An Integrated Memory Array Processor Architecture for Embedded Image Recognition Systems

Shorin Kyo, NEC Corporation
Shin'ichiro Okazaki, NEC Corporation
Tamio Arai, University of Tokyo
Conferência ISCA 2005,
Resumo por: Leandro Rodrigues Magalhães de Marco **RA** 009089

Existem muitos esforços para desenvolver processadores embarcados para aplicações multimídia, com performance suficiente e baixos custos (que envolvem também baixo consumo de energia). O artigo descreve o IMAP, que é um processador SIMD (Single Instruction, Multiple Data), que se propõe a ser um processador embarcado para sistemas de reconhecimento de imagem.

Tarefas de reconhecimento de imagens de vídeo, assim como aplicações multimídia, possuem um alto nível de pararelismo e necessitam de resposta em tempo real. Entretanto algoritmos de reconhecimento de imagens tem uma diversidade muito maior, o que torna necessário além de performance, flexibilidade.

Devido a essa necessidade por flexibilidade, foi adotada a estratégia de desenvolver técnicas de paralelização que permitam mapear diversas tarefas de reconhecimento de imagens em um processador SIMD altamente paralelo.

Para facilitar a técnica de paralelização, foram definidos sete grupos de operações que por sua vez foram classificadas em quatro categorias de acesso à memória. Esta classificação levou em conta critérios de localidade e confinamento da seqüencia de atualização de pixels. Com esses padrões em mente, foi feita a escolha de uma arquitetura em que cada PE (elemento de processamento) possuí uma quantidade de memória endereçável, e está conectado aos outros PEs em anel. Nesta arquitetura quatro técnicas de paralelização foram definidas. Cada uma para atender uma das categorias de acesso a memória.

Para uma implementação eficiente, e para utilizar um alto nível de paralelismo no IMAP, uma extensão da linguagem C, chamada C unidimensional (One Dimensional C, IDC) foi utilizada. No IDC, entidades associadas ao vetor de PEs são declaradas com a palavra-chave sep (ou separate). Quando o sep é utilizado, são denotadas operações explicitamente paralelas. Ou seja, diversos PEs utilizarão aquela entidade ao mesmo tempo.

O IMAP é constituído de um procesador de controle (CP), que é um RISC de 16 bits de propósito geral com um pipeline de 6 estágios, um vetor de 128 PEs (16 grupos de 8 PEs). Cada PE é um RISC de 8 bits com um pipeline de 3 estágios. O CP possuí 32 Kb de cache de programas e 2 Kb de cache de dados. Cada PE acessa 2 Kb de memória (IMEM). A soma de todas as IMEMs (256 Kb) é o espaço de trabalho dos PEs. Entidades declaradas com sep, ficam armazenadas na IMEM e são mapeadas na EMEM de 64 M que além de servir de swap para a IMEM, também é cache de programa e dados para o CP. O consumo de energia do IMAP é de em média 2 Watts.

Os resultados da utilização do IMAP de 100 MHz, comparado a um Pentium 4 de 2.4 GHz, utilizando códigos IPC para o IMAP e códigos seqüenciais para o Pentium 4 mostram um ganho proporcional ao nível de paralelismo esperado. Em todos os casos O IMAP supera o Pentium 4. Quando comparado também a códigos com otimizações MMX, códigos IDC rodando no IMAP são 3 vezes mais rápidos do que códigos MMX e 8 vezes mais rápidos do que código C sem otimizações. Foi feito um benchmark com uma aplicação de detecção de veículos. Este revelou dois casos em que o Pentium 4 se sai melhor que o IMAP. Em um dos casos o paralelismo existente era modesto comparado ao númeor de PEs. No outro caso, um processamento seqüencial era necessário.

Pelos resultados pode-se constatar a performance alcançada pelo IMAP. Um IMAP de 100 MHz, consumindo apenas 2 Watts, oferece uma performance em média quatro vezes maior que a de un Pentium 4 de 2.4 GHz que consome cerca de 100 Watts. Entretanto, o benchmark feito com uma aplicação do mundo real mostrou que em alguns casos o fato da arquitetura ser puramente SIMD inibe um uso mais eficiente. Direções futuras apontadas envolvem melhorias de performance e maior flexibilidade no controle do vetor de PEs.

Wrong Path Events: Exploiting Unusual and Illegal Program Behavior for Early Misprediction Detection and Recovery

D. N. Armstrong, H. Kim, O. Mutlu, and Y. N. Patt. Wrong Path Events: Exploiting Unusual and Illegal Program Behavior for Early Misprediction Detection and Recovery; pp. 119 - 128; The 37th Annual International Symposium on Microarchitecture, 2004

Rafael Augusto Scaraficci - 009649

Este artigo propõe uma solução em hardware para se tentar resolver *mispredicted branches* [1] antes que eles sejam resolvidos pelo processador, diminuindo, desta forma, a penalidade paga por se tomar (especular) um caminho de execução de maneira incorreta. Trata-se de um mecanismo especulativo que explora a execução de operações ilegais e operações que são executadas mais frequentemente nos *branches* especulados incorretamente. Estas operações são denominadas como *Wrong Path Events* (WPE) e consistem de eventos como acesso a uma posição de memória referenciada por um ponteiro NULL, tentativa de escrita numa página de leitura, leitura ou escrita num endereço desalinhado, exceções aritméticas, etc.

Os autores do artigo verificaram através de um simulador, que se utiliza de um mecanismo confiável para predição de *branches* com 64K entradas e com um pipeline de latência de 30 ciclos para *branches* especulados incorretamente, que nos benchmarks de inteiros do SPEC2000, 1,6% dos *mispredicted branches* produzem um WPE e em média esses WPEs ocorrem 51 ciclos antes dos *branches* que os originaram serem resolvidos pelo processador.

O mecanismo proposto é capaz de detectar um determinado conjunto de WPEs e identificar na maioria dos casos o *branch* responsável por tal evento, permitindo que o processador tome o caminho correto de execução antes que a instrução de *branch* seja executada. A identificação é feita através de um mecanismo de predição baseado em história, que memoriza a relação entre a instrução que gerou o WPE e o *branch* que foi especulado incorretamente. Este mecanismo baseia-se nas seguintes observações:

- Muitas instruções que geram um WPE se repetem durante a execução do programa;
- Se uma instrução A gera um WPE devido à especulação incorreta de um *branch* B, a próxima vez que a instrução A gerar um novo WPE provavelmente deve-se ao *branch* B;

Toda vez que o mecanismo de predição identifica um WPE ele toma uma das decisões:

- Bloqueia o *fetch* de instruções para economizar energia (mecanismo de predição não conseguiu identificar o *branch* responsável pelo WEP);
- Inicia a recuperação do *branch* que foi especulado e originou o WPE (mecanismo de predição conseguiu identificar o *branch* responsável pelo WEP);

Não há nenhuma garantia que o *branch* identificado tenha sido especulado incorretamente, portanto o processo de recuperação pode ser executado para um *branch* especulado corretamente, degradando o desempenho. No entanto, foi verificado que isso é pouco frequente para o SPEC2000 de inteiros. Em média, o mecanismo erra em 4% das vezes, acerta em 69% das previsões e nas demais vezes não consegue identificar o *branch*. O resultado final foi um ganho de aproximadamente 0,6% no IPC o que não justifica a implementação imediata deste mecanismo num processador real. Para ser aplicável faz-se necessário a descoberta de novos WPEs de forma a aumentar, principalmente o número de *mispredicted branches* que geram um WPE e também seria interessante que esses WEPs ocorressem mais precocemente, justificando a recuperação do caminho correto de execução antes que a instrução de *branch* seja executada pelo processador.

[1] Resolver um *mispredicted branch* significa detectar que o processador especulou um caminho de execução de maneira incorreta e recuperar-se desta predição indo para o caminho correto de execução.

Douglas José Soares Rodrigues – ra011104

Luiz André Barroso, Jeffrey Dean, Urs Hölzle
Web Search for a Planet: The Google Cluster Architecture
pp. 119-129 – IEEE Micro 2003

Poucas aplicações demandam tanta computação por requisição quanto os buscadores de internet. A abordagem do Google para a solução deste problema é a utilização de um cluster de mais de 15 000 PCs, executando software tolerante a falhas. Os mais importantes fatores que influenciam esta abordagem são o uso eficiente de energia e a taxa preço/desempenho. Neste ambiente, o consumo de energia e a refrigeração são fatores muito significativos.

O objetivo da arquitetura é prover a confiabilidade da aplicação no software, viabilizando a utilização de PCs comuns com elevado poder computacional, a um baixo custo. A confiabilidade da aplicação se deve à distribuição do serviço entre várias máquinas, e a detecção automática de falhas. Os computadores do cluster estão distribuídos em vários sites, e um mecanismo de load-balancing é responsável pela distribuição das requisições.

A execução da consulta consiste em dois passos: primeiro, as palavras que compõe a busca são localizadas em índices invertidos, que mapeiam uma palavra em uma lista de documentos; depois as listas de documentos são unificadas para ordenar os resultados. Este processo é altamente paralelizável, a partir da divisão do índice em vários pedaços, cada um com um subconjunto de documentos escolhido aleatoriamente.

Os racks do Google são compostos por 40 processadores 80x86. Diferentes gerações de processadores compõem os racks, desde o Celeron de 533 MHz ao Pentium II de 1.4 GHz. Cada servidor contém um disco IDE de 80 Gb. Os servidores em um mesmo rack se comunicam por uma interface Ethernet de 100-Mbps, enquanto os racks são ligados a um switch gigabit. O critério de seleção dos processadores é o custo por consulta, na forma da soma dos gastos (incluindo a depreciação) e os custos operacionais dividida pelo desempenho.

A tabela abaixo mostra algumas estatísticas dos servidores de índices

Característica	***Valor***
Ciclos por instrução	1.1
Branch mispredict	5%
Misses de instruções no cache L1	0.4%
Misses de dados no cache L1	0.7%
Misses no cache L2	0.3%

A aplicação apresenta um CPI razoavelmente alto, considerando que o Pentium III é capaz de despachar três instruções por ciclo. A mesma carga executando num Pentium 4 apresenta aproximadamente o dobro do CPI, e a mesma taxa de "branch mispredict", mesmo podendo fazer o despacho de mais instruções paralelamente, e tendo uma lógica de predição de branches mais elaborada. De fato, não há muito paralelismo em nível de instruções a ser explorado neste tipo de aplicação.

É observado um bom desempenho para o cache de instruções, reflexo do código de loop pequeno das consultas. Os dados do índice não se beneficiam de localidade temporal, devido à imprevisibilidade das buscas, entretanto a localidade espacial é altamente explorada, onde o pre-fetching do hardware (ou linhas de cache maiores) apresenta um desempenho melhor.

Universidade Estadual de Campinas
Cidade Universitária Zeferino Vaz, 24 de abril de 2006
João Paulo Porto 016377
Artigo: Improving program efficiency by packing instructions into registers, Hines S., Green J., Tyson G., Whalley D., ISCA 2005

Com o progresso científico-tecnológico da sociedade em que vivemos, o uso de sistemas embarcados (*embedded*) tem sido cada vez mais freqüente. Maneiras de diminuir o consumo de potência e o custo, bem como aumentar o desempenho, têm sido pesquisadas. Infelizmente, há poucas maneiras de melhorar todos os três parâmetros. É difícil encontrar, inclusive, maneiras de melhorar apenas um deles sem afetar negativamente, algum outro. A técnica que o artigo propõe consegue melhorar os três parâmetros.

Basicamente, a idéia é encontrar o conjunto de instruções mais freqüentes no código e colocá-las em um banco de registradores especial (IRF). Além do novo banco de registradores, foi adicionado uma tabela de constantes (IMM). A IMM (de entradas de 32 bits) contém as constantes mais presentes no código e são utilizadas para evitar o problema de ser necessário duas entradas para instruções que diferem apenas pelo imediato. A utilização desta técnica acaba compactando o código, aumentando a eficiência do estágio de *fetch* do pipeline do processador.

Para ilustrar o funcionamento da técnica, os autores do artigo apresentam uma versão modificada do processador MIPS. As alterações mais drásticas na ABI foram a inclusão de um novo formato de instrução e a modificação do formato das instruções: o campo *shamt* das instruções de formato R é utilizado, agora, para referenciar uma instrulção na IRF (a entrada zero da IRF é um *nop* que não é executado). Além disso, o campo *imm* das instruções do tipo I foi encolhido em 5 bits (para resolver este problema, o campo imm da instrução *lui* agora carrega constantes de 21 bits). Assim, com estas alterações é possível buscar duas instruções por *fetch*. Este tipo de compactação é chamado *loosely packed*.

Mas a modificação mais drástica foi a inclusão de novas instruções que permitem até cinco referências ao IRF. Estas instruções, cujo formato é chamado T, são ditas *tightly packed*. É possível haver até cinco referencias ao IRF em instruções do tipo T, ou 4 instruções e um parâmetro (um parâmetro é uma referência à IMM), ou ainda 3 instruções e dois parâmetros. O uso desta tabela para representação de constantes, ao invés de campos de imediatos nas próprias instruções representam uma melhora significante no total de instruções que são colocadas no IRF.

Para aumentar ainda mais a probabilidade de utilização da IRF, os autores propõe o uso de *positional registers*. Estes registradores não são referências aos registradores físicos da arquitetura, mas correspondem aos registradores utilizados em instruções anteriores à instrução atual.

Por fim, as instruções de desvio incondicional não podem ser facilmente compactadas, dado que o *offset* utilizado pode utilizar potencialmente todos os bits. Para solucionar isso, a técnica codifica os desvios cujo destino é representável em 5 bits (*offsets* variando de -16 a 15) como um desvio condicional que compara um registrador com ele mesmo, o que possibilita o uso de instrução do IRF com parâmetros.

Para fazer a geração de código para este processador alterado, é necessário um compilador especial. Este compilador deve detectar quais as instrução são candidatas a serem colocadas no IRF e fazer a alocação de registradores para estas instruções. A determinação de candidatos à alocação pode ser feita estaticamente ou com informação de *profiling*. Obviamente, a alocação utilizando informação sobre a execução do programa é mais eficiente. Entretanto, a alocação estática pode obter resultados razoáveis com o uso de heurísticas (e.g., favorecendo instruções de laços). O artigo apresenta o algorítmo utilizado para fazer a otimização.

Os resultados obtidos por esta técnica se mostraram deveras animadores. Devido à natureza das modificações, dificilmente o código utilizado nesta nova arquitetura seria maior que um código para a arquitetura simples. A técnica também reduziu o consumo de potência do processador (o que era esperado). O mais impressionante, contudo, foi a diminuição do tempo de execução de alguns dos programas do bechmark (provavelmente devido ao fato de ser necessário menos acesso à memória).

Os autores fazem ainda considerações sobre o aumento / diminuição do IRF. Em geral, o desempenho aumenta para IRF's maiores. Entretanto, a partir de 512 registradores de instrução ocorre uma degradação da performance da aplicação já que torna-se impossível representar mais que 2 instruções em uma instrução.

Para ambientes multi-processo, é possível utilizar uma IRF por processo, sendo necessário apenas o carregamento dos registradores com as instruções de cada processo. Entretanto é necessário salvar (e restaurar) os especificadores dos registradores posicionais.

MO401 1s2006 - Trabalho 1: Resumo do Artigo [KBG+04]

Tony Minoru Tamura Lopes - ra017502

24 de Abril de 2006

Em [KBG+04] é apresentada a arquitetura de processadores *Vector-Thread*(VT). Essa arquitetura tenta explorar extensivamente o paralelismo e a localidade, características fundamentais para se melhorar o desempenho com o aumento do número de transistores, quando se lida com atrasos e dissipação de calor nos circuitos. Para tal, ela une os modelos de execução vetorial e *multithread*, fornecendo uma ISA onde se pode utilizar o paralelismo e a localidade explicitamente, sem necessitar de uma área considerável no circuito para extrair o paralelismo dinâmicamente e realizar comunicação de longo prazo. Essa arquitetura visa, dessa forma, especialmente o domínio dos processadores embarcados.

O modelo de programação do VT utiliza de um processador de controle e um vetor de processadores virtuais (VPs). Os VPs possuem conjuntos de registradores próprios e executam grupos de instruções "RISC-like" empacotadas em um bloco atômico de instruções (AIBs). Não havendo um PC automático, o processador de controle utiliza de comandos *vector-fetch* para distribuir AIBS para todos os VPs, executando código em paralelo. Já para código de threads independentes, cada VP executa comandos *thread-fetch* e adquire suas próprias AIBs.

Esse mecanismo possibilita diversas formas de mapeamento de aplicações no VT, sendo a execução de *loops*, a principal. Nela cada iteração é executada por um VP e o código comum pode ser executado pelo processador de controle. Para permitir a execução de loops com depedência de dados, os VPs são conectados em uma topologia de anel unidirecional e comandos de recebimento e envio para os VP adjacentes, podem ser utilizados. Em contraste com as arquiteturas VLIW, só há a necessidade de um *vector-fetch* para a AIB da iteração, sendo os atrasos, pelas dependências entre os VPs, calculados dinâmicamente.

No caso de loops internos, os comandos de *thread-fetch* podem ser utilizados para controlar o fluxo interno em um VP. Essa habilidade de realizar *vector-fetches* e *thread-fetches* permite à arquitetura VT explorar o paralelismo e a localidade.

[KBG+04] apresenta também, um protótipo de processador chamado SCALE que implementa os conceitos da arquitetura VT, visando sistemas embarcados.

Para se obter menor área e consumo de energia, o SCALE utiliza baias de execução para os VPs com múltiplos *clusters*, onde cada um deles possue somente um subconjunto de funções e registradores. A execução atômica dos AIBs permite ao SCALE expor registradores internos dos *clusters*, utilizados pela ALU, ao programador. Há também os registradores privados que mantém seu valor entre AIBs e registradores compartilhados, utilizados para armazenar dados comuns aos VPs. O desacoplamento dos *clusters* e a comunicação entre *clusters* de baias adjacentes, permite a execução de diversos VPs simultaneamente em uma mesma baia.

O processador protótipo SCALE usado para testes possuía um processador escalar de controle MIPS, 4 baias de execução cada uma com 4 *clusters*, suportando até 128 processadores virtuais. A área estimada para sua implementação, incluindo memória cache, é de $10mm^2$. A freqüência de *clock* escolhida foi de 400mhz, considerando o consumo, complexidade de implementação e desempenho.

Os testes foram feitos usando o *benchmark* EEMBC, que abrange diversos domínios de software. O código para o SCALE foi produzido diretamente em *assembly* e foram utilizados códigos otimizados para outros processadores de sistemas embarcados usados na comparação. Analisou-se, nos testes, o mapeamento de paralelismo do SCALE, o aproveitamento da localidade e sua eficiência. Foi verificado que o SCALE obteve resultados competitivos, se comparado a outros processadores mais complexos, e que sua eficiência aumenta ao acrescentar novas baias de execução.

Por fim, a nova arquitetura VT se mostrou interessante, através da sua implementação satisfatória pelo SCALE, como uma opção de design menos complexa para processadores embarcados. O grande ganho neste aspecto está na economia de energia, sem perda de desempenho. A utilização da arquitetura VT para outros fins será vista em trabalhos futuros.

Referências

[KBG+04] R. Krashinsky, C. Batten, S. Gerding, M. Hampton, B. Pharris, J. Casper, and K. Asanović. The vector-thread architecture. In *31st International Symposium on Computer Architecture*, Munich, Germany, June 2004.

MO 401 – Trabalho 01
Aluno: Márcio Paixão Dantas (049174)
Artigo: [1] MISHRA, P., DUTT, N., *Modeling and validation of pipeline specifications*, ACM Transactions on Embedded Computing Systems (TECS),Vol. 3 , pp. 114-139.

O fluxo tradicional de projeto *hardware/software* para sistemas embutidos assume o uso de processadores já em uso no mercado e que possuem compilador e simulador (*software toolkit*) disponíveis. Se o processador com as características desejadas não existe ou não possui *software toolkit*, é preciso desenvolver um. Este é um processo que demanda bastante tempo. Um fator que contribui para esta demora é que os projetistas costumam fazer modificações na arquitetura programável para cumprir requisitos como: baixo consumo elétrico, melhor desempenho, menor área e maior densidade de código.

Implementar o *software toolkit* manualmente é impraticável devido à natureza competitiva do mercado moderno. Usar linguagens de descrição de arquitetura (ADL) resolve bem este problema, permitindo explorar os parâmetros do espaço de projeto e, ao mesmo tempo, gerar automaticamente compilador e simulador para a arquitetura especificada.

O trabalho [1] apresenta uma técnica de modelagem em grafos e validação de arquiteturas que captura estrutura e comportamento de unidades de processamento, interfaces de entrada/saída e memórias. O objetivo é verificar se o *pipeline* modelado está bem formado analisando aspectos estruturais da especificação.

À partir de uma especificação em ADL EXPESSION, são obtidos dois grafos, G_A e G_B. O primeiro modela a estrutura do pipeline e, o segundo, o seu comportamento. Além dos grafos, uma função de mapeamento entre G_A e G_B também é gerada (mapeamento estrutura-comportamento).

$G_A=(V_A,E_A)$, onde os vértices, em V_A, representam unidades de processamento, dispositivos de memória, portas e conexões. Já as arestas, em E_A, são subdivididas em duas categorias: arestas de *pipeline* (conectam unidades) e arestas de transferências de dados (todas as outras combinações de vértices).

$G_B=(V_B,E_B)$ é um grafo composto por diversos subgrafos disjuntos, no qual os vértices representam os campos de cada instrução e, as arestas (orientadas), ordenam os campos. Cada subgrafo conexo maximal representa uma instrução ou operação. E_B também pode ser subdividido em duas categorias: arestas de operação (ordem dos campos na instrução) e arestas de execução (ordem em que os campos são usados durante a execução).

A validação usa uma abordagem descendente, ao contrário da maioria das técnicas existentes. Por enquanto, só aspectos estruturais podem ser validados. O projetista especifica quais propriedades devem ser validadas para que a arquitetura esteja bem formada. Entre as propriedades já implementadas, estão: conectividade (todas as unidades do modelo devem pertencer a algum caminho, seja de transferências de dados ou de *pipeline*), caminhos falsos (todos os caminhos devem realmente poder ser usados por instruções), completude (todas as instruções devem ser executáveis), finitude (garante o término de uma operação no *pipeline*) e outras propriedades específicas de arquiteturas.

A técnica de validação em [1] foi capaz de identificar problemas em diversas arquiteturas diferentes (ARM, DLX, MIPS R10K, TI C6x e Power PC). Segundo os autores, novas propriedades podem facilmente ser adicionadas. Este seria um tipo mais simples de validação e seu uso seria complementar à abordagem tradicional.

Autor do Resumo:
Daniel Carlos Guimarães Pedronette
RA: 050269

Artigo:

Graph-based functional test program generation for pipelined processors

P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In Proceedings of Design Automation and Test in Europe (DATE) , pages 182–187, 2004.

A validação e verificação funcional de processadores é reconhecidamente uma tarefa complexa e custosa, representando um dos gargalos nas metodologias de design de processadores. Paralelamente, os requisitos de desempenho para sistemas embutidos têm se tornado cada vez mais exigentes, exigindo a implementação de técnicas de *pipeline*, o que por sua vez torna ainda mais difícil a tarefa de validação.

Sob esta perspectiva, é proposta uma metodologia gráfica de geração de testes funcionais para processadores com *pipeline*. A proposta pretende contribuir, gerando de forma automática um modelo gráfico do processador, gerando os testes funcionais sobre o comportamento do *pipeline* e ainda reduzindo o tempo necessário para geração dos testes.

No primeiro estágio de aplicação da metodologia, a arquitetura do processador é descrita utilizando-se ADL (*Architeture Description Language*). A representação contém informações sobre a estrutura, o comportamento do processador e o mapeamento entre estrutura e comportamento. Estruturalmente, são definidos quatro tipo de componentes: unidades funcionais (ALUs), unidades de armazenamento (registradores), portas e conexões, onde as conexões são definidas por *pipelines* ou barramentos de dados. Sobre o comportamento, as informações abrangem descrição das operações em: *opcode*, operandos e formato da instrução.

Ainda visando a descrição da arquitetura, a abordagem proposta faz uso do conceito de abstração funcional, que consiste no fato de que diferentes arquiteturas fazem uso das mesmas unidades funcionais (uma unidade de *fecth*, por exemplo), mas com parâmetros diferentes. Ou seja, a estrutura de cada unidade funcional é capturada utilizando-se funções genéricas, mas parametrizadas adequadamente para a arquitetura em questão. Uma unidade de *fecth*, por exemplo, possui vários parâmetros, como operações lidas por ciclo, esquema de predição de *branches*, etc.

Baseada na descrição da arquitetura, a estrutura do processador é modelada num grafo G = (V,E). Os vértices representam componentes do processador: unidades funcionais ou de armazenamento e as arestas representam ligações para transferência de dados: via *pipeline* ou barramentos. Cada vértice do grafo contém informações a respeito de suas entradas, saídas, operações suportadas e respectiva duração. Ainda para cada vértice (unidades estruturais) é gerada uma descrição SMV do seu comportamento através de funções genéricas, de acordo com a abordagem de abstração funcional.

A partir da definição do grafo que representa a arquitetura, torna-se necessário definir finalmente uma métrica de cobertura dos testes. A metodologia abordada propõe uma métrica baseada na cobertura funcional do *pipeline*, definindo todas as possibilidades de interação entre as instruções e os estágios do *pipeline*. Um grafo é considerado testado quando todos os seus vértices e arestas estiverem passado pelos quatro estados definidos: *active* (executando ou transferindo dados), *stalled* (devido a *hazards*), *exception* ou *flushed* (devido a *exception* de um nó pai). Assim, o algoritmo de geração de testes percorre o grafo, gerando propriedades de acordo com os estados acima citados, que por sua vez darão origem aos programas de teste.

Foi realizado um estudo de caso, aplicando a metodologia proposta ao processador DLX, num *pipeline* de 5 estágios. Considerando todos os cenários, foram gerados 223 programas de teste em 91 segundos numa máquina Sun UltraSPARC-II trabalhando a 333MHz, com 128M RAM.

A metodologia proposta reduz o tempo necessário para geração de testes e representa uma técnica bastante promissora, no que diz respeito a validação dirigida pela especificação. Atualmente, os testes gerados são aplicados a um simulador estrutural da arquitetura. Os trabalhos futuros incluem a aplicação destes testes na descrição RTL para a validação funcional de processadores com *pipeline*.

Título: Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance

Referência: Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance; pp. 64-75; ISCA '04

Autor do resumo: Daniel Henricus de Knegt Dutra Nicácio **RA:** 057612

Os processadores atuais precisam atender a dois requisitos: bom desempenho com um alto fluxo de *threads* e boa performance com uma única *thread*. A abordagem mais promissora é a construção de chips com multi-processadores. Sendo assim, o artigo tem como objetivo mostrar que o uso de processadores heterogêneos em um mesmo chip é mais eficiente do que o uso de processadores iguais.

A segunda parte do artigo apresenta duas grandes vantagens do uso de arquiteturas heterogêneas. A primeira delas é a eficiente adaptação para a diversidade de aplicações, ou seja, diferentes programas ou diferentes fases de um mesmo programa exigem diferentes tipos de processadores. A segunda vantagem é o uso eficiente do *die* para um dado paralelismo de *threads*, ou seja, para um certo número de *threads*, a área do *die* pode ser melhor aproveitada utilizando processadores de tamanhos e eficiências diferentes. No final desta parte, é apresentado um gráfico comparativo contendo diferentes combinações de processadores, mostrando a área utilizada por cada uma dessas combinações e seus *throughputs*.

A terceira seção do artigo descreve a metodologia utilizada para realizar as comparações entre as diferentes arquiteturas heterogêneas e homogêneas. Os hardwares utilizados foram os processadores da família *Alpha*: EV5(21164), EV6(21264) e um hipotético EV6+ com suporte *multi-thread*. A avaliação foi feita com o número de *threads* variando de um ao máximo número de contextos dos processadores. A métrica de comparação foi o tempo médio de resposta das aplicações, assim como o tamanho das listas de espera do sistema.

A quarta seção apresenta as vantagens de performance das arquiteturas heterogêneas e mecanismos de designação de tarefas para seus processadores, os quais contribuem para o aumento da eficiência das arquiteturas heterogêneas. O primeiro tópico desta parte é o escalonamento estático para diversidade entre *threads*. Utilizando esta técnica padrão, a arquitetura composta por 3 EV6 e 5 EV5 teve uma melhora média de 26% em relação à arquitetura com 4 EV6 e 23% em relação à arquitetura com 20 EV5. Esta comparação utilizando o alcance de 20 *threads* mostra um significativo ganho com o uso de processadores heterogêneos.

A segunda análise explora a eficiência de arquiteturas heterogêneas em sistemas abertos. Para tal, foi utilizado um *framework* de simulação que modela chegadas randômicas de tarefas com tamanhos também randômicos. Dessa forma, foi medido o tempo médio de resposta, levando em consideração o número de tarefas designadas em um intervalo de tempo. O resultado obtido entre as arquiteturas 4EV6 e a 3EV6 & 5EV5 mostra que a segunda se satura com um *throughput* muito maior do que a primeira. Além disso, o tempo de resposta da arquitetura heterogênea se degrada de forma muito mais suave quando submetida a uma alta carga de tarefas.

O terceiro tópico enfatiza o escalonamento dinâmico para *threads* diversas, detalhando políticas de designação de tarefas. As estratégias, chamadas de '*core sampling strategies'*, atribuem tarefas aos processadores de forma que o processador mais robusto nunca fique a toa e o sistema como um todo tenha o melhor desempenho possível. A primeira, *sample-one*, testa cada *thread* em cada diferente processador; a segunda, *sample-avg*, testa cada *thread* várias vezes em cada tipo de processador; e a terceira, *sample-sch*, testa cada forma de escalonamento das tarefas. O resultado da comparação mostra que a terceira técnica é a mais eficiente, aumentando a eficiência em relação à atribuição randômica em até 22%.

Estas estratégias requerem a minimização da sobrecarga das amostragens e uma reação rápida a mudanças do sistema. Com este fim, são sugeridas duas técnicas: variação do intervalo de tempo entre as amostragens e o monitoramento do IPC, este pode ser realizado em uma única *thread*, em todas elas, ou especificando um intervalo geral de IPC. Destas técnicas, a última apresentou o melhor resultado, com melhora de 20% em relação à técnica estática.

Para finalizar esta quarta seção, o artigo relata o uso de processadores com suporte *multi-thread* em arquiteturas heterogêneas. O resultado obtido foi um desempenho ainda maior para situações com grande quantidade de tarefas.

Por fim, o artigo conclui que arquiteturas multi-processadores heterogêneas provêm uma significante vantagem em relação às arquiteturas homogêneas de mesma área. Além disso, o artigo também conclui que o uso de estratégias de escalonamento em arquiteturas heterogêneas aumentam ainda mais o potencial destas arquiteturas.

Increased Scalability and Power Efficiency by Using Multiple Speed Pipelines

Refêrencia: Emil Talpes, Diana Marculescu. *Increased Scalability and Power Efficiency by Using Multiple Speed Pipelines;* ISCA 2005
Autor do resumo: Douglas Gielo Quinellato (RA: 057615)

Resumo

Um dois grandes problemas enfrentados pelos projetistas de arquiteturas é a baixa escalabilidade de alguns dos elementos do pipeline quando se aumenta a freqüência do clock e o tamanho dos pipelines. Vários estudos mostram que é grande a diferença entre a escalabilidade do front-end e o back-end do pipeline, sendo o primeiro (Issue Window) mais devagar e menos escalável. Em face disso, este artigo propõe dois mecanismos que permite que estas duas partes sejam separadas, funcionando em sua velocidade ótima.

O primeiro desses mecanismos é o **Dual clock issue window**, que permite que o front-end trabalhe em uma freqüência diferente do back-end; o segundo e o **Pré-Scheduled execution**, que tenta manter a execução das instruções o máximo de tempo na parte mais rápida do pipeline. Para isso é proposto o uso de uma **execution cache**(EC) logo apos a *issue window*. Este cache armazena as instruções já decodificadas e com registradores renomeados.

Estes mecanismos funcionam utilizando-se de dois modos de operação: **trace creation mode** e **trace execution mode**. No primeiro, as instruções são lidas do I-Cache, e entram no pipeline pelo *front-end*. As instruções decodificadas são executadas e armazenadas no execution cache. Quando ocorre um *branch misprediction a* próxima instrução é procurada no EC; se for achado, o front-end do pipeline é desligado e as instruções são executadas diretamente do back-end, sendo buscadas no EC. Instruções no EC são armazenadas em *issue order*. Para o primeiro caso, o pipeline trabalha numa freqüência menor (*baseline)*, ditada pelo Issue Window; no segundo caso, trabalha-se numa freqüência maior, devido ao desligamento do front-end.

Para a verificação do desempenho, foi utilizado um processador de 9 estágios, superescalar (4-way), com I*ssue Window* de 128 entradas, suportando *issue* de 6 instruções por ciclo. Caches L1 de 64K com tempo de acesso de dois ciclos foi considerado um design bem balanceado para o *baseline*. Foram utilizados os SPEC95 e SPEC 2000 para a medição do desempenho.

O uso de Dual clock issue window, junto com a limitação de renomeamento de registradores levou a necessidade de adicionar 3 estágios no pipeline, causando uma queda de desempenho de 10% em alguns *brenchmarks*. Porém, a arquitetura apresentada sobrepõe essa penalidade, conseguindo uma melhoria de 5%, operando no modo *baseline.* Esta arquitetura pode utilizar o EC em 88% do tempo, o que dá uma boa oportunidade de melhoria de velocidade utilizando-se um clock mais rápido nessa situação.

Utilizando clocks diferentes observamos um aumento no desempenho maior, variante de acordo com o aumento das freqüências do front e do back end. Com um aumento de 50% na freqüência, temos um aumento de 56% do desempenho, por exemplo.

O consumo de energia diminuiu também com esta arquitetura Com uma implementação em 130nm, temos uma diminuição de 30%, ficando em 20% a redução em 65nm.

Aluno: Fernando André Kronbauer RA: 057619

Artigo: Guilherme Ottoni, Ram Rangan, Adam Stoler, David I. August. *Automatic Thread Extraction with Decoupled Software Pipelining*. pp. 105 - 118; MICRO 2005

O artigo apresenta a técnica *Decoupled Software Pipelining* – DSWP. Trata-se de uma técnica de compilação que extrai paralelismo não especulativo dos laços de um programa, de forma que diferentes partes do laço possam ser executados paralelamente em diferentes núcleos de processamento, acelerando desta forma a execução do programa como um todo. É proposto no artigo um método completamente automático de extração de paralelismo em nível de linha de execução, no qual o paralelismo altamente granular inerente à maioria dos programas é extraído e transformado em linhas de execução paralelas de longa duração.

O artigo lembra outra técnica de extração de paralelismo, chamada DOACROSS, e a compara a DSWP. O paralelismo em DOACROSS é obtido pela execução concorrente de partes de cada iteração do laço, distribuídos em vários núcleos de processamento. As dependências são respeitadas pelo repasse de valores de um núcleo para outro, geralmente através de memória com sincronização. É apresentado o exemplo no qual uma lista ligada é percorrida, realizando-se algum processamento em cada nó. O carregamento do ponteiro para o próximo nó da lista precisa ser repassado de um núcleo de processamento para outro a cada iteração do laço. Enquanto o método DOACROSS sobrepõe a execução do corpo da iteração corrente com o carregamento do ponteiro do nó da próxima iteração, os custos de comunicação neste caso podem simplesmente anular quaisquer ganhos de paralelismo.

Em DSWP, em vez de colocar cada iteração do laço em um núcleo, quebra-se o corpo do laço em um número de partes igual ao número de núcleos de processamento disponíveis, de forma que os núcleos formem um *pipeline*, ou seja, valores produzidos por um núcleo de processamento servem como entrada para o núcleo seguinte no *pipeline*. Tomando novamente o exemplo da lista ligada, a parte de carregamento do ponteiro do próximo nó é realizada em um núcleo enquanto que o processamento do nó é realizado em outro. Desta forma a dependência crítica do laço não precisa mais ser roteada de um núcleo para outro. A transmissão dos valores de um núcleo para outro é feita em uma fila de mensagens implementada em *hardware*, com baixo custo de sincronização.

Ao contrário das técnicas DOACROS e outras técnicas de extração de paralelismo desenvolvidas anteriormente, DSWP requer que o fluxo de dados entre um núcleo e outro seja acíclico. Isto cria a oportunidade para que o processamento de cada núcleo seja o mais independente possível dos demais, aumentando a tolerância à latência de comunicação. Técnicas DOACROS acabam sendo mais restritivas do que DSWP, pois geralmente requerem que as iterações do laço sejam contadas, que operem somente em vetores, que tenham padrões simples de acesso à memória ou que tenham fluxo de controle simples ou inexistente.

O artigo segue apresentando o algoritmo do DSWP, aplicado no código assembler gerado pelo compilador, mas antes do processo de alocação de registradores. Este algoritmo constrói o grafo de dependência das instruções que compõe o laço, e identifica os seus componentes fortemente conectados. Estes componentes são utilizados pelo algoritmo para construir as partições que serão designadas a cada núcleo, de forma a garantir que o fluxo de dados seja acíclico entre as partições, e tentando balancear as cargas de cada núcleo através de uma heurística (o problema de balanceamento de cargas dos núcleos é NP-completo). Duas novas instruções são acrescentadas ao ISA: produce e consume. produce recebe como parâmetro o número da fila que receberá a mensagem, mais a mensagem em si, do tamanho de uma palavra do computador. consume por sua vez recebe como parâmetro o número da fila de onde retirará a mensagem ali depositada por uma instrução produce. Após a partição do corpo do laço, o compilador insere instruções produce e consume nas posições adequadas de forma a garantir que as dependências de dados, de controle e de sincronização de memória sejam satisfeitas.

Cada partição do laço que irá executar em paralelo é transformada pelo compilador em uma função auxiliar. O compilador então injeta código no início programa para que uma linha de execução auxiliar seja instanciada e execute estas funções no momento apropriado. Esta linha de execução executa um laço principal, onde fica aguardando em uma fila mestre de mensagens o valor do ponteiro da função que irá executar em seguida. A linha de execução principal, ao chegar ao laço otimizado por DSWP, executa uma operação produce passando como parâmetro a fila mestre e o endereço da função auxiliar a ser executada pela linha de execução auxiliar.

Os resultados apresentados pelo artigo foram produzidos através da adaptação do compilador IMPACT e simulação do código em um simulador validado de um processador Itanium 2. Devido a natureza altamente detalhada do simulador, os autores do artigo não conseguiram simular o código completo dos programas gerados. Em vez disso, as simulações detalhadas foram restritas aos laços otimizados pelo DSWP. Os resultados experimentais mostraram que DSWP pode ser aplicado a quase todos os laços de um programa. Utilizando um simulador de dois núcleos de processamento baseado no modelo validado de um núcleo do Itanium 2, e utilizando um compilador de otimização de código de alta qualidade, DSWP alcançou *speedup* médio de aproximadamente 19,4% em loops importantes dos *benchmarks*, traduzido em *speedup* médio de 9,2% sobre *benchmarks* completos. Os autores concluem o artigo reconhecendo áreas nas quais o método pode ser melhorado, como análise de memória mais adequada, otimizações adicionais para quebrar ciclos de dependência, heurísticas de particionamento mais elaboradas, e novas técnicas de otimização para reduzir o número de fluxos entre núcleos.

Título: Exploiting Vector Parallelism in Software Pipelined Loops
Autores: Samuel Larsen, Rodric Rabbah e Saman Amarasinghe
MIT Computer Science and Artificial Intelligence Laboratory
Congresso: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, páginas 119-129, 2005.

Resumo por Renato Silva das Neves – RA 057639

Os processadores atuais têm adicionado ao seu ISA instruções que operam sobre vetores, principalmente no caso de instruções multimídia. Uma maneira de otimizar o uso dos processadores, aumentando sua performance, é justamente explorar as características de paralelismo vetorial entre *loops* no código com *pipeline* de *software* (*software pipelining*). Essa é a proposta do artigo analisado, que utiliza a vetorização seletiva (*selective vetorization*) para dividir operações entre recursos escalares e vetoriais do processador, de uma maneira que maximize a performance quando um *loop* está em *pipeline* de *software*. Assim, em *loops* com um grande número de operações vetoriais, estratégias convencionais fazem com que todas operações de dados paralelas sejam vetoriais, deixando recursos escalares ociosos. Movendo algumas operações para as unidades escalares pode se fazer com que o *scheduling* seja mais efetivo.

O algoritmo utilizado para a vetorização seletiva foi baseado na heurística de particionamento em dois *clusters* de Kernighan e Lin's, dividindo instruções entre partições escalares ou vetoriais. O algoritmo é iterativo e funciona da seguinte forma: inicialmente todas operações são colocadas na partição escalar e cada iteração reposiciona cada operação que pode ser vetorial exatamente uma vez. Com cada movimento, o algoritmo calcula o custo do resultado da reconfiguração, registrando o custo mínimo encontrado. Uma vez que cada operação foi reparticionada, a configuração com o menor custo é usada como o ponto de partida da próxima iteração. O processo termina quando uma iteração falha ao tentar melhorar sua configuração inicial. No final, operações que ficaram na partição vetorial são passíveis de vetorização. O cálculo do custo da configuração é a principal parte do algoritmo e é definida como o peso do recurso mais usado, ou seja, o número de ciclos de máquina que o recurso mais usado está reservado. O cálculo desse custo é feito a partir do método *bin-packing*, onde cada operação adicionada em um *bin* (recurso visível ao compilador) aumenta o peso desse *bin*. Uma otimização efetuada na proposta do artigo foi que quando dois *schedulings* alternativos não aumentam o peso do recurso mais usado, a opção escolhida é a que minimiza a soma dos quadrados dos pesos dos *bins*. Isso gera um balanço de operações entre os *bins*. No pior caso, o algoritmo de vetorização seletiva resulta em uma complexidade de $O(n^3)$.

O algoritmo acima foi implementado no *beckend Trimaran*, uma infra-estrutura de compilação e simulação para arquiteturas VLIW. Também foi usado o *frontend* SUIF. Para todos os *benchmarks* utilizados (nove SPEC FP) foram aplicados um conjunto de otimizações padrões antes da vetorização seletiva, que só foi aplicada em *loops do*. A comparação de *speedup* foi realizada com um método de vetorização tradicional e outro de vetorização completa (*full*). A vetorização seletiva alcançou o máximo de 1.38x de *speedup*, com uma média de 1.11x, mostrando melhor performance em todos os casos, exceto em um, em que foi criado mais *schedules* compactos para *loops* críticos, aumentando o número de estágios do *pipeline* de *software*, levando a prólogos e epílogos mais longos. Resultados mostraram que para cada *benchmark* há um significante número de *loops* no qual a vetorização seletiva fornece vantagens. A comunicação de operandos entre recursos escalares e vetoriais, a partir de instruções *load* e *store*, é levada em conta nos cálculos do custo descritos acima e resultados também foram mostrados comprovando que considerando tal comunicação há uma melhora no *speedup*. Por último, as operações vetoriais de memória podem ser consideras alinhadas ou não alinhadas. Simulações mostraram que o *overhead* de alinhamento pode ser eliminado quando as operações são conhecidas em tempo de compilação como sendo alinhadas, pois assim não são consideradas durante a análise de custo do algoritmo de vetorização seletiva.

Arquitetura de Computadores - MO401

Aluno	Thiago Senador de Siqueira
RA	057642
Título do Artigo	Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling
Referência Bibliográfica	KUMAR, R. ZYUBAN, V. TULLSEN, D. M. **Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling**. In: *Proceedings of the 32nd International Symposium om Computer Architecture* (ISCA'05). 2005.
Número de páginas	12

As arquiteturas de processadores de alto desempenho têm se direcionado ao desenvolvimento de múltiplos núcleos de processamento (*cores*) em um único chip. Estas arquiteturas são conhecidas como "Arquiteturas *Multi-Core*". Tal tecnologia possibilita um maior poder de processamento, maior *throughput* e maior escalabilidade que as estruturas monolíticas. Entretanto, apesar do rápido desenvolvimento das arquiteturas *Multi-Core*, não há ainda na literatura um conhecimento profundo sobre questões de interconexão de núcleos e interatividade com outras estruturas (*cache*, IO, etc.) num mesmo chip. A busca por uma solução ideal tem de lidar com vários aspectos como performance, consumo de energia, área utilizada, taxa de transferência dentre outros. A questão da interconexão de núcleos é um elemento crítico de uma arquitetura *Multi-Core*.

O artigo em questão apresenta três mecanismos de interconexão de núcleos e interação com outras estruturas: 1) *Shared Bus Fabric* (SBF), que é um link de alta velocidade utilizado na comunicação entre núcleos, *caches*, IO e memória. 2) link ponto-a-ponto (P2P), utilizados em sistemas que possuem múltiplos SBF. 3) *Crossbar*, mecanismo com alta taxa de transferência utilizado em sistemas onde múltiplos núcleos compartilham uma mesma *cache* L2.

No artigo, são avaliados vários tipos de *overhead* associados às arquiteturas *Multi-Core*, como latência, área ocupada e consumo de energia. A área ocupada por um barramento é determinada pelo número de linhas (fios) vezes o comprimento destas linhas. Quanto maior a área ocupada, maior o *overhead* de área. O *overhead* de energia é gerado pela quantidade de linhas, repetidores e *switchs*. Quanto maior o número destes componentes, maior o overhead de energia. Já a latência de um sinal conduzido através de uma interconexão é determinada basicamente pelas latências das linhas, tempo de espera em filas de acesso a barramentos e tempo de arbitração (definição de prioridades).

Vários testes foram realizados com os diversos mecanismos de interconexão de núcleos. Com relação ao SBF, os testes de área revelaram que a área utilizada por tal mecanismo leva em consideração a fiação do circuito (quantidade de linhas) e a lógica relacionada à interconexão. O *overhead* causado pela área pode ser significante, por exemplo, em um *die* de $400mm^2$, o *overhead* de área para se interconectar 16 *cores* é 13%. Para 8 e 4 *cores* o *overhead* é de 8.7% e 7.2% respectivamente. Considerando que um *core* possui $10mm^2$, a área ocupada por um BSF é suficiente para se colocar de 3 a 5 *cores* extras ou de 4 a 6 MB de *cache* extra. Logo, concluiu-se que o overhead de área cresce rapidamente com a adição de novos *cores*. Os testes de consumo de energia de mecanismos BSF foram feitos através da soma da energia dissipada pelas linhas e pela lógica de interconexão. O *overhead* de energia de interconexão para um processador de 16 *cores* é maior que a energia consumida de 2 *cores*. O consumo de energia cresce superlinearmente com o número de unidades conectadas. Já os testes relacionados à performance de mecanismos BSF foram realizados em duas etapas: a primeira desconsiderava o *overhead* de interconexão. Nesta fase, através de um programa de uma única *thread*, foi possível constatar que a performance decresce com o aumento do número de *cores*, visto que cada *core* possui uma *cache* cada vez menor. Na segunda etapa, onde o *overhead* de interconexão é considerado, a performance diminui ainda mais rapidamente. Os valores para o *overhead* de performance para 4, 8 e 16 *cores*, considerando o *overhead* de interconexão, foram 10%, 13% e 26% respectivamente. Um outro resultado relacionado à performance considera a taxa de transferência entre *cores*. Se a taxa de transferência é reduzida em pequenos fatores, a degradação da performance pode ser recuperada com a utilização de *caches* maiores.

Os teste realizados com o mecanismo *Crossbar* avaliaram os *overheads* de sistemas onde os vários *cores* compartilham uma mesma *cache* L2. *Cache* compartilhada permite que o tamanho da mesma seja particionado dinamicamente melhorando a taxa global de *hits*. Nos testes de área, constatou-se que a complexa fiação do circuito resulta em *overhead* de área, assim como o compartilhamento de *cache*. Por exemplo, em um *die* de $400m^2$, o *overhead* de área para uma latência aceitável é 11,4%, 22,8% e 46.8% para barramentos com 2, 4 e 8 vias para a cache, respectivamente. Já o teste de energia de um *Crossbar* mostrou que o *overhead* é significativo: pode ser maior que o consumo de 3 *cores*. Com relação à performance, o compartilhamento de *cache*, em geral, mesmo desconsiderando os *overheads* de área de interconexões, não melhoram ou melhoram muito pouco a performance de um sistema *multi-core*. Logo, concluiu-se que o uso de *cache* compartilhada torna-se significativamente menos desejável em tais sistemas.

Os mecanismos de interconexão SBF maximizam o tamanho de um chip, permitindo assim a conexão de tantas unidades quanto necessárias em uma linha que percorre todo o chip. Entretanto, por causa das latências de longos SBF algumas alternativas devem ser avaliadas. Os autores propõem uma nova estratégia para interconexão através de mecanismos SBF e P2P. Considerando um chip com 8 *cores*, dividindo-se um único SBF ao meio e conectando estes dois SBFs menores através de um link P2P consegue-se uma performance 17% maior que a de um único SBF. Com esta estratégia, acessos locais a cache se beneficiam com a diminuição das distâncias, entretanto, os acessos remotos podem sofrer atrasos, visto que estes acessos percorrem a mesma distância e utilizam filas e mecanismos de arbitração adicionais entre interconexões.

Aluno	Edmar Welington Oliveira	**RA**	065819
Título do Artigo	The Impact of Performance Asymmetry in Emerging Multicore Architectures		
Referência Bibliográfica	BALAKRISHNAN, S.; RAJWAR, R.; UPTON, M.; LAI, K.; **The Impact of Performance Asymmetry in Emerging Multicore Architectures**. In: *Proceedings of the 32nd International Symposium om Computer Architecture* (ISCA'05). 2005. p. 506-517		

Processadores *multicore* são caracterizados por apresentar vários *cores* (núcleos de processamento) em um único *chip*. Tal tecnologia se apresenta como alternativa aos atuais processadores, dotados de um único *core* no *chip*. O desempenho assimétrico (*performance asymmetry*) em arquiteturas *multicore* é caracterizado quando núcleos individuais possuem diferentes desempenhos. Processadores *multicore* provêm um aumento na capacidade computacional sem que seja necessário um aumento de complexidade destes em nível de micro-arquitetura. Como resultado, possuem melhor desempenho por área e *watt* que os processadores de núcleo único (*single-core*).

O artigo apresenta um estudo detalhado sobre o comportamento de certas aplicações em arquiteturas *multicore* assimétricas. Inicialmente, são discutidos os efeitos da tecnologia *multicore* no desempenho dos processadores. Em seguida, ressalta-se a importância do estudo, destacando-o como o primeiro no que tange a avaliação do impacto do desempenho assimétrico das arquiteturas *multicore* no comportamento de aplicações *multithreaded.* Basicamente, o estudo procura analisar se a assimetria de desempenho em sistemas *multicore* tem um impacto negativo nas características das aplicações de *software*. Além disso, procura-se verificar se é possível prever o desempenho e a escalabilidade destas em configurações assimétricas.

Para tornar possível a análise dos resultados, é estabelecido um padrão básico de desempenho. Para tal, é observado o comportamento das aplicações em sistemas de desempenho simétrico. Para a realização dos testes, a freqüência individual dos núcleos que compõem o multiprocessador é variada. Segundo os autores, essa é uma técnica efetiva para se obter um desempenho assimétrico em sistemas de hardware. As aplicações avaliadas compreendem sistemas cliente/servidor (SPECjbb2000 e SPECjAppServer2002), servidores de banco de dados (TCP-H), WebServers (Zeus e Apache), aplicações científicas (SPEC OMP), aplicações multimídia (H.264) e uma ferramenta de desenvolvimento (PMAKE); todas configuradas segundo as regras providas pelas próprias organizações desenvolvedoras. Para formalização e apresentação dos resultados, os autores adotaram uma representação para as configurações assimétricas e simétricas. O rótulo de configuração *nf-ms/scale* significa *n cores* de alta velocidade e *m cores* de baixa; estes últimos rodando em uma escala de velocidade equivalente a *1/scale* a dos núcleos rápidos. As configurações simétricas adotadas para o estudo são 4f-0s, 0f-4s/4 e 0f-4s/8. Já as assimétricas compreendem 3f-1s/4, 3f-1s/8, 2f-2s/4, 2f-2s/8, 1f-3s/4 e 1f-3s/8.

O artigo apresenta a metodologia experimental adotada nos testes. Em seguida estes são realizados e os resultados obtidos analisados. Para o SPECjbb2000, o número de operações realizadas por segundo foi adotado como principal métrica de desempenho. Para configurações simétricas, o desempenho da aplicação se mostra estável e escalável, mas para as configurações assimétricas, observa-se certa instabilidade. A aplicação SPECjAppServer2002 se adapta dinamicamente às variações de desempenho. Para tal, faz uso de técnicas como balanceamento de carga. Esta característica provê estabilidade e escalabilidade, além de prevenir uma sobrecarga do sistema. Para as configurações simétricas, o TCP-H apresenta certa estabilidade e escalabilidade. Entretanto, para as configurações assimétricas, o mesmo não ocorre. Segundo os autores, a própria aplicação contribui para a instabilidade observada. Isto sugere a necessidade de se expor o desempenho assimétrico da arquitetura para a aplicação. O estudo revela que o desempenho do servidor Apache, atuando em configurações assimétricas e sobre uma carga leve, é significantemente instável e não escalável. De forma diferente, ao ser submetido a uma carga pesada, o mesmo se apresenta estável. Já para o servidor Zeus, independentemente da carga de trabalho inferida, observa-se grande instabilidade. Entretanto, este apresenta uma vazão 2.5 vezes maior que o servidor Apache. Para este teste, o artigo conclui que a instabilidade varia conforme a assimetria. A aplicação SPEC OMP, para as configurações simétricas, apresenta-se estável e escalável. Entretanto, para as configurações assimétricas, a aplicação apresentou um desempenho melhor, indicando, claramente, que a assimetria pode ser uma característica efetiva no desempenho. Para todas as configurações, observam-se a estabilidade e a escalabilidade das aplicações H.264 e PMAKE.

Ao final, o artigo apresenta uma tabela que resume, qualitativamente, os resultados obtidos, além de uma figura onde se observam, quantitativamente, a previsibilidade de desempenho e escalabilidade das aplicações avaliadas. Em seguida, com base nos resultados, são discutidas as questões chaves que deram motivação para a realização do trabalho. O artigo conclui ressaltando que o desempenho assimétrico das arquiteturas *multicore* pode ter um impacto negativo nas aplicações, dificultando a previsão do desempenho e escalabilidade destas. Isto ocorre porque os desenvolvedores assumem que todos os núcleos possuem igual desempenho; portanto, não há a preocupação sobre como tais núcleos se comunicam de forma a prover uma efetiva distribuição de tarefas e processamento. Mas os autores também destacam que alguns graus de assimetria de desempenho podem ser benéficos, fato que varia conforme a aplicação e a métrica de avaliação.

Título do Artigo: Compiler Optimizations for Transaction Processing Workloads on Itanium® Linux Systems

Citação Bibliográfica do Artigo: Gerolf Hoflehner, Knud Kirkegaard, Rod Skinner,Daniel Lavery, Yong-fong Lee, Wei Li. Compiler Optimizations for Transaction Processing Workloads on Itanium® Linux Systems; pp. 294 - 303; 37th Annual International Symposium on Microarchitecture, 2004.

Aluno: Mário Luiz Rodrigues Oliveira

RA: 066254

O artigo analisado apresenta e discute algumas otimizações que contribuem para melhorar o desempenho do código gerado pelo compilador C/C++ da *Intel* para o processador *Itanium 2*. Especificamente, o artigo apresenta otimizações para aproveitar a pilha de registradores do processador *Itanium* e propõe alterações no modelo de preempção do Linux.

O uso, em conjunto, de todas as otimizações apresentadas permitiram uma melhoria de mais de 40 % em aplicações envolvendo processamento de transações *on-line* (OLTP),utilizando-se como *workloads* a base de dados do *Oracle* e um sistema composto por 4 processadores *Itanium 2* executando o sistema operacional Linux.

O desempenho das aplicações OLTP no processador *Itanium 2* é prejudicado por *misses* na cache de dados, na cache de instruções e na ITLB e pelo tráfego de dados entre a memória e o *register stack engine (RSE)*. Assim, o artigo apresenta otimizações que tentam melhorar tais aspectos.

As otimizações descritas são:

- Redução do Tráfego RSE:o processador *Itanium* possui 128 registradores e desses 96 são usados para a chamada e retorno de procedimentos. A arquitetura do processador *Itanium* permite encolher a pilha de registadores antes de uma chamada de procedimento e restaurar o seu tamanho original mais tarde. Essa característica pode reduzir o número total de registadores consumidos pelo procedimento chamador e pelo procedimento chamado. Assim pode-se diminuir o tráfego de RSE.

- Escalonamento de código e especulação de controle: para tentar reduzir *misses* na cache de instruções pode-se usar as técnicas de especulação de controle (mover as instruções de *load* para antes das instruções de *branches*) e escalonamento de código (reordenar instruções para tentar minimizar os efeitos de *misses* na cache).

- Prefetching de instrução: também para tentar minimizar *misses* na cache de instruções pode-se buscar antecipadamente instruções que são alvo de uma instrução de desvio.

- Otimizações da disposição das funções: nessa categoria pode-se incluir 3 otimizações, a saber: funções *inline*, agrupamento de funções e separação de funções. Funções *inline* removem o *overhead*de chamada de funções, agrupamento de funções permitem reduzir *misses* na cache de instruções e separação de funções diminuem os *misses* em ITLB.

- Otimizações da disposição dos dados: duas otimizações são aplicadas pelo compilador nesse item. Primeiro, constantes e *strings* devem ser mantidas

numa área de somente leitura ao invés de ficar na seção de dados. E segundo, o compilador implementa heurísticas que tentam ordenar os dados locais na pilha baseado na freqüência e no tamanho.

- Otimizações Setjmp()/longjmp(): essas otimizações visam reduzir o *overhead* na chamada das funções setjmp()/longjmp().
- Modelo de preempção do Linux: essa otimização tem por objetivo diminuir o *overhead* em tempo de exeução causado pelo modelo genérico de preempção do Linux que requer do compilador a geração de código realocável.

Aluna: Carla Geovana do Nascimento Macário **RA:**895063

RENO: A Rename-Based Instruction Optimizer In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pags. 98 – 109. IEEE Computer Society Washington, DC, USA.2005

RENO (REnaming Optimzer) é um mecanismo de renomeamento de registros do MIPS-R10000 modificado para implementar dinamicamente algumas técnicas de otimização estática existentes, promovendo a eliminação de instruções e levando à diminuição na latência do grafo de fluxo de dados e ao aumento do *bandwidth* do core de execução. Com a implementação dinâmica, em contraposição à estática, tem-se maior espaço físico para renomeamento de registradores, inexistência de limitações presentes nos compiladores, otimização baseada em informação especulativa ou dinamicamente disponível sobre dependência de memória e executada apenas nos caminhos dinâmicos. Em caso de mis-especulação, ou de informação de memória errada, as instruções executadas são desfeitas, bem como as otimizaçoes executadas pelo RENO.

RENO unifica um novo mecanismo ($RENO_{CF}$) com outros previamente propostos ($RENO_{ME}$, $RENO_{CSE}$ e $RENO_{RA}$), promovendo uma sinergia entre eles. $RENO_{ME}$ elimina *moves* e é a mais simples das otimizações disponíveis. Requer, além de uma infra-estrutura de compartilhamento de registradores, um circuito para identificar *moves*. No renomeamento, o registrador de entrada de uma instrução *move* passa a apontar para o registrador correspondente à sua saída, eliminando a instrução, que deixa de ser executada. O $RENO_{CSE}$ faz a eliminação de subexpressões comuns, mantendo uma tabela com os valores disponíveis em cada registrador físico e com informaçoes do fluxo de dados da instrução que levou a este valor. No renomeamento é feita uma busca por tuplas com o mesmo código de operação que a instrução corrente. A existência desta tupla indica redundância e a instrução deve ser eliminada, passando sua saída a apontar para a saída do registrador correspondente à tupla identificada. O $RENO_{RA}$ implementa dinamicamente a alocação de registros, tendo como principal objetivo eliminar *loads* via integração de registradores (com o compartilhamento de registradores físicos). É usado para implementar especulação de *bypassing* de memória para pares de *load-store,* transformando cadeias produtor-store-load-consumidor em produtor-consumidor. O $RENO_{CF}$ implementa dinamicamente a técnica de *folding* de constantes em adições do tipo *register-immediate*, escolhidas por serem mais comuns nos programas (cálculo de endereços, etc.) e por gerarem operações menos complexas após o *folding* (com 2 operandos). Usa uma extensão do formato da tabela de mapeamento, armazenando, além das informações convencionais, um deslocamento que contém o valor resultante das adições. No mapeamento um acumulador armazena a(s) soma(s) do valor do registrador com o deslocamento, disponibilizando o resultado assim ele que for requisitado por alguma instrução subseqüente. Desta forma, e de maneira semelhante às técnicas anteriores, instruções intermediárias são eliminadas.

A implementação do RENO é simples, usando sempre o mesmo princípio: manipulação de tabelas de mapeamento e infra-estrutura de compartilhamento de registros físicos para promover o colapso (ou eliminação) de instruções dinâmicas (diminuindo a latência do fluxo de dados). A instrução eliminada é colocada num buffer de reordenamento, do qual poderá ser sertirada mas não executada (diminuindo o core de execução). Em relação ao MIPS RS1000, RENO usa um renomeador convencional e adiciona uma lógica de seleção de saída para cada slot de instrução. Há ainda um circuito paralelelo à estrutura do renomeador que acumula e seleciona deslocamentos no lugar de nomes de registros físicos. Na avaliação do RENO foram utilizados os programas de benchmark SPECint e MediaBech, executando numa arquitetura Alpha EV6 com o compilador OSF Digital com otimizações -O4. O $RENO_{ME}$ eliminou em média 4% das instruções dinâmicas e o $RENO_{CF}$ eliminou adicionalmente 12%(SPECint) e 16%(MediaBench). Considerando também o $RENO_{CSE/RA}$, o ganho a mais foi de 5%(SPECint) e 3%(MediaBench), o que em se tratando de *load*, é um importante. No geral as eliminações promoveram ganhos de performance de 8%(SPECint) e de 13%(MediaBench), variando conforme as instruções predominantes nos respectivos caminhos críticos (*load* ou operações de ALU).

Resumo: *MP3 Optimization Exploiting Processor Architecture And Using Better Algorithms*

Mancia Anguita; J Manuel Martinez. MP3 Optimization Exploiting Processor Architecture And Using Better Algorithms; IEEE Micro May/June 2005 Vol 25 N 3; pp 81-91

Autor do resumo: *Ricardo Massahiro Nishihara RA 936161*

Introdução

O tempo de execução de toda aplicação é impactado por fatores como: arquitetura e freqüência de relógio do processor utilizado, complexidade computacional dos algoritmos utilizados no programa, escolha do compilador e opções de compilação, e capacidade do programador para explorar (de maneira explícita ou implicita) aspectos da arquitetura do processador utilizado. O artigo tratado neste resumo procura através de experimentos quantificar a influência de tais fatores na implementação de um decodificador de áudio MP3. O artigo inicia com uma breve descrição do processamento realizado por um decodificador MP3, descreve implementações de decodificador realizadas pelos autores com diferentes níveis e tipos de otimizações (considerando principalmente o uso pelo programador de funcionalidades da arquitetura do processador e de algoritmos mais eficientes), descreve a metodologia de teste usada para comparação das implementações (otimizações), e finalmente mostra e discute resultados destes experimentos. Este resumo segue esta mesma organização.

Descrição do decodificador MP3 (MPEG Audio Layer 3)

Uma descrição dos estágios básicos de processamento de um decodificador MP3 (descrito em detalhes pelo padrão ISO/IEC 11172-3) é apresentada no artigo para um melhor entendimento dos requisitos demandados. Conforme mostra a Figura 1 do artigo, o decodificador é composto pelos estágios:

- *Pré-processamento* - esse estágio encontra as estruturas de quadro dentro do bitstream MP3, e extrai destes quadros: dados referentes ao áudio comprimido, e informações auxiliares necessárias ao processo de decodificação, tais como as tabelas de Huffman e os fatores de escala.
- *Decodificação do código de Huffman* - o processo de codificação de Huffman é um esquema de codificação sem perdas que gera palavras-código de tamanho variável (códigos de Huffman) a partir de símbolos de entrada. Este mapeamento palavras-código versus símbolos de entrada é baseado na distribuição estátistica da seqüência de simbolos de entrada. Procura-se associar (codificar) símbolos de entrada que ocorrem mais freqüentemente a palavras-código mais curtas, e palavras-códigos mais longas a símbolos de entrada menos freqüentes, com o intuito de comprimir dados. O processo de decodificação é baseado na consulta a tabelas de Huffman que mapeiam palavras-código a símbolos. A informação auxiliar extraída no estágio de pré-processamento especifica qual tabela deve ser usada no quadro corrente. O padrão MPEG Audio Layer III define 17 tabelas, sendo que a palavra-código mais longa tem comprimento de 19 bits. A utilização de um método de look-up direto pelo decodificador envolveria tabelas muito grandes e por esta razão uma representação mais compacta traduz cada tabela de Huffman em uma estrutura de consulta em árvore. A palavra código inteira é recuperada quando uma folha da árvore é atingida, sendo que a cada folha está associada a um símbolo de saída do decodificador de Huffman (neste caso, o valor de um coeficiente em freqüência escalonado).
- *Requantização* - Este estágio reconstrói os coeficientes em freqüência originais (ou seja em sua escala original) a partir dos coeficientes escalonados recuperados pelo decodificador Huffman e dos fatores de escala recuperados no estágio de pré-processamento.
- *Reordenamento* - O codificador realiza um reordenamento de blocos curtos para aumnetar a eficiência da codificação de Huffman, o decoficador então tem de realizar o processo inverso.
- *Decodificação estéreo* – Para explorar redundância entre canais estéreo o codificador pode codificar as amostras em MS/Stereo e Intensity Stereo e decodificador tem de extrair os canais independentes.
- *IMDCT (Inverse Modified Discrete Cossine Transform)* - Este estágio é o responsável pela execução da transformada inversa do cosseno discreto modificada, operação dual à MDCT (*Modified Discrete Cossine Transform*) realizada no codificador. Esta transformada inversa é aplicada sobre blocos subbanda de 18 coeficientes de freqüência. A Figura 3 do artigo, ilustra de maneira esquemática este processamento.
- *Síntese do filtro polifásico* - Este estágio é responsável pela execução da síntese do banco de filtros polifáficos, operação dual da etapa de análise do banco de filtro polifásico realizado no codificador. A Figura 4 do artigo ilustra de maneira esquemática este processamento. Trata-se do estágio do decodificador com maior demanda computacional. As operações de MDCT/IDCT e em combinação com a análise/síntese do banco de filtros polifásicos são responsáveis pelo mapeamento freqüência versus tempo do codec MP3.

Implementações MP3 usadas nos experimentos

Os autores do artigo implementaram várias versões do decodificador MP3, cobrindo diferentes níveis e tipos de otimização, as quais são resumidas a seguir:

- *Standard:* Esta versão foi implementada seguindo a documentação do padrão MPEG, e utiliza somente as tabelas definidas por este padrão.
- *Basic:* Melhoria da versão *standard* principalmente através do uso de funções da biblioteca padrão do compilador, que se valem de funcionalidades da arquitetura do processador utilizado. Dentre exemplos de tais otimizações são citadas: troca de divisões em ponto flutuante por multiplicações, e de algumas multiplicações inteiras por deslocamentos; troca de funções de biblioteca computacionalmente intensas como cossenos e potências por tabelas; troca de código de alto nível feito pelo programador por funções de biblioteca que utilizam instruções específicas do processador; uso de *loop unrolling* para alguns loops.
- *SIMD:* Melhoria da versão *basic* através do uso de instruções SIMD do processador. Instruções SIMD executam a mesma operação sobre vários dados em paralelo, o que pode ser utilizado de maneira mais eficiente pelo programador para aumentar o desempenho de operações matriciais e vetoriais. O decodificador MP3 é fortemente baseado em operações vetoriais de modo sua implementação pode se beneficiar destas instruções SIMD. Para esta versão foram desenvolvidas implementações (rotinas inline assembly) para os seguintes estágios do decodificador MP3: requantização, decodificação estéreo, IMDCT, e síntese de filtros polifásicos. Instruções SIMD também foram utilizadas para melhorias para inicialização e transferência de blocos de memória.
- *Algorithmic:* Melhoria da versão *basic* através do uso de melhores algoritmos para os seguintes estágios do decodificador MP3: síntese de filtros polifásicos (método de Konstantinides), IMDCT (método de Marovich) e decodificação de Huffman (algoritmo *tree-clustering*).
- *Algorithmic SIMD:* Baseada na versão *SIMD,* combinada com implementações SIMD dos novos algoritmos para os estágios de síntese de filtros polifásicos e IMDCT usados na versão *algorithmic.* A decodificação de Huffman também foi baseada no algoritmo *tree-clustering.*

Comparação de desempenho

Sobre as condições de teste descritas pelos autores vale mencionar:

- *Compiladores*: As 5 versões do decodificador MP3 (*standard*, *basic*, *SIMD*, *algorithmic*, e *algorithmic-SIMD*) foram compiladas usando três compiladores: Intel C++ 7.1, Microsoft Visual C++ 6, e Visual C++ . NET 2003. A Tabela 1 do artigo resume as diferentes opções de

compilação utilizadas para ajustar o desempenho destes códigos-fonte. Resumindo estas opções pode-se dizer que:

- O2: inclui otimizações clássicas que são independentes do processador, expansão de funções inline;
- G6: otimiza código para Pentium Pro, Pentium II, e Pentium III, gerando código que é compatível com processadores anteriores;
- G7: optimiza código para o Pentium 4, gerando código que é compatível com processadores anteriores;
- Intel QxK: permite vetorização usando instruções SSE e MMX incluídas no Pentium III e Pentium 4;
- Microsoft arch:SSE: utiliza instruções SSE and cmov.

- *Processadores*: As implementações foram testadas nos processadores: AMD Athlon, Intel Pentium III e Intel Pentium 4. A Tabela 2 apresenta detalhes sobre os processadores utilizados na avaliação, tais como: família, memória cache, memória, e sistema operacional.
- *Bitstream de teste*: Foi utilizado um bitstream conhecido como Tristana, cujas características são mostradas na Tabela 3 do artigo.

Os autores, em seus experimentos, mediram o número de ciclos de clock por quadro ao invés do tempo gasto para decodificar o quadro, para os resultados obtidos fossem independentes do freqüência de clock do processador.

Vale destacar alguns dos resultados apresentados:

- As Figuras 6 (*standard* x *basic*) e 7 (*basic* x *demais versões*) do artigo mostram o desempenho em termos de ciclos de clock por quadro para as versões: *standard*, *basic*, *SIMD*, *algorithmic*, e *algorithmic-SIMD*, para três processadores e três compiladores estabelcidos pelo setup de teste. Nota-se uma diminuição considerável dos estágios mais lentos: síntese de filtros polifásicos, IMDCT, e Huffman, a medida que aumenta a exploração de funcionalidades da arquitetura e melhores algoritmos.
- A Figura 6 do artigo mostra um maior número de ciclos requerido pelo processor Pentium 4 em relação ao Pentium III e ao Athlon. Este maior número é explicado pelo maior número de estádios do pipeline do Pentium 4, 20, contra 12 do PentiumIII e 10 do Athlon. De acordo com os autores este maior número de estágios do pipeline embora tenha aumentado a freqüência de clock do Pentium 4 em relação ao Pentium III, também aumenta a penalidade para o código não-otimizado.
- A Figura 8 do artigo mostra os valores de speedup obtidos pelas versões *SIMD*, *algorithmic*, e *algorithmic-SIMD* comparadas à versão *basic* compilada com o compilador Microsoft Visual C++ 6. A figura mostra que a versão *algorithmic-SIMD* compilada com o mesmo compilador é 4 vezes mais rápida para o Pentium 4, 5 vezes mais rápida para o Pentium III, e 4,5 vezes mais rápida para o Athlon.
- Os dados da Figura 8 também mostram que as opções de compilação Intel QxK e Microsoft arch:SSE (aplicadas sobre a versão *basic* do decodificador) obtem menos speedup que qualquer outro executável obtido a partir da versão *SIMD* do decodificador.

Conclusões

O artigo termina ressaltando 3 lições extraídas destes experimentos:

- *Explorar funcionaliades da arquitetura pode ser tão importante quanto escolher os algoritmos mais eficientes*: comparando as Figuras 7 e 8, pode-se verificar que tanto o uso de instruções SIMD (versão *SIMD*) quanto o uso de algoritmos mais eficientes (versão *algorithmic*) , proporcionam melhorias consideráveis.
- *Programadores podem explorar funcionalidades da arquitetura de maneira mais eficiente que os compiladores*: conforme mostram os resultados da Figura 8 do artigo, as opções Intel QxK e Microsoft arch:SSE obtem menos speedup que qualquer outro executável obtido a partir da versão *SIMD*.
- *A escolha das opções de compilação depende da aplicação*: Em todos os resultados mostrados no artigo há pouca diferença de desempenho entre as opções G6 e G7. Por outro, pode-se notar em alguns um aumento de desempenho através do uso das opções de vetorização QxK e arch:SSE (vide Figuras 7 e 8 do artigo), o que é de se esperar dado que a decodificação MP3 é bastante intensa em operações vetoriais.

Fábio Augusto Menocci Cappabianco RA991724

Para melhorar o acesso à memória, modos de acesso tais como acesso paginado e read-modify-write têm sido idealizados.

Na maioria dos acessos à memória de sistemas específicos pesados, acessos a vetores são realizados. O artigo apresentou uma nova estratégia para minimizar os acessos da memória DRAM, por maximizar o número de acessos paginados, a partir de três princípios: determinar o número e tamanho de memórias, a maneira de atribuir ou colocar os vetores na memória e a seqüência de acesso dos vetores por operações de programas.

Um acesso normal à memória é realizado por um estágio de decodificação de linha, onde uma linha inteira contendo m palavras é copiada em um buffer de linha, seguindo de um estágio de decodificação de coluna ou seleção do elemento procurado na linha copiada. Posteriormente uma leitura ou escrita é realizada. Ao final do processo um novo estágio de decodificação de linha é iniciado.

Em um acesso paginado, depois de uma leitura/escrita completada, caso o próximo acesso seja na mesma linha, então somente o segundo estágio de decodificação de coluna é realizado, pois a linha desejada já está no buffer.Para se maximizar o acesso a páginas mais eficiente, dois passos são realizados: (1) alocar e mapear os vetores do código nos limites da memória. (2) escalonar os acessos no código.

O primeiro passo tenta encontrar uma configuração de memória melhor para o segundo paso, de modo que a latência de acesso à memória seja reduzida. O algoritmo citado trabalha a fim de encontrar a menor latência dentro de um determinado limite de memória. Quanto maior a instância de memória maior a latência, porém menos espaço é utilizado.

Inicialmente cada vetor é alocado em uma instância de memória diferente. Unem-se, então, pares de memórias que aumentem menos a latência, sequencialmente, até que o tamanho da memória utilizada seja menor que o limite dado. Vetores agrupados na mesma memória não podem ser acessados ao mesmo tempo na execução de instruções.

No segundo passo, faz-se um reagendamento das instruções que acessam vetores na memória, maximizando os acessos paginados na estrutura do passo anterior. Para resequenciar o código, são calculadas a latência inicial e a latência após a realocação de cada operação individualmente em cada posição possível do código, que não altere a dependência de dados. A alteração que produz o maior número de acessos paginados é escolhida, fixando-se a posição da operação correspondente. Segue-se fazendo alterações até que todas as operações sejam fixadas. A subseqüência de alterações do total realizado que possui o maior número de acessos paginados é então escolhido.

O segundo passo é repetido tendo-se então o novo agendamento como inicial, até que nenhuma das possíveis subseqüências de alterações melhore o resultado. Depois disso, inicia-se novamente o processo, executando-se o primeiro passo e o segundo passo até que se encontre a melhor combinação de latência e acesso paginado.

A aplicação do método gerou uma melhora de até 18% da letência nos programas testados pelo autor.