satisfatório no Niagara. Uma unidade de aceleração de criptografia está disponível para cada pipeline para auxiliar no processamento de *Secure Socket Layer* (SSL). O pipeline implementado é capaz de despachar apenas uma instrução a cada ciclo e possui seis estágios: *fetch*, *thread select*, *decode*, *execute*, *memory* e *write back*.

No estágio *fetch*, a cache de instruções e a TLB de instruções (ITLB) são acessadas. A seguir, a associatividade é escolhida. O caminho crítico é determinado pelo acesso à ITLB, que possui 64 entradas e associatividade completa. Um multiplexador determina de qual dos quatro PCs a instrução deve ser buscada. O pipeline busca duas instruções a cada ciclo. Um bit de precodificação na linha de cache indica se as instruções são de longa latência.

No estágio *thread-select*, um multiplexador seleciona uma *thread* do conjunto de *threads* disponíveis e despacha uma instrução para os estágios seguintes. Este estágio também é responsável por manter os *buffers* de instruções. Instruções adquiridas durante o estágio *fetch* podem ser inseridas no *buffer* de instruções daquela *thread* caso os estágios seguintes do *pipeline* não estiverem disponíveis. Os registradores do *pipeline* para os dois primeiros estágios são replicados para cada *thread*.

Instruções da *thread* selecionada prosseguem para o estágio *decode*, que realiza a decodificação da instrução e acesso ao banco de registradores. As unidades de execução suportadas incluem uma unidade lógico-aritmética (ALU), deslocador (*shifter*), multiplicador e divisor. Um mecanismo de repasse está presente no *pipeline* para repassar resultados a instruções pendentes antes que o banco de registradores possa ser atualizado. Operações lógico-aritméticas e de deslocamento possuem latência de um ciclo e geram resultados no estágio *execute*. Instruções de multiplicação e divisão são de longa latência e causam a troca de contexto para outra *thread*.

A unidade de *load-store* contém a TLB de dados (DTLB), a DL1 e os *buffers* de *store*. O acesso à DTLB e à DL1 acontecem durante o estágio *memory*. Como no estágio *fetch*, o caminho crítico é determinado pelo acesso à DTLB, totalmente associativa em suas 64 entradas. A unidade *load-store* possui quatro *buffers* de 8 entradas, uma para cada *thread*. *Hazards* do tipo *read after write* (RAW) na memória podem ser detectados através da checagem dos valores contidos nestes *buffers*. Os *buffers* de *store* possuem suporte para repassar valores a instruções *load* e resolver estes *hazards* RAW. Os endereços de destinos de operações *store* contidas no *buffer* são checados após o acesso ao TLB na parte inicial do estágio *write back*. Dados de operações *load* estão disponíveis para instruções dependentes na parte final do estágio *write back*, e instruções com latência de apenas um ciclo atualizam o banco de registradores neste estágio.

A lógica de seleção de *threads* decide qual *thread* estará ativa durante os estágios *fetch* e *thread select*. Portanto, se o estágio *thread select* escolhe uma *thread* para despachar uma instrução para o estágio *decode*, o estágio *fetch* seleciona a mesma para

Tabela 1. Comparativo entre as características do Pentium Extreme Edition e do Niagara

Característica	**Pentium Extreme Edition**	**Niagara**
Freqüência de *clock*	3.2 Ghz	1.2 Ghz
Profundidade do *pipeline*	31 estágios	6 estágios
Potência consumida	130 W (@ 1.3 V)	72 W (@ 1.3 V)
Tamanho do *die*	206 mm^2	379 mm^2
Número de transistores	230 milhões	279 milhões
Número de núcleos de processamento	2	8
Número de *threads*	4	32
Cache IL1	12 kμop (8-*way*)	16 kB (4-*way*)
Cache DL1	16 kB (4-*way*)	8 kB (4-*way*)
Latência de instrução *load* (IL1)	1.1 ns	2.5 ns
Cache L2	Duas cópias de 1MB (8-*way*)	3 MB (12-*way*)
Latência de instrução *load* (L2)	7.5 ns	19 ns
Largura de banda da cache L2	~180 GB/s	76.8 GB/s
Latência de instrução *load (Memória Principal)*	80 ns	90 ns
Largura de banda da Memória Principal	6.4 GB/s	25.6 GB/s

acessar a cache. A lógica de seleção de *threads* utiliza informação adquirida de vários estágios do *pipeline* para decidir quando selecionar ou ignorar uma *thread*. Por exemplo, o estágio *thread select* pode determinar o tipo da instrução utilizando um bit precodificado na IL1, enquanto outras situações que levam a latência prolongada de uma instrução são detectáveis apenas mais tarde no *pipeline*. Portanto, o tipo da instrução pode fazer com que sua *thread* seja ignorada pelo mecanismo de seleção durante alguns ciclos seguintes, enquanto que uma excessão detectada durante o estágio *memory* causa a anulação de intruções da mesma *thread* em estágios iniciais e preempção da *thread* durante o processamento da excessão.

Para instruções com baixa latência, como as lógico-aritméticas, Niagara implementa o repasse de resultados a instruções dependentes para resolver dependências RAW. Instruções *load* possuem três ciclos de latência antes que seu resultado seja visível a instruções seguintes. Tais intruções de longa latência podem causar *hazards* no *pipeline*, e resolvê-los requer parar a *thread* correspondente até que a condição de *hazard* não exista mais. Portanto, após uma instrução *load* de uma determinada *thread* ser despachada, a próxima instrução da mesma *thread* deve esperar dois ciclos para então ser executada. Da mesma forma, quando uma instrução de desvio de controle é encontrada, o Niagara não tenta executar especulativamente instruções de qualquer dos alvos do desvio. O processador simplesmente ignora a *thread* correspondente enquanto o alvo do desvio é resolvido, e durante este tempo tenta despachar instruções não especulativas das outras *threads* do mesmo grupo.

Em um *pipeline* com *multithreading*, *threads* que competem por recursos compartilhados também estão sujeitas a *hazards* estruturais. Recursos como a ALU que possuem latência de apenas um ciclo não apresentam nenhum problema, porém a unidade de divisão, que possui *throughput* de menos de uma instrução por segundo, representa um problema de escalonamento. Neste caso, qualquer *thread* que precisa executar uma instrução de divisão precisa esperar até que a unidade correspondente esteja disponível. O escalonador garante a atribuição justa de recursos ao escalonar a *thread* que foi executada o menos recentemente. Quando a unidade de divisão estar ocupada, no entanto, outras *threads* podem utilizar outros recursos disponíveis, como a ALU ou unidade *load-store*.

A política de seleção de *threads* é a troca de contexto a cada ciclo, dando prioridade à *thread* executada o menos recentemente. As *threads* podem tornam-se indisponíveis por causa de instruções de longa latência como *loads*, desvios de controle, multiplicações, divisões e operações de ponto flutuante. Elas também podem tornar-se indisponíveis devido a *stalls* no *pipeline* como *misses* na cache, excessões e competição por recursos. O escalonador considera que os dados de operações *load* serão encontrados na cache e portanto pode despachar instruções dependentes de maneira especulativa. No entanto esta *thread* recebe prioridade menor para escalonamento se

Tabela 2. Relação de performance entre o Pentium Extreme Edition e o Niagara sob diferentes tipos de carga de trabalho

Aplicação	Pentium Extreme Edition	Niagara
Código paralelo	1	2-3
Código seqüencial	5-7	1
Performance/Watt Código paralelo	1	4-5
Performance/Watt Código seqüencial	3-4	1

comparada a outras *threads* que podem despachar instruções não especulativas.

Niagara usa um protocolo simples de coerência de caches. As caches L1 são *write-through*, sem alocação durante *stores*. As linhas das caches L1 podem estar no estado válido ou inválido. A cache L2 intercala seus dados em quatro bancos com granularidade de 64 bytes. A cache L2 mantém um diretório que armazena quais são as caches L1 que possuem cópias dos dados, em uma granularidade igual ao tamanho das linhas das caches L1. Uma instrução *load* cujo dado falha na cache L1 é entregue ao banco de origem dos dados na cache L2, juntamente com o código que identifica a cache requisitante. Neste banco de cache L2, o endereço que causou o *miss* é usado para acessar a linha e retorná-la para a cache L1, e o código da cache é usado para atualizar o diretório. Uma instrução de *store* subseqüente para a mesma linha, emitida da mesma ou de outra cache L1, irá causar a verificação do diretório e a invalidação das linhas em outras caches L1 que compartilhan os mesmos dados. As instruções de *store* não atualizam as caches locais enquanto a cache L2 não tiver sido atualizada. Durante este período, a operação de *store* pode passar dados para a mesma *thread* mas não para outras *threads*. Desta forma, uma operação de *store* adquire visibilidade global na cache L2. O *crossbar* estabelece a ordenação de acessos à memória entre acessos a um mesmo ou diferentes bancos e garante a entrega de transações às caches L1 na mesma ordem. A cache L2 segue uma política *copy-back*, escrevendo na memória linhas modificadas e descartando linhas não modificadas durante uma reposição. O acesso direto à memória por parte de diferentes dispositivos de I/O também são ordenados através da cache L2.

3.COMPARATIVO

Para que possa ser feita uma comparação justa entre o Niagara e um processador com Hyper-Threading, escolheremos um processador da Intel construído com tecnologia semelhante e destinado ao mesmo mercado de aplicações comerciais para o qual o Niagara foi desenvolvido. Laudon [6] faz tal comparativo, o qual reproduziremos aqui. É importante notar que Laudon é um dos engenheiro da Sun Microsystems responsável pela criação do Niagara, e portanto seus resultados podem ser tendenciosos. No entanto, os resultados por ele encontrados parecem coerentes, e o fato de terem sido publicados em um periódico respeitável corroboram com tal impressão. Logicamente, é factível a construção de *bentchmarks* que favoressam um produto ou outro. Resultados publicados pela Hewlett-Packard [7], por exemplo, comparam um sistema com processadores Opterom da AMD com o Sun Fire T2000, apresentando números diferentes dos de Loudon.

O estudo de caso analisa dois processadores construídos com a tecnologia de 90 nm. O processador com Hyper-Threading é o Pentium Extreme Edition, que contém um par de núcleos superescalares, 4-*issue* e execução fora de ordem. Os núcleos usam *pipelining* profundo (31 estágios) e operam na freqüência máxima de 3.2 GHz. A TC tem capacidade para 12 mil μops, a cache L1 de dados possui capacidade de 16 kB, e cada núcleo possui cache L2 própria de 1 MB *on-chip*. O chip conecta-se a um controlador de memória externo através de um barramento frontal de 800 MHz. Cada núcleo suporta Hyper-Threading. Uma comparação entre as características do Pentium Extreme Edition e o Niagara é feita na Tabela 1. Figuras 1 e 2 mostram fotografias dos *dies* de ambos os chips.

Figura 1. Fotografia do *die* do Pentium Extreme Edition

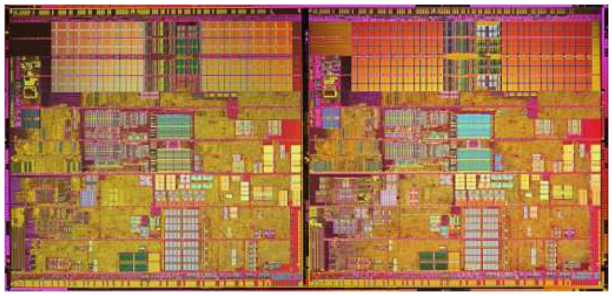
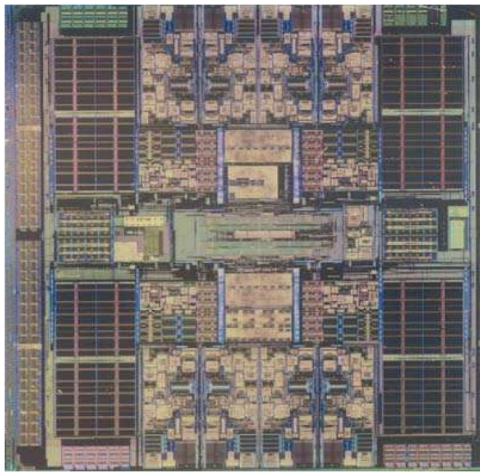


Figura 1. Fotografia do *die* do Niagara



Os *bentchmarks* selecionados foram o SPEC JBB 2000, TPC-C, TPC-W, e XML Test. SPEC JBB 2000 emula um sistema de três camadas enfatizando a performance da implementação regras de negócio empresariais em Java no servidor. TPC-C é um *bentchmark* de processamento de transações online. TPC-W é um

bentchmark que simula as atividades de um servidor de transações orientado a negócios pela Internet. XML Test é um teste de processamento de XML desenvolvido pela Sun Microsystems.

Pentium Extreme Edition possui área não maior que 55% a área do Niagara, porém com seu *pipelining* profundo consome 80% mais potência que este. Com apenas 4 *threads*, a performance em termos de *throughput* do Pentium Extreme edition é apenas uma fração da performance do Niagara em várias aplicações comerciais. Após fatorar o *throughput* mais elevado e menor consumo de potência, Niagara apresenta uma vantagem significativa em termos de performance/Watt, conforme a Tabela 2. Podemos também notar as diferentes ênfases no projeto dos dois processadores. As caches L1 e L2 são semelhantes, porém o Pentium Extreme Edition possui menor latência devido sua ênfase na performance de cada *thread* tomada separadamente. A latência da memória principal é parecida, mas o Niagara provê significativamente mais largura de banda para alimentar as 32 *threads*.

Os resultados obtidos são apresentados na Tebela 2, que relaciona a performance dos dois processadores. Claramente, a performance de *throughput* do Niagara é atingida sacrificando a performance de aplicativos de *thread* única (*single-threaded*). O Pentium Extreme Edition possui performance 5 a 7 vezes maior que o Niagara para os programas do SPEC CPU2000. No entanto, no âmbito comercial ao qual Niagara se destina a maior parte dos aplicativos possui paralelismo abundante, e as vantagens em performance e consumo de potência do Niagara em código paralelo podem levar a relação performance/watt a um patamar 4 a 5 vezes melhor.

Os núcleos de processamento simples do Niagara possuem algumas limitações. A primeira é menor performance de cada *thread* em cada núcleo de processamento. Uma possível limitação adicional resulta na eficiência de uso de área de cada núcleo. Por ser possível colocar vários núcleos no mesmo *die*, o número total de *threads* no mesmo processador pode tornar-se bastante grande. Este grande número de *threads* é vantajoso quando a carga de trabalho possui paralelismo suficiente, porém a menor performance de cada *thread* pode tornar-se um problema quando não há paralelismo suficiente a ser explorado. Por outro lado, processadores superescalares com a tecnologia Hyper-Threading são atraentes para sistemas que primam o desempenho máximo de *threads* individuais, como estações de trabalho monousuário.

Aplicativos comerciais que fazem uso pesado de operações de ponto flutuante possivelmente apresentarão melhor performance e relação performance/Watt se executados no Pentium Extreme Edition, pelo fato do Niagara possuir apenas uma unidade de execução de operações de número flutuante compartilhada entre todas as *threads* do processador. A Sun Microsystems provavelmente tentará remediar esta desvantagem implementando uma unidade de ponto flutuante para cada *pipeline* em versões futuras do UltraSPARC T1.

4.CONCLUSÕES

Os dois processadores representam estratégias diferentes na forma que oferecem a capacidade de *multithreading*. Provavelmente, cada gerente de tecnologia quererá realizar *bentchmarks* próprios com os aplicativos que precisa executar antes de efetuar a compra de um sistema com processadores de um tipo ou de outro. Compatibilidade com código legado pode também pesar na escolha.

5.REFERÊNCIAS

[1] Ungerer, T., Robič, B., Šilic, J. A Survey of Processors with Explicit Multithreading, *ACM Computing Surveys*, 35, 1 (março de 2003), 29-63.

[2] Marr, D. "Hyper-Threading Technology in the Netburst® Microarchitecture", *14th Hot Chips* (agosto de 2002).

[3] Rupert, G. "Hyperthreading hurts server performance, say developers", Reportagen online na ZDNet UK (novembro de 2005) http://news.zdnet.co.uk/0,39020330,39237341,00.htm

[4] "Developing Multithreaded Applications: A Platform Consistent Approach", tutorial online oferecido pela Intel em 2006, http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/hyperthreading/53797.htm

[5] Kongetira, P., Aingaran, K., Olukotun, K. Niagara: A 32-Way Multithreaded Sparc Processor, *IEEE MICRO* (março e abril de 2005).

[6] Laudon, J. Performance/Watt: The New Server Focus, *ACM SIGARCH Computer Architecture News*, 33, 4 (setembro de 2005).

[7] "The Real Story about Sun's CoolThreads (aka Niagara)", material de marketing publicado pela Hewlett-Packard em janeiro de 2006, http://h71028.www7.hp.com/ERC/cache/280124-0-0-0-121.html?ERL=true

Coerência de Memórias Cache e Modelos de Consistência de Memória

Renato Silva das Neves
RA 057639
Universidade Estadual de Campinas
Instituto de Computação
Campinas, Brasil
renato.neves@students.ic.unicamp.br

ABSTRACT

*A*rquiteturas de computadores com memória compartilhada e múltiplos processadores, cada um com seu esquema de memórias cache, têm que considerar problemas de coerência de dados entre memória e caches. Existem duas abordagens principais em hardware que surgiram para manter a coerência, uma baseada em monitoração (*snooping*) e outra baseada em diretórios. Vários protocolos implementam tais abordagens e alguns deles serão discutidos neste trabalho. A questão sobre modelos de consistência de memórias em sistemas com multiprocessadores, tais como consistências sequencial e relaxadas, também será abordada.

Keywords

Coerência de cache, consistência de memória, *snooping*, diretórios, *write-once*, berkeley, illinois, firefly, consistência sequencial, consistência relaxada.

1. INTRODUÇÃO

Atualmente, é comum observar os processadores atuais com um ou dois níveis de memória cache. Caches são memórias rápidas que possuem cópias dos dados e instruções da memória principal usados mais recentemente, fornecendo um acesso com menor latência para o processador [1]. Assim, o uso de memórias cache surgiu como um método eficiente para reduzir o tempo de acesso médio à memória em computadores monoprocessados, pelo fato de explorar a localidade de referências à memória no tempo (localidade temporal) e no espaço (localidade espacial), fazendo com que a maioria das requisições de memória sejam supridas pela própria cache, evitando assim acessos à memória principal, que é um gargalo nos sistemas computacionais modernos.

Com a dificuldade em aumentar a velocidade dos microprocessadores atuais, a tecnologia de arquiteturas multiprocessadas com memória compartilhada está se tornando cada vez mais uma alternativa viável para o desenvolvimento de sistemas com melhor desempenho. Por exemplo, chips multicore não são mais restritos a apenas servidores ou sistemas de alto-desempenho, já estão se expandindo para computadores desktop, laptops, videogames e em outra série de dispositivos [8]. Daqui a alguns anos será difícil encontrar sistemas com um único processador disponíveis no mercado.

Em arquiteturas multiprocessadas o uso de caches também permite uma melhora em desempenho, evitando na maioria dos casos a latência da memória principal. Usando uma memória principal compartilhada, o compartilhamento de código e dados se torna mais simples entre os processos sendo executados em um ambiente paralelo (multiprocessado). Esse compartilhamento pode resultar em várias cópias de um mesmo bloco em uma ou mais caches ao mesmo tempo. Se os processadores atualizarem suas cópias livremente, os dados poderão ser vistos de forma incoerente, com um processador tendo um valor diferente de um mesmo dado presente em um bloco em relação a outro processador. Esse é o chamado problema de coerência de memórias cache e algum mecanismo deve ser implementado, em software ou hardware, para evitar tal problema. Protocolos em hardware adicionam um hardware especial que detecta acessos à cache e implementa um protocolo de coerência de cache que é transparente ao programador e compilador [6]. Em uma abordagem em software, o compilador deve gerar código consistente, se beneficiando do mecanismo de cache e de algumas instruções especiais que podem ser usadas para manter a coerência.

A coerência de memórias cache poderia ser garantida simplesmente se o valor retornado por uma instrução de carga fosse sempre o último valor gravado por uma instrução de armazenamento para uma mesma posição de memória. Porém, como vários processadores estão executando instruções em paralelo de forma assíncrona, garantir esta propriedade não é uma tarefa trivial. Assim, um modelo de consistência de memória deve garantir aos programadores uma visão da ordem de execução no tempo de instruções de armazenamento e carga na memória, em diferentes processadores, assim como permitir operações de sincronização, a fim de que a consistência possa ser mantida [7].

As seções a seguir irão abordar em maiores detalhes aspectos sobre a coerência de memórias cache, mostrando alguns protocolos implementados em hardware que se baseiam em monitoração e em diretórios. Uma série de protocolos de monitoração serão apresentados, refletindo diferentes implementações encontradas nos sistemas atuais. Uma breve discussão sobre modelos de consistência de memórias também será apresentada, seguida pela conclusão do trabalho.

2. COERÊNCIA DE MEMÓRIAS CACHE

A coerência de memórias cache pode ser realizada por software ou por algum harware especial. As soluções por software geralmente se baseiam em uma análise dos códigos dos programas pelo compilador, que pode verificar conjuntos de dados compartilhados e, por exemplo, não permitir que sejam armazenados na cache. Isso pode levar a uma utilização ineficiente da cache, visto que alguns dados compartilhados podem ser de uso exclusivo de um processador em algum intervalo de tempo. Outras técnicas permitem uma análise mais detalhada dos códigos, procurando determinar períodos em que blocos compartilhados podem ser mantidos na cache e inserindo novas instruções para manter a coerência nestes períodos. As abordagens em software têm a vantagem de evitarem o uso de hardware adicional para manter a coerência de cache, mas por serem efetuadas em tempo de compilação, tendem a tomar decisões que não aproveitam a cache da melhor maneira possível [10].

As soluções em hardware devem ser capazes de detectar acessos à memória incoerentes e garantir a coerência dos blocos da cache (invalidando ou atualizando o bloco), em tempo de execução [7]. Basicamenta tais protocolos se dividem em duas classes: protocolos de monitoração e protocolos baseados em diretório. Os protocolos de monitoração (*snoopy protocols*) são implementados nos controladores de cada cache e ficam monitorando o barramento para saber se eles possuem alguma cópia do bloco solicitado. Assim, as informações de compartilhamento do bloco são mantidas em cada cache. Em contrapartida, o esquema baseado em diretório mantém as informações de cópias e compartilhamento dos blocos da cache de todos os processadores centralizado em um local, chamado diretório. Esses esquemas serão abordados com mais detalhes nas próximas seções.

2.1 Protocolos de Monitoração

Existem dois protocolos de monitoração básicos, chamados protocolo de invalidação de gravação e protocolo de atualização de gravação. No protocolo de invalidação de gravação, a cada escrita realizada em um bloco de uma cache de um processador, uma mensagem é enviada pelo barramento com o objetivo de invalidar todas as cópias presentes nos demais processadores. Assim, esse processador fica com acesso exclusivo à linha da cache para efetuar operações de escrita, dando a idéia de leitores-escritores, ou seja, podem haver vários leitores para uma linha da cache, mas apenas um escritor. No caso do protocolo de atualização de gravação, ao ser realizada uma escrita em uma linha da cache por um processador, ela é atualizada em todas as cópias presentes nas caches dos outros processadores.

Em geral, o mais eficiente desses protocolos depende do padrão de leituras e escritas à memória [10]. A abordagem de invalidação é a mais usada nos sistemas multiprocessados atuais, pois sugere menor tráfego no barramento. Assim, uma série de protocolos de invalidação serão descritos a seguir.

2.1.1 Write-Once

Historicamente é o primeiro protocolo de invalidação que foi proposto na literatura [3] (figura 1). Nesse esquema, os blocos em uma cache podem estar em um de quatro estados: inválido, válido (não modificado, possivelmente compartilhado), reservado (não é necessário um *write-back* e é a única cópia em qualquer cache) e *dirty* (é necessário um *write-back* e é a única cópia em qualquer cache).

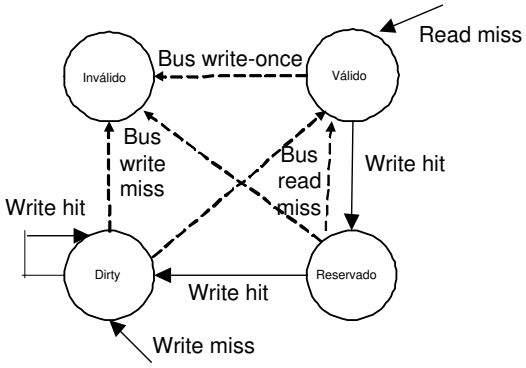


Figure 1: Protocolo Write-Once.

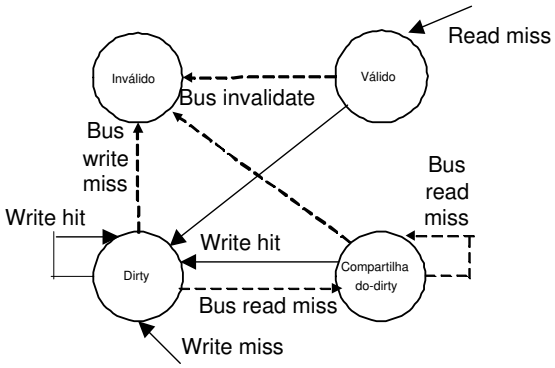


Figure 2: Protocolo Berkeley.

Em uma falha de leitura (*read miss*), o bloco que está no estado *dirty* fornece o bloco solicitado, realizando um *write back* para a memória. Caso nenhum bloco está no estado *dirty*, então o bloco vem da memória. Todas as caches com uma cópia do bloco definem seu estado como válido.

Em um acerto de escrita (*write hit*), caso o estado do bloco seja *dirty*, a escrita é realizada sem alteração de estado. Se o estado for reservado, então o estado do bloco é mudado para *dirty*. Se o estado for válido, o bloco é gravado na memória por *write-through* e seu estado mudado para reservado. O estado válido quer dizer que é um dado compartilhado e deve ser colocado no barramento a informação para as outras cópias serem invalidadas.

Em uma falha de escrita (*write miss*), o bloco é carregado da memória ou do bloco de outra cache cujo estado seja *dirty*. Uma vez que o bloco foi carregado, a escrita é realizada e o estado é modificado para *dirty*. Todas as outras caches invalidam suas cópias.

2.1.2 Berkeley

Desenvolvido pela Universidade da Califórnia para ser aplicado em uma máquina RISC multiprocessada, esse esquema (figura 2) usa transferências diretas entre caches [5]. Seus possíveis estados são: inválido, válido (não modificado, possivelmente compartilhado), compartilhado-*dirty* (modificado e possivelmente compartilhado) e *dirty* (modificado e não há nenhuma outra cópia em caches). Um bloco no estado compartilhado-*dirty* ou *dirty* só pode estar em uma cache,

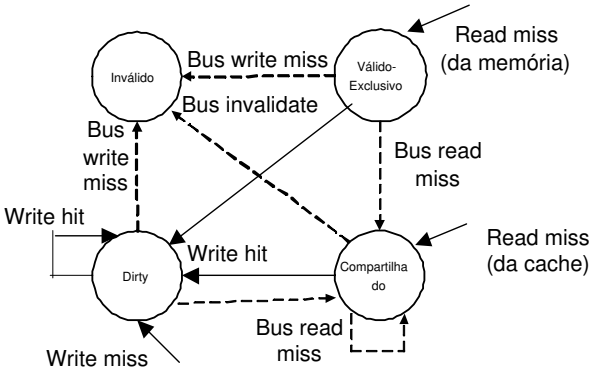


Figure 3: Protocolo Illinois.

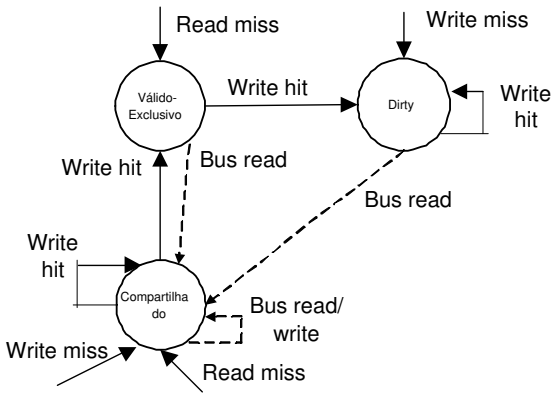


Figure 4: Protocolo Firefly.

mostrando que a cache é a proprietária do bloco. A diferença é que se em uma cache o estado é compartilhado-*dirty*, nas outras caches o bloco pode ser válido. Blocos no estado compartilhado-*dirty* e *dirty*, quando escolhidos para substituição na cache, são gravados na memória principal por *write-back*.
Em uma falha de leitura, a cache que tem o bloco nos estados compartilhado-*dirty* ou *dirty* transfere diretamente o bloco para a cache que fez o pedido, mudando seu estado para compartilhado-*dirty*. Se nenhuma cache possui o bloco ou ele está em qualquer outro estado, então o bloco vem da memória principal. O bloco da cache que fez o pedido tem seu estado alterado para válido.
Em um acerto de escrita, se o estado do bloco é *dirty*, então o estado não é alterado. Se o estado é válido ou compartilhado-*dirty*, o estado é alterado para *dirty* e um *broadcast* é feito pelo barramento para que todas as outras caches possam invalidar suas cópias.
Em uma falha de escrita, o bloco é colocado no estado *dirty* e todas as cópias em outras caches são invalidadas.

2.1.3 *Illinois*

Este método [9] (figura 3) se baseia no fato de que falhas de acesso a blocos podem ser tratadas procurando por blocos em outras caches ou na memória principal e também assume que a cache que requisitou o bloco pode determinar a origem do bloco. A melhora de performance proposta por esse método é que ele percebe que invalidações para acertos de escrita não são necessárias em blocos privados não modificados, pois é possível determinar a partir dos estados se o bloco é ou não compartilhado. Assim como os anteriores, também são usados quatro estados: inválido, válido-exclusivo (não modificado, é a única cópia em todas as caches), compartilhado (não modificado e possivelmente tem outras cópias) e *dirty* (modificado e é a única cópia em todas as caches). Também é usada a abordagem de *write-back*, mas somente quando o bloco está no estado *dirty*.
Em uma falha de leitura, a cache com maior prioridade que tenha uma cópia vai colocar o bloco no barramento e gravar o bloco na memória principal se o estado for *dirty*. Todas as caches que possuam uma cópia do bloco mudam seu estado para compartilhado. Se o bloco vir da memória principal, então nenhuma outra cache tem o bloco e o estado é definido como válido-exclusivo.
Em um acerto de escrita, um bloco *dirty* é escrito sem transições e um bloco válido-exclusivo é mudado para *dirty*. Se o bloco é compartilhado, então é necessário invalidar todas as cópias existentes, e o bloco gravado também é definido como *dirty*.
Em uma falha de escrita, todas as caches com cópias são invalidadas e o bloco é carregado como *dirty*.

2.1.4 *Firefly*

É o esquema (figura 4) usado no *workstation* Firefly [11]. Neste caso são usados três estados: válido-exclusivo (não modificado, é a única cópia nas caches), compartilhado (não modificado, possivelmente com outras cópias) e *dirty* (modificado, é a única cópia nas caches). Também é usado *write-back* para blocos *dirty* a serem substituídos em caso de conflito. A principal diferença dessa técnica é que ela não gera invalidações (não apresenta o estado inválido), permitindo vários escritores. O dado de cada escrita para um bloco compartilhado é transmitido para cada cache que possua uma cópia.
Em uma falha de leitura, a cache que possui o bloco responde fornecendo o bloco com o dado correto. Neste caso o estado do bloco de todas as caches que possuam uma cópia são definidos como compartilhado e caso o bloco seja *dirty* ele é gravado na memória principal. Se nenhuma cache possui o bloco, este vem da memória principal e é definido para ficar no estado válido-exclusivo.
Em um acerto de escrita, nenhuma alteração de estado é realizada caso o estado seja *dirty*. Se o estado for válido-exclusivo, então é modificado para *dirty*. Quando o estado for compartilhado é que aparece a principal modificação desse protocolo: como pode haver cópias em outras caches e não há invalidações, então o dado gravado precisa ser atualizado em todas as cópias existentes. Uma linha de barramento especial é usada para determinar se existem cópias ou não para uma determinada linha da cache: se essa linha estiver com sinal alto, o compartilhamento existe e então o estado é mantido em compartilhado. Caso contrário, não há cópias e o estado pode ser definido como válido-exclusivo.
Em uma falha de escrita, se nenhuma cache possui o bloco, este é carregado da memória no estado *dirty* e depois escrito. Se alguma cache o possui, então o bloco é carregado no estado compartilhado e a palavra é escrita na memória, sendo que os blocos que possuem uma cópia são atualizados.

2.1.5 *MESI*

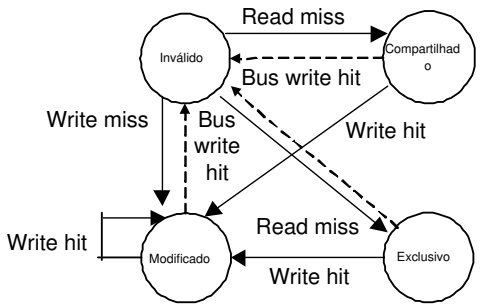


Figure 5: Protocolo MESI.

No protocolo MESI [10] (figura 5), utilizado no Pentium II, um bloco pode ter um de quatro estados: inválido, exclusivo (a linha de cache é igual a da memória principal e é a única cópia entre as caches), compartilhado (a linha da cache é igual a da memória principal e pode estar compartilhado) e modificado (bloco modificado, diferente da memória principal e é a única cópia entre as caches).
Em uma falha de leitura, se alguma cache tem o bloco no estado modificado, então ela fornece o bloco para a cache que o requisitou e ambos definem seu estado para compartilhado. Caso alguma cache tenha o bloco no estado compartilhado ou exclusivo, então a cache que fez o pedido do bloco faz a leitura do bloco da memória principal e todas caches com cópias mudam seu status para compartilhado. Se nenhuma cache possui o bloco requisitado, então o processador carrega o bloco da memória principal e define seu estado como exclusivo.
Em um acerto de escrita, caso o estado do bloco seja modificado, então nenhuma transição é necessária. Se o estado for exclusivo, então o estado é alterado para modificado. Caso o estado do bloco seja compartilhado, é necessário invalidar outras cópias existentes. O bloco passa de compartilhado para modificado.
Em uma falha na escrita, o bloco é carregado da memória principal, caso não existam cópias, é marcado como modificado e logo após é realizada a escrita. Se alguma cache possui uma cópia do bloco, ela escreve no barramento o bloco e marca seu estado para inválido. O processador da cache que fez a requisição lê esta linha, realiza sua escrita e marca como modificada. Todas as outras cópias existentes são invalidadas.

2.2 Protocolos de Diretório

Os protocolos de monitoração são limitados a sistemas multiprocessados com tipicamente menos que 32 processadores, porque o barramento se torna o gargalo para um grande número de processadores. Protocolos de coerência baseados em diretório eliminam a necessidade de um barramento compartilhado, pois mantêm informações de compartilhamento dos dados armazenadas nas caches em diretórios. Isso permite que os sistemas sejam mais escaláveis, com um número maior de processadores, em que informações sobre coerência podem ser enviadas entre processadores usando conexão ponto-a-ponto, sem a necessidade de *broadcast* para todos os processadores em um barramento compartilhado, que é o que acontece com os protocolos de monitoração [6].
As implementações de diretório associam uma entrada de diretório a cada bloco de memória, mas não impede que a propósito de otimizações seja relacionada uma entrada a cada bloco da cache. Além disso, cada entrada possui informação adicional sobre quais caches (processadores) possuem cópia do bloco, para uma posterior busca e/ou invalidação. O diretório pode ser centralizado, juntamente com uma memória compartilhada centralizada, ou também pode ser distribuído, assim como a memória. Neste caso, o diretório não se torna um gargalo, permitindo que acessos a diferentes entradas de diretórios possam ser realizados de forma distribuída [4].
Os esquemas baseados em diretório se apresentam basicamente em três categorias possíveis: diretórios com mapeamento completo, diretórios limitados e diretórios encadeados. Em diretórios com mapeamento completo, cada entrada de diretório tem um bit por processador mais um bit de *dirty*. Em diretórios limitados, existem um número fixo de bits para indicar cópias em processadores, restringindo o número de cópias simultâneas de qualquer bloco. Em diretórios encadeados, um mapeamento completo é emulado distribuindo o diretório entre as caches [1].
Os esquemas baseados em diretório tipicamente funcionam da seguinte maneira: os diretórios mantêm informação sobre quais processadores possuem em suas caches determinados blocos. Um processador deve ter acesso exclusivo ao bloco do diretório antes de ser permitido a ele realizar uma escrita no bloco. Assim, o diretório envia uma mensagem para todos os processadores que possuem cópias do bloco para invalidá-los e espera as confirmações das requisições de invalidação, para logo após ter acesso exclusivo ao bloco. Quando um processador tenta ler um bloco que é de exclusividade de outro processador, uma falha de leitura da cache ocorre e o diretório irá enviar ao processador proprietário uma mensagem para ele realizar um *write back* na memória. O bloco pode então ser compartilhado para leitura pelo processador original e pelo o que requisitou o bloco [6].
Em [4] um exemplo simples de protocolo de diretório é apresentado, que implementa as duas operações citadas anteriormente: falha de leitura da cache e gravação de um bloco da cache. Para implementar isso o protocolo mantém três estados para cada bloco da cache:

- Compartilhado - o bloco na memória está atualizado e um ou mais processadores possuem em sua cache tal bloco;
- Não inserido na cache - nenhum processador tem uma cópia do bloco da cache;
- Exclusivo - o bloco na memória está desatualizado e somente um processador tem a cópia do bloco na cache (proprietário).

Neste protocolo o diretório é armazenado de forma distribuída. Assim surgem três conceitos: nó local, de onde uma requisição foi realizada, nó inicial, onde o diretório está localizado, e nó remoto, que possui uma cópia do bloco da cache, seja ela exclusiva ou compartilhada. As possíveis mensagens entre nós podem ser vistas na tabela 1 [4].

Quando um bloco está no estado não inserido na cache e ocorrer uma falha de leitura, o estado do bloco se torna compartilhado e a cache local recebe os dados da memória.

Table 1: Mensagens enviadas entre nós para manter coerência em um protocolo baseado em diretórios.

Tipo de Mensagem	Origem	Destino	Conteúdo da Mensagem	Função dessa Mensagem
Falha de Leitura	Cache Local	Diretório Inicial	P,A	O processador P tem uma falha de leitura no endereço A; solicitar dados e fazer de P um compartilhador de leitura.
Falha de Gravação	Cache Local	Diretório Inicial	P,A	O processador P tem uma erro de gravação no endereço A; solicitar dados e fazer de P o proprietário exclusivo.
Invalidar	Diretório Inicial	Cache Remota	A	Invalidar uma cópia compartilhada de dados no endereço A.
Buscar	Diretório Inicial	Cache Remota	A	Buscar o bloco no endereço A e enviá-lo a seu diretório inicial; mudar o estado de A na cache remota para compartilhado.
Buscar/invalidar	Diretório Inicial	Cache Remota	A	Buscar o bloco no endereço A e enviá-lo a seu diretório inicial; invalidar o bloco na cache.
Resposta de valor de dados	Diretório Inicial	Cache Local	D	Retornar um valor de dados da memória inicial.
Write back de dados	Cache Remota	Diretório Inicial	A,D	Executar o *write back* de um valor de dados no endereço A.

Caso ocorra uma falha de escrita, o processador solicitante se torna proprietário, definindo o estado do bloco como exclusivo. O conjunto de compartilhadores do bloco reflete esse estado, mantendo indicação de cópia de bloco somente para esse processador.
Quando o bloco está no estado compartilhado e ocorrer uma falha de leitura, o processador solicitante recebe os dados da memória e é adicionado ao conjunto de compartilhadores do bloco. Se houver uma falha de escrita, todos os processadores recebem mensagens para invalidarem seus blocos de cache e o estado do bloco se torna exclusivo, indicando no conjunto de compartilhadores que o único processador a possuir uma cópia foi o que realizou a solicitação.
Quando o bloco está no estado exclusivo e ocorrer uma falha de leitura, o processador proprietário recebe uma mensagem de busca de dados, mudando o estado do bloco para compartilhado e fornecendo ao solicitante o bloco desejado. É adicionado ao conjunto de compartilhadores o processador solicitante. Ao substituir um bloco, o processador proprietário deverá realizar *write back* de dados, atualizando a cópia em memória e removendo o processador do conjunto de compartilhadores de tal bloco. Em falhas de escrita, uma mensagem é enviada ao antigo proprietário invalidando seu bloco na cache. Logo depois, o conjunto de compartilhadores mantém somente o processador solicitante, mantendo o estado do bloco como exclusivo para ele.

3. MODELOS DE CONSISTÊNCIA DE MEMÓRIA

O modelo de consistência de memória especifica a ordem na qual operações em memória serão vistas pelo programador. Uma leitura em uma posição de memória deveria retornar o valor da última escrita na mesma posição de memória. Em sistemas multiprocessados, onde há vários processadores lendo e escrevendo em posições de memória, essa noção de "último" não é bem definida.
Alguns métodos de consistência de memória foram propostos na literatura, dentre eles a consistência sequencial e modelos de consistência relaxados, que podem ser classificados de acordo com as ordenações que relaxam em consistência do processador, consistência fraca ou consistência de liberação [4].
A consistência sequencial fornece a noção de que todas as operações de memória parecem ser executadas uma a cada instante e que todas as operações de um único processador são executadas na ordem descrita pelo programa que está sendo executado no processador. Segundo [2], duas condições são suficientes para manter a consistência sequencial:

1. Antes de uma instrução de carga ser permitida executar em relação a qualquer outro processador, todas as instruções anteriores de carga e armazenamento devem ser executadas, e

2. Antes de uma instrução de armazenamento ser permitida executar em relação a qualquer outro processador, todas as instruções anteriores de carga e armazenamento devem ser executadas.

Os modelos de consistência relaxados levam em consideração justamente essa característica de sistemas multiprocessados: leituras e escritas fora de ordem. A fim de manter a consistência sequencial, instruções de sincronização são usadas para impor a ordenação. Há três casos de relaxamento de ordenação: escrita para leitura, escrita para escrita e todas as ordens.

No caso da escrita para leitura, é permitida uma leitura ser reordenada em relação a escritas anteriores do mesmo processador. Isso pode violar a consistência sequencial dos programas da figura 6. Assim, as duas condições para consistência sequencial são relaxadas para [2]:

1. Antes de uma instrução de carga ser permitida executar em relação a qualquer outro processador, todas as instruções anteriores de carga devem ser executadas, e

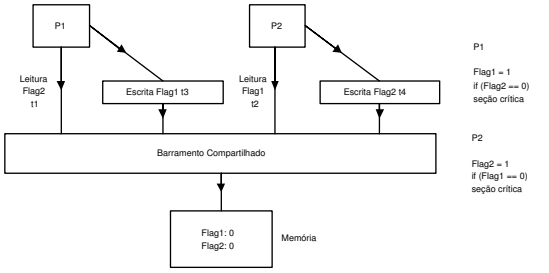


Figure 6: Relaxamento de escrita para leitura.

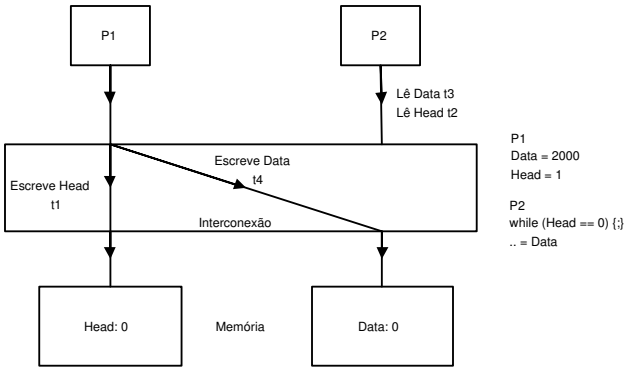


Figure 7: Relaxamento de escrita para leitura e de escrita para escrita.

2. Antes de uma instrução de armazenamento ser permitida executar em relação a qualquer outro processador, todas as instruções anteriores de carga e armazenamento devem ser executadas.

No caso de escrita para leitura e de escrita para escrita, é permitido que escritas para diferentes posições em um mesmo processador sejam colocadas em *pipeline* ou sobrepostas, podendo acessar a memória ou cache fora da ordem de execução do programa. Isso pode violar a consistência sequencial dos programas da figura 7, onde P1 e P2 são processadores que possuem programas diferentes que compartilham dados. A sequência cronológica das operações é indicada por t1, t2, t3 e t4.

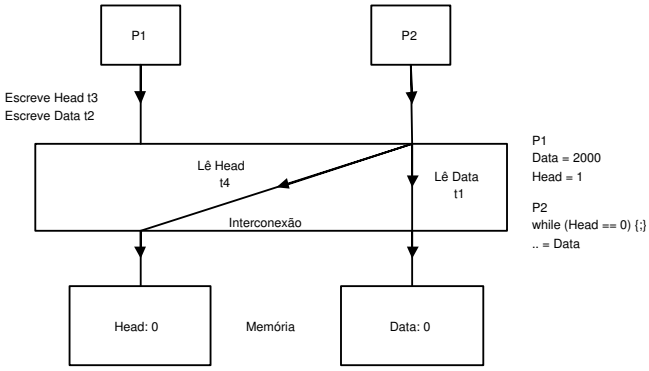


Figure 8: Relaxamento para todas as ordens.

No caso de todas as ordens de instruções serem relaxadas, é permitido que leituras e escritas sejam reordenadas em relação às leituras e escritas subsequentes. Isso pode violar a consistência sequencial dos programas da figura 8. Um dos modelos que é derivado deste caso é o chamado consistência fraca, que se utiliza de pontos de sincronização para manter a consistência das leituras e escritas. Neste caso três condições devem ser satisfeitas, segundo [2]:

1. Antes de uma instrução de carga ou armazenamento ser permitida executar em relação a qualquer outro processador, todos pontos de sincronização anteriores devem ser executados, e

2. Antes de um ponto de sincronização ser permitido executar em relação a qualquer outro processador, todas instruções de carga e armazenamento devem ser executadas, e

3. Pontos de sincronização são sequencialmente consistentes em relação a outro ponto de sincronização.

Assim, enquanto que o mecanismo de coerência de cache garante que as escritas em todas as caches para um bloco específico terão a mesma ordem lógica, a consistência de memória define ao programador como a ordem das escritas para diferentes blocos será percebida por cada processador.

4. CONCLUSÃO

O presente trabalho apresentou uma revisão bibliográfica sobre dois problemas existentes nos sistemas multiprocessados modernos: coerência de memórias cache e consistência de memória. Dentre o problema de coerência de cache foram discutidos mecanismos baseados em software, que a partir de análise de código pelo compilador gera código coerente, e mecanismos baseados em hardware, que são os mais comuns em arquiteturas atuais. Basicamente mecanismos em hardware se dividem em protocolos de monitoração e baseados em diretório. Para sistemas multiprocessados com barramento compartilhado e com um número limitado de processadores, as técnicas baseadas em monitoração são as preferidas, variando desde protocolos simples até protocolos que implementam máquinas de estados mais complexas a fim de melhorar o desempenho das caches. Basicamente tais protocolos baseados em estados mantém um ou dois estados que indicam que o bloco é exclusivo para uma determinada cache e quaisquer gravações invalidam todas as cópias existentes nas caches, marcando tal bloco como exclusivo ou *dirty*. Outro estado, em geral, indica se o bloco é possivelmente compartilhado ou não. Para sistemas com um número maior de processadores, técnicas baseadas em diretório são as preferidas. Um exemplo simples foi apresentado, implementando um protocolo de coerência de caches baseado em diretórios distribuídos e uma máquina com três estados possíveis para cada bloco de cache.

Por último foi realizada uma breve discussão à respeito de modelos de consistência de memória, que definem a ordem na qual as instruções serão vistas pelo programador, ou seja, quando um valor gravado será retornado por uma leitura. A coerência de memórias cache complementa esse conceito, definindo que valores podem ser retornados por uma leitura.

Mecanismos de sincronização são fundamentais para manter a consistência de memória em arquiteturas modernas, pois permitem que a consistência sequencial seja garantida somente em certos pontos e não em todas as instruções, degradando a performance do sistema.

5. REFERENCES

[1] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.

[2] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, New York, NY, USA, 1990. ACM Press.

[3] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 124–131, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[4] J. L. Hennessy and D. A. Patterson. *Arquitetura de Computadores uma Abordagem Quantitativa*. Editora Campus, 2003.

[5] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *ISCA '85: Proceedings of the 12th annual international symposium on Computer architecture*, pages 276–283, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[6] R. Lawrence. A survey of cache coherence mechanisms in shared memory multiprocessors. 1998.

[7] D. J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons. *ACM Comput. Surv.*, 25(3):303–338, 1993.

[8] M. M. K. Martin. Formal verification and its impact on the snooping versus directory protocol debate. *Proceedings of the 2005 International Conference on Computer Design*, pages 543–549, 2005.

[9] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, New York, NY, USA, 1984. ACM Press.

[10] W. Stallings. *Arquitetura e Organização de Computadores*. Prentice Hall, 2003.

[11] C. Thacker. Private communication. 1984.

Bluetooth – Características, protocolos e funcionamento

Thiago Senador de Siqueira
RA 057642
Instituto de Computação
Universidade Estadual de Campinas
thiagosenador@yahoo.com.br

RESUMO

Bluetooth é um padrão de comunicação sem fio de curto alcance, baixo custo e baixo consumo de energia que utiliza tecnologia de rádio. Sua especificação é aberta e está publicamente disponível. Uma Bluetooth Wireless Personal Area Network (BT-WPAN) consiste de piconets. Cada piconet é um conjunto de até oito dispositivos Bluetooth. Um dispositivo é designado como mestre e os outros como escravos. Através de uma pilha de protocolos bem definida e de um conjunto mínimo de componentes de hardware, dispositivos Bluetooth têm ganhado uma parcela significativa do mercado wireless.

1. INTRODUÇÃO

Bluetooth é uma especificação industrial para *Personal Area Networks* (PANs), também conhecido como IEEE 802.15.1. *Bluetooth* provê uma forma de conectar e trocar informações entre dispositivos como PDAs, telefones celulares, laptops, PCs, impressoras, câmeras digitais dentre outros, através de freqüência de rádio de curto alcance, segura, de baixo custo e globalmente disponível [1][2][5]. A especificação *Bluetooth* foi inicialmente projetada com o objetivo de desenvolver dispositivos interconectáveis de baixo consumo de energia, através de freqüências de rádio de curto alcance – de 1 a 100 metros, dependendo da categoria do dispositivo.

Se estiverem um ao alcance do outro, dois dispositivos que implementam a especificação *Bluetooth* podem se comunicar, mesmo se não estiverem na mesma sala, mas, respeitando uma distância máxima de até 100 metros. A transferência de dados e de voz varia em muitos dispositivos *Bluetooth* dependendo de requisitos como quantidade de energia necessária na transmissão, taxa de transferência e distância [1].

Uma das grandes vantagens da utilização de *Bluetooth* é a possibilidade de criar PANs de forma *ad-hoc*, ou seja, um dispositivo que entra ao alcance de outro, automaticamente se conecta e estes constituem uma PAN.

Neste cenário, um dispositivo recebe o papel de mestre, enquanto que os outros recebem o papel de escravo. A este conjunto de mestre e escravos dá-se o nome de *piconet*. Cada *piconet* pode ter até 8 dispositivos. Quando duas *piconets* se conectam, através de um dispositivo em comum, forma-se a chamada *scatternet*.

Outra característica marcante na especificação *Bluetooth* é a segurança. Através de mecanismos como *frequency hopping*, autenticação de códigos PIN (*Personal Identification Number*) e criptografia de 128 bits garante-se comunicação segura e livre de interferências entre dispositivos *Bluetooth* [3].

Este trabalho está organizado da seguinte maneira: na seção 2 é apresentada uma visão geral da especificação *Bluetooth* com funcionalidades e características básicas. Na seção 3 são analisados software e hardware Bluetooth. O mecanismo de comunicação entre dispositivos *Bluetooth* é apresentado na seção 4. E, finalmente, na seção 5, as conclusões obtidas a respeito da especificação *Bluetooth*.

2. VISÃO GERAL

Bluetooth é um padrão de comunicação sem fio de curto alcance, baixo custo e baixo consumo de energia que utiliza tecnologia de rádio. Embora tenha sido imaginada como uma tecnologia para substituir cabos pela Ericsson (a maior fabricante de celulares, hoje Sony-Ericsson *Corporation*) em 1994, *Bluetooth* tem se tornado largamente utilizado em inúmeros dispositivos e já representa uma parcela significativa do mercado *wireless*. Dentre os dispositivos que utilizam *Bluetooth* podemos incluir os dispositivos inteligentes, como PDAs, telefones celulares, PCs,

periféricos, como mouses, teclados, *joysticks*, câmeras digitais, impressoras e dispositivos embarcados, como os utilizados em automóveis (travas elétricas, cd *player*, etc).

O nome *Bluetooth* nasceu no século X com o rei da Dinamarca, rei Harald Bluetooth, que se engajava na diplomacia entre os países da Europa, fazendo com que estes estabelecessem acordos comerciais entre si [3]. Os projetistas de *Bluetooth* adotaram tal nome para sua especificação pelo fato desta permitir que diferentes dispositivos possam se comunicar um com outro.

O projeto da especificação *Bluetooth* começou quando a empresa de celulares Ericsson se juntou a outras empresas como Intel *Corporation*, *International Business Machines Corporation* (IBM), Nokia *Corporation* e Toshiba *Corporation* para formarem o *Bluetooth Special Interest Group* (SIG) em 1998. Em 1999 outras empresas se juntaram ao SIG como 3Com *Corporation*, Lucent/Agere *Technologies Inc*., Microsoft *Corporation* e Motorola *Inc*. O trabalho conjunto de todos os membros do SIG permitiu o desenvolvimento da especificação *Bluetooth*, através de padrões abertos para assegurar uma rápida aceitação e compatibilidade com as tecnologias disponíveis no mercado [7]. A especificação resultante, desenvolvida pelo *Bluetooth* SIG é aberta e inteiramente disponível. *Bluetooth* já é adotada por mais de 2100 companhias ao redor do mundo. A tecnologia *Wireless Personal Area Network* (WPAN), baseada na especificação *Bluetooth*, é agora um padrão IEEE sob a denominação 802.15.1 WPANs [4].

A especificação *Bluetooth* define como dispositivos *Bluetooth* são agrupados para propósito de comunicação. Levando em consideração o alcance das ondas de rádio dos dispositivos *Bluetooth*, estes são classificados em três classes:

- Classe 3 – alcance de no máximo 1 metro;
- Classe 2 – alcance de no máximo 10 metros;
- Classe 1 – alcance de no máximo 100 metros.

Uma *Bluetooth Wireless Personal Area Network* (BT-WPAN) consiste de ***piconets***. Cada *piconet* é um conjunto de até oito dispositivos *Bluetooth*. Um dispositivo é designado como mestre e os outros como escravos. Duas *piconets* podem se conectar através de um dispositivo *Bluetooth* comum a ambas (um *gateway*, *bridge* ou um dispositivo mestre) para formarem uma ***scatternet***. A figura 1 mostra uma *scatternet* formada por duas *piconets*. Estas *piconets* interconectadas dentro de uma *scatternet* formam uma infra-estrutura para *Mobile Area Network* (MANET) e podem tornar possível a comunicação de dispositivos não diretamente conectados ou que estão fora de alcance de outro dispositivo.

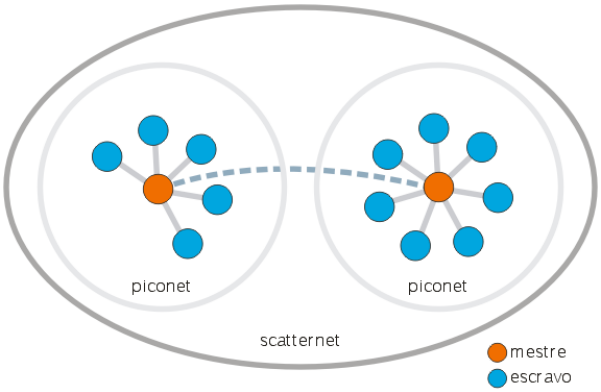


Figura 1 – Uma *scatternet* formada de duas *piconets*

Implementações atuais de dispositivos *Bluetooth* baseiam-se basicamente em conexões ponto-a-ponto. Entretanto, a especificação *Bluetooth* não define apenas soluções ponto-a-ponto como também topologias mais complexas [1]. O objetivo é a formação de *scatternets* que forneçam comunicação efetiva e eficiente através de vários de nós com tempo de resposta aceitável e baixo consumo de energia para o desenvolvimento de aplicações fim-a-fim.

3. ARQUITETURA

A arquitetura *Bluetooth* consiste basicamente de dois componentes: um *transceiver* (hardware) e uma pilha de protocolos (software). Esta arquitetura oferece serviços e funcionalidades básicas que tornam possível a conexão de dispositivos e a troca de uma variedade de tipos de dados entre estes dispositivos.

A freqüência utilizada por dispositivos *Bluetooth* opera em uma faixa de rádio não licenciada ISM (*industrial, scientific, medical*) entre 2.4 GHz e 2.485 GHz. O sistema emprega um mecanismo denominado *frequency hopping*, que "salta" constantemente de freqüência para combater interferência e enfraquecimento do sinal. A cada segundo são realizados 1600 saltos de freqüência. A taxa de transmissão pode alcançar 1 Megabit por segundo (Mbps) ou, com o mecanismo *Enhanced Data Rate*, recentemente introduzido na última especificação *Bluetooth*, a 2 ou 3 Mbps [5][6].

Durante uma operação típica, um canal físico de rádio é compartilhado por um grupo de dispositivos que estão sincronizados a um *clock* comum e a um padrão de saltos de freqüência. Um dispositivo provê a sincronização de referência e é chamado de mestre. Todos os outros dispositivos são conhecidos como escravos [1][4][6]. Como mencionado na seção anterior, um grupo de dispositivos sincronizados desta maneira forma uma *piconet*. Esta é a maneira fundamental de comunicação através de *Bluetooth*.

Dispositivos em uma mesma *piconet* utilizam um padrão de saltos de freqüência que é algoritmicamente determinado por atributos na especificação *Bluetooth* e pelo *clock* do dispositivo mestre. Este salto de freqüência se baseia em um algoritmo pseudo-randômico ordenando 79 freqüências, em intervalos de 1 MHz, dentro da faixa ISM. O padrão de salto de freqüência pode ser adaptado para excluir a porção de freqüências que está sendo utilizada e interferindo os dispositivos. Este mecanismo de *frequency hopping* auxilia na coexistência de dispositivos Bluetooth com outros (*non-hopping*) sistemas ISM que se encontram na mesma localização.

O canal físico é sub-dividido em unidades de tempos denominados *slots*. Dados são transmitidos entre dispositivos *Bluetooth* em pacotes que são posicionados nestes *slots* [1][2][6]. Quando as circunstâncias permitem, é possível alocar um número consecutivo de *slots* em um único pacote. O mecanismo de *frequency hopping* entra em cena tanto na emissão quanto na recepção de pacotes. A especificação *Bluetooth* provê o efeito de transmissões *full-duplex* através do uso de esquemas de divisão de tempo (*time division duplex*) [4].

Dentro de um canal físico, um *link* físico é formado entre quaisquer dois dispositivos, e transmitem pacotes em ambas as direções. Em um canal físico de uma *piconet* há restrições sobre qual dispositivo pode formar um *link* físico. Existe um *link* físico entre cada escravo e o mestre. Em uma *piconet*, não há formação de *links* físicos diretamente entre escravos.

3.1 A pilha de protocolos *Bluetooth*

A especificação *Bluetooth* divide a pilha de protocolos em três grupos lógicos. São eles: grupos de protocolos de transporte, grupo de protocolos de *middleware* e o grupo de aplicação [1][4], como ilustrado na figura 2.

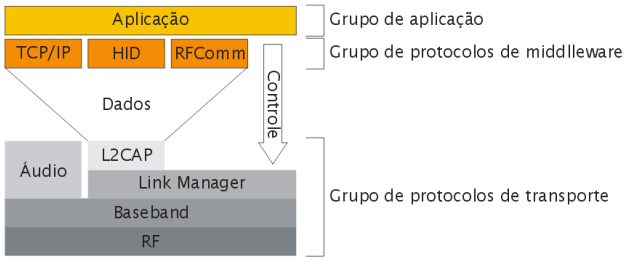


Figura 2 – Pilha de protocolos Bluetooth

O grupo de protocolos de transporte permite dispositivos *Bluetooth* localizar outros dispositivos e gerenciar *links* físicos e lógicos para as camadas superiores. Neste contexto, protocolos de transporte não se equivalem aos protocolos da camada de transporte do modelo OSI (utilizado na especificação de protocolos de rede). Ao invés disso, estes protocolos correspondem às camadas físicas e de enlace do modelo OSI. As camadas de rádio freqüência (RF), *Baseband*, *Link Manager*, *Logical Link Control and Adaptation* (L2CAP) estão incluídas no grupo de protocolos de transporte. Estes protocolos suportam tanto comunicação síncrona quanto assíncrona e todos estes são indispensáveis para a comunicação entre dispositivos *Bluetooth* [1].

O grupo de protocolos de *middleware* inclui protocolos de terceiros e padrões industriais. Estes protocolos permitem que aplicações já existentes e novas aplicações operem sobre *links Bluetooth*. Protocolos de padrões industriais incluem *Point-to-Point Protocol* (PPP), *Internet Protocol* (IP), *Trasmission Control Protocol* (TCP), *Wireless Application Protocol* (WAP), etc. Outros protocolos desenvolvidos pelo próprio SIG também foram incluídos como o RFComm, que permite aplicações legadas operarem sobre os protocolos de transporte *Bluetooth*, o protocolo de sinalização e controle de telefonia baseada em pacotes (TCS), para o gerenciamento de operações de telefonia e o *Service Discover Protocol* (SDP) que permite dispositivos obterem informações sobre serviços disponíveis de outros dispositivos [4].

O grupo de aplicação consiste das próprias aplicações que utilizam *links Bluetooth*. Estas podem incluir aplicações legadas ou aplicações orientadas a *Bluetooth*.

Pode-se resumir as características das camadas da pilha de protocolos *Bluetooth* da seguinte maneira [1][5][7]:

- **Camada de Rádio (RF)**: a especificação da camada de rádio corresponde essencialmente

ao projeto dos *tranceivers Bluetooth* e será discutida na próxima seção;

- **Camada *Baseband***: esta camada define como dispositivos *Bluetooth* localizam e se conectam a outros dispositivos. Os papéis de mestre e escravo são definidos nesta camada, assim como os padrões de saltos de freqüência utilizados pelos dispositivos. Mestre e escravos só se comunicam entre si através de *slots* de tempo pré-definidos. É nesta camada também onde se definem os tipos de pacotes, procedimentos de processamento de pacotes, estratégias de detecção de erros, criptografia, transmissão e retransmissão de pacotes. Esta camada suporta dois tipos de *links*: *Synchronous Connection-Oriented* (SCO) e *Assynchronous Connection-Less* (ACL). *Links* SCO são caracterizados pela periódica atribuição de um *slot* de tempo a um dispositivo e é utilizado basicamente na transmissão de voz, que requer transmissões de dados rápidas e consistentes. Um dispositivo que estabeleceu um *link* SCO possui, em essência, determinados *slots* de tempo reservados para seu uso. Seus pacotes são tratados como prioritários e serão processados antes de pacotes ACL. Já um dispositivo que opera sobre um *link* ACL pode enviar pacotes de tamanho variável de 1, 3 ou 5 *slots* de tempo. Entretanto, este tipo de *link* não possui reserva de *slots* de tempo para seus pacotes;
- ***Link Manager***: esta camada implementa o *Link Manager Protocol* (LMP), que gerencia as propriedades do meio de transmissão (ar) entre os dispositivos. O protocolo LMP também gerencia a alocação de taxa de transferência de dados, taxa de transferência de áudio, autenticação através de métodos de desafio-resposta, níveis de confiança entre dispositivos, criptografia de dados e controle do gasto de energia;
- **Camada L2CAP**: a camada L2CAP serve de interface entre os protocolos de camadas superiores e os protocolos de transporte de camadas inferiores. Esta camada também é responsável pela fragmentação e remontagem de pacotes.

3.2 Hardware Bluetooth

Todos os dispositivos que implementam a especificação *Bluetooth* devem possuir minimamente seis componentes de hardware [2], como mostra a figura 3. São eles:

- ***Host Controller***: responsável pelo processamento de código de alto nível, tanto de aplicações quanto de algumas camadas inferiores da pilha de protocolos *Bluetooth* – controle de link lógico, L2CAP, RFComm e outras funcionalidades;
- ***Link Control Processor***: um microprocessador responsável pelo processamento das camadas mais baixas da pilha de protocolos como *Link Manager* e *Link Controller*. Em algumas aplicações embarcadas, pode estar combinado com o *Host Controller* em um único chip;
- ***Baseband Controller***: bloco lógico responsável pelo controle do *transceiver* de rádio freqüência (RF);
- ***Transceiver* RF**: contém o sintetizador de rádio freqüência, filtros Gaussianos, recuperação de *clock* e detector de dados;
- **RF *Front-End***: contém o filtro de banda passante da antena, amplificador de ruídos e amplificador de energia. É responsável pela troca de estados – emissor x receptor.
- **Antena**: pode ser interna ou externa e pode estar integrada em componentes de terceiros.

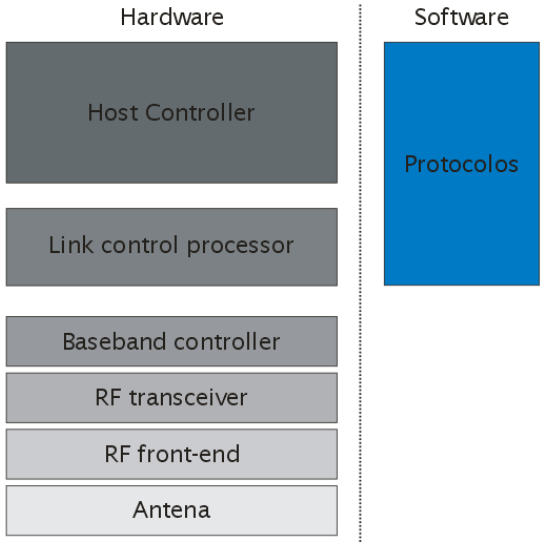


Figura 3 – Diagramas de blocos de hardware e software da especificação Bluetooth.

A maioria dos desenvolvedores de dispositivos *Bluetooth* tem adotado uma abordagem de projeto de multi-chip, utilizando componentes CMOS para o núcleo de gerenciamento de banda e microprocessadores para processamento de rádio freqüência (processamento de sinais), como mostra a figura 4. Enquanto esta abordagem ajuda a simplificar o design do chip, algumas desvantagens como o alto número de componentes, espaço de placa inadequado e outras questões de integração podem aumentar os custos de implementação de forma significativa. A implementação típica de um sistema de rádio *Bluetooth* envolve um número de componentes consideravelmente caros, como dispositivos de rádio freqüência e filtros de interferência.

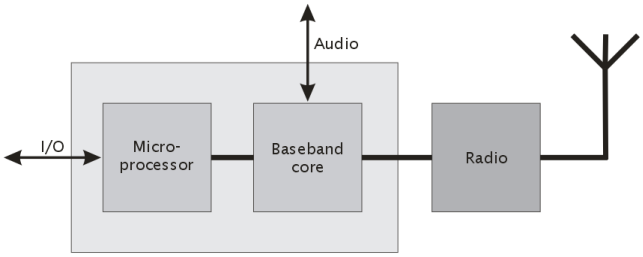


Figura 4 – Abordagem multi-chip de dispositivos *Bluetooth*

Alguns desenvolvedores de dispositivos *Bluetooth*, entretanto, têm adotado uma abordagem diferente da anterior: implementar toda a especificação em um único chip (figura 5). Nesta abordagem, todos os componentes – microprocessador, *baseband* e rádio, são implementados inteiramente em um único componente CMOS. [6] A principal vantagem desta abordagem é a redução de custos na fabricação dos chips.

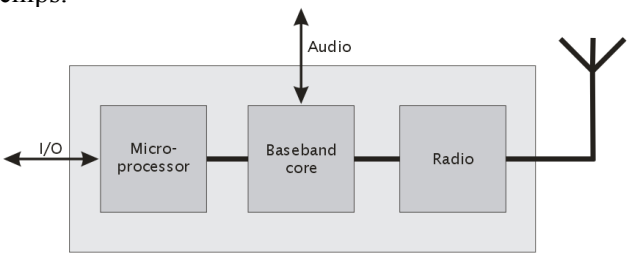


Figura 5 – Abordagem de um único chip

4. O PROCESSO DE COMUNICAÇÃO

Um *transceiver Bluetooth* é um dispositivo que opera em uma faixa de rádio não licenciada (ISM) a 2.4 GHz. Na maioria dos países, há 79 canais disponíveis. Entretanto, alguns países permitem apenas o uso de 23 canais. A taxa de transferência nominal de cada canal é de 1 MHz [1][4].

Quando conectado a outros dispositivos *Bluetooth*, um dispositivo troca de freqüência a uma taxa de 1600 vezes por segundo. Cada freqüência é utilizada apenas 625 microssegundos [6].

A especificação *Bluetooth* utiliza o esquema de divisão de tempo (*Time Division Duplexing* - TDD) e divisão de tempo com múltiplos acessos (*Time Division Multiple Access* - TDMA) na comunicação de dispositivos. Como discutida na seção anterior, a transmissão de dados é feita através de slots de tempos. Um único *slot* possui 625 microssegundos de comprimento, representando um pacote de dados que ocupa um único *slot*. Na camada de *baseband*, um pacote consiste de um código de acesso, um cabeçalho e *payload*, como mostra a figura 6.

Figura 6 – Estrutura de um pacote de uma *piconet*

O código de acesso contém o endereço da *piconet* (para filtrar mensagens de outra *piconet*) e possui geralmente 72 bits de comprimento. O cabeçalho possui 18 bits e inclui o endereço de um dispositivo escravo ativo na rede *Bluetooth*. O campo *payload* é onde trafega os dados da aplicação. Pode conter de 0 a 2745 bits de dados.

Em uma *piconet*, o mestre transmite em *slots* de tempo pares enquanto que os escravos transmitem apenas em *slots* de tempo ímpares. Em cada *slot* de tempo, devido ao mecanismo *frequency hopping*, um canal de freqüência diferente é utilizado, ou seja, após cada envio ou recebimento de pacotes, o canal é trocado, antes mesmo da transmissão do próximo pacote.

Um dispositivo *Bluetooth* pode estar em um dos seguintes estados: espera, solicitação, página, conectado, transmissão, bloqueado, escuta e estacionado [1], como mostra a figura 7. Um dispositivo está no estado de **espera** quando está ligado mas ainda não se juntou a uma *piconet*. Este entra no estado de **solicitação** quando envia requisições de busca de outros dispositivos com os quais possa se conectar. Um dispositivo mestre de uma *piconet* existente pode também estar no estado de **página**, enviando mensagens à procura de dispositivos que possam se juntar e sua *piconet*.

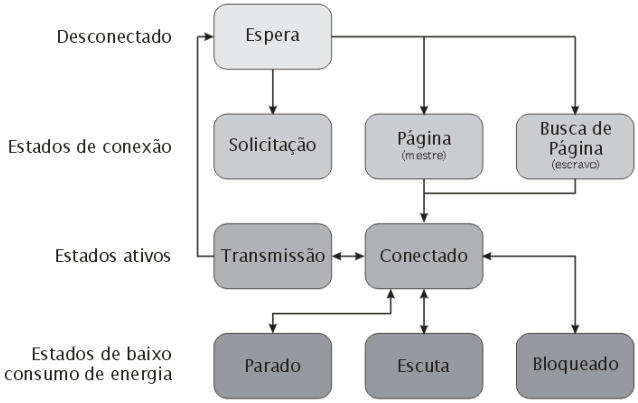


Figura 7 – Diagrama de estados da especificação *Bluetooth*

Quando uma comunicação é bem sucedida entre mestre e um novo dispositivo, este novo dispositivo assume o papel de escravo e entra no estado de **conectado**, e então recebe um endereço que o identifica na *piconet*. Enquanto conectado, um escravo pode transmitir dados, quando o mestre o permite fazê-lo. Durante a transmissão de seus dados, os escravos estão no estado de **transmissão**. Ao fim de sua transmissão, este retorna ao estado de conectado.

O estado de **escuta** é um estado de baixo consumo de energia onde um escravo "dorme" por um número pré-definido de *slots*. O dispositivo então acorda para realizar a transmissão de dados em seu slot de tempo apropriado. Após a transmissão o dispositivo escravo retorna então para o estado de escuta até que seus próximos *slots* de tempo designados cheguem. O estado de **bloqueado** é outro estado onde se verifica o baixo consumo de energia, em que o escravo não está ativo por um período pré-determinado de tempo. Entretanto, não há transferência de dados dentro do estado bloqueado.

Quando um dispositivo escravo não tem dados a serem enviados ou recebidos, o dispositivo mestre pode instruí-lo a entrar no estado de **estacionado**. Quando no estado de estacionado, o dispositivo escravo perde seu endereço atual na *piconet*, o qual será dado a outro escravo que o mestre está retirando do estado de estacionado.

5. CONCLUSÕES

A tecnologia sem fio *Bluetooth* aborda vários pontos chaves que facilitam sua vasta adoção: 1) é uma especificação aberta e que está publicamente disponível; 2) sua tecnologia sem fio de curto alcance permite dispositivos periféricos comunicarem entre si através de uma interface simples, o ar, ao contrário das tecnologias de cabos, que utilizam conectores de uma grande variedade de formas, tamanhos e números de pinos; 3) a especificação *Bluetooth* suporta transferências tanto de voz quanto de dados, tornando-se uma tecnologia ideal na comunicação de dispositivos heterogêneos; e 4) *Bluetooth* utiliza uma faixa de freqüências não regulamentada e vastamente disponível em qualquer lugar do mundo.

Referências

[1] McDermott-Wells, P. Bluetooth Overview. *IEEE Potencials Magazine*. December 2004, pp.33-35.

[2] Johnson, D. Hardware and software implications of creating Bluetooth Scatternet devices. In: *Proceedings of the IEEE AFRICON*. 2004, pp. 211-215.

[3] Wikipedia. Bluetooth. Disponível em <www.wikipedia.com/eng/bluetooth>. Último acesso em 12 de junho de 2006.

[4] Bluetooth SIG. Specification of the Bluetooth System. Disponível em <www.bluetooth.com>. Último acesso em 15 de junho de 2006.

[5] Chomienne, D. Eftimakis, M. Bluetooth Tutorial. Disponível em <www.newlogic.com/products/Bluetooth-Tutorial-2001.pdf>. Último acesso em 15 de junho de 2006.

[6] Kardach, J. Bluetooth Architecture Overview. *Intel Technology Journal*. 2000.

[7] Miller, B. A. Bisdikian, C. Bluetooth Revealed. *Upper Saddle River*. Prentice Hall, 2001.

Tecnologia de Discos

Edmar Welington Oliveira
RA 065819

edmmarwt@gmail.com

RESUMO
Discos rígidos ainda constituem o elemento fundamental de todo e qualquer subsistema de armazenamento. Ao longo dos anos, avanços tecnológicos permitiram que eles armazenassem quantidades cada vez maiores de dados, e que os mesmos fossem acessados em tempos cada vez menores. Tecnologias de disco, comumente, são classificadas em função de sua interface com o sistema. Baseado nisto, este artigo apresenta as principais tecnologias, bem como a evolução destas ao longo dos anos.

Palavras-Chave
Discos, tecnologias, interfaces.

1. INTRODUÇÃO
O desempenho dos sistemas computacionais depende, primordialmente, das propriedades do processador, dispositivos de armazenamento e tecnologias de interconexão. Com relação aos sistemas de armazenamentos baseados em discos magnéticos, tais propriedades se referem às velocidades nas quais os dados, no disco, são armazenados e recuperados. Melhorias nas tecnologias de armazenamento são cruciais para o desempenho de sistemas.

Novas fontes de informação como imagens, sons e vídeos demandam alta capacidade de armazenamento e desempenho dos discos rígidos. Melhorias nas propriedades dos discos também são necessárias para atender às empresas, com cada vez maiores quantidades de informação a armazenar. Tão importante quando as características físicas do disco é sua conexão com o sistema e o quanto de desempenho é possível obter através desta.

Ao longo dos anos observa-se que, embora inferior a outros dispositivos, a evolução dos discos rígidos e das interfaces destes com os computadores procuram atender à demanda cada vez maior por capacidade de armazenamento e velocidades de transferência de dados.

2. DISCOS - VISÃO GERAL [1,3]
Os discos rígidos são dispositivos de armazenamento não voláteis – não necessitam de energia para conservar os dados neles gravados.

Os dados de um disco rígido estão armazenados em um conjunto de discos formados por um substrato coberto por uma camada magnética extremamente fina. É neste material magnético que se encontra o princípio de armazenamento. Na verdade, quanto mais fina for a camada de gravação, maior será sua sensibilidade e, consequentemente, maior será a densidade de gravação permitida por ela Poderemos, então, armazenar mais dados, criando discos rígidos de maior capacidade. O substrato, não-magnético, é a base desses discos, e geralmente é feito de liga de alumínio ou de uma mistura de vidro e cerâmica, cada vez mais utilizada por permitir a fabricação de discos mais finos e resistentes.

Os discos rígidos possuem, além de controladores responsáveis por comandar operações internas e fazer a comunicação com a interface do barramento, um conjunto de linhas de transmissão compartilhadas que conecta os dispositivos de entrada e saída ao processador e à memória.

A figura 1 ilustra os componentes de um disco rígido moderno, manufaturado em um compartimento selado com a tecnologia *Head Disk Assembly* (HDA), na qual os discos magnéticos estão hermeticamente fechados para evitar contaminações e danos à superfície de armazenamento.

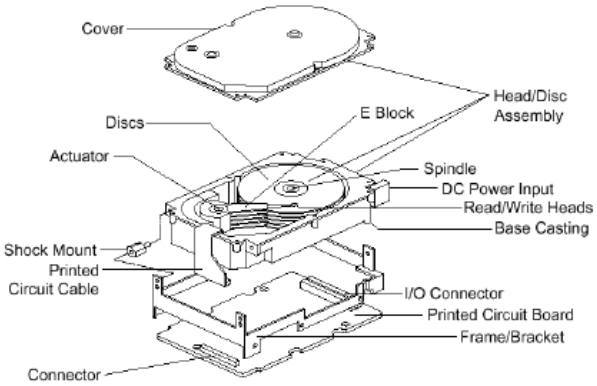


Figura 1 – Componentes do disco rígido.

Conforme mostra a figura 2, os discos magnéticos formam uma pilha suportada por um eixo central cuja rotação os faz girar. Entre eles existem braços mecânicos que movimentam as cabeças responsáveis pela leitura e gravação em cada uma das superfícies. As trilhas - zonas concêntricas de gravação - são divididas em setores. Setores de discos diferentes que estão relativamente na mesma posição podem ser lidos e gravados simultaneamente, formando cilindros.

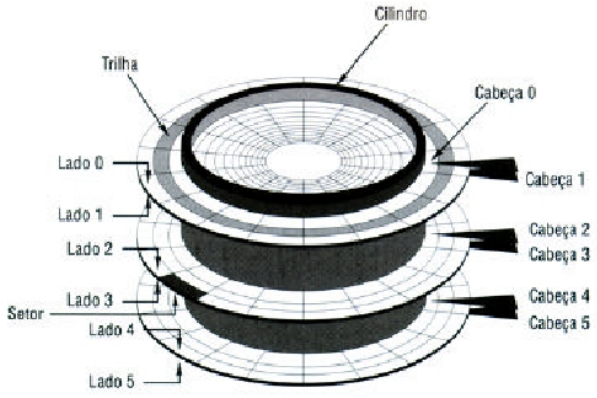


Figura 2 – Geometria de um disco rígido

Dois modos de endereçamento são comuns nos discos rígidos. O mais antigo, denominado Cylinder-*Head*-*Sector* (CHS), é altamente ligado à configuração espacial ilustrada na figura 2. Nesse modelo, geralmente as operações de leitura e escrita tendem a ser concentradas em localidades próximas, para evitar os *delays* causados pela movimentação dos mecanismos internos do drive.

Os discos rígidos modernos, entretanto, não respeitam essa divisão tradicional, gravando mais dados onde há maior densidade magnética. Juntamente com essa tendência, os discos passam a ser endereçados como um *array* unidimensional de setores pelo *Logical Block Addressing* (LBA). O LBA não garante qualquer proximidade física entre dois endereços vizinhos nesse *array*, ficando a cargo da lógica de controle interna dos discos fazer o mapeamento na mídia.

3. PERSPECTIVA HISTÓRICA [6,10]

O primeiro projeto de unidade de disco foi o IBM 305 RAMAC (*Random Access Method of Accounting and Control*), lançado em 1956. Antes disso, as formas de armazenamento disponíveis eram memória principal, fitas e tambores magnéticos. O armazenamento do IBM 305, que pesava uma tonelada e ocupava cerca de 8,5m^3, consistia de uma pilha de 50 discos de 24 polegadas que giravam a 1200rpm e eram capazes de armazenar 5 milhões de caracteres de 7 bits – pouco mais do que 4MB. O acesso à mídia era feito a taxas de até 8,8KB/s através de dois braços mecânicos independentes que se moviam no sentido vertical, para selecionar um dos discos, e no sentido horizontal, para selecionar a trilha de gravação.

O próximo avanço importante no caso dos discos rígidos foi a unidade de disco rígido removível desenvolvida pela IBM em 1962; ela tornou possível compartilhar os dispendiosos componentes eletrônicos das unidades de disco e ajudou os discos a superar as fitas como meio de armazenamento preferencial. O IBM 1211 de 1962 tinha uma densidade de área de 50.000 bits por polegada quadrada e um custo de aproximadamente $800 por MB.

A segunda inovação importante nos discos foi o projeto denominado *Winchester*, em 1973. Tal disco foi a primeira tecnologia popular de discos rígidos em compartimentos selados com a tecnologia HDA. O primeiro disco hermeticamente fechado comercializado pela IBM tinha dois eixos, cada um com um disco de 30MB; o apelido "30-30" para o disco levou o nome de *Winchester* – o rifle esportivo mais popular da América, o *Winchester* 94, ficou conhecido como "30-30" devido ao calibre de seu cartucho.

O primeiro disco rígido construído para uso específico em microcomputadores foi lançado em 1979 pela *Shugart Associates* – hoje *Seagate Technologies*. O ST-506 era capaz de armazenar até 40MB e alcançar uma taxa de transferência de dados de 645KB/s. A conexão com o controlador era realizada através de um cabo de controle e outro de dados.

Com a introdução dos IBM *Personal Computers* (PCs) no mercado, diversas outras companhias começaram a fabricar discos rígidos que utilizavam os mesmos conectores e sinais utilizados no ST-506. A interface se popularizou, tornando-se um padrão no início da década de 80.

A primeira tentativa de imprimir melhoramentos às interfaces ST-506 surgiu em meados da década de 80 com o padrão *Enhanced Small Device Interface* (ESDI), desenvolvido por um consórcio de empresas lideradas pela *Maxtor Corporation*. Mantendo o cabeamento utilizado pelo ST-506, a interface ESDI moveu algumas das funções do controlador para o disco rígido, solucionando problemas de compatibilidade existentes no ST-506 e possibilitando taxas nominais de transferência de até 20MB/s. A interface foi generalizada para permitir a conexão de outros dispositivos como discos removíveis e fitas magnéticas, tendo inclusive tornado-se um padrão do *American National Sciences Institute* (ANSI).

Entretanto, no início da década de 90, o padrão ESDI caiu em desuso devido às complexidades do controlador e do cabeamento e também graças à maturidade dos padrões ATA e SCSI, que ofereciam maior confiabilidade e desempenho.

4. INTERFACE ATA [2,9]

Em 1986, a *Compaq Computer*, a *Western Digital* e uma divisão da *Control Data Corporation* desenvolveram, como precursor da interface *Advanced Technology* (AT), a *Advanced Technology Attachment* (ATA), que recebeu este nome por ter sido primeiramente adotada em computadores PC/AT.

Uma evolução subseqüente ocorreu na medida em que todos os computadores passaram a incluir discos rígidos. Os *chipsets* das placas-mãe trazem, desde então, controladores ATA embutidos, e os discos rígidos são, hoje, conectados diretamente ao barramento do PC. A interface ATA tornou-se, a partir de meados da década de 90, o padrão de conexão de discos rígidos mais utilizado em microcomputadores. Sua introdução no mercado proporcionou um novo patamar de desempenho, confiabilidade e compatibilidade.

4.1 Modos de Transferência

Os modos de transferência especificam de que forma e com qual velocidade ocorre a transferência de dados. Como a interface ATA é, essencialmente, um canal de comunicação, os modos de transferência devem ser suportados tanto pelos discos rígidos como pelo *Basic Input/Output System* (BIOS) da placa mãe. O BIOS é o software de mais baixo nível, executado pelo computador tão logo este é ligado, que possibilita o acesso a diversos dispositivos de hardware, inclusive ao controlador ATA.

4.1.1 PIO

O modo PIO (*Programmed* I/O) é uma forma de acesso a disco efetuada com ajuda da CPU. O padrão ATA estabelece cinco modos que diferem nas taxas em que os dados são transferidos. A tabela 1 mostra os cinco modos PIO, com suas respectivas taxas máximas de transferência, tempo de ciclo e o padrão ATA em que foi definido.

Tabela 1. Modos de transferência PIO

Modo	Taxa máxima de Transferência	Ciclo	Padrão
0	3.3MB/s	600ns	ATA
1	5.2MB/s	383ns	ATA
2	8.3MB/s	240ns	ATA
3	11.1MB/s	180ns	ATA-2
4	16.7MB/s	120ns	ATA-2

4.1.2 DMA

O modo DMA (*Direct Memory Access*) transfere dados entre disco e memória sem interferência da CPU, que fica liberada para realizar outras tarefas. O padrão original previa um único modo DMA: DMA 0. mais tarde foram incorporados os modos DMA 1 e DMA 2, os quais diferem apenas quanto às taxas máximas de transferência de dados. Muitos modos DMA foram definidos para a interface IDE/ATA. Tais modos foram agrupados em duas categorias. O primeiro conjunto de modos é denominado DMA *single word* enquanto que o segundo é denominado DMA *multiword.*

4.1.2.1 DMA single word

Quando estes modos são utilizados, cada transferência move apenas uma simples palavra de dados (uma palavra é um termo utilizado para definir dois bytes). Existem três modos DMA *single word*, todos definidos no padrão ATA original.

Tabela 2. Modos de transferência DMA *single word*

Modo	Taxa máxima de Transferência	Ciclo	Padrão
0	2.1MB/s	960ns	ATA
1	4.2MB/s	480ns	ATA
2	8.3MB/s	240ns	ATA

4.1.2.2 DMA multi word

Por questões de eficiência, os modos DMA *single word* foram, rapidamente, substituídos pelos modos denominados DMA *multi word*. A tabela 3 detalha tais modos.

Tabela 3. Modos de endereçamento DMA *multi word*

Modo	Taxa máxima de Transferência	Ciclo	Padrão
0	4.2MB/s	480ns	ATA
1	13.3MB/s	150ns	ATA-2
2	16.7MB/s	120ns	ATA-2

Como as transferências DMA *multiword* são mais eficientes e possuem maiores taxas máximas, os modos DMA *single word* foram rapidamente abandonados tão logo o padrão ATA-2 foi largamente adotado. Os modos *single word* foram, inclusive, removidos do padrão ATA na definição da especificação ATA-3. Portanto, todos os acessos DMA atuais (incluindo *Ultra* DMA) são *multiword*.

4.1.5 UDMA

Claramente, os discos rígidos estavam ficando cada vez mais rápidos e, obviamente, a taxa máxima do modo DMA *multi word* (16.7MB/s) tornou-se insuficiente para tais discos. Os engenheiros perceberam que a tarefa de elevar a taxa máxima de transferência não era fácil; aumentar a velocidade da interface poderia provocar problemas relacionados à interferência de transmissão. Então, ao invés de tornar a interface mais rápida, uma abordagem diferente foi desenvolvida. O resultado foi a criação de um novo tipo de modo de transferência DMA, chamada de modo Ultra DMA.

A primeira especificação do Ultra DMA foi definida no padrão ATA/ATAPI-4 e incluiu três modos: Ultra DMA 0, 1 e 2. Nos padrões ATA subseqüentes foram adicionados novos modos. A tabela 4 resume todos os modos Ultra DMA.

Tabela 4. Modos de endereçamento UDMA.

Modo	Taxa máxima de Transferência	Ciclo	Padrão
0	16.7MB/s	240ns	ATAPI-4
1	25.0MB/s	160ns	ATAPI-4
2	33.3MB/s	120ns	ATAPI-4
3	44.4MB/s	90ns	ATAPI-5
4	66.7MB/s	60ns	ATAPI-5
5	100.0MB/s	40ns	-
6	133.0MB/s	-	ATAPI-6

Na verdade, não existe uma única especificação para interfaces ATA, mas uma família de padrões que evoluíram ao longo do tempo, impulsionados pela necessidade de captar as constantes inovações tecnológicas.

4.2 ATA (ATA-1)

O primeiro documento de definição de interface ATA foi submetido para aprovação do ANSI em 1990, mas sendo aprovado e publicado apenas quatro ano depois no padrão X3.221-1994. Essa interface também é chamada de ATA-1, para distingui-la de suas sucessoras.

ATA-1 especifica três modos de transferência PIO: 0, 1 e 2. Também foram definidos no primeiro padrão ATA os modos DMA *single-word* 0, 1 e 2 e o modo DMA *multi-word* 0.

4.3 ATA-2

A primeira especificação ATA, contudo, mostrou-se inadequada para discos cada vez mais rápidos e que demandavam novas funcionalidades. O resultado foi o desenvolvimento de uma nova interface, a qual foi publicada em 1996 no padrão ANSI X3.279-1996 *AT Attachment Interface with Extensions* (ATA-2).

ATA-2 introduziu os modos PIO 3 e 4, e os modos DMA *multi-word* 1 e 2. Os termos "*Enhanced* IDE" (EIDE), "Fast IDE, "*Fast* ATA" e "*Fast* ATA-2" são usados no mercado como sinônimos do padrão ATA-2.

4.4 ATA-3

A especificação ATA-3, publicada no padrão ANSI X3.298-1997, é uma revisão dos padrões ATA a ATA-2. ATA-3 acrescenta ao padrão ATA uma tecnologia de auto-detecção de condições adversas e falhas conhecida como *Self- Monitoring Analysis and Reporting Technology* (SMART), e a capacidade de proteger os discos rígidos através de senhas (*Security Feature*).

4.5 ATA/ATAPI-4

A interface ATA foi originalmente concebida exclusivamente para discos-rígidos. Dispositivos como CD-ROMs e drives de fita utilizavam interfaces proprietárias ou interface de disquetes. Criou-se então uma extensão do protocolo ATA com suporte a esses demais periféricos: *AT Attachment Packet Interface* (ATAPI).

Internamente, ATAPI comunica-se com os dispositivos através de pacotes de comandos, fazendo com que eles se comportem como um disco ATA. A extensão ATAPI foi incorporada na quarta geração do padrão ATA, publicada em 1998 no padrão NCITS 317-1998, ou ATA/ATAPI-4.

Além dos modos de transferência especificados pelos padrões ATA anteriores, ATA/ATAPI-4 introduziu os modos UDMA 0, 1 e 2. Os modos UDMA 0 e 1 nunca chegaram a ser implementados pelos fabricantes de discos.

Os discos ATA/ATAPI-4 também são conhecidos no mercado como "Ultra ATA", "ATA/33", "Ultra ATA/33" e "Ultra DMA/33", por causa do *throughput* máximo de 33.3 MB/s. Neste padrão, alguns comandos obsoletos foram retirados e outros mais avançados foram adicionados. Além disso, erros de transmissão passaram a ser checados via código *Cyclical Reduncy Checking* (CRC) – protocolo para verificação de erros comuns, usado para garantir a integridade dos dados.

4.6 ATA/ATAPI-5

Em 2000 foi publicada a especificação ATA/ATAPI-5 no padrão NCITS 340-2000.

ATA/ATAPI-5 suporta os modos UDMA 3 e 4. Também devido às taxas máximas de transferências do modo UDMA 4, discos rígidos ATA/ATAPI-5 são popularmente denominados "ATA/66" ou "Ultra ATA/66".

Para garantir a confiabilidade de transmissões em freqüências mais altas, o padrão ATA/ATAPI-5 tornou obrigatório do uso de um cabo de 80 vias, que nada mais é do que o tradicional cabo de 40 vias no qual para cada via de sinal há uma via de aterramento, impedindo interferências entre as linhas. O uso deste cabo era opcional na especificação ATA/ATAPI-4.

4.7 ATA/ATAPI-6

Publicada em 2001 no padrão NCITS 347-2001, a interface ATA/ATAPI-6, também conhecida como "Ultra ATA/100" e "Ultra ATA/66+", suporta o modo UDMA 5 e disponibiliza recursos de controle automático de emissão de ruídos no disco rígido (*Hard Disk Noise Reduction*).

4.8 ATA/ATAPI-7

O padrão ATA/ATAPI-7 é a especificação mais recente da interface ATA. Publicada em 2002 no padrão NCITS 361-2002,

ela suporta o modo UDMA 6, sendo também conhecida como "Ultra ATA/133". ATA/ATAPI-8 está em desenvolvimento.

A despeito das crescentes taxas de transmissão dos modos UDMA introduzidos pelos padrões ATA mais recentes, até o momento não há discos rígidos capazes de sustentar transferências acima de 80 MB/s.

A tabela 5 mostra um resumo das informações a respeito do padrão ATA. Devido à sua retirada do padrão ATA, os modos DMA *single word* não estão representados na tabela.

Tabela 5. Resumo da especificação ATA.

Generation	Standard	Year	Speed	Key features
IDE		1986		Pre-standard
	ATA	1994		PIO modes 0,1,2 multiword DMA 0
EIDE	ATA-2	1996	16 MB/sec	PIO modes 3-4, multiword DMA modes 1-2, LBAs
	ATA-3	1997	16 MB/sec	SMART
	ATA/ATAPI-4	1998	33 MB/sec	Ultra DMA modes 0-2, CRC, overlap, queuing, 80-wire
Ultra DMA 66	ATA/ATAPI-5	2000	66 MB/sec	Ultra DMA mode 3-4
Ultra DMA 100	ATA/ATAPI-6	2001	100 MB/sec	Ultra DMA mode 5, 48-bit LBA
Ultra DMA 133	ATA/ATAPI-7	2002	133 MB/sec	Ultra DMA mode 6

5. INTERFACE SCSI [2,9]

Small Computer System Interface (SCSI) é um protocolo de nível mais alto que o ATA. Na verdade, apesar de ser comumente referida como uma interface, SCSI e mais do que isso: um barramento de sistema com controladores "inteligentes" no qual os dispositivos trabalham em conjunto para controlar o fluxo de dados no canal de comunicação. Além disso, SCSI não é um padrão concebido unicamente para discos rígidos, podendo ser utilizado como barramento de interconexão de periféricos dos mais diversos tipos. Portanto, ao contrário de ATA, SCSI engloba diversos padrões de cabeamento. Num barramento SCSI os dispositivos competem pelo controle do canal. Além disso, estão disponíveis comandos de alto nível através dos quais operações como rebobinar uma fita magnética ou formatar um disco rígido podem ser iniciadas em um dispositivo sem qualquer intervenção subseqüente da CPU, o que representa ganho de desempenho significativo para sistemas multitarefa.

O desenvolvimento do padrão SCSI tem origem em 1979 quando a *Shugart Associates* lançou a *Shugart Associates Systems Interface* (SASI) – uma predecessora rudimentar de SCSI que implementava apenas uma pequena parte das funcionalidades das interfaces modernas, cujos objetivos eram suportar endereçamento LBA ao invés de CHS, realizar transferências paralelas de 8 bits e abolir as linhas exclusivas de controles em favor de comandos genéricos executados nos mesmos canais de transmissão de dados.

As interfaces SCSI eram capazes de transmissões de no máximo 1,5 MB/s. Entretanto, elas representaram uma idéia inovadora por serem a primeira proposta de uma interface inteligente de armazenamento para computadores de pequeno porte.

O padrão conta com 3 nomes comumente utilizados: SCSI-1, SCSI-2 e SCSI-3. Todos vêm sendo modularizados com funcionalidades (que também proporcionam diversos nomes ao SCSI) incluídas ou não pelos fabricantes.

5.1 – SCSI 1

Em 1981, a *Sughart* uniu-se à *NCR Corporation* para padronizar a interface SASI. No ano seguinte, as duas companhias formaram o comitê técnico ANSI X3T9.2. Diversas mudanças foram feitas na definição SASI para incorporar novas funcionalidades e obter melhorias de desempenho. O resultado deste trabalho resultou com a publicação, em 1986, do padrão ANSI X3.131-1986 – a primeira especificação formal da interface SCSI.

Conhecido como SCSI, esse padrão é a definição básica do barramento SCSI, seu protocolo de sinalização e um conjunto de comandos de 6 e 10 bytes. Ele especifica um barramento de 8 bits (sendo 1 bit de paridade) no qual podem ser conectados até 8 dispositivos. Esse barramento é capaz de transferir até 3.5MB/s em modo assíncrono e até 5MB/s em modo síncrono, e o tamanho máximo dos cabos de conexão é limitado a 6cm.

O suporte nativo ao padrão foi adotado pala maioria dos computadores *Apple Macintosh*, que vinham de fábrica com uma porta SCSI para conexão de dispositivos como drives de disco e impressoras. Muitos dos primeiros discos SCSI eram, na verdade, discos ST-506 ou ESDI com um adaptador SCSI embutido.

O padrão definiu apenas especificações básicas como tamanho de cabo, características da sinalização, comandos e modos de transferência. No entanto, surgiram dificuldades para sua popularização, já que não havia garantias de que dispositivos diferentes poderiam trabalhar em conjunto. Atualmente o padrão está obsoleto, tendo sido, inclusive, removido pela ANSI.

5.2 – SCSI-2

Um ano antes da publicação do padrão SCSI-1, iniciou-se o trabalho da especificação SCSI-2, que foi aprovada em 1994 e publicada como o padrão ANSI X3.131-1994. na verdade, o padrão foi originalmente liberado em 1990 com X3.131.1990, mas foi recolhido para novas modificações e somente foi aprovado formalmente quatro anos mais tarde.

Para solucionar os problemas de incompatibilidade do padrão anterior, SCSI-2 especifica um subconjunto mínimo de comandos que deve ser implementado por todos os dispositivos, o *Common Command Set* (CCS). Os objetivos mais importantes dessa evolução incluíam aumento do desempenho e confiabilidade e a formalização dos comandos SSI, principalmente após a confusão que surgiu com as implementações não padronizadas do SCSI original.

O padrão SCSI-2 oferece novos comandos para possibilitar o suporte a outros dispositivos, como drives de CD-ROM e *scanners*, estendendo o conjunto inicial de comando que era focado na comunicação com discos rígidos. Outra nova funcionalidade consiste na capacidade de executar múltiplas requisições de entrada e saída entre os dispositivos do barramento de forma simultânea. Com a especificação do padrão SCSI-2, surgiram também as seguintes variantes:

- *Fast* SCSI, que dobrou a taxa máxima de transmissão de dados com a redução do tamanho dos cabos para 3m, alcançando 10MB/s.
- *Wide* SCSI, que dobrou o tamanho do barramento para 16 bits, possibilitando a conexão de até 16 dispositivos.
- *Fast Wide* SCSI que, reunindo as modificações das duas variantes anteriores, ainda viabilizou taxas de transmissão de até 20MB/s.

5.3 – SCSI-3

O trabalho de definição do padrão SCSI-3 começou em 1992, antes mesmo do lançamento oficial de SCSI-2, cuja especificação consumiu cerca de 8 anos de trabalho. Diante das dificuldades de se formalizar um padrão tão abrangente e da proliferação de extensões proprietárias que ocorre no mercado quando inexistem padrões oficiais, optou-se por dividir SCSI-3 em uma coleção de padrões distintos, porém correlatos, que

poderiam ser desenvolvidos mais rapidamente e de forma razoavelmente independente. O relacionamento entre esses padrões é definido pelo SCSI-3 *Architecture Model* (SAM) – documento que organiza e classifica os vários padrões que existem no SCSI-3. A forma mais implementada de SCSI, a qual era anteriormente conhecida apenas como SCSI, tornou-se a SCSI-3 *Parallel Interface* (SPI) no SCSI-3.

Existem, atualmente, várias versões de SPI, cada uma definindo novas características e velocidades de transmissão para dispositivos SCSI convencionais paralelos.

5.3.1 – SPI-1

Com a generalização do padrão SCSI, todas as funcionalidades "tradicionais" do barramento SCSI paralelo para discos rígidos passaram a ser conhecidas como SCSI-3 *Parallel Interface* (SPI), e foram normalizadas num conjunto de documentos de especificação do protocolo de comunicação e das características da camada física.

Do padrão SPI-1 originaram-se duas variantes:

- *Ultra* SCSI, que dobrou a freqüência de transmissão *Fast* SCSI, alcançando até 20MB/s
- *Ultra Wide* SCSI, que dobrou a freqüência de transmissão de *Fast Wide* SCSI, alcançando até 40MB/s

5.3.2 – SPI-2

A segunda geração da SPI, publicada em 1999, trouxe melhorias de desempenho e novas características eletrônicas que foram consolidadas no mercado através das implementações "Ultra2 SCSI", que dobrou a freqüência de transmissão de Ultra SCSI, alcançando até 40MB/s e Ultra2 Wide SCSI, que dobrou a freqüência de transmissão de Ultra Wide SCSI, alcançando até 80MB/s.

5.3.3 – SPI-3

A SPI-3 tornou-se um padrão oficial em 2001. A confiabilidade da comunicação foi melhorada nessa versão da interface com a introdução da checagem de erros de transmissão via CRC. Ganhos de desempenho foram obtidos com recursos como:

- Empacotamento de comandos (*packetization*): reduz o overhead associado com cada transferência de dados
- *Quick Arbitration and Selection* (QAS): representa uma mudança na forma como dispositivos determinam qual deles tem o controle do barramento SCSI.
- Validação de domínio (*domain validation*): aumenta a robustez do processo pelo qual diferentes dispositivos SCSI determinam uma taxa de transferência de dados ótima.

Além disso, a taxa máxima de transmissão foi dobrada novamente, alcançando até 160MB/s. Dessa vez, a freqüência do barramento não foi aumentada: os dados passaram a ser transmitidos tanto na borda de subida quanto na borda de descida do *clock*.

Os discos rígidos que implementam a interface SPI-3 foram denominados genericamente pela SCSI *trade Association* como Ultra3 SCSI, mas devido ao fato desse termo não os obrigar a terem os mesmos conjuntos de funcionalidades, eles são mais comumente conhecidas no mercado pelas variantes Ultra160 SCSI (até 160MB/s) e Ultra160+SCSI.

5.3.4 – SPI-4

Publicada em 2002 no padrão NCITS 362-2002, a interface SPI-4 dobra novamente a freqüência do barramento, passando a suportar o modo Ultra320 SCSI (até 320MB/s).

A figura mostra a diferença de taxas de transmissão entre os 3 padrões SCSI (Incluindo as variantes pertencentes ao padrão SCSI-3).

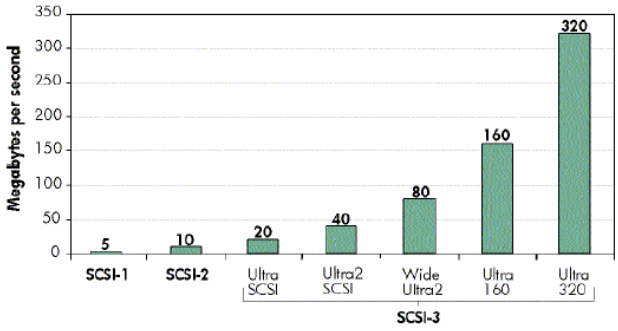


Figura 3 – Diferenças de taxas de transmissão

A tabela resume as informações a respeito da especificação SCSI e de suas variantes.

Tabela 6. Resumo da especificação SCSI.

Interconnect	Standard	Year	Speed	Key features
SASI		1979		Shugart Associates
SCSI-1	SCSI-1	1986	3 MB/sec 5 MB/sec	Asynchronous Synchronous
SCSI-2	SCSI-2	1990	10 MB/sec	CCS
SCSI-3	Split command sets, transport protocols, and physical interfaces into separate standards			
Fast	SPI	1992	20 MB/sec	
Ultra	SPI-1	1995	40 MB/sec	
Ultra 2	SPI-2	1999	80 MB/sec	
Ultra 3	SPI-3	2001	160 MB/sec	Packetized , QAS
Ultra 320	SPI-4	2002	320 MB/sec	

6. INTERFACE SERIAL ATA [1,2,4]

O padrão Serial ATA ou SATA (Serial Advanced Technology Attachment) é uma tecnologia para discos rígidos que surgiu no mercado no ano de 2000 para substituir a tradicional interface PATA (Parallel ATA), também conhecida como ATA ou IDE.

O nome de ambas as tecnologias já indica a principal diferença entre elas. A tecnologia PATA faz transferência de dados de forma paralela, enquanto que no SATA a transmissão é em série.

Nas transmissões paralelas, os dados fluem byte a byte entre o micro e o periférico. Como um byte é formado por oito bits, micro e dispositivo são ligados por cabos com pelo menos oito condutores, uma para cada bit. Já nas portas seriais os dados fluem bit a bit, um após o outro. Na origem, cada byte é desmontado (serializado) e os bits que o formam são transmitidos sequencialmente. No destino, são recebidos e remontados (desserializados) para reconstituir o byte original. Para isso, bastam dois condutores: uma para transportar os bits, outro para funcionar como terra.

A transmissão paralela é, evidentemente, mais rápida, mas possui inconvenientes como necessidade de sincronia na transmissão dos oito bits de um byte, cabos mais pesados e sujeitos a interferências, hardware mais caro e ruído – perda de dados ocasionada por interferência.

Para lidar com o problema de ruído nos discos PATA, os fabricantes utilizam certos mecanismos; um deles é recomendar a utilização de um cabo IDE – cabo que liga o disco rígido á placa-mãe do computador – com 80 vias ao invés dos tradicionais cabos de 40 vias. As vias a mais atuam como uma espécie de blindagem contra ruídos. No caso do padrão SATA, tais ruídos praticamente não existem, mesmo porque seu cabo de conexão ao computador possui apenas 4 vias e também é blindado. Isto acaba trazendo outro ponto de vantagem ao SATA, pois como o

cabo tem dimensão reduzida, o espaço interno do computador é melhor aproveitado, facilitando inclusive a circulação de ar.

Há outra característica interessante no padrão SATA: discos rígidos que utilizam essa tecnologia não precisam de *jumpers* para identificar o disco *master* (mestre ou primário) ou *slave* (escravo ou secundário). Isto ocorre porque cada dispositivo usa um único canal de comunicação (ao contrário do PATA que permite até dois dispositivos por canal), atrelando sua capacidade total a um único disco. No entanto, para não haver incompatibilidade com dispositivos Paralell ATA, é possível instalar esses aparelhos com interfaces seriais através de placas adaptadoras. Muitos fabricantes de placas-mãe oferecem estas com ambas as interfaces.

Outra novidade é a possibilidade de uso da técnica "*hot-swap*", que torna possível a troca de um dispositivo Serial ATA com o computador ligado. Tal recurso é muito útil em servidores que precisam de manutenção/reparos, mas não podem parar de funcionar.

A primeira versão do padrão SATA trabalha com taxa máxima de transferência de dados de 150MB/s. Essa versão recebeu os seguintes nomes: SATA 150, SATA 1.0, SATA 1,5 Gbps ou simplesmente SATA I.

Não demorou muito para surgir uma versão denominada SATA II, cuja principal característica é a velocidade de transmissão de dados a 300MB/s, o dobro do SATA I.

È necessário fazer uma observação quanto ao aspecto de velocidade de transmissão. Na prática, dificilmente os valores mencionados (150MB/s e 300MB/s) são alcançados. Essas taxas indicam a capacidade máxima de transmissão de dados entro o disco rígido e sua controladora (presente na placa-mãe), mas dificilmente são usadas em sua totalidade, já que isso depende de uma combinação de fatores como conteúdo da memória, processamento, tecnologias aplicadas no disco rígido, etc.

Há outra ressalva importante a ser feita: a entidade que controla o padrão SATA (formada por um grupo de fabricantes e empresas relacionadas) chama-se, atualmente, SATA-IO (SATA International Organization). O problema é que o nome anterior dessa organização era SATA-II, o que gerava certa confusão com a segunda versão do SATA. Aproveitando essa situação, muitos fabricantes inseriram selos da SATA-II em seus discos rígidos SATA I para confundir os usuários, fazendo-os pensar que tais discos eram, na verdade, da segunda geração de discos SATA.

6.1 Tecnologias relacionadas ao SATA

Os fabricantes de discos SATA podem adicionar tecnologias em seus produtos para diferenciá-los no mercado ou para atender a uma determinada demanda, o que significa que certo recurso não é obrigatório em um disco rígido só por este ser SATA.

6.1.1 NCQ

NCQ (*Native Command Queuing*) é tido como obrigatório no SATA II, mas é opcional no padrão SATA I. Trata-se de uma tecnologia que permite ao disco rígido organizar as solicitações de gravação ou leitura de dados numa ordem que faz com que as cabeças se movimentem o mínimo possível, aumentando, pelo menos teoricamente, o desempenho do dispositivo e sua vida útil. Para usufruir dessa tecnologia, não só o disco tem que ser compatível com ela, mas também a placa-mãe, através de uma controladora apropriada.

6.1.2 xSATA

Basicamente, o xSATA é uma tecnologia que permite ao disco rígido utilizar menos energia elétrica. Para isso, o disco rígido pode assumir três estados: ativo, parcialmente ativo ou inativo. Com isso, o disco rígido vai receber energia de acordo com sua utilização no momento.

6.1.3 *Staggered spin-up*

Esse é um recurso muito útil em sistemas RAID, por exemplo, pois permite ativar ou desativar discos rígidos trabalhando em conjunto sem interferir no funcionamento do grupo de discos. Além disso, a tecnologia SSU também melhora a distribuição de energia entre os discos.

6.1.4 *Hot plug*

Em sua essência, a tecnologia Hot Plug permite conectar o disco ao computador com o sistema operacional em funcionamento. Este é um recurso muito usado em discos do tipo removível.

6.2 Conectores e cabos

Como dito, os cabos do padrão SATA são diferentes dos cabos do PATA, justamente por utilizar apenas quatro vias. Como conseqüência, seu conector também é menor, como ilustra a figura.

Além do cabo de dados, o conector do cabo de alimentação também é diferente no padrão SATA. Uma característica importante desse conector é que sua retirada do disco rígido é mais fácil, se comparado ao padrão PATA.

7. INTERFACE SAS [7]

Serial Attached SCSI é um novo padrão SCSI onde a comunicação é feita em série, em vez de em paralela, como no SCSI tradicional.

O trabalho de especificação do SAS teve inicio no ano 2001 e ficou sob responsabilidade das empresas Compaq/HP, LSI, Logic, Maxtor e Seagate Technologies. A especificação inicial de 2004 definiu uma taxa de transferência de 3Gbps (300MB/s). Posteriormente foram definidas taxas de 6Gbps (600MB/s) e 1.2GBps (1200MB/s). O padrão SAS permite total compatibilidade com o padrão Serial ATA (SATA). Enquanto o SATA é destinado ao mercado de desktops enquanto que o padrão SAS é destinado ao mercado de servidores.

Sua grande vantagem em relação ao padrão Serial SCSI existente atualmente (Fibre Channel, FC) é que ele permite o uso de discos de várias taxas de transmissão, usando a taxa máxima do dispositivo. O Fibre Channel nivela por baixo, ou seja, se no sistema há um disco lento misturado com outros rápidos, o barramento passa a operar na velocidade do dispositivo mais lento, comprometendo o desempenho do sistema como um todo.

Assim como o padrão SATA, SAS é hot swap, permitindo a troca de discos rígidos mesmo com o micro ligado. Outras características do SAS são:

- Melhoria no desempenho e confiabilidade
- Capacidade de redundância de cabos no mesmo disco
- Interface serial ponto-a-ponto de simples cabeamento
- Possibilidade de aumento de configuração e desempenho
- Capacidade de expansão e atualização
- Possibilidade de clientes e usuários escolherem entre discos SAS de dupla redundância de cabos e alto desempenho ou convencionais discos SATA de alto desempenho e baixo custo no mesmo sistema.

8. INTERFACE SSA [7]

Serial Storage Architecture (SSA) é um protocolo de transmissão serial usado para conectar discos rígidos a servidores capaz de conectar até 192 discos rígidos com taxas de transmissão de até 80MB/s. Ela é tolerante a falhas no sentido de que defeitos em um único cabo não interrompem o acesso aos dados. SSA caiu em desuso com o surgimento do padrão *Fibre Channel*.

9. INTERFACE FIBRE CHANNEL [7]

Fibre Channel é tanto um padrão de comunicação quanto um

protocolo de transporte aberto - conforme definido por normas ANSI (Comitê X3T11) -, implementado em fibras óticas ou fiações de cobre, sendo, portanto, um padrão para meios físicos de comunicação. Este padrão é amplamente utilizado na conectividade física do que é conhecido hoje como "*SANs – Storage Area Networks"*, ou seja, redes especializadas na interconexão de dispositivos de armazenamento a servidores. A palavra "*Fibre*" foi criada pelo comitê quando o uso de fiações de cobre foi introduzido no padrão. Originalmente, referia-se somente a fibras óticas.

Fibre Channel Aribtrated Loop ou FC-AL é uma das topologias implementadas pelas normas *Fibre* (sendo as outras a *Point-to-Point* e o *Fibre Channel Switched Fabric*, ou FC-SW). Todas estas topologias são utilizadas para interconectar dispositivos de armazenamento à seus servidores, atendendo a um grande número de dispositivos. Atualmente, a FC-AL é usada na maior parte dos subsistemas de armazenamento para MainFrame.

Uma vez conectados fisicamente, os dispositivos participantes de uma rede FC-AL iniciam o que é chamado de LIP *(Loop Initialization Primitive)*, durante o qual os mesmos são identificados. Após o LIP, a rede entra em um estado de gerenciamento, controlado pelo dispositivo de mais baixo endereço físico, o qual foi identificado por uma outra primitiva, chamada LISM (*Loop Initialization Select Master*). No caso dos subsistemas de grande porte, esta função cabe à placa adaptadora.

Após terem sido executadas as primitivas de inicialização, qualquer dispositivo que queira iniciar uma conexão terá, a semelhança do padrão SCSI, que arbitrar. Caso haja mais de um dispositivo tentando a arbitragem ao mesmo tempo, o que tiver o menor endereço físico (AL-PA – *Arbitrated Loop Physical Address*) vence, ganhando controle sobre o loop, e podendo estabelecer uma conexão com outro nó e utilizar, durante este tempo, toda a largura de banda disponível para sua transmissão. A maior parte das implementações FC-AL conta com o Fairness Algorithm, o que impede que dispositivos de mais alta prioridade (menor AL-PA) possam monopolizar a conexão física.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Anderson, D.; Dykes, J.; Riedel, E. ***More Than an Interface – SCSI vs. ATA***. *In Proceedings of the 2nd Annual Conference on File and Storage Technology* (FAST), March 2003.

[2] Patterson, D., Hennessy, J. ***Computer Organization and Design: the hardware/software interface***. Morgan Kaufmann Publishers, 2005.

[3] Bosch, P. *Mixed Media File Systems*. PhD thesis. University of Twente, 1999.

[4] Intel, 2005. ***Disk Interface techonology – Quick reference Guide***.

[5] Zelenovsky, R.; Mendonça, A. **PC: um guia de Hardware e interfaceamento**. 3. ed. Rio de Janeiro, 2002.

[6] Kozierok, C. *The PC Guide – Hard Disk Drives*. Disponível em <http://www.pcguide.com/ref/hdd/>

[7] IDC, 2005. ***Evolution in Hard DiskDrive Technology: SAS an d SATA, White Paper.*** Disponível em <http://www.idc.com>

[8] Guia do Hardware, 2005. Disponível em <http://www.guiadohardware.com.br>

[9] HP Invent. ***Serial Attached SCSI, General Overview.*** Disponível em <http://www.hp.com>

[10] Hennesy, J. L.; Patterson D. A. Arquitetura de Computadores, Uma Abordagem Quantitativa. 3. ed.

Trace Cache

Mário Luiz Rodrigues Oliveira - RA 066254
Instituto de Computação/UNICAMP
Av. Albert Einstein, 1251
Campinas/SP, Brasil
mario.oliveira@students.ic.unicamp.br

RESUMO

Processadores superescalares demandam um aumento no *bandwidth* do mecanismo de busca de instruções para melhorar o desempenho. Uma cache de instruções convencional não é capaz de suprir esses processadores com um grande número de instruções por ciclo, pois na maioria das vezes não dispõe de quantidades razoáveis de instruções em um mesmo bloco básico. Uma solução para tal problema é adicionar à cache de instruções convencional uma *trace cache*, a qual captura uma seqüência dinâmica de instruções e as armazena na *trace cache*. Essa técnica foi proposta, em 1996, por *Eric Rotenberg*, o qual mostrou que ela melhora o desempenho em 28%, em média, quando comparada à mecanismos de busca seqüenciais. Para auferir esse desempenho utilizou-se *benchmarks* com inteiros. O presente artigo mostra a proposta inicial de Robenterg, o uso de *trace cache* no processador Pentium 4 e algumas idéias para melhorar o desempenho da técnica de *trace cache*.

Termos gerais

Cache, Desempenho

Palavras-chave

Trace cache, memória cache, desempenho

1. INTRODUÇÃO

A organização de computadores de superescalares é dividida em dois mecanismos: o mecanismo de busca de instruções e o mecanismo de execução de instruções. Entre esses dois mecanismos pode-se colocar *buffers*, tais como filas ou estações de reservas. Conceitualmente, o mecanismo de busca atua como um produtor, o qual busca, decodifica e coloca as instruções no *buffer* e o mecanismo de execução atua como um consumidor, o qual remove as instruções do *buffer* e as executa, de acordo com a disponibilidade de recursos e dados.

Para melhorar o desempenho de processadores superescalares deve-se empregar técnicas agressivas que explorem paralelismo no nível de instrução (ILP) afim de executar várias instruções no mesmo ciclo. Entretanto para tirar proveito do ILP, deve-se ter o máximo possível de instruções no *buffer* existente entre os mecanismos de busca e execução.

Não haveria problemas se esse *buffer* fosse preenchido com instruções de um bloco básico, uma vez que todas as instruções do bloco básico podem ser executadas em paralelo. Um bloco básico é definido como um trecho do programa que não apresenta instruções de desvio, a não ser eventualmente a última instrução e assim, inexistindo conflitos estruturais e de dados, um bloco básico pode ser executado em paralelo aumento o desempenho. Todavia, a quantidade de instruções em um bloco básico é pequena. A tabela 1, adaptada de [7], mostra que o bloco básico, para programa de inteiros, possue em média 4 ou 5 instruções e dessa forma o uso de ILP dentro de apenas um bloco básico é limitado. Para melhorar o desempenho deve-se prover ao mecanismo de execução mais de um bloco básico por ciclo. A introdução de múltiplos preditores de *branches* por ciclo permite alimentar o *buffer* com múltiplos blocos básicos por ciclo e dessa forma consegue-se uma melhoria no desempenho do processador.

Benchmark	**Branch Tomados %**	Tam. médio do bloco	Número de instruções entre branch tomados
Eqntoot	86.2 %	4.20	4.87
Espresso	63.8 %	4.24	6.65
Xlisp	64.7 %	4.34	6.70
Gcc	67.7 %	4.65	6.88
Sc	70.2 %	4.71	6.71
Compress	60.9 %	5.39	8.85

Table 1: Estatísticas de branch e blocos básicos

O restante desse texto está organizado em 6 seções. Na próxima seção explica-se o que é e como funciona a técnica *trace cache*. Na seção 3 mostra-se o uso de *trace cache* no Pentium 4. Na seção 4 apresenta-se a melhoria de desempenho obtida com a adoção da técnica *trace cache*. A conclusão é apresentada na seção 5 e as referências na seção 6.

2. TRACE CACHE

O trabalho da unidade de busca é alimentar a unidade de decodificação. Entretanto as instruções são colocadas na

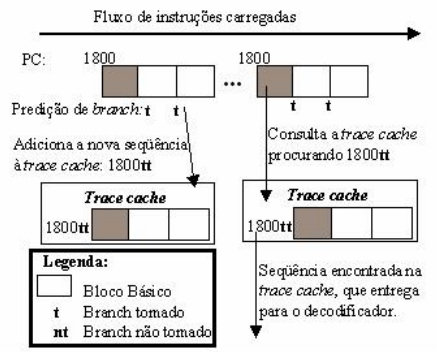


Figure 1: Visão geral do comportamento de uma *trace chace*

cache de instruções na ordem de compilação, o que é favorável a códigos que não tenham *branchs* tomados ou que possuem blocos básicos compostos por muitas instruções. Porém, como visto na tabela 1, esse não é o caso de programas de inteiros.

Uma forma de aumentar a quantidade de instruções disponíveis ao decodificador é através de *trace cache*, técnica proposta por [7] em 1996. A *trace cache* é uma cache adicional à cache de instruções que captura a seqüência dinâmica de instruções de um programa. Uma *trace* é uma seqüência de no máximo **n** instruções e no máximo **m** blocos básicos, iniciando em qualquer ponto da seqüência dinâmica de instruções. O limite **n** é o tamanho da linha da *trace cache* e **m** é o *throughput* do preditor de *branches*. Uma *trace* é completamente especificada através de um endereço inicial dado pelo *Program Counter - PC* e de uma seqüência de saídas dos **m - 1** preditores de *branches*, os quais indicam o caminho a ser seguido, ou seja, se o *branch* foi tomado ou não tomado. A figura 1, retirada de [6], mostra uma visão geral do comportamento de uma *trace cache*.

A *trace cache* é preenchida conforme a execução do programa, a cada *trace* diferente encontrada uma nova linha é alacoda na *trace cahce*. É interessante notar que para um mesmo PC pode-se ter mais de uma linha alocada se a seqüência de saída do preditor de *branch* for diferente. Algumas implementações transformam isso em duas linhas na *trace cache*com o mesmo PC, porém com *bits* diferentes para indicar o novo caminho a ser seguido. Outras implementações permitem apenas uma linha com um determinado PC, nesse caso a *trace* antiga é substituída pela nova.

Ao encontrar o mesmo *trace* novamente, a *trace cache* entrega ao decodificador uma linha, ou seja, todas as instruções constantes em uma *trace*. Caso contrário as instruções são buscadas na cache de instruções convencional.

Pode-se perceber pelo mecanismo de funcionamento da *trace cache* que um bom preditor de *branch* é necessário para se obter bom desempenho, pois a cada predição errada são perdidos vários ciclos até que a instrução seja carregada da memória para cache e dessa para o decodificador do processador.

A *trace cache* apresenta, assim como outros tipos de caches, tamanho limitado e dessa forma deve-se estabelecer algum critério de substituição para as linhas da *trace cache*. Assim, em sua implementação, são usados *bits* adicionais em cada linha e um desses *bits* é o *bit* de validade indicando quais linhas são válidas e um algoritmo de substituição, como por exemplo, o algoritmo LRU [6].

Uma desvantagem no uso de *trace cache* é o fato dela armazenar as mesmas instruções várias vezes na cache de instruções, pois *branches* que fazem escolhas diferentes resultam na inclusão das mesmas instruções como partes de *traces* diferentes, cada um deles ocupando espaço na *trace cache*[3].

3. TRACE CACHE NO PENTIUM 4

Algumas arquiteturas substituiram a cache de instruções convencional por uma *trace cache*, como é o caso dos processadores da *Intel* baseados na micro-arquitetura *NetBurst*, dentre eles o Pentium 4. Esses processadores executam micro-operações que são derivadas do conjunto de instruções IA-32 e por sua característica superescalar executam até três instruções em um ciclo de clock [3] [4].

O uso de *trace cache* no Pentium 4 apresenta vantagens, tais como:

- cada linha da *trace cache* é preenchida com um conjunto de instruções pertencentes a mais de um bloco básico;
- as instruções são armazenadas decodificadas.

A cache nível 1 do Pentium 4 é uma *trace cache* e entrega três micro-operações por ciclo de clock para o mecanismo de execução. Ela é capaz de armazenar até 12.000 micro-operações decodificadas e possui uma taxa de acerto similar à uma cache de instruções convencional com tamanho entre 8 Kbytes e 16 Kbytes [4].

4. MELHORIAS NO DESEMPENHO

O alto desempenho em processadores superescalares é obtido através da execução de várias instruções em paralelo. Para tanto duplicam-se várias unidades funcionais e quanto mais dessas unidades funcionais forem usadas ao mesmo tempo, melhor o desempenho conseguido.

Como visto na Seção 2, *trace cache* é uma proposta para aumentar a quantidade de instruções entregues ao mecanismo

de execução do processador em cada ciclo. Em [7], o ganho médio obtido com o uso de *trace cache* foi de 28% em *benchmarks* com inteiros em comparação com alguns mecanismos convencionais de busca, a saber:

- buscar um bloco básico por ciclo de clock;
- buscar três blocos básicos por ciclo de clock;
- *Branch Address Cache*, maiores detalhes sobre essa técnica podem ser encontrados em [8];
- *Collapsing Buffer*, detalhes dessa técnica podem ser obtidos em [2].

Para auferir essa porcentagem de ganho utilizou-se os *benchmarks* de inteiros SPEC92 e IBS. Robenterg notou, também, que o desempenho da *trace cache* pode ser melhorado aumentando o seu tamanho e/ou a sua associatividade.

Outros autores trabalhando com a técnica de *trace cache* apresentaram novas idéias para melhorar o seu desempenho, alguns trabalhos nesse sentido são:

- *trace cache* seletiva;
- *trace cache* baseada em blocos.

A *trace cache* seletiva é descrita em [5] e sua ideía principal é construir apenas os *traces* com maior freqüência, ou seja, os *traces* que apresentam a maior porcentagem de acertos.

Em [1] é apresentada a *trace cache* baseada em blocos, a qual armazena ponteiros para os blocos que constituem a *trace* ao invés de guardar as próprias instruções da *trace* explicitamente. Tais ponteiros são armazenados em uma tabela de *trace*.

5. CONCLUSÃO

A *trace cache* foi proposta como uma forma de aumentar o *bandwidth* no mecanismo de busca dos processadores superescalares, os quais necessitam de um grande número de instruções decodificadas por ciclo, a fim de apresentarem um ganho significativo de desempenho. E nesse sentido, a técnica de *trace cache* apresentou-se como uma boa solução para o problema.

Cumpre salientar que o desempenho desse tipo de cache está associado a um bom preditor de *branch*, pois quanto melhor os resultados dos preditores de *branches*, melhores os resultados obtidos pela *trace cache*. Dado que atualmente a quantidade de trabalhos na área de predição de *branches* é grande e apresenta resultados muito bons, tal fator não torna-se um empecilho à aplicação da técnica de *trace cache*.

Por fim, nota-se a utlização dessa técnica em novos processadores, como é o caso do Pentium 4 da *Intel*.

6. REFERÊNCIAS

[1] Bryan Black, Bohuslav Rychlik, and John Paul Shen. The block-based trace cache. *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 196 – 207, Maio 1999.

[2] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. Optimization of instruction fetch mechanisms for high issue rates. *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 333 – 344, Junho 1995.

[3] John L. Hennessy and David A. Patterson. *Arquitetura de Computadores: Uma Abordagem Quantitativa*. Editora Campus, São Paulo, 2003.

[4] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the pentium 4 processor, Junho 2006.

[5] Jie S. Hu, Mary Jane Irwin, N. Vijaykrishnan, and Mahmut Kandemir. Selective trace cache: A low power and high performance fetch mechanism, Junho 2006.

[6] Danilo Lacerda. Trace caches: alternativa inteligente à cache de instruções, Junho 2006.

[7] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 24 – 35, Dezembro 1996.

[8] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. *Proceedings of the 7th International Conference on Supercomputing*, pages 67 – 76, Julho 1993.

Software Pipelining

Carla Geovana do Nascimento Macário
RA: 895063

RESUMO

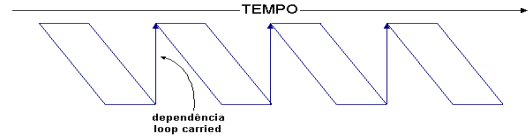
Software pipelining constitui uma das principais técnicas utilizadas pelos compiladores para aumentar o paralelismo na excução de programas. Isto deve-se ao fato de que, em geral, o tempo de execução de loops domina o tempo de execução de um programa. Portanto, atenção especial é direcionada para melhorar o seu desempenho. Este trabalho apresenta um visão geral da técnica de software pipelining, os principais conceitos e desafios envolvidos, trazendotambém uma breve descrição dos principais algoritmos existentes.

Palavras-Chave
paralelismo, software pipelining, modulo scheduling, identificação do kernel.

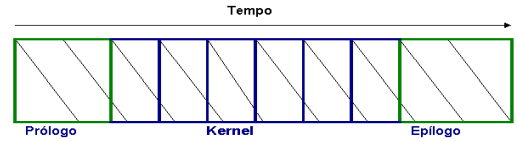
1.INTRODUÇÃO

Cada vez mais busca-se por melhorar o desempenho na execução dos programas e a exploração estática de paralelismo em nível de instrução (ILP) é uma das chaves para isso. Neste caso, o compilador, mediante uma análise prévia, promove um conjunto de transformações capazes de expor mais paralelismo ao código em questão, de maneira a usar diferentes unidades de processamento disponíveis.

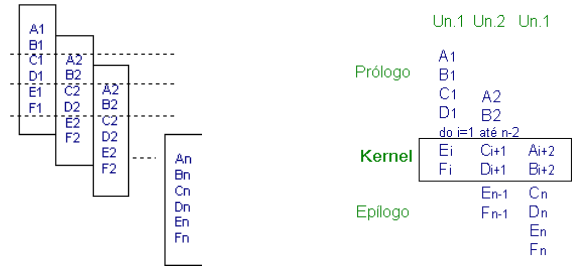
Diversas são as transformações promovidas e dentre elas merecem destaque as que buscam paralelizar os loops, pois em geral o tempo de execução de loops domina o tempo de execução de um programa. Técnicas para isso são o *Loop Unrolling* e o *Software Pipeline*. Na primeira é feito um desenrolamento (desdobramento) do código que compõe o loop, repetindo-o, com os devidos ajustes, tantas vezes quantas forem suas iterações, tornando-o um programa seqüencial. Como limitações tem-se o aumento do tamanho do código e possibilidade de falha em registradores, já que é alta a probabilidade de não ser possível alocar registradores para todas as iteraçõres desdobradas. A segunda técnica, o software pipelining reorganiza as operações do loop de forma a permitir que um conjunto destas operações seja executado em paralelo, promovendo um desenrolamento simbólico e infinito. Com isso, cria-se um novo corpo para o loop, denominado k*ernel* ou *pattern,* que é a parte a ser paralelizada. Considera, então, que embora as instruções ABC que compõem uma iteração do loop possam ser paralelizadas entre si, mais paralelismo pode ser obtido se todas as iterações do loop também puderem ser paralelizadas, numa execução sobreposta, com uma nova iteração tendo sua execução iniciada em intervalos regulares. Para isso leva-se em conta que um loop $\{ABC\}^n$, onde n é o número de iterações a serem executadas, pode ser escrito na forma $A\{BCA\}^{n-1}BC$. Embora as operações não mudem, e nem o número de vezes que serão executadas, o corpo do loop passa a conter operações de provenientes de diferentes iterações. O novo loop passa a ser composto de um prólogo, operações iniciais para preencher o pipeline , kernel e epílogo, instruções para esvaziar o pipeline, respectivamente A, BCA e BC no exemplo dado. A figura 1 ilustra o efeito de uso da técnica software pipelining.



a) loop sem software pipeling



b) loop com software pipeling



c) escalonamento do software pipeling

Figura 1 – Software Pipeling

Este trabalho contém uma breve descrição apresenta uma visão geral desta técnica, seus desafios e os principais algoritmos desenvolvidos para sua implementação. A maior parte do texto foi baseada no survey de Allan [Allan et al 1995] que contém uma descrição detalhada da técnica, conceitos envolvidos e algoritmos existentes

2.SOFTWARE PIPELING: VISÃO GERAL

Software pipelining é um excelente método para promover paralelismo em loops, tendo sido motivado pelo trabalho de Patel para escalonamento de hardware pipelining [Patel & Davidson 1976]. Sua viabilidade deu-se com a tecnologia de processadores paralelos, sendo desenvolvido por Rau e Aiken o primeiro compilador que realmente tirava proveito das características da arquitetura paralela para implementar paralelismo em loops [Lam 2004].

A idéia básica da técnica de software pipeling é promover um novo escalonamento do loop, ou seja reformá-lo, de tal forma que cada iteração possa ser iniciada antes que outra termine, potencializando, assim, o paralelismo. Percebe-se então que a identificação deste novo loop, ou *kernel*, é chave para o sucesso da técnica. Vários itens devem ser levados em conta neste processo e um deles é conhecer as dependências entre operações, já que operações dependentes não podem ser executadas em paralelo.
Um grafo de dependência de dados (DDG) é a ferramenta comumente usada para representar as dependências existentes entre operações de uma iteração, onde os nós representam as operações e os arcos que representam uma ligação do tipo "deve seguir". Um arco *loop carried* indica dependências "deve seguir" entre iterações diferentes e um *loop independent* dependência numa mesma iteração. No escalonamento das operações, as que são dependentes não podem ser executadas em paralelo (seja num mesmo processador ou em vários deles). Associado a cada nó há duas informações (diff ,min). *Min* indica o atraso que deve haver para a próxima instrução ser iniciada, sendo usado para especificar operações multi-ciclos (p.ex. de ponto flutuante). *Diff* indica o número de iterações relativas a uma dependência entre iterações. A figura 2 ilustra o código de um loop, o DDG correspondente que contém um loop carried e três independent e o escalonamento correspondente. Note que no loop carried corresponde a um ciclo e seu diff tem valor 1, pois a operação 1 (a[i+1]=a[i]+2) tem dependência com a próxima.

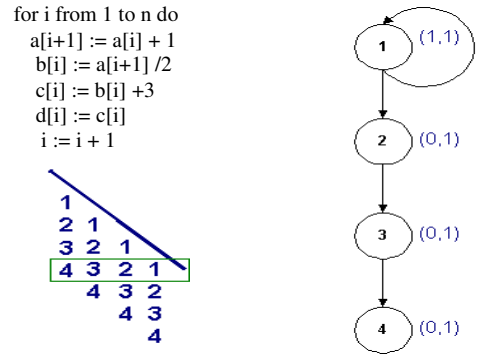


Figura 2 – Grafo de Dependências de Dados e escalonamento baseado nas dependências

Na geração do escalonamento, estas informações são levadas em conta e as dependências *loop carried* são preservadas. Portanto, na definição do novo escalonamento e do novo loop é importante definir o intervalo de tempo necessário para iniciar cada iteração. Este intervalo denomina-se Intervalo de Iniciação (*II*) e quanto menor ele for, maior *troughput*. O *II* mínimo (*MII*) é o menor *II* possível para um escalonamento válido para o software pipelining. Geralmente não é fácil identificar o *II* apenas analisando-se o DDG. Na verdade, este é um problema NP-Completo. Mas se *MII* é encontrado e ele é igual ao *II* do escalonamento obtido, então este é um escalonamento ótimo do ponto de vista do *throughput* [Rau 1994].

Alguns fatores afetam a determinação do *II*. Um deles é a dependência de dados representada pelo DDG. O *II* deve considerar as dependências existentes respeitando os devidos atrasos. O outro fator é o uso de recursos (unidades funcionais, bus, etc). A utilização de recursos é calculada totalizando os requerimentos impostos por cada operação de uma iteração. A figura 3 ilustra tabelas de reservas utilizadas para a identificação de recursos necessários para operações de adição e de multiplicação. Numa breve análise deste exemplo, verifica-se que estas operações não poderiam ser excutadas em paralelo, pois ambas usam os dois Source Bus disponíveis no primeiro ciclo da execução. No processo de escalonamento é usada uma estrutura de dados para

representar esta informação de maneira a tornar possível o cálculo do *II* acomodando adequadamente o uso dos recursos por ciclo. Portanto, o *MII* deve ser igual ou maior que as restrição de recursos (RecMII) e de dependência (ResMII) existentes. Então, MII = max(ResMII, RecMII).

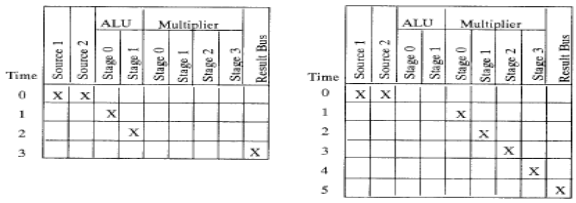


Figura 3 – Tabelas de Reservas de Recursos

Diversos métodos são usados para calcular o *II.* Dente eles temos*:* enumeração simples de ciclos; algoritmo de menor caminho, que usa fecho transitivo; menor caminho iterativo e programaçao linear.

3.CLASSIFICAÇÃO DOS ALGORITMOS

Alguns problemas podem existir no software pipeling, como por exemplo não se conseguir chegar a um *kernel* viável. Portanto a escolha do algoritmo de escalonamento a ser utilizado e a técnica adotada é importante. Os algoritmos de software pipelining geralmente dividem-se em 2 grupos segundo a abordagem utilizada para promover o escalonamento, Escalonamento Módulo e Identificação do Kernel, os quais serão discutidos a seguir.

3.1. Escalonamento Módulo

O Escalonamento Módulo foi proposto por Rau [Rau & Glaeser 1981] e consiste num framework que especifica um conjunto de restrições a serem seguidas para gerar um escalonamento válido. Esta abordagem prevê um escalonamento inicial do *kernel*, que depois, usando técnicas de movimentação, é melhorado. Como vantagem tem-se que não há muita expansão de código, já que não exige desenrolamento de código para efetuar o escalonamento.

No escalonamento obtido, espera-se que na repetição da iteração em intervalos regulares (o *II*), não haja conflito de recursos e que as restrições de dependência existentes sejam respeitadas. Em outras palavras, durante o escalonamento para geração do novo *kernel*, quando uma operação é colocada em uma dada localização, seja garantido que nenhuma dependência de dados ou conflito de recursos será violado durante a sobreposiçao de iterações. A dificuldade da técnica está em garantir que a colocação das operações é legal de tal forma que sucessivas iterações sejam escalonadas de maneira idêntica. Uma vez determinado o escalonamento é efetuada a movimentaçao do código e sua estrutura completa pode, então, ser definida.

Para promover o escalonamento inicial, é necessário o conhecimento do *II*, que é estimado em função das restrições existentes, não sendo necessariamente o melhor, ou seja aquele que promove o melhor *throughput*. O algoritmo é repetido testando valores menores para o *II*, até que uma solução seja encontrada, satisfazendo as restrições existentes.

No caso da existência de caminhos múltiplos decorrentes de desvios condicionais, antes de proceder o escalonamento, o código deve ser transformado em um único caminho usando algum algoritmo específico para isso, pois o seu uso em loops com desvios torna-se muito complicado. Diversas instâncias de algoritmos foram criadas sob este framework, sendo que elas diferenciam-se exatamente no tratamento dos caminhos múltiplos. Os princi e serão aboradados a seguir.

Redução Hierárquica

Este algoritmo proposto por [Lam 1988] é uma extensão do escalonamento módulo, com algumas melhorias, bastante difundido, sendo uma referência para a maioria dos trabalhos desenvolvidos desde então. Sua principal motivação foi tornar o software pipelining aplicável a todos loops, inclusive àqueles contendo instruções condicionais. Esta abordagem escalona o loop hierarquicamente, começando com as construções de loops mais internos. À medida que cada construção é escalonada ela é reduzida a um nó simples do DDG, com este nó representando todas as restrições existentes de seus componentes em relação aos demais nós. Com isso reduz toda a construção condicional para um único caminho que representa a união do uso de recursos. Com isso, o nó pode então

ser escalonado considerando as "novas" restrições existentes. O processo continua até que todo o programa esteja representado por um único nó. A figura 4 ilustra este processo no DDG.

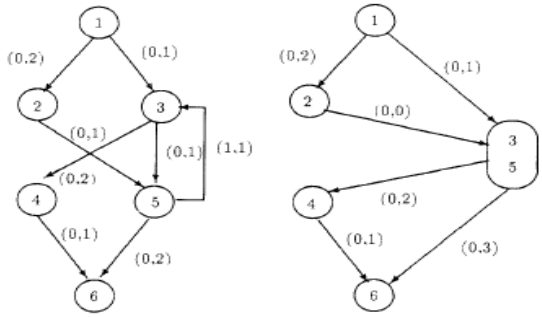


Figura 4 – DDG na Redução Hierárquica

Uma de suas principais contribuições é a expansão de variável, uma otimizaçao feita quando um mesmo registrador é usado em todas as iteraçoes. Neste caso, para permitir o paralelismo, é necessária alguma modificação. Inicialmente o algoritmo identifica os registradores que são usados em todas as iterações, fazendo o escalonamento normal do loop, como se não houvesse a dependência. O escalonamento resultante é usado para identificar onde é necessária a troca de registradores, considerando a sua duração (primeira e última vez que é usado), efetuando-a quando necessária para permitir o paralelismo. Por exigir o desenrolamento inicial para identificação dos conflitos de registradores, esta técnica pode levar a muita expansão de código. A figura 5 exemplifica um trecho de código onde esta operação foi efetuada.

```
Def(R1)          Def(R1)
op               op       Def(R2)
Use(R1)       L:Use(R1)   op       Def(R1)
                          Use(R2)  op       Def(R2) CJump L
                                   Use(R1)  op
                                            Use(R2)
```

a) código da iteração

b) iteração após a expansão de variável

Figura 5 – Expansão de Variável

Escalonamento Predicado

Esta é outra abordagem para promover o tratamento de desvios condicionais em loops. Neste caso, cada operação tem uma entrada predicada de 1-bit. Se o bit é falso, a operação não deve ser executada e o hardware a trata como um no-op. Caso contrario, ela é executada de forma usual. A execução predicada geralmente elimina a necessidade de desvios condicionais e é uma extensão da técnica *if-conversion* que transforma um código não estruturado em código livre de desvios condicionais. Uma alternativa ao uso de bit predicado é o uso de instruções predicadas específicas (invalidade, etc).

A execução predicada usa uma alternativa à expansão de variável, que evita duplicação de código. É usado um arquivo de registros rotativo (*rotating register file*) que funciona como uma lista circular, tornando disponível a cada iteração apenas uma janela de registradores que implementam de maneira lógica os registradores usados na iteração.

Esta técnica é mais flexivel que a anterior, pois na redução hierárquica o código condicional é escalonado antes das demais operações. Neste caso, uma decisão arbitrária mal feita pode impactar as demais que sequer foram ainda consideradas. Na execução predicada todas operações são carregadas e apenas aquelas que devem ser executadas (bit true) são efetivadas e portanto, o escalonamento leva todas em conta. Entretanto, tem como desvantagem a execução de todas as operações

3.2. Identificação do Kernel

Diferente do escalonamento módulo, os algoritmos desta classe promovem o desenrolamento do loop para que consigam identificar um padrão. Assim é possível que em alguns casos o seu custo se torne impraticável, dada a expansão de código existente.

Compreende os seguintes passos: desenrolamento do loop, anotando as dependências existentes; escalonamento das operações respeitando as dependências; tentativa de identificação de um conjunto de operações que se repete para formar o corpo do novo loop; reescrita do loop, considerado o novo kernel. A questão em relação a esta abordagem é: e se não surgir o novo bloco? Normalmente usam-se técnicas de movimentação para isso. A seguir as principais técnicas desenvolvidas seguindo esta abordagem.

Perfect pipelining

Esta é uma das técnicas mais difundidas, e assim com a redução hierárquica, deu origem a vários outros trabalhos. Aiken e Nicolau [Aiken & Nicolau 1988]

introduziram o perfect pipeline como um algoritmo ou framework que combina movimentação de código com escalonamento para melhorar o paralelismo. Usa alguns testes para recolocar o problema segundo novos parâmetros, respondendo à pergunta: "Como o software pipelining seria afetado se a arquitetura fosse modificada para suportá-o"?

Os autores colocaram em seu trabalho que as técnicas disponíveis até então não exploravam toda a capacidade de paralelismo disponível, mesmo quando os recursos existentes eram ótimos. Com isso, propuseram o perfect pipelining e provaram que esta técnica levava a um escalonamento ótimo, ou seja, considerando as depedências de dados existentes, nenhum paralelismo melhor podia ser obtido em melhor tempo, fazendo uso efetivo dos processadores e dos recursos diponíveis. O problema é encontrar uma utilização ótima dos rescursos, mesmo quando todos os recursos necessários nào estão disponíveis.

A abordadem é simples: considera o histórico de execução de um loop e baseado nas i primeiras iterações, escalona estas iterações o mais breve possível, considerando a dependência entre as iterações como o fator para definir o nível de paralelismo entre elas. Apesar do pressuposto de que para revelar um padrão para o kernel deve ser escalonado um conjunto grande de iterações, os autores mostraram que a quantidade de iteraçoes a serem executadas para revelar este corpo é mínimo.

Para a identificaçao do corpo do loop, são escalonadas as operações que não estão no caminho crítico, buscando-se com isso, um *kernel* compacto. Nesta atividade são usadas técnicas de movimentaçao do código. Inicialmente o escalonamento é feito como se não houvesse restrições de recursos, ou seja, eles fossem infinitos. Após a identificaçao do prólogo, epílogo e *kernel*, é feita uma análise da quantidade de processadores disponíveis, veriicando se eles são suficientes para sua paralelização. Caso não sejam, um conjunto de heurísticas simples é usado para reduzir o largura do *kernel*, sem contanto, aumentar o seu tamanho, tornando-o adequado aos recursos disponíveis.

Da mesma forma que no escalonamento módulo, o corpo do loop não pode conter desvios além daqueles de teste de saída. Portanto, técnicas de *if-conversion* devem ser aplicadas antes da utilização do algoritmo.

Modelo de Redes de Petri

Consiste numa variação do algoritmo anterior, usando redes de petri para resolver o problema de reconhecimento do kernel [Rajagopalan & Allan, 1993]. Poderoso como o perfect pipelining, substitui as técnicas *adhoc* utilizadas por uma fundamentação matemática. No algoritmo proposto a rede de Petri é usada para mapear as depedências cíclicas do loop. Apesar do DDG mostrar as dependências "deve seguir" entre operações, ele não é capaz de identificar as operações que estão prontas para serem executadas. Neste sentido a rede de Petri é utilizada, funcionando como um DDG com o estado do escalonamento corrente embutido, o que pode ajudar bastante no processo de identificação do *kernel*. Cada operaçao é representada por uma transição e os arcos representam as dependências entre elas. Apesar das vantagens apresentadas, não é uma técnica muito difundida.

Técnica de Vergdahl

Representa um método exaustivo de identificação do *kernel* no qual todas as soluções possíveis de escalonamento do loop são representadas e dentre elas a melhor é escolhida. O método usa programação dinâmica para chegar a um grafo de decisões a partir do desenrolamento de loop. Neste grafo os nós representam as operações escalonadas e os arcos as possíveis decisões de escalonamento. Como o escalonamento em si já é um problema NP-completo, esta técnica é inviável na prática. Entretanto, é capaz de fornecer heuríticas bastante úteis para os demais algortimos existentes.

Enhanced Pipeline

É uma extensão do Perfect Pipelining, desenvolvido para tirar proveito de arquiteturas que suportam a execução multi-predicada. Para isso, integra transformação de código com escalonamento conseguindo obter bons resultados. O algoritmo usa movimentaçao de código, por meio de renomeação e substituição de foward, mas retém o corpo do loop, deixando de ser necessária a identificação do kernel. Apesar de ser um algoritmo de difícil entendimento, traz benefícios que nenhum outro conseguiu.

4. ESTADO ATUAL

Embora os algoritmos apresentados sejam relativamente antigos, nenhum novo objeto de impacto foi proposto. O que tem sido apresentado ao longo destes anos são trabalhos desenvolvidos buscando melhorar os algoritmos existentes em pontos específicos. Neste sentido, podemos citar os trabalhos desenvolvidos por [Chabin et al. 2005] e [Rong et al. 2005] que tratam da alocação de registros entre iterações, o de [Zhuge et al. 2003] e o de [Kim et al. 2003] que apresentam abordagens para reduzir o tamanho do código. Em especial tem-se o trabalho de retrospectiva de Lam [Lam 2004] que mostra que software pipelining é efetivo em máquinas VLIW sem exigir suporte complicado de hardware .

5. CONCLUSÃO

Quando se pensa em melhorar desempenho de um programa via paralelismo, software pipelining aparece como uma das técnicas mais importantes existentes, dado o domínio do tempo de execução de um loop sobre o tempo de execução do programa. Esta técnica tem como principais objetivos (e desafios) identificar o corpo do loop a ser paralelizado e diminuir ao máximo o intervalo de execução (*II*) entre suas iterações. Esta não é uma atividade simples, consistindo em um problema NP-Completo. Vários algoritmos surgiram para realizar esta atividade, buscando obter este escalonamento que execute em um tempo polinomial. Dentre eles temos o Escalonamento Módulo, Escalonamento por Redução Hierárquica e o Perfect Pipelining, que ao longo dos anos vêm sofrendo algumas modificações e ajustes, mas continuam dominando o cenário .

Software pipeling é uma técnica tão importante que chega a dirigir o projeto de arquiteturas de computadores. Entretanto, Monica Lam mostra em seu trabalho que ele é efetivo, mesmo em máquinas que não foram projetadas especificamente para estes fim [Lam 2004].

Este trabalho apresentou um visão geral da técnica de software pipelining, os principais conceitos e desafios envolvidos. Além disso, trouxe uma breve descrição dos principais algoritmos existentes, provendo ao leitor uma boa idéia da importância desta técnica.

REFERÊNCIAS BIBLIOGRÁFICAS

Aiken A, Nicolau A. Perfect Pipelining: A New Loop Parallelization Techinique, *Proceedings of the 2nd European Symposium in Programming,* p. 221-235, March 21-24, 1988

Allan V.H, Jones R.B, Lee R. M, Allan S. J. Software pipelining *ACM Comput. Surv.* 27 3 1995 367-432.

Chabini N., Aboulhamid E.M., Chabini I., Savaria Y., Scheduling and optimal register placement for synchronous circuits derived using software pipelining techniques, *ACM Transactions on Design Automation of Electronic Systems* (TODAES), v.10 n.2, p.187-204, April 2005

Kim S., Moon S.M., Park J., Ebcioglu K., Unroll-Based Copy Elimination for Enhanced Pipeline Scheduling, *IEEE Transactions on Computers*, v.51 n.9, p.977-994, September 2002

Lam Software pipelining: an effective scheduling technique for VLIW machines *SIGPLAN Not.* V 23 7 1988 p 318—328.

Lam M. Software pipelining: an effective scheduling technique for VLIW machines *SIGPLAN Not*. V 39 4 2004 p 244-256.

Patel J. and Davidson E.S. Improving the throughput of a pipeline by insertion of delays ISCA '76: *Proceedings of the 3rd annual symposium on Computer architecture* 1976 159—164 ACM Press New York, NY, USA

Rajagopalan M. , Allan V. H. Efficient scheduling of fine grain parallelism in loops MICRO 26: *Proceedings of the 26th annual international symposium on Microarchitecture* 1993 p.2--11Austin, Texas, United States.

Rau B. R., Glaeser C. D. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing MICRO 14: *Proceedings of the 14th annual workshop on Microprogramming* 1981 p183-198 Chatham, Massachusetts, United States.

Rau B. R. Iterative modulo scheduling: an algorithm for software pipelining loops MICRO 27: *Proceedings of the 27th annual international symposium on Microarchitecture* 1994 63—74 San Jose, California, United States

Rong H., Douillet A., Gao G. R., Register allocation for software pipelined multi-dimensional loops, *ACM SIGPLAN Notices*, v.40 n.6, June 2005

Zhuge Q., Xiao B., Sha E.H.M., Code Size Reduction Techinique and Implementation for Software-Pipelined DSP Applications, *ACM Transactions on Embedded Coputing Systems*, vol.2, no. 4, November 2003, p. 590-613.

Arquitetura Raw

Ricardo M Nishihara
RA 936161 - UNICAMP
ricardo.nishihara@terra.com.br

RESUMO

Este trabalho descreve a arquitetura Raw desenvolvida pelo Laboratório de Ciência da Computação do Massachusetts Institute of Technology - MIT. A arquitetura proposta tem como desafio viabilizar um processador de propósito geral que tenha bom desempenho tanto em aplicações computacionais que envolvam *streaming de dados*, quanto em programas sequenciais típicos. No presente trabalho são apresentadas além da descrição dos principais aspectos desta arquitetura também algumas considerações sobre os desafios e características software de suporte (run-time e compilador) requerido para o bom desempenho do sistema. Também são fornecidas informações sobre testes comparativos envolvendo esta arquitetura e microprocessadores comerciais de complexidade equivalente.

1. INTRODUÇÃO [1,2]

O grande avanço da tecnologia de circuitos integrados tem expandido de maneira considerável o número de aplicações implementadas em VLSI. Tais aplicações incluem tanto programas sequenciais executados sobre microprocessadores de propósito geral (os quais utilizam técnicas para exploração de paralelismo no nível de instrução - ILP), quanto aplicações "streaming" com alto grau de paralelismo, implementadas até o momento, principalmente em hardware dedicado através de circuitos integrados específicos à aplicação - ASICs. Muitos destes circuitos ASIC "altamente paralelos" implementam algoritmos cujas demandas computacionais estão ainda muito acima das capacidades disponibilizadas pelos microprocessadores de propósito geral atuais. Exemplos de tais circuitos são decodificadores / processadores de áudio e vídeo, aceleradores gráficos, controladores de rede, codificadores para criptografia, etc.

O objetivo principal do projeto Raw foi fazer um melhor uso de tal avanço tecnológico propondo uma arquitetura que viabilizasse novos microprocessadores de propósito geral, capazes de executar com bom desempenho tanto aplicações sequenciais típicas explorando ILP, quanto um maior número de aplicações "*streaming*" com alto grau de paralelismo, e que até aqui vem sendo implementadas principalmente através de hardware dedicado.

Com o intuito de viabilizar esta inovação, os idealizadores da arquitetura Raw identificaram quatro fatores como responsáveis principais para o sucesso dos ASICs na implementação de aplicações "streaming" altamente paralelas:

1. Especialização: As implementações baseadas em ASIC podem especializar as operações requeridas pela aplicação no nível de *gate*, ao passo que estas mesmas operações implementadas em um microprocessador tem de ser feitas através de combinações de "sub-operações" definidas pelo conjunto de instruções do microprocessador. Esta implementação de operadores especializados viabilizada pelos ASICs, resulta em ganhos expressivos de desempenho. Por exemplo, a implementação de um operador específico como uma operação de ponto flutuante incompatível implementada em hardware dedicado tem o potencial de ser executada em um ou alguns ciclos de relógio, a passo que uma implementação equivalente em microprocessador pode requerer várias instruções e demandar vários ciclos de relógio para sua execução.

2. Maior utilização de recursos em paralelo: Abordagens baseadas em ASICs são capazes de viabilizar um grande número de operadores e canais de comunicações operando em paralelo, e tal capacidade costuma ser um requisito de aplicações "streaming". Embora existam microprocessadores superescalares e VLIW capazes de executar mais de uma instrução em um único ciclo de relógio (como por exemplo Itanium II capaz de executar até 6 instruções por ciclo), circuitos dedicados implementados em ASIC como por exemplo aceleradores gráficos costumam executar centenas ou milhares de operações paralelas no nível de *word* por ciclo.

3. Gerenciamento de condutores e dos atrasos de propagação associados: No projeto de ASICs o gerenciamento dos atrasos dos condutores é feito através do *placement/routing* adequado das várias unidades funcionais (ou operadores) requeridas pela aplicação. Com exemplos de boas práticas de *placement/routing* podemos citar: a colocação de mais próxima de unidades que se comunicar mais freqüentemente; a implementação de canais de comunicação dedicados nos pontos que requerem maior largura de banda; e a colocação de registradores *pipeline* entre operadores distantes, convertendo atrasos de propagação em ciclos de latência associados ao acesso destes registradores. Através destas estratégias, obtem-se um compromisso entre paralelismo e latência, de modo a maximizar o número de recursos usufruindo de um maior número de sinais para comunicação.

4. Gerenciamento de pinos: Implementações ASICs em geral não são limitadas por gargalos como a hierarquia do sistema de memória, pois procuram utilizar os pinos do encapsulamento de modo a melhor se adequar às necessidades da aplicação. Tal abordagem visa minimizar latências tanto no acesso a memórias DRAMs externas e maximizar a largura de banda de E/S, facilitando a interface com dispositivos E/S que demandam mais eficiência como CCDs, conversores A/D, matriz de sensores, etc. Os atuais mecanismos de E/S disponibilizados por microprocessadores de propósito geral ainda são ineficientes, principalmente devido ao uso do sistema de memória (baseado em DRAMs) como *buffer* intermediário.

O objetivo da arquitetura Raw é portanto, viabilizar um microprocessador que contemple os fatores mencionados, sem deixar de lado funcionalidades típicas de um processador de propósito geral. Neste sentido a arquitetura Raw adota a seguinte abordagem:

1. Implementação através de mecanismos de hardware especializado de operadores requeridos para a exploração de ILP

em aplicações seqüênciais, e/ou para o aumento de eficiência em aplicações "*streaming*". A maioria destes mecanismos é exposta ao software através da arquitetura do conjunto de instruções – ISA. Dentre estes mecanismos incluem-se: operações inteiras e de ponto flutuante usuais, operações especializadas para manipulação multigranulares (nos níveis de *bit*, *byte* e *word*), operações de roteamento de operandos entre unidades funcionais adjacentes, e bypass de operandos entre unidades funcionais, registradores, filas de E/S, e cache de dados.

2. A arquitetura Raw tem como paradigma fundamental replicar em grande número os operadores mencionados no item 1, e expô-los ao software através do ISA. Deste modo, o usuário (compilador ou programador) pode via software fazer uso destes recursos para explorar melhor tanto o ILP existente nas aplicações sequênciais quanto o alto grau de paralelismo das aplicações "*streaming*".

3. A arquitetura Raw gerencia o impacto dos atrasos de propagação nos condutores que inteligam unidades funcionais do processador expondo ao software operadores relativos aos canais de comunicação interligando estas unidades. Deste modo o software de suporte pode considerar as latências associadas ao realizar a organização do transporte de dados escalares e *streaming* entre estas unidades, e também criar novos padrões de comunicação dedicados a uma dada aplicação. Estes operadores relativos aos canais de comunicação quando considerados em conjunto provêm uma abstração para uma rede de operandos escalares de baixa latência que também pode ser usada na exploração de ILP.

4. As abstrações expostas pelo ISA referentes aos "pinos", permitem o gerenciamento via software de sistemas de memória cache e de interfaces E/S de alto desempenho.

O presente texto visa apresentar a arquitetura Raw desenvolvida pelo Laboratório de Ciência da Computação do Massachusetts Institute of Technology – MIT, e para tanto possui a seguinte organização: na seção 1 são apresentadas as motivações do projeto Raw; na seção 2 são considerados os principais os componentes definidos por esta arquitetura; na seção 3, são feitas algumas algumas considerações sobre características e desafios associados ao software de suporte (run-time e compilador) requeridos para um bom desempenho de um sistema Raw; na seção 4 são fornecidas informações sobre testes comparativos envolvendo esta arquitetura e microprocessadores comerciais de complexidade equivalente; e finalmente na seção 5 são apresentados os comentários finais.

2. ARQUITETURA RAW [3,4]

O paradigma fundamental da arquitetura Raw baseia-se na obtenção de um processador a partir da replicação de elementos de processamento mais simples e idênticos denominados *tiles*, sendo que cada *tile* é responsável pelo processamento de seu próprio fluxo de instruções.

Em sua implemetação inicial, conforme mostra a Figura 1, um processador RAW foi construído a partir de um conjunto de 16 *tiles* interconectados em um arranjo *mesh-2D*. Cada *tile* é dotado de um processador principal (dotado de um pipeline baseado no MIPS 32 bits e de memórias SRAM de instruções e de dados), de um roteador programável dedicado roteamento estático e dois roteadores dinâmicos.

2. ARQUITETURA RAW

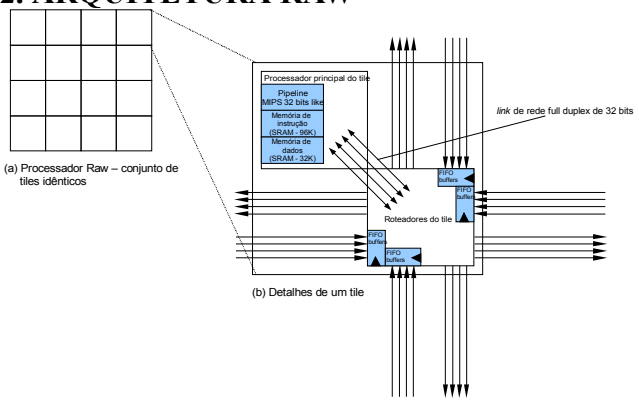


Figura 1 – Processador Raw – Um conjunto de *tiles* idênticos conectados através de um arranjo *mesh-2D*

Ainda conforme mostrado na Figura 1, os *tiles* são interconectados através de quatro redes *full duplex* de 32 bits integradas ao *chip*. Duas dessas redes são ditas estáticas, ou seja, com as rotas são especificadas em tempo de compilação, e as outras são ditas dinâmicas, ou seja, cujas rotas são especificadas em tempo de execução. Cada *tile* é conectado apenas a seus quatro vizinhos mais próximos (nas direções norte, sul, leste e oeste), ou seja o condutor mais longo do sistema tem no máximo a largura de um *tile*. Esta uma propriedade é muito importatnte para a escalabilidade da arquitetura. As referidas redes são expostas pelo ISA da arquitetura Raw ao software, permitindo que o programador ou compilador programe diretamente os canais de comunicação para controlar a transferência de dados entre os processadores principais dos *tiles,* numa abordagem análoga ao *placement/routing* realizado no projeto de circuitos ASICs para operadores especializados.

Ao contrário dos processadores superescalares, a arquitetura Raw não incorpora ao hardware estruturas lógicas para *register renaming*, ou para *scheduling* dinâmico de instruções. Ao invés disso, a orientação é mover estes mecanismos para o software e manter o *tile* o mais simples e pequeno possível, com intuito de: maximizar o número de *tiles* integrado no chip, aumentar a freqüência de relógio do sistema e aumentar a oferta de recursos computacionais que podem ser usados em paralelo.

Componentes do *tile*

Processador principal

O processador principal do *tile* é composto basicamente de um processador RISC *pipeline* implementando um conjunto de instruções MIPS de 32 bits com algumas modificações (tais como implementação de operações de manipulação multigranular, ou seja, nos níveis de *bit*, *byte* e *word; e* roteamento de operandos escalares entre unidades funcionais adjacentes).

Considerando o protótipo da implementação inicial, o processador principal conta com 32K de SRAM para memoria de dados, e 96K de SRAM para memoria de instrução. A memória de instrução não faz uso de cache, sendo sua virtualização implementada via software. A memória de dados pode fazer ou não uso de cache. Nos casos em que não se faz de caching em hardware a virtualização da memória de dados também fica a cargo do software.

A Figura 2 mostra o diagrama de blocos do pipeline RISC implementado no processador principal de um *tile*. Ele é bastante similar a um *single issue in order pipeline*, a menos da presença de FIFO buffers que constituem a interface do processador principal com as redes para comunicação entre *tiles*. É importante destacar o projeto bastante agressivo usado na integração destas interfaces de rede dentro pipeline do processador principal do *tile*. Tal abordagem foi adotada visando minimizar as latências da comunicação entre *tiles*. As interfaces de rede não só foram mapeadas como registradores, como foram também integradas diretamente ao *path* de *bypass* do *pipeline*, ou seja, este pipeline é capaz de ler dois valores direto da rede, executar uma operação sobre eles e enviar o resultado de volta para a rede sem acessar o banco de registradores.

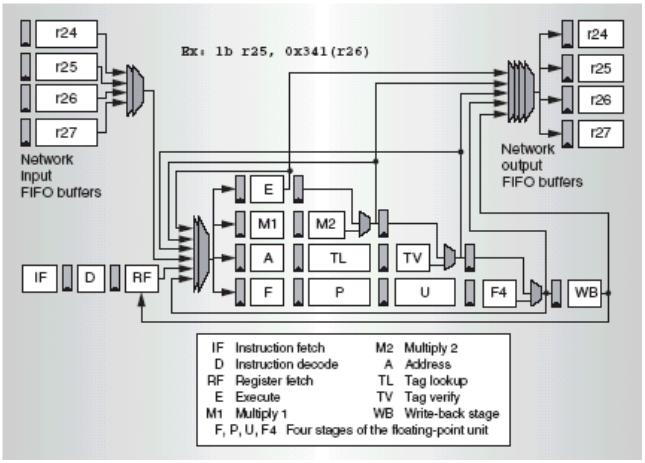


Figura 2 – Diagrama de blocos do pipeline RISC usado no processador principal de um *tile* da arquitetura Raw

Conforme ilustra a Figura 2, os registradores de 24 a 27 são mapeados para os quatro *ports* de rede. Assim por exemplo, uma leitura a partir do registrador 24 removerá um elemento do FIFO buffer associado, enquanto uma escrita enviará um dado para rede. Se nenhum dado está disponível no FIFO buffer de entrada, ou se não há espaço no FIFO buffer de saída para armazenar este dado, o pipeline do processador principal do tile sofrerá um *stall*.

Cada FIFO buffer de saída é conectado à saída de cada um dos estágio do pipeline. Assim os FIFO buffers podem retirar o valor "mais antigo" do pipeline assim que ele estiver pronto, ao invés de esperar atpe o final do estágio de *write-back*. Essa lógica é similar a lógica tradicional de bypass exceto pelo fato de que é dada prioridade às instruções mais antigas ao invés das instruções mais novas.

Ainda como uma pequena amostra de modo como estes mecanismos são expostos ao sofware ou programador, a Tabela 1 e a Figura 3 mostram respectivamente a definição dos *aliases* dos registradores especiais 24 a 27 associados aos FIFO buffers da interface de rede, e um pequeno trecho de código em linguagem assembly ilustrando sua utilização dos *ports* de rede.

Registrador	Alias	Descrição
$24	$csti	Port de entrada para a rede estática
$25	$csgn[i/o]	Port de E/S para a rede dinâmica de uso geral
$26	$csti2	Segundo port de entrada para a rede estática
$27	$cmn[i/o]	Port de E/S para a rede dinâmica reservada à memória

Tabela 1 – Mapeamento de registradores aos *ports* das redes estáticas e dinâmicas

```
# XOR register 2 with 15,
# and put result in register 31
xori $31,$2,15
# get two values from switch,
# add to register 3, and put
# result in register 9
addu $9,$csti2,$csti
# an ! indicates that the result
# of the operation should also
# be written to $csto
and! $0,$3,$2
# load from address at $csti+25
# put value in register 9 AND
# send it through $csto port
# to static switch
ld! $9,25($csti)
# jump through value specified
# by $csti2
jr $csti2
```

Figura 3 – Trecho de código assembly ilustrando o acesso às interfaces com as redes para comunicação entre tiles

Roteador estático

O roteador estático é um processador pipeline de 5-estágios que controla dois *crossbar switchers* e duas redes físicas. Cada crossbar rotea valores entre sete entidades: o processador pipeline do roteador estático, os quatro *tiles* vizinhos (norte, sul, leste e oeste), o processador principal do *tile*, e o outro crossbar.

O processador pipeline do roteador estático dispõe de uma memória de instruções com 8096 palavras e implementa um conjunto de instruções restrito com palavras de largura de 64 bits. Cada instrução codifica um pequeno comando (branches com ou sem decremento e acessos a um pequeno banco de registradores) e 13 rotas (uma para cada saída de crossbar).

Para cada palavra de dado transmitida entre *tiles* sobre a rede estática, deve existir uma instrução na memória de instrução de cada roteador estático, que faz parte do caminho percorrido pela palavra de dado. Estas instruções são programadas em tempo de compilação e são virtualizadas via software da mesma maneira qua as instruções do processador principal do *tile*. Portanto, os roteadores estáticos coletivamente reconfiguram inteiramente o

padrão de comunicação da rede em uma base ciclo a ciclo. Deve-se ainda ressaltar que em razão da memória de instrução do roteador ser virtualizada tem-se uma enorme flexibilidade na criação de novos padrões de comunicação.

Em razão do roteador estático saber qual rota será executada muito antes da chegada de uma palavra, preparações relativas a rota a ser seguida podem ser executadas em *pipeline.* Isso permite que a palavra seja roteada imediatamente após sua chegada ao roteador. Como veremos mais adiante a obtenção de baixa latência no roteamento estático é crítica para a exploração de ILP em aplicações seqüênciais.

Outra característica crítica para exploração de ILP, é o controle de fluxo do roteador estático. O roteador estático prossegue para a próxima instrução somente depois que todas as rotas de uma dada instrução são completadas. Isso assegura que os *tiles* destino recebam as palavras sempre em uma ordem conhecida, mesmo que ocorram situações como erro na predição de desvios, interrupções, *cache misses*, ou outros eventos não preditos.

Roteadores e redes dinâmicas

Conforme mencionado anteriormente, cada *tile* conta também com 2 roteadores dinâmicos. Quando consideramos o conjunto interligado de todos roteadores dinâmicos de todos os *tiles*, tem-se um par de redes dinâmicas para comunicação entre *tiles.* Para enviar uma mensagem em uma dessas redes, o usuário envia uma única palavra de cabeçalho especificando: o *tile* destino (ou *port* de E/S), um campo de usuário e o comprimento da mensagem. O usuário pode enviar até 31 palavras de dados em uma mensagem.

Uma das principais preocupações do uso de redes dinâmica é a ocorrência de *deadlocks* relativos a negociação de acesso aos *buffers* da rede (localizados nos roteadores dinâmicos distribuidos). Existem duas soluções clássicas para este problema: eliminar a ocorrência de *deadlock*, ou prover mecanismos que permitam recuperação a partir de sua ocorrência. Eliminação de *deadlock* requer que os usuários restrinjam seu uso a um conjunto de práticas comprovadamente livres de *deadlock*. Recuperação a partir de deadlock não coloca restrições à utilização da rede, mas requer que dados armazenado nos *buffers* da rede sejam movido para algum outro recurso externo de memória, quando for detectada a condição de *deadlock*.

A solução proposta pela arquitetura Raw utiliza duas redes dinâmicas fisicamente idênticas: Uma rede para acesso à memória com um modelo de uso restrito e eliminação de *deadlock*, e uma rede de acesso geral com modelo de uso irrestrito com um esquema de recuperação de *deadlock.*

Somente os clientes de acesso a rede privilegiado (como sistema operacional, cache de dados, interrupções, dispositivos de hardware, DMA, e *ports* E/S) podem usar a rede dinâmica de acesso à memória. O protocolo de acesso desta rede limita modo de uso permitido aos clientes de modo a eliminar *deadlocks*. Por exemplo o cliente não pode "bloquear" no envio de uma mensagem a menos que ele possa garantir de recursos em seus buffers de entrada para receber todas as mensagens a ele endereçadas.

Os demais clientes da rede dinâmica só podem utilizar a rede de propósito geral contam com o mecanismo de recuperação de *deadlock* implementado no sistema, para garantir a continuidade de sua interação. Por este mecanismo o sistema operacional programa um contador configurável no processador principal do *tile* para detectar se as palavras estão esperando muito tempo para serem enviadas. Este contator informa a ocorrência de *deadlock*, disparando uma interrupção que remove dados dos buffers da rede para memórias DRAM externas através da rede de acesso à memória. Uma segunda interrupção virtualiza o *port* de entrada da rede de proposito geral e permite a recuperação dos dados a partir das memórias DRAM externas.

3. SUPORTE DE SOFTWARE [1,5]

Considerações sobre o *run-time* software

Atingir bom desempenho em programas com padrões de desvios dependentes de dados e com operações de memória baseados em ponteiros requer mecanismos capazes de analisar e reagir ao comportamento do programa em tempo de execução. Em processadores superescalares, tais mecanismos são providos por unidades de hardware tais como caches, buffers para predição de desvios, lógica para *register renaming*, e unidades para escalonamento de instrução.

Em um processador Raw tais funcionalidades devem ser providas pelo *run-time* software do sistema. Por exemplo, o processador Raw implementa *caching* da memória de instruções em software. Neste caso o *run-time* software gerencia a hierarquia de memória executando a checagem de cada acesso de memória e provendo o mapeamento corrente do endereço requisitado. O compilador por sua vez tem neste situação duas atribuições: inserir código referentes aos estes testes (feitos a cada referência de memória), e eliminar os testes detectados como redundantes durante a compilação. Embora o *caching* via software seja mais custoso que a implementação em hardware, o primeiro pode valer-se de algoritmos mais sofisticados, e permitir se necessário a adequação de tais algoritmos a necessidades específicas da aplicação. Isto pode elevar a taxa de *hits* do sistema de *caching* compensando custos e overheads adicionais. Outro mecanismo que requer um suporte similar tanto do *run-time* software quanto do compilador é uso de execução com especulação. De maneira geral, os custos adicionais de se implementar tais serviços em software precisam ser reduzidos, e esta redução pode vir tanto através do compilador, a partir da eliminação de operações desnessárias, quanto a partir de algoritmos melhores de suporte. Como já mencionado o principal benefício de mover-se mecanismos de controle dinâmico para o software é a simplicação de hardware de controle, liberando mais espaço para maior disponibilização de recursos computacionais em paralelo, e permitindo maiores freqüência de relógio.

O desempenho geral de um sistema baseado na arquitetura Raw depende do compromisso de alguns fatores. Em razão do suporte a comportamentos dinâmicos dos programas ter sido movido do hardware para o software, um programa executará mais instruções que um sistema que implemente este suporte em hardware. Por outro lado, o hardware mais simples da arquitetura Raw poderá rodar a freqüências de clock maiores, e disporá de mais recursos computacionais para explorar o paralelismo da aplicação.

Considerações sobre o compilador

Como aspectos principais relacionados ao compilador dedicado a um sistema baseado na arquitetura Raw, devemos destacar: a alocação de recursos, a exploração de paralelismo de fina granularidade, e o escalonamento das comunicações entre *tiles.*

Alocação de recursos

Na execução de um programa no processador Raw, tanto o *run-time* software quanto o programa executado são mapeadas para uma coleção de *tiles* idênticos. Ao invés de uma alocação de uma alocação fixa pré-definida em hardware, a arquitetura Raw permite que os *tiles* sejam alocados de uma maneira específica definida pela aplicação. Deste modo, mais recursos podem ser alocados para as partes mais críticas do run-time software ou do programa em execução.

Além disso, o programa em execução pode ser dividido em regiões paralelas de maior granularidade. Cada uma destas regiões paralelas pode ser executada em múltiplos *tiles*, os quais em conjunto comportam-se como um único processador lógico. O número de *tiles* alocados para uma dada região depende da existência de paralelismo de granularidade mais fina dentro desta região.

Os *tiles* dentro de uma dada região comunicam-se entre si usando principalmente comunicações estáticas, enquanto as regiões comunicam-se entre si ou com run-time software usando mensagens dinâmicas.

O tamanho e o número de regiões alocadas pode ser decidido pelo compilador baseando-se na natureza do paralelismo existente na aplicação. Por exemplo, pode-se usar um pequeno número de regiões grandes quando um alto grau de paralelismo de granularidade fina for detectado pelo compilador, e um grande número de aplicações de regiões pequenas quando for detectado somente paralelismo de maior granularidade.

O compilador para um sistema Raw portanto, deve ser capaz de analisar um programa como um todo, com intuito de obter os requisitos demandados pelo run-time software, e detectar a disponibilidade de paralelismo de diferentes granularidades.

Exploração de paralelismo de granularidade fina

Uma oportunidade provida pela arquitetura Raw, devido às baixas latências associadas a comunicação entre *tiles* via rede estática, é a capacidade de explorar eficientemente o paralelismo de granularidade fina ao longo dos tiles. Nesta abordagem, múltiplos *tiles* podem trabalhar juntos para explorar o paralelismo no nível de instrução - ILP de um fluxo único de instruções. O compilador toma este fluxo único de instruções como entrada, particiona-o em multiplos fluxos de instruções, mapea cada fluxo particionado para um *tile*, e finalmente escalona a comunicação estática entre os fluxos particionados.

Esta abordagem da arquitetura Raw para exploração de ILP difere da utilizada em processadores superescalares e VLIW. Ao contrário dos processadores superescalares, um processador Raw explora ILP sem fazer uso de lógica complexa implementada em hardware. Os mecanismos de exploração são movidos para o software e são realizados de maneira estática pelo compilador. Em constraste com os processadores VLIW cada unidade de processamento da arquitetura Raw, ou seja o *tile*, tem seu próprio fluxo de instruções. Essa abordagem é mais flexível porque não requer que a execução seja feita em passos estanques determinados pelos blocos de múltiplas instruções contidas na palavra longa do VLIW.

A arquitetura Raw também oferece oportunidades para novas otimizações por parte do compilador, como por exemplo, novas formas de realizar o *spill* de registradores. Em processadores convencionais, o spill de registradores é feito para a memória. O processador Raw permite que o *spill* de registradores seja alternativamente feito para os *tiles* vizinhos.

Escalonamento de comunicação

Compiladores tradicionais realizam o escalonamento de um único tipo de evento (ou seja, instruções) ao longo de uma dimensão única (neste caso, o tempo). Devido a presença da rede estática, um compilador para arquitetura Raw enfrenta um problema mais complexo: ele tem de escalonar tanto instruções quanto eventos de comunicação, sendo que ambos tipos de eventos têm de ser escalonados temporalmente e espacialmente.

A compilação de código para uso da rede estática resulta em desafios relacionados a correção e desempenho. Com relação à correção, o código de roteamento tem de ser livre de *deadlock*. Com relação ao desempenho, o compilador deve ser capaz de organizar cuidadosamente a comunicação de maneira a minimizar a ocorrência de *stalls* ao longo da rede. Estes dois aspectos tem ainda sua complexidade aumentada na presença de eventos dinâmicos como desvios, *cash misses*, e mensagens dinâmicas.

Quando uma aplicação faz uso tanto de redes estáticas quanto dinâmicas, aspectos relativos à interação destas duas rede também devem ser considerados. O compilador tem de garantir que os programas gerados livres de *deadlock* neste novo contexto, e provavelmente terá de valer-se de otimizações temporais para compensar as diferenças de desempenho e comportamento das redes estáticas e dinâmicas (a temporização nas redes estáticas é altamente previsível, enquanto os atrasos devido às mensagens dinâmicas podem ser grandes e altamente imprevisíveis).

4. IMPLEMENTAÇÃO E TESTES

Nesta seção apresenta-se algumas informações relativas à implementação do protótipo ASIC do processador Raw, e sobre os testes comparativo da arquitetura Raw e um microprocessador comercialmente existente de complexidade equivalente – Pentium 3.

Protótipo de validação implementado [2,3]

Os proponentes desta arquitetura Raw implementarão um chip com um array de *16 tiles* usando como tecnologia de implementação o processo ASIC da IBM SA-27E (0.15-micron, 6 níveis, cobre). Foi utilizado um die de tamanho 18.2 x 18.2 mm e um encapsulamento CCGA *(ceramic column grid array package)* de 1657 pinos, com 1080 deles, pinos de E/S HSTL (*high speed transceiver logic)*. O chip consome em média 18.2 watts operando à 425MHz, e opera em temperatura ambiente com freqüência de relógio nominal de 425MHz/500Mhz com alimentação de 1.8V/2.2V. Esses valores são da mesma ordem dos obtidos por outros processadores IBM usando o mesmo processo, como exemplo: o PowerPC 405GP que opera na faixa de 266-400 Mhz, e o PowerPC 440GP que opera na faixa de 400-500 Mhz. A Figura 4 mostra fotos do die do chip Raw e da motherboard construída a partir do chip.

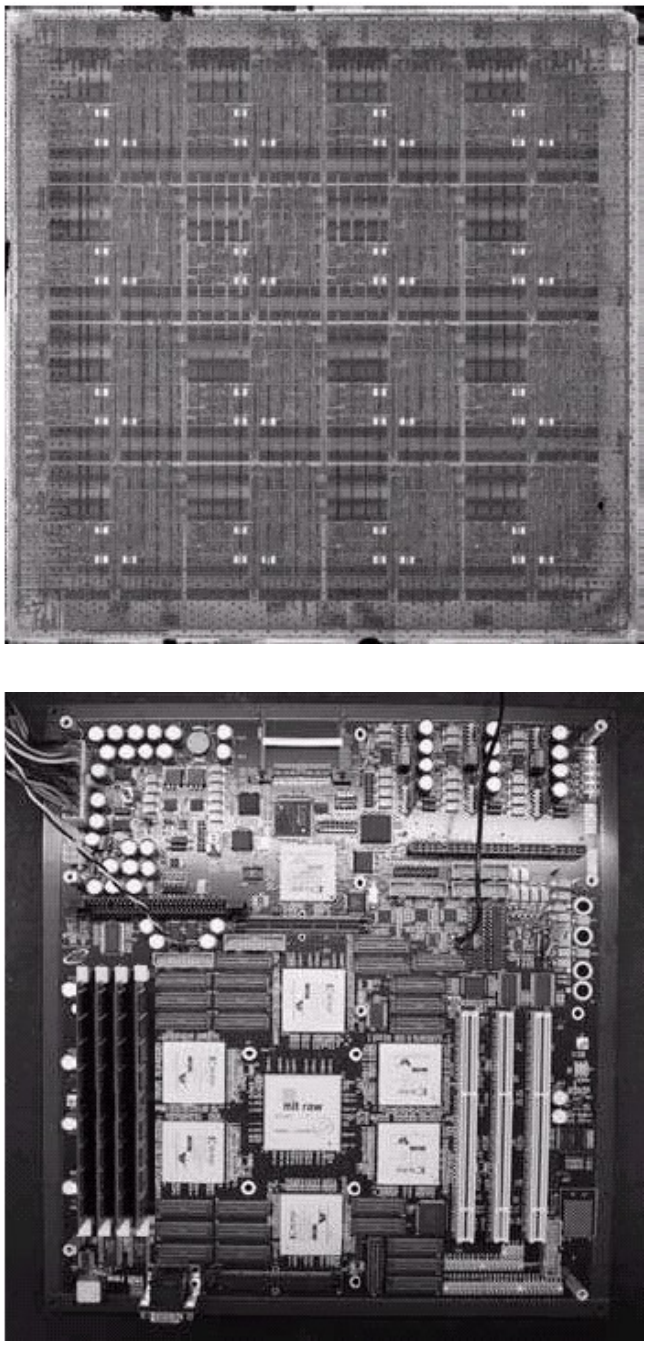


Figura 4 – Fotos do chip Raw e da motherboard construída em torno do chip Raw

Testes comparativos [2]

A referência [2] descreve em detalhes uma avaliação comparativa realizada com a implementação do processador Raw. Nesta subseção destacaremos alguns pontos da metodologia utilizada e comentaremos alguns resultados dos testes realizados.

Comparação com um microprocessador comercial existente

A metodologia usada na avaliação do processador baseou-se na sua comparação direta com um microprocessador existente comercialmente. Tal abordagem além de mais honesta por não esconder ineficiênciais do compilador e da arquitetura, facilita a comparação indireta da arquitetura Raw com outras alternativas desenvolvidas por outros grupos, uma vez que, a comparação de um dada alternativa com o mesmo sistema comercial de referência pode ser extentida para uma comparação com a arquitetura Raw.

O processador Pentium 3 foi o escolhido por ser a implementação de processador Intel mais próxima do protótipo Raw. O Pentium 3 foi implementado usado um processo com a mesma geração de litografia do protótipo Raw, e possui latências associadas às unidades funcionais muito próximas as do protótipo Raw.

A Tabela 2 a seguir mostra uma comparação dos parâmetros de implementação do protótipo Raw e do Pentium 3.

Parameter	Raw (IBM ASIC)	P3 (Intel)
Lithography Generation	180 nm	180 nm
Process Name	CMOS 7SF (SA-27E)	P858
Metal Layers	Cu 6	Al 6
Dielectric Material	SiO_2	SiOF
Oxide Thickness (T_{ox})	3.5 nm	3.0 nm
SRAM Cell Size	4.8 μm^2	5.6 μm^2
Dielectric k	4.1	3.55
Ring Oscillator Stage (FO1)	23 ps	11 ps
Dynamic Logic, Custom Macros (SRAMs, RFs)	no	yes
Speedpath Tuning since First Silicon	no	yes
Initial Frequency	425 MHz	500-733 MHz
Die Area[2]	331 mm^2	106 mm^2
Signal Pins	~ 1100	~ 190
Vdd used	1.8 V	1.65 V
Nominal Process Vdd	1.8 V	1.5 V

Tabela 2 – Parametros de implementação: Rw x Pentium 3

Validação usando simulador

Embora o protótipo em hardware do processador Raw estivesse disponível, optou-se por realizar a avaliação através de um simulador do processador Raw com precisão de ciclo de relógio. Segundo os proponentes da arquitetura Raw, tal simulador tem exatamente o mesmo comportamento RTL do chip Raw implementado pelo processo ASIC da IBM, e a razão para este procedimento foi a necessidade de normalização de alguns itens para viabilizar a comparação. Neste sentido menciona-se por exemplo a normalização de latências associadas à motherboard e às memórias DRAM, e a troca do sistema de caching por software do Raw (um algoritmo ainda em fase de pesquisa) por um sistema de caching convencional implementado em hardware para que uma comparação mais direta com o Pentium 3.

Ferramentas de SW utilizadas

A Tabela 3 a seguir lista as ferramentas de sofware utilizadas nos testes de comparação reportados em [2] pelos sistemas Raw e Pentium.

Pentium 3	Raw
> gcc 3.3 -O3 -march=pentium3 -mfpmath=sse > Intel Performance Primitives > LAPACK/BLAS com SSE para rotinas de algebra linear	> rawcc – compilador desenvolvido internamente > Streamit - compilador desenvolvido internamente > gcc 3.3

Tabela 3: Ferramentas de SW usadas nos testes de comparação

Sumário dos resultados dos testes

A Figura 5 apresenta um sumário dos resultados dos testes de desempenho envolvendo inúmeras aplicações. Os resultados são apresentados na forma de *speedup* relativos ao Pentium 3. As aplicações foram divididas em 4 classes:

1. ILP: Na qual foram colocadas aplicações sequenciais convencionais. Tipicamente, a única forma de paralelismo disponível nestas aplicações é o ILP. Para esta avaliação foram selecionados programas com diferentes graus de ILP.

2. Stream: Nesta classe foram colocadas aplicações "streaming" que lidam com grandes conjuntos de dados, ou podem manipular fluxos contínuos de dados em tempo-real.

3. Server: Para medir o desempenho do processador Raw em termos de workload de servidor, foi conduzido o seguinte experimento para se obter resultados SpecRate: para cada subconjunto de aplicações Spec 2000, executou-se uma cópia independente deste subconjunto em cada um dos 16 *tiles*, e mediu-se o throughput total deste workload em relação a uma única execução no Pentium 3.

4. Bit-Level: Inclui duas aplicações manipulação no nível de bit em geral implementadas em FPGA e ASIC: 802.11a ConvEnc, 8b/10b Encoder.

A partir do gráfico da Figura 5, pode-se fazer várias observações. Com relação as aplicações seqüenciais típicas observa-se que o Pentium 3 saiu-se melhor que o Raw para aplicações com baixo de grau de ILP, enquanto ocorre o oposto em aplicações com maior grau de ILP como Vpenta. Para aplicações "streaming" e vetoriais, o Raw supera o Pentium 3 por fatores de 10 a 100 vezes. Um servidor construido a partir de 16 processadores Pentium 3 foi escolhido como o melhor sistema servidor. Note que um sistema com um único chip Raw fica a um fator de apenas 3 abaixo deste melhor servidor para a maioria das aplicações. Com relação as aplicações da classe *bit-level*, o desempenho do Raw fica de um fator de 2 a 3 vezes abaixo das implementações em ASIC.

A tabela 4 a seguir sumariza as funcionalidades primárias responsáveis pelos ganhos de performace do processador Raw.

Classe	Benchmark	S	R	W	P
ILP	Swim, Tomcatv, Btrix, Cholesky, Vpenta, Mxm, Life, Jacobi, Fpppp-kernel, SHA, AES, Encode, Unstructured, 172.mgrid, 173.applu, 177.mesa, 183.equake, 188.ammp, 301.apsi, 175.vpr, 181.mcf, 197.parser, 256.bzip2, 300.twolf	X	X	X	
Stream	Beamformer, Bitonic Sort, FFT, Filterbank, FIR, FMRadio, Mxm, LU fact., Triang. solver, QR fact., Conv., Copy, Scale, Add, Scale & Add, Acoustic Beamforming, FIR, FFT, Beam Steering, Corner Turn, CSLC	X	X	X	X
Server	172.mgrid, 173.applu, 177.mesa, 183.equake, 188.ammp, 301.apsi, 175.vpr, 181.mcf, 197.parser, 256.bzip2, 300.twolf		X		X
Bit-level	802.11a ConvEnc, 8b/10b Encoder	X	X	X	

Tabela 4: Utilização de funcionalidades Raw. S = Especialização R = Utilização de Recursor Paralelos. W = Gerenciamento dos atrasos dos condutores. P = Gerenciamento de Pinos.

5. CONCLUSÃO

Este trabalho apresenta a arquitetura Raw proposta com desafio viabilizar um processador de propósito geral que tenha um bom desempenho tanto em aplicações computacionais altamente paralelas que envolvam *streaming de dados*, quanto nos programas sequenciais tradicionais (através da exploração de ILP). A arquitetura Raw procura atingir este objetivo a partir da de uma arquitetura reconfigurável construida a partir da replicação de elementos de processamento conhecidos como tiles, e da implementação de redes integradas de comunicação entre tiles de baixíssima latência. Estes recursos são expostos pelo ISA ao software, e deste modo, o usuário (compilador ou programador) pode fazer uso deles para explorar melhor tanto o ILP existente nas aplicações sequênciais quanto o forte paralelismo das aplicações "*streaming*". Os resultados dos testes de avaliação reportam ganhos de desempenho moderados em relação a processadores convencionais de complexidade equivalente (Pentium 3) na exploração de ILP de aplicações seqüências (fatores de 2 a 6 vezes) e ganhos expressivos de desempenho quando aplicações "streaming" são consideradas (fatores de 10 a 100 vezes).

REFERENCES

[1] E.Waingold, et al. Baring It All to Software: Raw Machines. IEEE Computer 30, 9 (September 1997), pp. 86–93.

[2] M. B. Taylor, et al. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. Proceedings of International Symposium on Computer Architecture, June 2004

[3] M. B. Taylor, et al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. IEEE Micro (Mar 2002), pp. 25–35.

[4] M. B. Taylor. Design Decisions in the Implementation of a Raw Architecture Workstation. MIT MS Thesis, Cambridge, MA, September, 1999

[5] A. Agarwal, et al. The Raw Compiler Project. Proceedings of the Second SUIF Compiler Workshop, Stanford, CA, August , 1997.

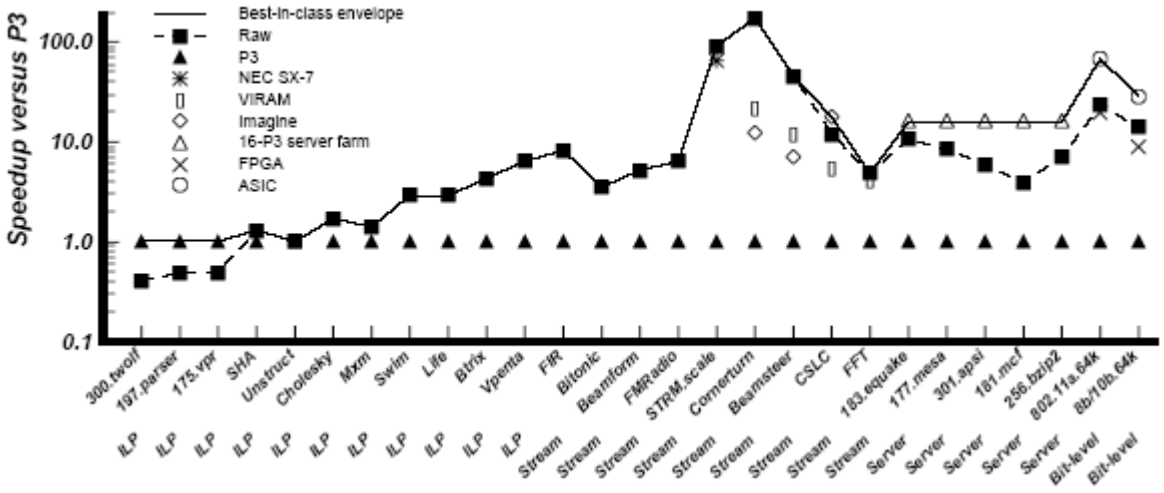


Figura 5 – Desempenho comparativo para várias aplicações

Conjunto de instruções Multimídia em Processadores de Propósito Geral

Fábio Augusto Menocci Cappabianco
Universidade Estadual de Campinas
Caixa Postal 6176, 13084-971 Campinas, SP - BRASIL
Campinas-SP, Brasil
fabioamc@ic.unicamp.br

ABSTRACT

Nos últimos anos, os microprocessadores receberam um novo impulso de aplicações multimídia, como áudio, vídeo e gráficos 3D. Tornou-se necessário obter maior desempenho e um tratamento mais específico da parte de processadores de propósito geral. Este trabalho é um survey a respeito de instruções multimídia em microprocessadores de propósito geral. São apresentados um histórico da computação multimídia, as principais arquiteturas desenvolvidas, uma descrição das instruções mais comuns e uma comparação de desempenho entre arquiteturas. Também são abrangidos os principais problemas e dificuldades encontrados na programação com instruções multimídia e algumas possíveis soluções e melhoras.

Categories and Subject Descriptors

H.5.1 [**Information Systems**]: Multimedia information systems; C.1.2 [**Processor Architectures**]: Multiple data Stream Architectures, SIMD; C.4 [**Performance of Systems**]

General Terms

Design Studies Performance

Keywords

MMX, SSE, 3DNow!, AltiVec, MDMX, VIS, VMX, multimedia, architecture, parallel, SIMD

1. INTRODUÇÃO

Desde a introdução dos primeiros conjuntos de instruções específicos para multimídia em processadores de propósito geral, aplicações de áudio, vídeo, gráficos 3D e outras têm tomado uma parcela cada vez maior de tempo e importância para os usuários [10].

Esta adaptação não foi trivial e ainda permite diversas melhorias, devido ao comportamento atípico dos programas multimídia. Na década passada, por exemplo, enquanto programas comuns utilizavam dados de 32 e 64 bits, aplicações de áudio e vídeo processavam dados de 8 e 16 bits [22, 21]. Além disso, algumas destas aplicações são preponderantemente seqüenciais. Portanto, foi fundamental tirar maior proveito de paralelismo com estruturas especializadas para decodificação de instruções e para a memória.

As soluções mais utilizadas atualmente são de sistemas embarcados especializados, para uma determinada aplicação multimídia, com baixo custo e baixo consumo de energia, e processadores de propósito geral como PC's para soluções genéricas que serão enfocados neste artigo.

A seção 2 contém uma breve revisão histórica do processamento multimídia. A seção 3 descreve os principais conjuntos de instrução para processadores de propósito geral, na seção 4, encontra-se uma descrição das principais instruções fornecidas pelos fabricantes e, por fim, a seção **??** conclui o trabalho.

2. HISTÓRICO DO PROCESSAMENTO MULTIMÍDIA

O processamento multimídia por computadores tem atualmente aplicações em diversas arquiteturas.

A arquitetura DSP, que ainda é bastante utilizada, remonta de 1980. Ela baseava-se em um pipeline formado por um multiplicador acoplado a uma ALU convencional. Assim, podia realizar operações do tipo MAC (multiply-and-accumulate) em paralelo. Por apresentar esta propriedade, as DSP's eram utilizadas como chips para modems e para codificação de voz, extremamente velozes para a época em comparação aos processadores de propósito geral. [12]

Em 1991, os processadores de arquitetura DSP ganharam campo em processamento de vídeo devido ao aumento de sua freqüência de execução e a utilização em modo de operação single instruction stream, multiple data stream(SIMD). Em 1993, os processadores DSP ganharam campo em processamento multimídia de aplicações no formato MPEG-2, devido ao uso de very long instruction words(VLIW) no modo de operação SIMD, denominado SIMD+ [15]. Estes processadores passaram a ser muito utilizados em receptores e tratadores de sinal de TV a cabo e de vídeo sob demanda, DVD's e outros dispositivos.

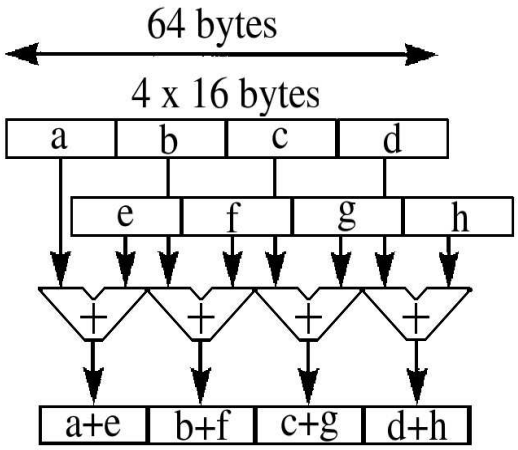


Figure 1: Exemplo de adição de palavras pequenas.

Alguns chips de aplicação específica também ganharam bastante espaço. Os chips de aceleração gráfica 3D em PC's, por exemplo, são um grande avanço no tratamento de funções multimídia, desde o início dos anos 90. Eles tratam de maneira extremamente rápida a renderização de polígonos em cenas tridimensionais [9]. Aplicações em FPGA's também ganharam seu espaço para processamento de imagens e sons se mostrando uma alternativa barata e de fácil implementação, com um desempenho muito acima dos processadores de propósito geral [19, 2].

Os microprocessadores, por fim, começaram a ficar interessantes para o tratamento de aplicações multimídia a partir do momento em que apresentaram um conjunto de instruções específico para elas. Como os dados multimídia são representados por 8 ou 16 bits e as arquiteturas possuíam registradores e ALU's que tratavam menos 32 bits, algum paralelismo pôde ser obtido por dividir uma ALU ou registrador de palavras longas em várias ALU's ou registradores de palavras pequenas, denominadas palavras ou dados particionados. Desta forma cada registrador e ALU particionados ficam responsáveis por processar um dado multimídia.A Figura 1 ilustra uma adição de palavras particionadas de 16 bits provenientes da divisão de uma palavra de 64 bits utilizando também várias ALU's pequenas que juntas formam uma ALU grande.

O paralelismo em microprocessadores de aplicações multimídia iniciou-se com instruções gráficas [8]. A seguir, aplicou-se também à decodificação de imagens [13]. Por fim, abrangeu o processamento de vídeos. A Tabela 1 contém alguns dos conjuntos de instrução mais conhecidos neste contexto.

Estes conjuntos de instrução foram tentativas de gerar um suporte mais apropriado para aplicações variadas de multimídia. Um novo conjunto de instruções era idealizado na tentativa das empresas de atender melhor à demanda do mercado e assim conseguir estabelecer o seu padrão como o principal. Atualmente, o padrão mais utilizado é o SSE em suas três versões, adotado tela Intel e pela AMD. Segue uma breve descrição destes conjuntos.

3. CONJUNTOS DE INSTRUÇÃO

Como mostra a Tabela 1 existe uma grande variedade no tipo e tamanho das extensões multimídia. Inicialmente pode-se classificar as extensões em três classes:

1. as que compartilham o banco de registradores inteiro e, conseqüentemente suas unidades de processamento, com os registradores e instruções multimídia;
2. as que compartilham o banco de registradores e unidades de processamento de ponto flutuante com os registradores e instruções multimídia;
3. as que possuem um banco de registradores exclusivo para registradores e instruções multimídia.

A primeira classe caiu em desuso por questões arquiteturais que tornam mais complexa a implementação e pode acarretar em perda de desempenho. Também por tomar espaço de operações com ponteiros, constantes e variáveis de desvios que são inteiros com dados multimídia, piorando ainda mais o desempenho dos programas.

A segunda classe ainda é utilizada em algumas arquiteturas, pois dificilmente instruções de ponto flutuante são utilizadas juntamente com instruções multimídia.

Já a terceira classe é a mais comum atualmente, principalmente por causa da demanda por registradores multimídia de 128 bits, para ganhar mais paralelismo, enquanto os registradores de ponto flutuante variavam entre 64 e 80 bits.

A seguir, há uma breve descrição de cada uma destes padrões quanto às suas instruções, metas de projeto e outras observações.

3.1 MAX-1 da HP

MAX-1 da Hewlett Packard foi a primeira extensão multimídia para microprocessadores que operava apenas sobre números inteiros. Ele permitia a decodificação de software MPEG-1 de resolução 352x240 pixels em um CPU PA-7100LC de 80MHz. Antes de possuir instruções multimídia, a estação HP720 de 50MHz processava um vídeo sem áudio em 4-5fps. O PA-7100LC foi idealizado para realizar operações multimídia, sem impactar no agendamento, complexidade, período de clock e área de chip, sendo portanto, bastante simples [14]. Como se tratava de uma arquitetura RISC e seguiu-se as mesmas diretivas de projeto da mesma, observou-se as operações multimídia mais freqüentes, quebrando-as em primitivas mais rápidas que foram implementadas em hardware [13]. Só com esta melhoria um vídeo com áudio podia ser executado a 30fps.

3.2 VIS da Sun

As operações do Visual Instruction Set (VIS) também atuam apenas sobre inteiros, utilizando, porém o banco de registradores de ponto flutuante. O VIS foi implementado inicialmente nos processadores UltraSPARC I, II e III e sua

Table 1: Principais conjuntos de instrução multimídia

Conjunto de Instuções	Empresa	Ano	instruções	banco de registradores
MAX-1	HP	1994	9	Integer(31x64b)
VIS	SUN	1995	121	FP(32x64b)
MAX-2	HP	1995	8	Integer(32x64b)
MDMX	MIPS	1996	29	FP(32x64b)+Ac(1x192b)
MIPS-64	MIPS	1996	74	FP(32x64b)
MMX	Intel	1997	57	FP(8x64b)
MVI	DEC	1997	13	Integer(31x64b)
Extended MMX	Cyrix	1997	12	FP(8x64b)
3DNow!	AMD	1998	21	FP(8x64b)
SSE	Intel	1999	70	8x128b
AltiVec	Motorola	1999	162	32x128b
MIPS-3D	MIPS	1999	23	FP(32x64b)
Enhanced 3DNow!	AMD	1999	24	FP(8x64b)
SSE2	intel	2000	144	8x128b
3DNow! Professional	AMD	2000	70	8x128b
SSE3	intel	2004	157	8x128b

principal motivação foi de criar uma plataforma padrão para aplicações multimídia como visualização 3D e codificação MPEG-1, MPEG-2. Antes do VIS, aplicações gráficas necessitavam de hardware especializado, e a partir de então o custo de sistemas diminuiu e slots de extensão foram economizados. O conjunto de instruções do VIS foi gerado, baseando-se em aplicações gráficas e multimídia. Todas as instruções são caracterizadas por: serem executadas em apenas um ciclo ou serem particionada em pipeline facilmente; serem aplicáveis a diversos algoritmos; e não afetarem o período de clock [24].

3.3 MAX-2 da HP

O MAX-2 fez parte de uma segunda abordagem multimídia utilizada pela HP em uma processadores da arquitetura PA-RISC 2.0. Novamente o MAX-2 oferece um conjunto reduzido de primitivas para as instruções multimídia, no entanto a arquitetura PA-RISC possui recursos adicionais. Por exemplo, duas abordagens diferentes foram utilizadas para multiplicação, dependendo do fluxo de mídia a ser processado. Para aplicações de áudio e gráficos 3D, uma unidade multiplicadora e acumuladora para ponto flutuante é utilizada. Para aplicações de baixa precisão, como multiplicação por constantes, estas são realizadas por séries de shifts e adições provenientes de instruções MAX-2 [**?**].

3.4 MDMX da MIPS

O padrão MCMX especifica que um banco de processadores de ponto flutuante contendo 32 registradores de 64 bits seja compartilhado para o uso de instruções multimídia sobre números inteiros. Além destes, foi especificado um registrador de 192 bits com o propósito de ser acumulador de operações. Este acumulador é especialmente útil pois operações de multiplicação ou uma sequência de somas pode ocupar mais bits que os registradores fontes suportam, obtendo-se assim resultados mais precisos [20].

3.5 MIPS-64 da MIPS

O conjunto de instruções MIPS-64 foi o primeiro a implementar instruções multimídia para registradores de ponto flutuante a serem executados dentro do microprocessador. Seu principal objetivo foi complementar o MDMX para aplicações de OpenGL como geometria 3D e Virtual Reality Modeling Language(VRML) dentre outras [11].

3.6 MMX da Intel

O padrão MMX se limitou a instruções multimídia de números inteiros, utilizando o conjunto de registradores de ponto flutuante a partir do processador Pentium P55. Isto impossibilitava que operações multimídia e de ponto flutuante fossem executadas simultaneamente. Foi com certeza o mais popular de todos os conjuntos de instrução até a própria Intel lançar o conjunto de instruções SSE [18].

A prioridade do projeto do MMX foi de aprimorar o processamento de aplicações multimídia, de comunicações de internet, incluindo vídeos nos formatos MPEG-1 e MPEG-2, síntese de músicas, compressão e reconhecimento de fala, processamento de imagens, gráficos 3D, vídeo conferências, áudio e modems. Na avaliação da Intel, desejou-se executar com maior eficiência as seqüências de código mais freqüentes e desejou-se também manter a compatibilidade com a arquitetura dos processadores anteriores.

3.7 MVI da DEC

As instruções Motion Vídeo Instructions (MVI) foram incorporadas primeiramente pelo Alpha 21164PC. A DEC, idealizadora do MVI, foi comprada pela HP. O principal foco da DEC, para o padrão MVI eram a reprodução de teleconferências e DVS a 30 fps com áudio stereo. Desejava-se a reprodução comparável aos hardwares dedicados. [3] A DEC implementou apenas 13 instruções de multimídia alegando que a largura de banda de memória era insuficiente para alimentar o processador com tanto paralelismo. A DEC alegou também que o conjunto MMX era complexo devido às deficiências das arquiteturas anteriores que necessitavam ser cobertas [17].

3.8 Extended MMX da Cyrix

Assim como a AMD a Cyrix licenciou o uso da extensão MMX da Intel. A Cyrix não produz mais processadores desde 1997 quando juntou-se à National Semiconductors.

A Cyrix expandiu as instruções de MMX para evitar destruição de dados durante operações. Enquanto o padrão MMX utilizava como registrador de destino um dos registradores de origem, o padrão Extended MMX implicitamente escolhia um registrador de destino diferente, dependendo dos registradores de origem [25].

3.9 3DNow! da AMD

Depois de licenciar o padrão MMX da Intel a AMD expandiu este conjunto de instruções, criando o padrão 3DNow! para o processador K6. Esta extensão utiliza as mesmas instruções multimídia para números inteiros e o mesmo banco de registradores. A inovação foi incrementar instruções para números de ponto flutuante. Duas operações de ponto flutuante de precisão simples eram executadas ao mesmo tempo. O motivo da AMD utilizar a expansão MMX da Intel foi de não querer gastar recursos com uma arquitetura que requeresse maior área ou de complexidade, sem muitos benefícios de desempenho [16].

3.10 SSE da Intel

Streaming SIMD Extension (SSE) foi a primeira expansão da Intel ao conjunto de instruções MMX. Quando proposta era denominada Internet Streaming SIMD Extension (ISSE), mas a aplicação foi além da Internet. Ela foi desenvolvida para cálculos de geometria 3D, renderização, codificação e decodificação de vídeos e reconhecimento de voz. A grande diferença desta arquitetura é de suportar cálculos com números de ponto flutuante. Além disso, incorporou instruções sugeridas por vendedores de software multimídia.

O conjunto de instruções SSE trouxe outra grande alteração na arquitetura Intel, o que não ocorria desde a migração de 16 para 32 bits do 80286 para o 80386. Esta mudança foi a adição de um novo conjunto de registradores exclusivo para operações multimídia, tirando os dados multimídia dos registradores de ponto flutuante. Isto simplificou a implementação de novos processadores e possibilitou a utilização de instruções de ponto flutuante ao mesmo tempo que instruções multimídia eram processadas.

O Pentium III foi o primeiro processador a adotar o padrão SSE. Ele foi desenvolvido para executar duas operações de ponto flutuante de 64 bits por ciclo de clock, operando assim com até quatro dados de 32 bits, somando 128 bits ao mesmo tempo [23].

3.11 AltiVec da Motorola

O conjunto de instruções AltiVec foi utilizado pelo processador MPC 7400 da estação Apple G4, adicionando ao PowerPC unidades de execução de 128 bits. Ao contrário de muitas arquiteturas, que investiram limitadamente em instruções multimídia, o AltiVec ocupa grande parte da área do processador com inovações para o processamento multimídia, pois está mais focado em funcionalidades que em custos. Além disso, o paralelismo oferecido pelo AltiVec se aplica não apenas às instruções multimídia, mas a qualquer uma que possa atuar paralelamente.

O AltiVec, assim como o SSE utiliza um banco de registradores exclusivo para o processamento multimídia. Este banco de registradores, sendo de 32 bits facilita o trabalho de compiladores para paralelizar códigos de programas em pipeline e loop unrolling. As instruções desta extensão também são bem peculiares, pois permitem a especificação de quatro operandos (três fontes e um destino). Isto proporciona um grande largura de banda, permite instruções extras como multipply-add, permute, etc e evita trocas de valores entre registradores que perdem dados como no caso do SSE, acelerando bastante o processamento [4].

3.12 MIPS-3D da MIPS

O conjunto de instruções MIPS-3D consiste em uma extensão especifica para aplicações (ASEs), que são opcionais para se adicionar ao MIPS-64. Também foi projetado de maneira simples para sistemas embarcados de baixo consumo, a fim de realizarem processamento 3D [6].

3.13 Enhanced 3DNow! da AMD

Esta extensão da AMD aos seus conjuntos de instrução MMX e 3DNow! foi utilizada no processador Athlon. Ela adicionou operações particionadas de inteiros e de pontos flutuantes para que ficasse funcionalmente equivalente ao SSE.

3.14 SSE2 da Intel

Este conjunto de instruções foi inserido no processador Pentium 4. Ele adicionou basicamente instruções inteiras de multimídia realizadas nos registradores extras de 128 bits, que eram realizadas nos registradores de ponto flutuante e adicionou o tipo de dado de dupla precisão para aplicações científicas, de engenharia e raytracing [1].

3.15 3DNow! Professional da AMD

A extensão 3DNow! Professional foi utilizada no processador Athlon MP da AMD. Ela contém um conjunto extra de instruções para fazer total compatibilidade com o conjunto de instruções SSE e adicionalmente ao seu próprio conjunto de instruções Enhanced 3DNow! [5].

3.16 SSE3 da Intel

O SSE3 foi introduzido no mercado na versão Prescott do Pentium 4. As instruções novas deste conjunto ajudam a sincronizar threads e auxiliam em aplicações específicas como mídia e jogos. [7]

4. INSTRUÇÕES MULTIMÍDIA

Esta seção descreve as principais instruções multimídia dos conjuntos apresentados anteriormente. Primeiramente serão apresentadas as instruções sobre números inteiros, depois sobre as de ponto flutuante, seguidas das polimorfas, seguidas das de comparação e de fluxo e por último as sobre memória.

4.1 Instruções Aplicadas a Inteiros

Todas as extensões apresentadas tratam de números inteiros, apesar de haver grandes variações de tipo e largura dos dados utilizados em cada uma delas. A Tabela 2 lista as instruções inteiras nas arquiteturas que foram examinadas na seção anterior.

4.1.1 Conversão de Tipo e Largura de Dados

Mudança na largura do dado é fundamental para instruções multimídia. A operação chamada empacotamento reduz a largura de um dado. Por exemplo, um registrador de 64

Table 2: Instruções multimídia de números inteiros das arquiteturas. Os números representam a quantidade de bits dos operandos de cada instrução. Quando precedidos de 'U','S' e 'FP' operam sobre inteiros sem sinal inteiros sinalizados e números de ponto flutuante, respectivamente. Quando seguidos de '*', significa que a instrução é possível, mas através de outras instruções.

	AMD	Cyrix	DEC	HP	Intel	MIPS	Motorola	Sun
Add/Sub Resto	8,16,32	8,16,32	-	16	8,16,32,64	8,16	8,16,32	16,32
Add/Sub Sat.	8,16	8,16	-	16	8,16	S16	8,16,32	-
Média	U8,U16	U8	-	S16	U8,U16	-	8,16,32	-
Min/Max	U8,U16	S16	U8,S16	-	U8,S16	U8,S16	8,16,32	-
Mul Trucada	US16	S16	-	-	US16	-	S16	S16,S32
Mul Arredond	S16	S16	-	-	-	-	S16	-
Mul Completa	US16	S16	-	-	S16,U32	U8,S16	US8,US16	-
Right Shift Lgc	16,32,64	16,32,64	-	16	16,32,64	8,16	8,16,32	-
Right Shift Art	16,32	16,32	-	16	16,32	8,16	8,16,32	-
Left Shift Lgc	16,32,64	16,32,64	-	16	16,32,64	8,16	8,16,32	-
Merge	8,16,32	8,16,32	-	-	8,16,32	32	8,16,32	8
Align/Rotate	-	-	-	-	-	8	8	8
Shuffle	$16(4^4)$	-	-	$16(4^4)$	$16(4^4),32(4^4)$	8(8),16(8)	$8(32^16)$	-
Align/Rotate	16	-	-	-	16	-	8,16,32	-
Pack FP Satur	FP32→S32,16	-	-	-	FP32→S32	-	FP32→US32	-
Pack FP Trunc	-	-	-	-	FP64→FPS32	-	-	-
Pack 32b Satur	S32→S16	S32→S16	-	-	S32→S16	-	S32→S16	S32→8,16
Pack 16b Satur	S16→US8	S16→US8	-	-	S16→US8	-	S16→US8	S16→U8
Pack 32b Trunc	32→16*,8*	32→16*8*	32→8	32→16	32→16*,8*	32→16	32→16	-
Pack 16b Trunc	16→8*	16→8*	16→8	-	16→8*	16→8	16→8	-
Unpack FP	-	-	-	-	FP32→FP64	-	-	-
Unpack 32b	S32→FP32	-	-	-	S32→FP32,64	-	US32→FP32	-
Unpack 16b	US16→U*FP32	U16→U32*	-	16→32*	U16→U32*	U16→U32	U16→U32*	U16→U32
Unpack 8b	U8→U16*	U8→U16*	8→16,32	-	U8→U16*	US8→US16	U8→U16*	U8→U16*

bits contendo 4 palavras de 16 bits pode ser convertido por colocar os 8 bits menos ou mais significativos de cada palavra de 16 bits em 32 bits de outro registrador. A operação de desempacotamento, por sua vez aumenta a largura de um dado adicionando zeros à esquerda ou expandindo conforme o sinal do dado representado.

A expansão MVI possui empacotamento apenas dos bits mais significativos e desempacotamento apenas com expansão de zeros. A MAX, por sua vez, não possui operações diretas de empacotamento desempacotamento, utilizando outros recursos para executar tais operações.

Apesar dos dados em PC's atuais sejam de precisão de 32 e 64 bits, dados multimídia muitas vezes operam sobre 8 e 16 bits, por isso, as operações diversas de empacotamento e desempacotamento são extremamente importantes.

4.1.2 Saturação e Overflow

Tradicionalmente, nas instruções com número inteiros, se um ocorrer um overflow ou um underflow, a operação retorna o resto da divisão do valor resultante pelo número máximo ou mínimo que a arquitetura pode representar, respectivamente. Este comportamento é indesejável em muitas aplicações multimídia. Em processamento de imagens por exemplo, em uma soma de brilhos de pixels, caso o valor total exceda o máximo representável é desejável que o total retornado pela operação seja o brilho máximo e não o resto da divisão do total pelo número pelo máximo. A Figura 2 demonstra este tipo de operação que é denominada 'saturada'.

A implementação de aritmética saturada é um pouco diferente da aritmética comum, e exige um tratamento diferente para números com e sem sinal, o que aumenta um pouco a complexidade do hardware. Isso pesa na escolha do banco de registradores a ser utilizado pelas instruções multimídia. Caso sejam implementadas sobre os registradores inteiros como no caso das extensões MAX e MVI, pode haver aumento no número de ciclos para as operações básicas aritméticas, ou um aumento no período do clock. Isto com certeza seria uma característica indesejável.

4.1.3 Adição e Subtração

A Tabela **??** lista os tipos de adição e subtração nas arquiteturas que foram examinadas na seção anterior. A extensão AltiVec destaca-se por incluir instruções de 32 bits para adição e subtração saturada. No entanto, operações multimídia sobre som e imagem utilizam dados de 8 ou 16 bits, o que reduz a abrangência desta. Há um contraste com a extensão VIS da Sun que não possui adições ou subtrações com saturação, já que foi projetada para atuar preferencialmente sobre imagens. Apesar das extensões MIPS só possuírem operações de adição saturada para inteiros sinalizados de 16 bits, o acumulador da arquitetura consegue lidar eficientemente com dados enormes devido ao seu tamanho de 192 bits.

Somente o conjunto da DEC não possui adições sobre números particionados, requerendo operações intermediárias para evi-

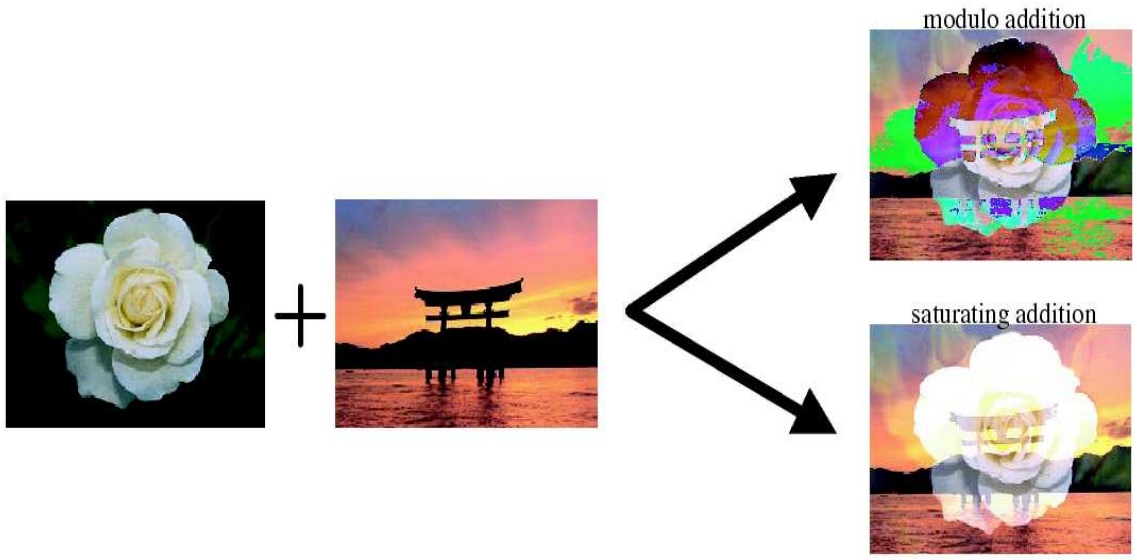


Figure 2: Adição de brilhos de pixels em imagem saturada e imagem de resto.

tar overflow e underflow.

4.1.4 Soma das Diferenças Absolutos

Uma das poucas instruções que a DEC colocou na extensão MVI é a de calcular a soma de da diferença absoluta de bytes desempacotados. O conjunto MMX, ao contrário, apesar de ser muito mais rica não possui tal instrução. Ela foi incluída nas arquiteturas da Intel, Cyrix e AMD em suas respectivas expansões do conjunto MMX, assim como a VIS. Esta instrução produz um grandioso benefício para instruções multimídia com um paralelismo de 8x para arquiteturas de 64 bits e substitui muitas operações escalares. Na decodificação de vídeos MPEG-2, por exemplo, a porção central do código pode ser substituída por duas operações de soma das diferenças absolutas.

4.1.5 Multiplicações

A multiplicação de elementos particionados, envolve a multiplicação dos elementos empacotados respectivos. As multiplicações particionadas são as instruções com a maior diferença entre as implementações. Elas são mais demoradas, levando de três a cinco ciclos e ocupam três vezes mais espaço no chip. Além disso, são difíceis de manejar, devido ao resultado ocupar até o dobro do espaço dos operandos fontes.

A complexidade da multiplicação particionada impediu que as extensões que possuem seu banco de registradores multimídia compartilhado com o banco de registradores inteiros implementássem-nas, que são o caso da MVI, MAX-1 e MAX-2. Por esta e outras razões, o compartilhamento do banco de registradores inteiros mostrou ser uma má idéia.

Por ser maior que os operandos de origem, o resultado da multiplicação teve diversos tratamentos.

1. **Redução: retorna a soma de elementos particionados do resultado da multiplicação, atuando como um multiply-add. A extensão MMX implementou esta instrução.**
2. **Par/impar: selecionar os elementos pares ou ímpares do resultado da multiplicação. AltiVec implementou este tratamento.**
3. **Truncado: São pré-determinados alguns bits que são perdidos após a multiplicação. Normalmente os bits menos significativos. A extensão MMX implementou esta instrução.**
4. **Registrador de destino com maior largura: Este é o caso do acumulador de 192 bits implementado na arquitetura MIPS com conjunto de instruções MDMX. Os acumuladores, no entanto, mostraram ser ao mesmo tempo um obstáculo para o aumento da velocidade de processamento, uma vez que todas as operações que causam overflow ou underflow passam por ele. Além disso, o acumulador impede o processamento superscalar como demonstra a história da arquitetura [11].**
5. **Primitivas de multiplicação: Esta solução adotada pela VIS é problemática. Ao invés de gerar o resultado completo de uma multiplicação, esta extensão possui primitivas ou sub-instruções para gerar o resultado. Isto aumenta as dependências de nomes, o que afeta o paralelismo e gera a necessidade por mais registradores.**

Uma opção muito sugerida por pesquisadores para minimizar os conflitos de instruções que geram números mais

largos que os operandos de origem é de utilizar uma representação dos dados diferente para armazenamento e para execução de instruções multimídia. Esta diferença de tamanhos reduz o gasto com instruções de empacotamento/desempacotamento, pois os dados são lidos da memória para o formato de execução e são rearranjados com saturação ou resto, se necessário para a armazenagem [21].

4.1.6 shifts ou deslocamentos

Os shifts servem para realizar multiplicações e divisões por potências de dois facilmente, alinhar dados, entre outras aplicações. São especialmente importantes em arquiteturas que não fornecem instruções de multiplicação.

4.1.7 Operações de comunicação de dados

Este tipo de instruções serve basicamente para rearranjar palavras particionadas em um registrador e entre registradores. Segue uma lista com as principais instruções de comunicação.

- *Merge:* alterna os elementos da parte superior ou inferior de dois registradores, a instrução
- *Align ou rotate:* permite o re-ajuatamento de palavras particionadas em dois registradores, as instruções
- *Insert e extract:* permitem que um elemento empacotado seja extraído como um escalar ou que um escalar seja inserido como elemento empacotado, a instrução.
- *Shuffle:* Permite que se escolha uma ordem diferente para elementos de um palavra. Os conjuntos de instrução MDMX, MAX-1, MAX-2, MMX e AltiVec. Apesar de se mostrar uma excelente instrução é muito custosa para se implementar, pois exige um registrador auxiliar e muitos barramentos adicionais para a execução.

4.2 Instruções Aplicadas a Números de Ponto Flutuante

A maioria das aplicações de ponto flutuante para multimídia trabalha com dados de precisão simples (32 bits), ou precisão dupla (64 bits). Todas as extensões multimídia iniciaram com suporte apenas para dados de precisão simples. A primeira dar suporte a precisão dupla foi a SSE2 da Intel. A partir de então, as outras extensões passaram a ter este recurso.

Apenas quatro linhagens de conjuntos de instrução aplicam-se a ponto flutuante multimídia: SSE, 3DNow!, AltiVec e MIPS-64/3D.

Além das instruções básicas aritméticas, as instruções de ponto flutuante particionadas incluem algum tipo de aproximação de inversão e raiz quadrada. Estas instruções são muito utilizadas pela geometria 3D. Elas são implementadas através de tabelas pré-estabelecidas, que facilitam seu cálculo. As extensões SSE e AltiVec retornam valores com precisão de 12 bits para estas operações, a 3DNow! retorna 14 e 15 bits e a MISP-3D retorna precisão de 24 bits.

Exceções são um problema para operações de ponto flutuante, pois é muito complexo determinar o elemento que gerou a exceção. Portanto, os conjuntos 3DNow! não geram instruções. O conjunto de instruções AltiVec, por outro lado, ao invés de gerar uma interrupção, faz com que o elemento do resultado da operação, onde ocorre overflow ou underflow, sofra alguma alteração, assim como no caso da saturação.

Os conjuntos SSEs incluem registradores de controle/status para exibir as exceções, no entanto o elemento particionado que sofreu a exceção não é apontado. O MIPS-64 tem uma abordagem semelhante aos SSEs.

4.3 Operações Polimorfas

Este tipo de instrução é aplicado aos dados independentemente de seu particionamento. As extensões MVI e MAX-1 e 2 não possuem nenhuma adição de lógica para suas arquiteturas, pelo fato de pertencerem à primeira classe de conjunto multimídia, que compartilha o banco de registradores e unidades de processamento inteiros. Estas unidades de processamento já são providas da lógica necessária, e portanto, não requer instruções adicionais.

No caso do MIPS, somente a extensão MDMX provê lógica adicional, utilizada sobre as unidades de processamento de ponto flutuante compartilhadas. As demais arquiteturas possuem este tipo de instrução.

4.4 Comparação e Controle de Fluxo

A computação em paralelo fica complicada caso uma das operações deva ser realizada condicionalmente. Como visto anteriormente, não há como gerar um flag ou exceção diferente para cada operação executada paralelamente. Duas soluções para o problema foram idealizadas.

A primeira é utilizar elementos máscara. Estes são conjuntos de bits do tamanho do elemento comparado compostos só de zeros (0x0000 para 16 bits) ou somente de uns (0xffff para 16 bits). Estes elementos são utilizados em conjunto com operações lógicas do tipo AND, NAND ou OR para atingir o resultado desejado. A vantagem de se utilizar este tipo de tratamento é de não criar dependências extras com os elementos máscara, mas ter somente a dependência do resultado da operação lógica.

A segunda consiste em gerar um vetor de bits, onde um único bit representa o resultado de uma comparação lógica para cada elemento particionado. Os bits restantes podem ser armazenados se necessário.

4.5 Memória

As operações de memória para multimídia são apenas Load e Store. Não utiliza-se referência a memória em nenhuma instrução de nenhum dos conjuntos de instrução multimídia apresentados.O tipo mais comum de leitura e escrita são os seqüenciais, indicados apenas pelo endereço do dado a ser lido/escrito e seu tamanho.

No entanto, algumas aplicações são mais difíceis de serem tratadas no seu fluxo de dados da memória. Isso acontece muito em imagens com três canais, RGB, onde cada canal é armazenado seqüencialmente, e deseja-se adquirir as três componentes de um pixel que estão situadas em posições

diferentes da memória, porém com certa regularidade de intervalo. Assim, o load e store poderiam ser realizados a partir de três atributos: largura dos elementos lidos; deslocamento do dado lido a partir do endereço base para iniciar a leitura/escrita; e distância de um endereço efetivo de um elemento para o próximo a ser lido.

Mais detalhes sobre instruções de comparação e controle de fluxo e de memória encontram-se em [22, 21, 12].

5. CONCLUSÕES

O aumento da demanda e do número de aplicações multimídia levou à introdução dos conjuntos de instruções específicos para multimídia em processadores de propósito geral.

Como visto, surgiram diversas extensões de grandes empresas, com a finalidade de atender a esta demanda. Diferentemente do que ocorreu em relação às instruções comuns de propósito geral, os projetos mais ousados e com mais recursos foram mais bem sucedidos. Enquanto a MVI e a MAX-1 e 2 tiveram desempenho baixo, arquiteturas mais complexas como SSE e AltiVec obtiveram um resultado melhor.

Outra visão dada pelo artigo foi a divisão das classes de expansões multimídia, quanto ao banco de registradores e seus recursos de processamento. Dentre as classes, a que destacou mais foi a que utiliza os bancos de registradores específicos para multimídia, pois facilitam a implementação e fornecem maior grau de paralelismo.

A última contribuição foi a visão geral dos tipos de instruções multimídia existentes, com uma análise crítica de algumas delas e a correlação com as extensões que as fornecem.

6. REFERENCES

[1] A. Kumar. *SSE2 Optimization - OpenGL Data Stream Case Study*, 1998.

[2] M. G. Alina N. Moga. Parallel image component labeling with watershed transformation. *IEEE Transactions on Pattern Analisys and Machine Inteligence*, 19(5):441–450, May 1997.

[3] R. Carlson, D.A.; Castelino and R. Mueller. Multimedia extensions for a 550-mhz risc microprocessor. *IEEE Journal of Solid-State Circuits*, 32:1618–1624, Nov 1997.

[4] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scale. Altivec extension to powerpc accelerates media processing. *Micro, IEEE*, 20:85–95, Apr 2000.

[5] J. Huynh. The amd athlontm mp processor with 512kb l2 cache. *AMD White Paper*, pages 1–16, May 2003.

[6] M. T. Inc. A process-portable 64b embedded microprocessor with graphicsextensions and a 3.6 gb/s interface. *IEEE International Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC.*, pages 234–235,451, 2001.

[7] Intel. Processors - define sse2 and sse3. *Intel White Paper*, page 1, Oct 2005.

[8] Intel corporate literature. *i860TM microprocessor family programmers reference manual*, 1992.

[9] Y.-W. Jeon, Y.-S. Kwon, Y.-H. Im, J.-H. Lee, S.-J. Nam, B.-W. Kim, and C.-M. Kyung. 3d graphics system with vliw processor for geometry acceleration. *Proceedings of the Second IEEE Asia Pacific Conference on ASICs - AP-ASIC*, pages 367–370, Aug 2000.

[10] M. V. Jesus Corbal, Roger Espasa. Mom: a matrix simd instruction set architecture for multimedia applications. *ACM/IEEE conference on Supercomputing - CDROM*, pages 1–12, Jan 1999.

[11] D. P. John Hennessy. *Computer Architecture A Quantitative Approach, Fourth Edition*. Elsevier, 2006.

[12] T. Kuroda, I.; Nishitani. Multimedia processors. *Proceedings of the IEEE*, 86:1203–1221, Jun 1998.

[13] R. Lee. Accelerating multimedia with enhanced microprocessors. *Micro, IEEE*, 15:22–32, Apr 1995.

[14] R. Lee. Realtime mpeg video via software decompression on a pa-risc processor. *'Technologies for the Information Superhighway' Compcon.*, pages 186–192, Mar 1995.

[15] T. Nishitani. Trend and perspective on domain specific programmable chips. *Signal Processing Systems, SIPS - IEEE Workshop on Design and Implementation.*, pages 1–8, Nov 1997.

[16] G. W. F. Oberman, S.; Favor. Amd 3dnow! technology: architecture and implementations. *Micro, IEEE*, 19:37–48, Apr 1999.

[17] M. M. P Rubinfeld, B Rose. Motion video instruction extensions for alpha. *Digital Equipment Corporation-enlight.ru*, pages 1–13, Oct 1996.

[18] U. Peleg, A.; Weiser. Mmx technology extension to the intel architecture. *Micro, IEEE*, 16:42–50, Aug 1996.

[19] I. Rambabu, C.; Chakrabarti and A. Mahanta. Flooding-based watershed algorithm and its prototype hardware architecture. *Image and Signal Processing, IEE Proceedings-Vision*, 151:224–234, Jun 2004.

[20] Silicon Graphics. *MIPS Extension for Digital Media with 3D*, 1997.

[21] A. Slingerland, N.; Smith. Measuring the performance of multimedia instruction sets. *IEEE Transactions on Computers*, 51:1317–1332, Nov 2002.

[22] N. T. Slingerland and A. J. Smith. Multimedia instruction sets for general purpose microprocessors: a survey. Technical report, Berkeley, CA, USA, 2000.

[23] T. Thakkur, S.; Huff. Internet streaming simd extensions. *Computer*, 32:26–34, Dec 1999.

[24] J. Tremblay, M.; O'Connor. Ultrasparc i: a four-issue processor supporting multimedia. *Micro, IEEE*, 16:42–50, Apr 1996.

[25] VIA-Cyrix. *Application Note 108: Cyrix Extended MMX Instruction Set*, 1998.