Paralelismo em nível de *Threads*[*]

Erikson Freitas de Morais (RA – 079781)
Instituto de Computação – IC
Caixa postal: 6176
Universidade Estadual de Campinas – UNICAMP
emorais@ic.unicamp.br

RESUMO

Limites físicos de temperatura e velocidade de processamento têm precionado projetistas de processadores a buscarem formas de mlhorar o desempenho das CPU's modernas. O caminha encontrado foi o paralelismo. Máquinas superescalares conseguem executar instruções simultaneamente devido a seu número de unidades funcionais, mas não conseguem evitar certos atrasos causados pela dependência entre as instruções que precisam ser processadas. Máquinas de múltiplos núcleos conseguem execuções simultâneas mas ainda possuem desperdício devido a dependências ocorridas durante a execução. Com esses e outros impulcionadores, surge o conceito de *Simultaneous Multi-threading* – SMT que promete bons resultados a custos reduzidos. Esse novo conceito trás melhorias consideráveis ao trhoughput dos processadores, principalmente aqueles dedicados a embiente de servidores. Este trablho mostra esse novo conceito de forma introdutória e apresenta algumas implementações comerciais encontradas.

Palavras Chaves

Threads, Paralelismo, Chip-Multithreading (CMT), Simultaneous Multithreading (SMT)

1. INTRODUÇÃO

Com os processadores atingindo limites físicos como os impostos pela termodinâmica, e ainda, cada vez mais memória sendo requisitada para o processamentto dos programas a velocidades cada vez mais distantes da velocidade dos processadores, nota-se nesse ponto um gargalo limitador que se evidencia nas máquinas atuais. No entanto as coorporações responsáveis pela fabricação de processadores assumem o desafio de buscar novas alternativas com o intuito de melhorar o desempenho dos sistemas aliado a um menor consumo de energia.

[*](Trabalho de disciplina) Apresentado na disciplina de Arquitetura de Computadores 1, como requisito final de integralização.

Uma solução adotada pelos projetistas de *hardware* é o uso de paralelismo em nível de *threads*, obtendo resultados interessantes. Alguns processadores comerciais dedicados a servidores, cujo ambiente explora macivamente o uso de *threads*, são desenvolvidos usando arquitetura de processadores SMT - *Simultaneous Multi-threading*. Como exemplos podemos citar o sistema *Hyper-Threading* da intel [10], Niagara e Niagara 2 da SUN [11] [9] e o Power 5 da IBM [8], ambos dedicados à exploração do uso de *threads* para obter melhor desempenho.

Ao longo de décadas vários projetos adotaram técnicas tradicionais como *pipeline* e a superescalaridade [6, 13, 17, 1]. Sempre com o objetivo de explorar o paralelismo e assim melhorar o tempo de resposta dos computadores. Um *pipeline* superescalar consegue atingir o paralelismo em nível de instruções, executando mais de uma instrução não dependente por cíclo, por possuir maior número de unidades funcionais e implementar técnicas para solução de falsas dependêncas [15, 1]. Esse tipo de projeto aumenta o desempenho na execução de tarefas com auto grau de paralelismo em nível de instrução, mas apenas uma *thread* por vez.

O paralelismo em nível de *threads*, além lidar com abordagens de maior granularidade [1], explora o paralelismo não apenas no nível do *hardware* mas também no fluxo de instruções. Isso significa que é possível mais de uma *thread* ser executada por vez, diferentemente da superescalaridade, onde o paralelismo ocorre apenas nas instruções de uma única *thread*.

Outra técnica que vem sendo bem aceita para o aumento de desempenho é o uso de múltiplos núcleos CMPs - *Chip Multiprocessor* [14, 2, 9, 1]. Se existem vários núcleos em um porcessador, ele pode executar simultaneamente tatas *threads* quantos núcleos existirem. Isso significa que em um processador de 4 núcleos, é possível que existam 4 *threads* sendo executadas simultaneamente, e ainda, os núcleos podem ter sido implementados com superescalaridade e SMT, gerando um ambiente ainda mais favorável ao uso de paralelismo mas com um aumento nas áreas dos CMPs. Os processadores com múltiplos núcleos que supotam múltiplas *threads* são conhecidos como CMT ou *Cip Multithreading* [16, 1].

Se muitas *threads* executam ao mesmo tempo, possivelmente a memória será bastante requisitada e precisa atender à grande demanda. Processadores comerciais têm adotado o

compartilhamento de *cache L2*, favorecendo o modelo de programação por memória compartilhada. Mas o princípio da localidade pode ser afetado nesse caso.

É sabido que programas, ou *threads* que trabalham com dados contíguos em memória proporcionam melhor desempenho do *hardware*. Com múltiplas *threads* descorrelacionadas executando simultaneamente, existe uma grande possibilidade de muitas falhas ocorrerem na *cache* e com isso o desempenho seria degenarado pelo grande número de acessos ao próximo nível da hierarquia de memória [20].

Soluçõe sinteressantes são propostas para que seja resolvido esse tipo de problema, como é o caso da arquitetura Niagara da Sun. O sistema divide a *cache* em blocos distintos para cada dois núcleos. Isso proporciona acessos múltiplos e com diminuição da interferência de outros núcleos [9].

A próxima seção mostra uma visão geral do projeto de *hardware* por SMT - *Simultaneous Multi-Threading* visto principalmente em [5]. Em seguida apresentamos, a carater ilustrativo, algumas implementações disponíveis comercialmente, como Power 5 da IBM, Niagara da SUN e a tecnologia Hyper-Threading da Intel, ambas explorando o uso de paralelismo no nível de *threads*.

2. SIMULTANEOUS MULTI-THREADING (SMT)

Durante a fase de projeto de um novo processador, uma das principais preocupações é com o desempenho daquela nova máquina. A comunidade de desenvolvimento de processadores tem se preocupado com esse assunto e propondo diverssas alternativas para ganho em desempenho, como por exemplo aumento de memória interna ao chip e a integração de sistemas como I/O e acelaração gráfica. Mas os ganhos nesses casos são superficiais.

O ganho de fato, se dá quando é aumentada a capacidade de processamento. Dadas as atuais circunstâncias de consumo de energia, aquecimento e área ocupada pelos processadores, o paralelismo tem sido o caminho para aumentar a capacidade de processamento.

Em uma máquina superescalar convencional, existe um nível de paralelismo que permite a execução de várias instruções simultaneamente, devido ao número de unidades funcionais existentes. Porém, as instruções paralelizadas são pertencentes à mesma *thread*. Além disso, muitos ciclos de *clock* são perdidos em *stalls*, principalmente devido a latência no acesso à memória quando ocorre *cache miss*.

Colocar múltiplos núcleos superescalares em um único chip não se torna uma solução tão eficiente, porque as aplicações com baixo nível de paralelismo faz com que os processadores não consiguam explorar todo o seu potencial e assim parte do *hardware* fica ocioso. Além disso, a situação se repete quando existe um baixo nível de *threads* paralelizávies. A melhor solução é utilizar uma abordagem capaz de explorar todos os tipos de paralelismo. SMT consegue atingir esse objetivo [5].

No modelo SMT os processadores podem executar *threads* com origens em diverssas formas. Elas podem ser parte de uma aplicação multithread, programas paralelizados ou programas independentes em um sistema de multiprogramado. O paralelismo no nível de instruções é explorado dentro da própria *thread*, como em uma arquitetura superescalar convencional. SMT consegue juntar caracteristicas de *hardware* superescalar com processamento multithread, resultando em um porcessador que dispacha múltiplas instruções de múltiplas threads por ciclo de clock.

A Figura 1 ilustra a diferença entre arquiteturas superescalares, Multithreading e SMT, com exemplo de execução em cada uma [5]. Na figura cada linha representa um ciclo de clock e cada bloco representa uma unidade funcional utilizada naquele ciclo. Então existem desperdícios verticais e desperdícios horizontais, representados pelos blocos não utilizados. Os desperdícios horizontais caracterizam unidades funcionais não utilizadas no ciclo e desperdícios verticais caracterizam os *stalls*.

Na Figura 1(a) é apresentado um processador superrescalar típico capaz de executar um programa (*thread*) por vez com execução de múltiplas instruções simultâneas, mas nem todas unidades funcionais são utilizadas por haver alguma dependência entre as instruções da mesma *thread*. Consequentemente temos desperdício horizontal e vertical. Uma arquitetura multithreading tenta resolver os deperdícios verticais trocando as threads em execução evitando os *stalls*, como na Figura 1(b), mas os *stalls* horizontal ainda ocorrem. Os processadores SMT conseguem escolher instruções de *threads* diferentes (Figura 1(c)) em cada ciclo. Com isso, os desperdícios horizontais são minimizados com uma gama de escolha maior e também os desperdícios verticais, pois tal como nas máquinas multithreading, os *stalls* são substituídos por instruções de outras threads.

Entretanto, uma implementação SMT pode ter impactos negativos seja no tempo de projeto ou na meta para os ciclos de clock. Os processadores SMT são derivados de projetos superescalares e por isso várias alternativas de melhoria são propostas modificando características nesse sentido, como [5]:

- Distribuir as unidades funcionais ao longo de banco de registradores duplicados (Alpha 21264);
- Construir blocos de *caches* intercalados com acessos múltiplos e independentes;
- Dividir o estágio de despacho para diminuir o tamanho da fila e reduzir o *delay*. Em [5] é proposto um estágio de *fetching* bastante eficiente;
- Reduzir o número de registradores e as portas de *cache*, trabalhando com um número reduzido de dispachos.

Algumas características são comuns a todas as implementações de SMT:

1. A maior parte dos resursos são compartilhados entre as *threads*. O modo como é feito o compartilhamento pode afetar o desempenho geral [12].
2. O processador deve garantir a melhor distribuição de recurssos entre as *threads*.

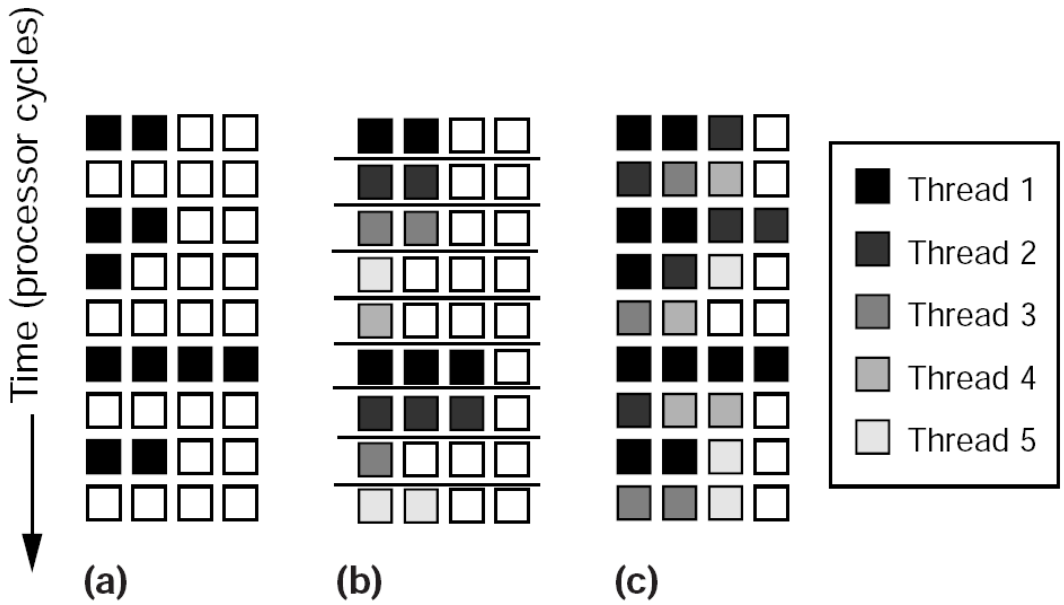

Figure 1: Particionamento de tarefas para as unidades funcionais. A figura mostra como as diferentes arquiteturas dividem o trabalho em suas unidades funcionais. Em (a) o projeto superescalar convencional, em (b) um projeto multithread superescalar e em (c) um projeto SMT. Cada linha representa um issue slot por clock. Cada quadrado branco significa um slot não usado naquele clock.

3. O aumento no ganho de prerformance é proporcionalmente maior que o aumento na área do *die*, dado que as características superescalres facilitam a evolução para SMT.

4. o Sistema Operacional vê um único porcessador físico como vários processadores lógicos.

Na proxima seção apresentamos algumas implementações comerciais baseadas no conceito de SMT.

3. IMPLEMENTAÇÕES DE SMT

Academicamente, os processadores baseados em SMT forma propostos no anos 90 [5, 7], com projetos reais alimentados no final da década, como o Alpha 21464/EV-8 da Dgital. Por motivos economicos e comerciais o projeto não foi finalizado mas serviu de base para um projeto maior na década seguinte.

3.1 SUN Niagara

O SUN Niagara, ou UltraSPARC T1, é um processador desenvolvido para o mercado de servidores de alto *workload*, com webservers e datacenters. É composto de 8 núcleos de processamento, com verssões de 4 e 6, e cada núcleo é capaz de processar até 4 *threads* simultaneas num total de 32 *threads* sendo executadas em um único processador físico simultaneamente [9].

Suas origens tem raízes em um projeto chamado Hydra desenvolvido pelo professr Kunle Olukotun, sobre chips multiprocessados. A empresa criada para produzir o processdor, resultado de tal projeto, passou por dificuldades financeiras e então foi comprada pela SUN Microsystems. O projeto Hydra se tornou a base do que veio a se tornar o processador Niagara.

Esse processador promete uma mudança radical no projeto de processadores, dado que o uso macisso de *threads* de baixa frequencia foi escolhido e ao invés da otimização de desempenho de poucas *threads* em frequencia elevada. A baixa frequencia permite menor aquecimento e portanto, menor consumo de energia. O grande número de threads permite um maior *throughtput* e portanto melhhor desempenho do *hardware*.

Cada grupo de 4 *threads* compartilham o pipeline de um núcleo, chama de *Sparc Pipe* [9], diminuindo de forma considerável a latência dos acessos à memóira, pois no caso de um *stall* uma instrução de outra *thread* é escolhida a custo quase zero, pois a mudança de contexto nãoinvolve multo trabalho.

Com campacidade de executar 32 *threads* simultaneamente, uma interface de memória eficiente é de suma inportância para o Niagara. A alta demanda de dados é suprida usando 4 controladores DDR2 integrados suportanto uma velocidade de mais de 20 GB/s. Essa integração faz com que o controlador trabalhe na mesma velocidade do processador e assim as informações da memória são recebidas mais rapidamente. Além disso cada núcleo possui um *cache* L1 de 24KB com 16KB para instruções e 8kB para dados, e todos compartilham um mesmo *cache* L2 de 3MB. O *cache* L2 é dividido em 4 bancos acessados através de um barramento *crossbar* com uma taxa de transferência altíssima na faixa de 200GB/s. O sistema *crossbar* também é usado para fazer acessos de I/O, como mostra a Fugura 2 [9].

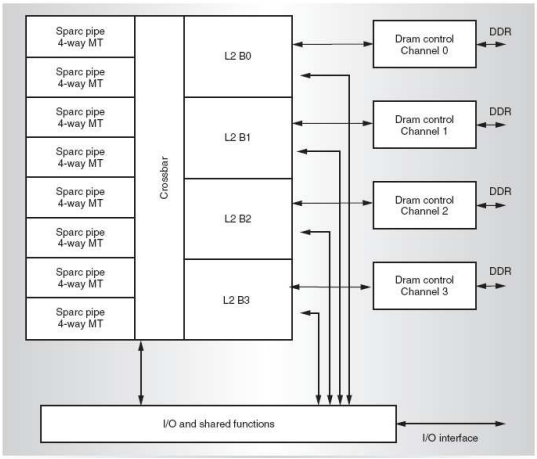

Figure 2: Esquema interno do processador Niagara.

Cada pipeline é responsável pelo gerenciamento de um grupo de 4 *threads*. Apesar da complexidade causada pelo grande número de *threads* total, pipeline é muito similar ao modelo tradicional de pileline com 5 estágios [6]. No Niagara existem apenas 6 estágios: *fetch*, *thread slect*, *decode*, *execute*, *memory* e *writeback*. A Figura 3 mostra um diagrama de blocos em alto nível para o pipeline do Niagara [9].

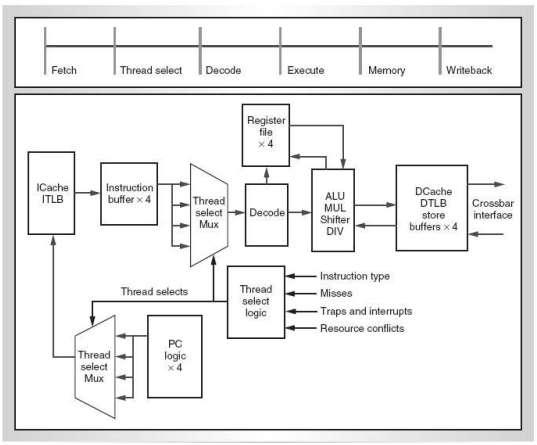

Figure 3: Peipeline do processador Niagara

Algumas limitações são encontradas no Niagara, como por exemplo o fato de possuir apenas uma unidade de ponto flutuante para os 8 núcleas existentes. Assim, o Niagara se torna incapaz de processar mais que 1–3% de operações em ponto flutuante.

Essa limitação foi revista no projeto seguinte da família Niagara, o Niagara 2.

3.2 Niagara 2

O segundo processador da série Niagara (Niagara 2) vem melhorar os resultados encontrados no porjeto de seu antecessor, o Niagara 1. As melhorias são percebidas principalemnte no que diz respeito ao sistema de ponto flutuante. A nova verssão possui uma unidade ponto flutuante para cada núcleo o que unido às demais melhorias relacionadas

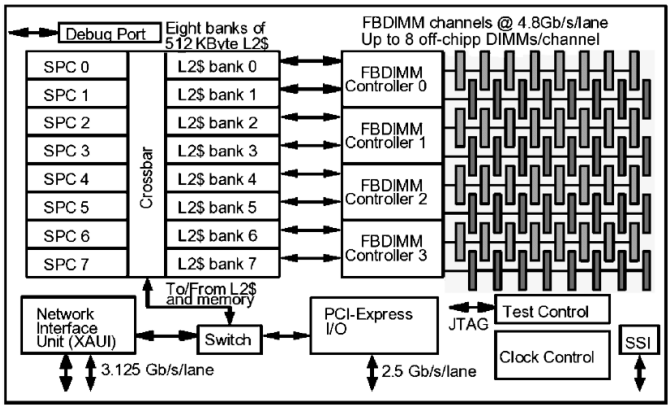

Figure 4: Diagrama de blocos do Niagara 2.

deu ao novo processador um desempenho 10 vezes maior em processamento de ponto flutuate. [11].

A arquitetura Niagara 2 possui 8 núcleos trabalhando a $1.4GH_z$ capazes de executar 8 *threads* em cada um, totalizando 64 *threads* em execução. Isso foi possível acrescentando mais uma unidade de execução em cada núcleo, permitindo duas *threads* simultâneas por pipeline. A Figura 4 mostra o diagrama de blocos do Niagara 2.

A arquitetura Niagara 2 possui 4MB de *cache* L2 divididos em 8 bancos de 512KB cada. A comunicação com os bancos é feita através de um barramento de alta velocidade (*crossbar*) tal como no seu antecessor. Além disso outros componentes foram incorporados como por exemplo duas portas Ethernet de 10Gb, uma PCI-Express 8x e quatro controladores de memória, cada um controlando dois canais. Cada núcleo possui a sua unidade de ponto flutuante e o processador possui uma unidade para cálculos criptográficos, que unidos deram ao Niagara 2 um grande ganho em desempenho, quando comparado a seu antecessor.

3.3 Intel Hyperthreading

A primeira implementação da tecnologia SMT na linha Intel foi realizada na família de processadores Xeon, em 2002. A Empresa batizou sua implementação de *Hyper-Threading Tecnology* – HTT. Nessa tecnologia cada processador físico executa dois processadores lógicos [10]. Posteriormente os processadores Pentium 4 ajudaram a popularizar essa tecnologia oferencendo o recurso HTT para usuários domésticos.

O projeto Xeon leva em consideração três pontos principais:

- minimizar o aumento do tamanho do *die*;
- permitir que o *stall* em um processador lógico não interfira no andamento do outro processador;
- caso apenas um processador lógico esteja inativo, a performance geral deve ser a mesma de um processador sem HTT.

A consequencia é que o modo de utilização de cada recurso do processador físico, que pode ser *replicado*, *tatalmente*

compartilhado ou *particionado*, foi decidido individualmente sendo que no caso de recursos compartilhados, eles devem ser recombinados quando apenas uma *thread* estiver ativa.

Existem duas partes no pipeline Xeon: *front end* e o *backk end*. O primeiro é responsável pela obtenção das instruções que serão executadas nos estágios posteriores do pipeline. As instruções são obtidas do *Trace Cache* (TC), que funciona como uma cache L1. O TC possui um apontador de instruções para cada processador lógico e usa alternadamante cada um deles em cada ciclo de clock, desde que um deles não sofra um *stall*. Nesse caso o outro será utilizado seguidas vezes até que o sistema volte a seu estado normal.

As instruções armazenadas no TC são micro-instruções (uops). Consequentemente, na ocorrência de um *cache hit* ela será enviada diretamnte para o final do *front end*, o *uop queue* (Figura 5(a)).

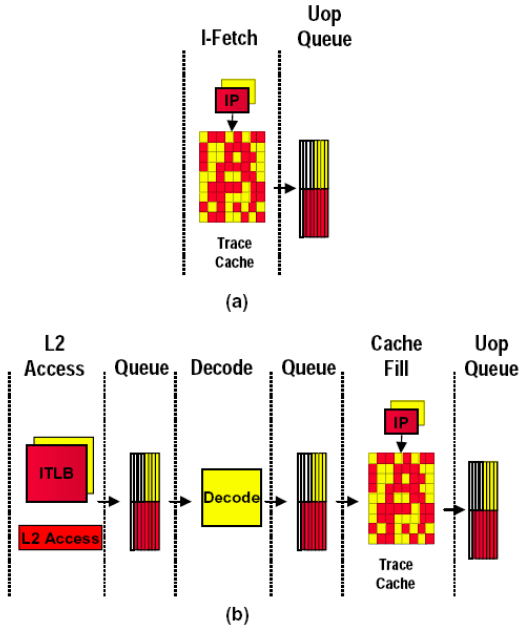

Figure 5: *Front end* **do pipeline Xeon. Em (a)** *chache hit*. **Em (b)** *cahe miss*.

Na ocorrência de um *cache miss*, a instrução é obtida do *cache* L2, decodificada e inserida no TC (Figura 5(b)).

O *back end pipeline* é responsável pela execução das instruções disponíveis na *uop queue*. Para aumentar o paralelismo, as instruções podem ser executadas fora da ordem original dada na *thread* de entrada, desde que não haja dependência entre as instruções (*out-of-order execution* – OOO).

Dessa forma, o estágio *rename* faz o mapeamento dos registradores arquiteturais de cada processador lógico para os registradores físicos disponíveis e passa as instruções para o estágio *queue* dividindo as instruções em 2 filas: uma para operações de leitura/escrita na memória e outra para as demais operações. Uma vez nas filas, as instruções são enviadas para execução através do estágio *scheduler*, que envia até 6 *uops* por ciclo, aos estágios seguintes da execução. Então a escolha é feita baseada na dependência entre as instruções. O último estágio é o *retirement*, que garante o termino das instruções na ordem original de entrada. A Figura 6 ilustra o processo.

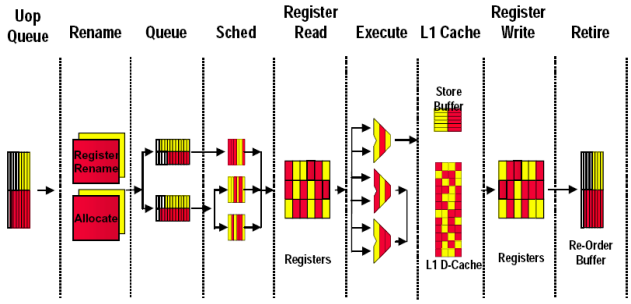

Figure 6: Detalhe do pipeline Out-of-order de um processador Xeon.

Os processadores com suporte a HTT possuem 2 modos de operação

- *Single-Tash*(ST)
- *Multi-Tash*(MT)

No modo MT é disponibilizado para o Sistema Operacional– SO 2 processadores lógicos que compartilham boa parte dos recursos do processador físico. Se necessário, o SO pode mudar o estado para ST (ST-0 ou ST-1) especificando qual processador lógico estará ativo. Isso é feito enviando uma instrução *halt* para um dos processadores. Para voltar ao modo Mt, basta enviar uma interrupção ao processador inativo.

Além do TC, existem *caches* L1, L2 e L3, todos compartilhados pelos dois processadores lógicos. Mesmo com a possibilidade de conflitos degenerativos à performance em consequencia ao compartilhamento, ele pode ser usado para implementar o sistema de *help threads* [19, 18]. Nesse sitema uma *thread* auxiliar é usada para fazer o *pré-fetching* do cache para a *thread* principal. Esse artifício é particularmente útil em aplicações com puco paralelismo inerente.

O trabalho desenvolvido em [3], mostra testes de performance para aplicações de vídeo aplicados na arquitetura Intel com a tecnologia Hyper-Threading.

3.4 IBM Power 5

A IBM lancou em 2001 o processador Power 4. O projeto do Power 4 integra dois núcleos em um único processador, que consegue resolver o paralelismo no nível de *threads* de forma transparente usando os núcleos replicados. Além disso o Power 4 possui um sistema de execução de instruções fora de ordem que possibilita escolher outra instrução no caso de um *cache miss* [8].

O processador Power 5 é a nova geração na linha do Power 4 na arquitetura PowerPC. É um processador multi-core (2 núcleos por *die*) e SMT, com 2 *threads* simultâneas por núcleo [4]. Os projetistas primaram pela compatibilidade com

o Power 4 durante o projeto, de forma que os binários compilados para Power 4 continuam funcionando na nova arquitetura.

Assim como os processadores com a tecnologia HTT, o Power 5 possui 2 modos de operação: *single-threaded* (ST) e *enhanced SMT*. Apesar do modo SMT ser exclusivo do Power 5, a estrutura do pipeline é idêntica ao Power 4, permitindo que sistemas otimizados para executar no Power 4 executem eficientemente no Power 5 [8].

O pipeline é composto de vários estágios, porém os estágios são agrupados em 3 macro-estágios: *instruction fetch*, *group formation and instruction decode* e *out-of-order processing*, tal com na Figura 7.

No 1^o macro-estágio do pipeline, o Power 5 usa 1 registrador PC (*program counter*) para cada *thread*. As instruções são trazidas do *cache* L1, compartilhado entre as *threads*, de forma alternada entre os PCs. Em cada ciclo, são trazidas até 8 instruções do IC (*instruction cache*), porém todas da mesma *thread*. Após trazidas ao pipeline, as instruções são examinadas para detecção de branches no estágio *branch prediction* – BP sendo que a predição de branches é realizada através de 3 tabelas de histórico de branches (BHT). Essas tabelas são compartilhadas pelas 2 *threads*.

No macro-estágio seguinte as instruções são decodificadas. No estágio D0, as instruções de cada *thread* são colocadas em 2 filas e, baseado na prioridade de execução de cada *thread*, o processador seleciona as instruções de uma das filas e forma um grupo (estágios D1, D2 e D3), onde as instruções serão decodificadas em paralelo. Ainda nesse macro-estágio, o processador aloca os recursos necessários para execução do grupo e faz o despacho no estágio GD, para o macro-estágio final.

No último macro-estágio, as instruções do grupo são separadas para o processamento fora de ordem (OOO).

O Power 5 apresenta 2 características que aprimoram o modo de operação SMT: balanceamento dinâmico de recursos e prioridade ajustável de *threads*.

O objetivo do balanceamento dinâmico de recursos é garantir que ambas as threads usem o processador de forma justa e eficiente. Para isso, o uso dos recursos é monitorado e, de acordo com fatores como prioridade das *threads*, incidência de *stall* e utilização de recursos, o particionamento dos recursos entre as *threads* é alterado dinamicamente.

Já a prioridade ajustável de *threads* permite que o *software* executado em qualquer nível, seja o SO, *middleware* ou aplicação final, altere a prioridade de uma *thread*

Apesar do modo SMT geralmente extrair um desempenho mais eficiente do processador, existem aplicações onde o uso de apenas uma *thread* oferece um melhor desempenho. Nessas situações, o SO pode instruir o processador a operar no modo *single-thread* (ST).

O Power 5 suporta 2 tipos de modo ST: dormente (*dormant*) ou nulo (*null*). Do ponto de vista do *hardware*, a única diferença entre os 2 tipos é que o primeiro pode ser desativado através de interrupções ou instruções especiais. Já para o SO, no modo nulo é como se o processador suportasse apenas 1 *thread*.

4. CONCLUSÕES

A preformance dos processadores que utilizam do conceito de *Simultaneous Multi Threading* – SMT é mostrada em diverssos trabalhos, [8, 3, 10, 11, 9] cada qual utilizando uma implementação de arquitetura diferenciada. Em todos os casos vistos, o aumento de performance é bastante superior ao aumento no tamanho do *die*, por exemplo, os processadores da família Xeon da Intel, tiveram um ganho médio de 21% ao custo de 5% de aumento [10] e o Power 5 da IBM, proporcionaram uma melhorar de 100% a um cursto de 24% a mais na área de cada núcleo [8].

As arquiteturas SMT podem traser mudançca significaticas no projeto e manufatura de processadores na direção de melhores desempenhos a custos menores, tanto na produção quanto durante o seu uso, como os processadores Niagara da Sun, que trabalham a frequancias menores e com isso consomem menos energia, porém com um *throughput* bem elevado.

5. REFERÊNCIAS

[1] M. A. Z. Alves, H. C. F. F. R. Wagner, and P. O. A. Navaux. Influência do compartilhamento de cache l2 em um chip multiprocessado sob cargas de trabalho com conjuntos de dados contíguos e não contíguos. In *VIII Workshop em Sistemas Computacionais de Alto Desempenho –WSCAD*, pages 27 — 34, 2007.

[2] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 282–293, New York, NY, USA, 2000. ACM.

[3] Y.-K. Chen, M. Holliman, and E. Debes. Video applications on hyper-threading technology. *Multimedia and Expo, 2002. ICME '02. Proceedings. 2002 IEEE International Conference on*, 2:193–196 vol.2, 2002.

[4] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson. Design and implementation of the power5/spl trade/ microprocessor. *Integrated Circuit Design and Technology, 2004. ICICDT '04. International Conference on*, pages 143–145, 2004.

[5] S. Eggers, J. Emer, H. Leby, J. Lo, R. Stamm, and D. Tullsen. Simultaneous multithreading: a platform for next-generation processors. *Micro, IEEE*, 17(5):12–19, Sep/Oct 1997.

[6] J. L. Hennessy and D. A. Patterson. *Arquitetura de Computadores Uma Abordagem Quantitativa*. Editora Campus, 3^a ed. edition, 2003.

[7] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An

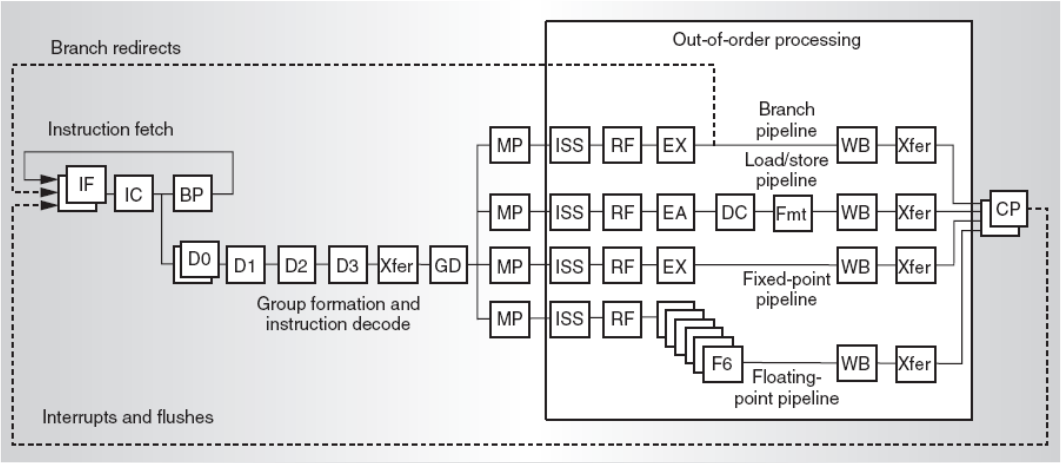

Figure 7: Pipeline de Instruções do processador Powe 5. IF=Instruction Fetch, IC=Instruction Cache, BP=Branch Predict, D0=Decode Stage 0, Xfer=Transfer, GD=Group Dispatch, MP=Mapping, Iss=Instruction Issue, RF=Register File Read, EX=Execute, EA=Comper Address, D=Dada Cache, F6=Six-Cycle float point execution pipe, Fmt=Data Format, WB=Write BAck e CP=Group Commit.

elementary processor architecture with simultaneous instruction issuing from multiple threads. *SIGARCH Comput. Archit. News*, 20(2):136–145, 1992.

[8] R. Kalla, B. Sinharoy, and J. Tendler. Ibm power5 chip: a dual-core multithreaded processor. *Micro, IEEE*, 24(2):40–47, Mar-Apr 2004.

[9] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21–29, March-April 2005.

[10] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 06:4 — 15, frebuary 2002.

[11] U. Nawathe, M. Hassan, K. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-core, 64-thread, power-efficient sparc server on a chip. *Solid-State Circuits, IEEE Journal of*, 43(1):6–20, Jan. 2008.

[12] S. Raasch and S. Reinhardt. The impact of resource partitioning on smt processors. *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 15–25, Sept.-1 Oct. 2003.

[13] W. SAtallings. *Arquitetura de Organização de Computadores*. Prentice Hall, 2005.

[14] B. Shinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. In *IBM Journal of Research and Development*, volume 49, 2005.

[15] J. Smith and G. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, Dec 1995.

[16] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *11th International Symposium on High-Performance Computer Architecture – HPCA*, pages 248 — 252, 2005.

[17] T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.

[18] H. Wang, P. H. Wang, R. D. Weldo, S. M. Ettinger, H. Saito, M. Girkar, S. S. wei Liao, and J. P. Shen. Speculative precomputation: Exploring the use of multithreading for latency tools. *Intel Technology Journal*, pages 22 — 35, february 2002.

[19] P. Wang, J. Collins, D. Kim, B. Greene, K.-M. Chan, A. Yunus, T. Sych, S. Moore, J. Shen, and H. Wang. Helper threads via virtual multithreading. *Micro, IEEE*, 24(6):74–82, Nov.-Dec. 2004.

[20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.