Processadores SMT e paralelismo em nível de threads

Felipe Leme
IC/UNICAMP
ra930886@ic.unicamp.br

RESUMO

A arquitetura de multi-threads executadas simultaneamente (*SMT – Simultaneous Multithreading*) oferece um aumento na performance do processador a um custo reduzido no aumento da área do *die* – tipicamente, um ganho de 20% na performance a um custo de 5% a mais de transistores. O ganho de perfomance vem da melhor utilização dos recursos do processador e da mitigação do impacto causado pela latência da memória; já o baixo aumento da área vem do aproveitamento de características já existentes nos processadores super-escalares.

Nesse artigo, serão apresentadas as arquiteturas de alguns processadores SMT comerciais.

1.INTRODUÇÃO

Com os processadores atingindo o limite de aquecimento e o acesso à memória tornando-se um gargalo cada vez maior, as empresas buscam alternativas para produzir processadores que ofereçam melhor desempenho porém com menor consumo de energia. E a principal tendência nesse sentido é priorizar o paralelismo em nível de *threads* e *cores* em detrimento ao paralelismo em nível de instruções.

Nesse artigo, será apresentada a arquitetura de processadores SMT (*Simultaneous Multi-Threading*), que explora o paralelismo em nível de threads em um mesmo processador. O artigo está dividido da seguinte forma: na seção 2 será descrita a arquitetura SMT; a seção 3 e sub-seções descreverão algumas implementações comerciais de processadores SMT e a seção 4 concluirá o artigo.

2.ARQUITETURA SMT

Em um processador super-escalar convencional, apenas as instruções de uma thread são executadas por vez. Embora esse tipo de processador ofereça paralelismo em nível de instruções (já que o pipeline processa pedaços de várias instruções simultaneamente), muitos ciclos são perdidos em *stalls* (principamente devido a latência no acesso à memória quando da ocorrência de um *cache miss*) e o processador não é utilizado em sua totalidade (devido ao baixo paralelismo de algumas instruções) [1]. A figura 1(a) representa a utilização temporal do processador; cada linha representa um ciclo e os blocos representam a utilização das unidades funcionais. Blocos vazios representam desperdício de uso do processador; stalls são desperdícios verticais e unidades funcionais não utilizadas em um ciclo são desperdícios horizontais.

Para diminuir o desperdício vertical, os processadores multi-threads trocam a thread em execução quando da ocorrência de um stall; essa troca de contexto também é chamada de troca leve (pois requer poucos ciclos), para diferenciar da troca realizada pelo sistema operacional (S.O.), que pode custar até centenas de ciclos. A figura 1(b) ilustra essa situação.

Já os processadores SMT vão um passo além na diminuição do desperdício: além de não precisar trocar o contexto das threads (desperdício vertical), como o processador tem as instruções de mais de uma thread disponível simultaneamente, ele pode escolher melhor quais instruções enviar ao pipeline, melhorando o paralelismo das instruções (o que diminui o desperdício horizontal) e mitigando a latência causada pelos caches misses (diminuição do desperdício vertical). A figura 1(c) ilustra a situação.

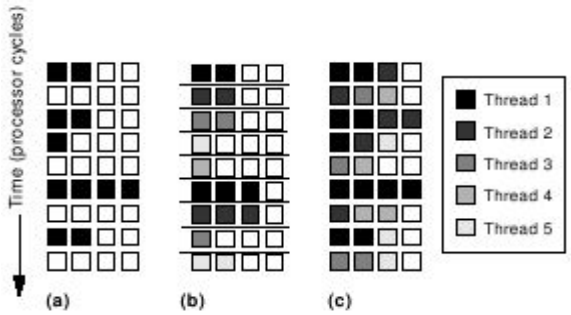

Figura 1 Exemplo de utilização das unidades funcionais em processadores (a) super-escalar convencional, (b) multi-threads e (c) SMT. (Fonte: [1])

Embora cada implementação de SMT tenha sua arquitetura própria, algumas caracteristicas são comuns a todas:

- o estado arquitetural (como registradores lógicos) é replicado para cada thread.
- a maior parte dos recursos do processador (como unidades funcionais) são compartilhados (embora alguns sejam replicados) entre as threads; a maneira como os recursos são compartilhados ou particionados pode afetar a perfomance geral do processador [7]
- o processador deve garantir o uso justo (*fairness*) dos recursos entre as threads.
- o aumento no ganho de performance é proporcionalmente maior que o aumento na área do *die*, pois as características dos processadores super-escalares facilitam a evolução para SMT.
- para o S.O., um único processador físico aparece como múltiplos processadores lógicos.

3.IMPLEMENTAÇÕES COMERCIAIS

Embora processadores SMT já tenham sido propostos academicamente desde o início dos anos 90 [1,2], apenas no final da década surgiram os primeiros projetos comerciais (como o Alpha 21464/EV-8 [3], da Digital, anunciado em 1999, porém não finalizado devido às fusões e venda de departamentos da empresa) e somente no início da década seguinte os processadores SMT realmente chegaram ao mercado.

Nessa seção, serão descritos alguns dos principais processadores SMT disponíveis atualmente (ou com previsão de lançamento próximo) no mercado.

3.1 Intel Hyper-Threading Technology (HTT)

3.1.1 Visão Geral

O primeiro processador da Intel a incorporar o conceito de SMT (que a empresa batizou de Hyper-Threading Technology ou HTT) foi a família Xeon, em 2002. A arquitetura HTT consiste em 2 processadores lógicos para cada processador físico; o número de processadores físicos depende do modelo do processador.

Posteriormente, os processadores Pentium 4 passaram a oferecer HTT, ajudando a popularizar a tecnologia (devido principalmente aos esforços mercadológicos da Intel); nas seções seguintes, porém, será descrita a arquitetura do Xeon, baseada em um dos únicos *papers* da Intel sobre o assunto [4].

3.1.2 Design

O design do Xeon levou em consideração 3 objetivos primários:

- minimizar o aumento no tamanho do *die*;
- permitir que o stall em um processador lógico não interferisse com o andamento do outro;
- caso apenas um processador lógico esteja ativo, ele deve obter a mesma performance de um processador sem HTT.

Em consequência desses objetivos, o modo de utilização (replicado, totalmente compartilhado ou particionado) de cada recurso do processador físico foi decidido individualmente e, no caso de recursos compartilhados, eles devem ser re-combinados quando apenas uma thread estiver ativa.

3.1.3 Front End Pipeline

O pipeline do Xeon é dividido em 2 partes: o *front end* e o *back end*.

O *front end* 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 o cache L1 de instruções. O TC possui 1 IP (*instruction pointer*) para cada processador lógico e usa alternadamente cada IP por ciclo (a não ser que um dos processadores sofra um stall – nesse caso o IP do outro processador é usado seguidamente até o término do stall). As instruções armazenadas no TC são na verdade micro-instruções (*uops*), já decodificadas; consequentemente, na ocorrência de um *cache hit*, a instrução já é enviada para o estágio final do *front end*, o *uop queue*, como mostra a figura 2(a). Na ocorrência de um cache miss, a instrução é obtida do cache L2, decodificada e inserida no TC, como mostra a figura 2(b).

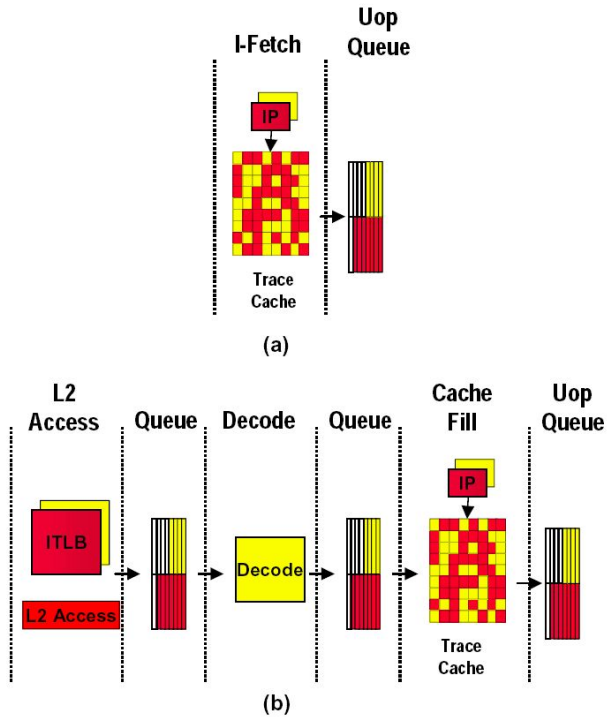

Figura 2 Front end pipeline do Xeon nas situações de (a) cache hit e (b) cache miss. (Fonte: [4])

3.1.4 Back End Pipeline

O *back end pipeline* é responsável em executar as instruções disponíveis na *uop queue*. Para aumentar o paralelismo, as instruções podem ser executadas fora da ordem original da thread *(OOO – out-of-order execution)*, desde que não haja dependência entre as instruções. Sendo assim, 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*, que divide as instruções em 2 filas: uma para operações de leitura/escrita de 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 para execução; a escolha é feita baseada na dependência entre as instruções. O último estágio é o *retirement*, que garante que as instruções serão finalizadas na ordem original de execução. A figura 3 ilustra o processo.

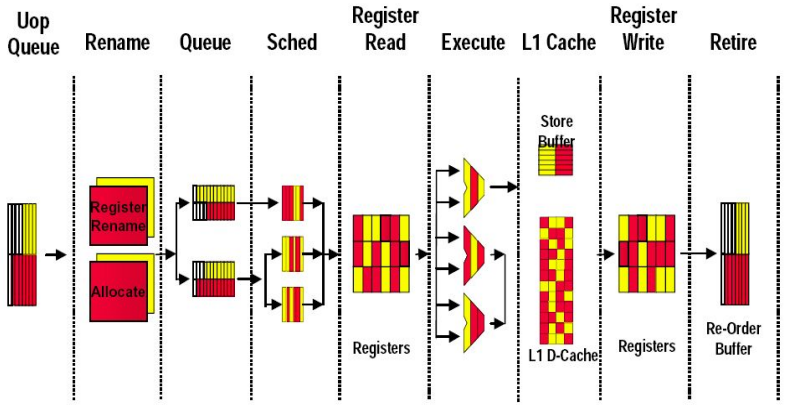

Figura 3Back end pipeline do Xeon. (Fonte: [4])

3.1.5 Modos de operação

Os processadores com suporte a HTT possuem 2 modos de operação: *Single-Task (ST)* ou *Multi-Task (MT)*. No modo MT, o S.O. enxerga 2 processadores lógicos que compartilham boa parte dos recursos do processador físico. Caso o S.O. deseje passar para o modo ST (ST-0 ou ST-1, quando apenas o processador lógico 0 ou 1, respectivamente, estará ativo), ele envia a instrução halt para um dos processadores, de tal forma que apenas o outro processador ficará ativo e consumindo 100% dos recursos físicos disponíveis. Para voltar ao modo MT, basta o O.S. enviar ao processador inativo uma interrupção.

3.1.6 Demais Caches

Além do TC, a família Xeon possui um cache de dados L1 e caches L2 e L3 e os 3 caches são compartilhados pelos 2 processadores lógicos. Embora esse compartilhamento possa causar conflitos que consequentemente diminuiriam a performance, ele também pode ser usados para implementar o mecanismo de *helper threads* [5, 6], onde uma thread auxiliar é usada para fazer o *pré-fetch* do cache para a thread principal – esse artifício é particularmente útil em aplicações com pouco paralelismo inerente.

3.2 IBM Power5

O processador Power5 – sucessor do Power4 na arquitetura PowerPC – é um processador multi-core (2 cores por *die*) e SMT (com 2 threads simultâneas por core) [9].

3.2.1 Modos de Operação e Pipeline

Assim como o HTT, o Power5 também apresenta 2 modos de operação: *single-threaded* (ST) e *enhanced SMT*. Apesar do modo SMT ser exclusivo do Power5, a estrutura do pipeline é idêntica ao Power4, permitindo que sistemas otimizados para executar no Power4 executem eficientemente no Power5 também.

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*, como mostra a figura 4.

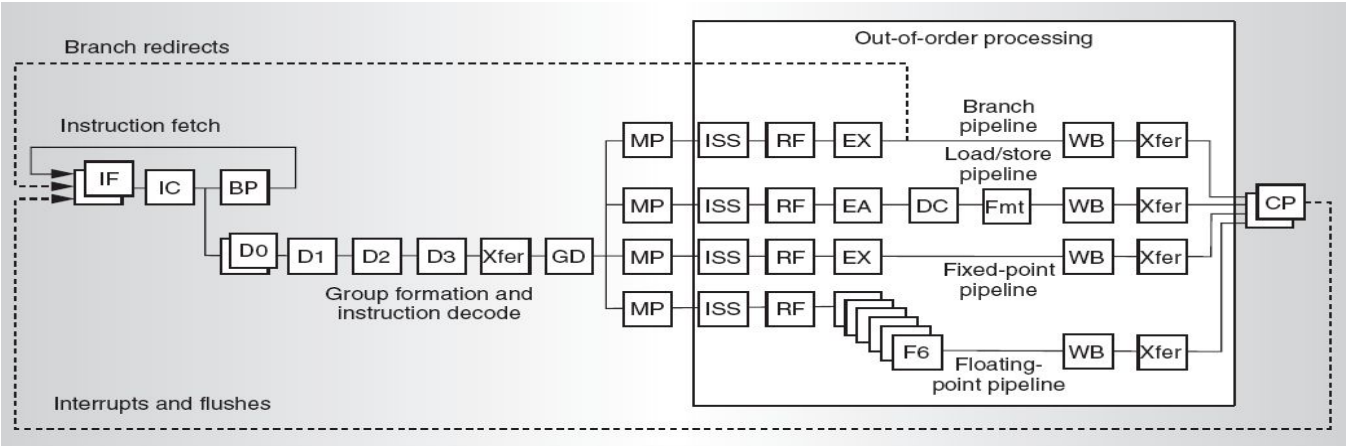

Figure 3. Power5 instruction pipeline (IF = instruction fetch, IC = instruction cache, BP = branch predict, D0 = decode stage 0, Xfer = transfer, GD = group dispatch, MP = mapping, ISS = instruction issue, RF = register file read, EX = execute, EA = compute address, DC = data caches, F6 = six-cycle floating-point execution pipe, Fmt = data format, WB = write back, and CP = group commit).

Figura 4Pipeline de 3 macro-estágios do Power5 (Fonte: [9])

No 1º macro-estágio do pipeline, o Power5 usa 1 registrador PC (*program counter*) para cada thread e as instruções são trazidas do cache L1 (que é 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 (estágio *BP – branch prediction*) e a predição de *branches* é realizada através de 3 tabelas de histórico de branches (*BHT*), que são compartilhadas pelas 2 threads.

No macro-estágio seguinte é feita a decodifição das instruções. 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 (estágio *GD*) para o macro-estágio final.

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

A figura 5 mostra o *workflow* do processamento das instruções.

3.2.2 Características Enhanced SMT

O Power5 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 – em qualquer nível, seja o S.O., *middleware* ou aplicação final – altere a prioridade de uma thread.

3.2.3 Modo de operação ST

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 S.O. pode instruir o processador a operar no modo *single-thread* (*ST*).

O Power5 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 S.O., no modo nulo é como se o processador suportasse apenas 1 thread.

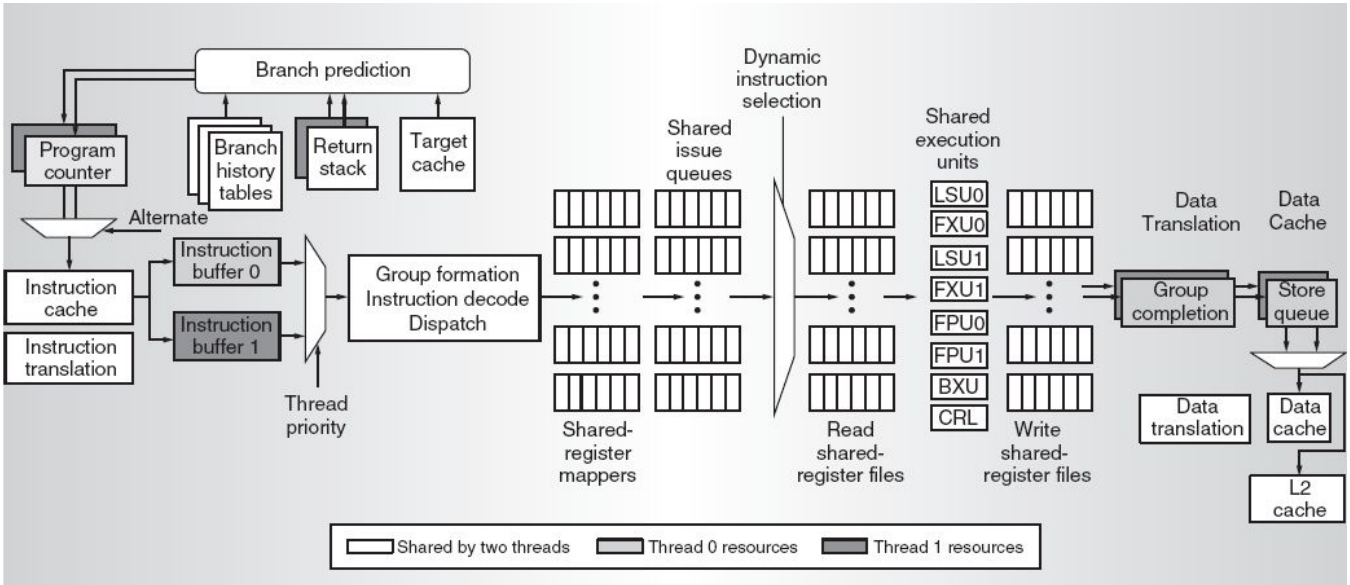

Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

Figura 5Fluxo de execução das instruções no Power5 (Fonte: [9])

3.3 Sun Niagara

3.3.1 Visão geral

O processador Niagara da Sun Microsystem promete uma mudança radical [10] no *design* de processadores para uso em servidores comerciais: em vez de otimizar o desempenho para 1 ou 2 threads executando em alta frequência, o processador é focado no uso maciço de threads (32) em menor frequência. O grande número de threads simultâneas possibilita um maior *throughput* e a menor frequência diminui a dissipação de calor (a expectativa é que o consumo de energia de um processador Niagara seja da ordem de apenas 60W!) – duas características importantes para servidores comerciais alocados em *data centers*.

Para suportar a execução simultânea de 32 threads, os *designers* do Niagara aproveitaram características de processadores multi-core e multi-threads: cada processador é composto de 8 cores, que por sua vez são compostos por um grupo de 4 threads (*thread group*); cada thread group compartilha o pipeline do core (chamados de *Sparc pipe)*. Dessa forma, a latência causada pelo acesso à memória é praticamente nula, pois no caso de stall em uma thread, o core seleciona outra thread do grupo e a envia ao pipe; essa troca de contexto tem custo 0 em ciclos, pois os recursos físicos do core são compartilhados pelas threads.

O alto paralelismo das threads demanda um acesso eficiente à memória e outros recursos da CPU. Para isso, cada pipe possui caches L1 para instruções e dados (16kb e 8kb, respectivamente) e as 32 threads compartilham um cache L2 de 32 MB e 4 bancos; o acesso ao cache L2 e ao sistema de I/O é feito através de um barramento *crossbar* que provê um *bandwidth* de mais de 200 GB/s. Já a memória consiste de 4 canais de DDR2 DRAM, suportando uma velocidade de mais de 20 GB/s. A figura 6 ilustra o diagrama do processador Niagara.

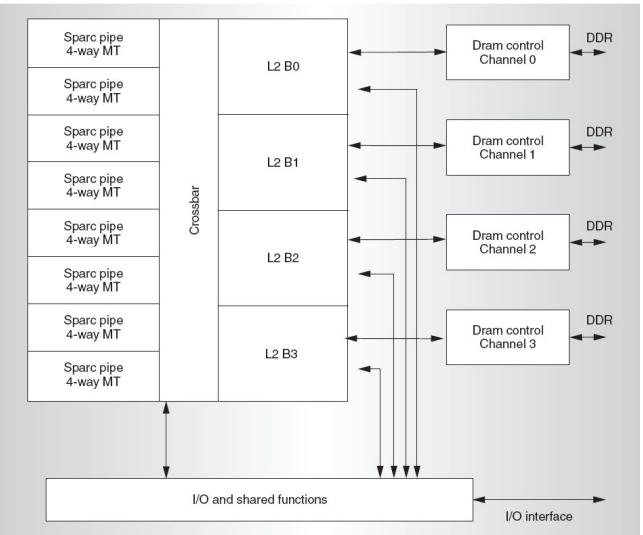

Figura 6 Estrutura do processador Niagara. (Fonte: [10])

Outra característica importante do processador é que as aplicações existentes poderão usar as múltiplas threads sem precisar de nenhuma re-compilação: para o S.O., as 32 threads serão vistas como 32 processadores lógicos. Certamente o S.O. terá que ser modificado para tirar melhor proveito das threads; porém, como o S.O. (Solaris) também é fornecido pela Sun, essa adaptação será transparente para as aplicações.

Nas sub-seções seguintes, serão avaliados mais detalhadamente alguns componentes do processador.

3.3.2 Sparc Pipeline

Cada pipeline é responsável pelo gerenciamento de um grupo de 4 threads. As threads do grupo compartilham alguns recursos – como o cache L1, a TLB, as unidades funcionais e a maior parte dos registradores de pipeline, e possui outros de uso exclusivo – como um conjunto de registradores e buffers de instruções e dados.

Apesar da complexidade causada por várias threads, o pipeline é simples e consiste de apenas 6 estágios: *fetch, thread select*, *decode, execute, memory* e *writeback*. A figura 7 ilustra o pipeline e as unidades funcionais usadas em cada estágio.

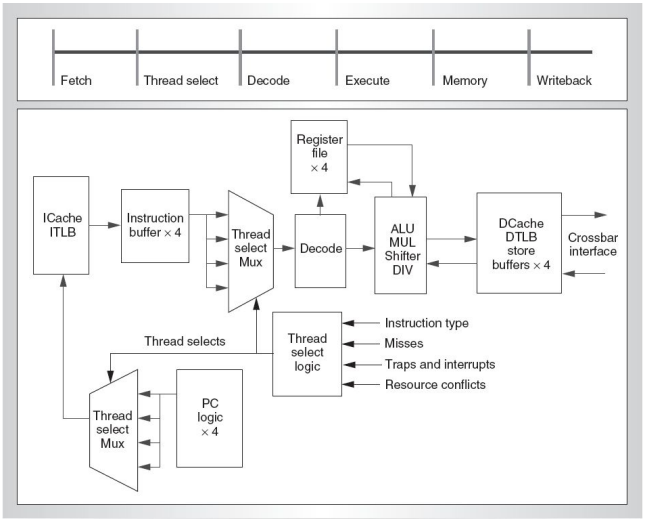

Figura 7 Pipeline do processador Niagara. (Fonte: [10])

O Sparc Pipeline é muito similar ao modelo clássico de pipeline com 5 estágios [11]; a principal diferença é a adição do estágio *thread select*.

Conforme mostra a figura 7, a lógica de seleção de threads é usada nos 2 primeiros estágios, onde apenas uma thread por ciclo é selecionada para execução; consequentemente, os registradores desses estágios são replicados para cada thread (que também possuim um *Program Counter* exclusivo). No estágio seguinte (*decode*), a instrução é decodificada e sua latência determinada (através de um bit da instrução). Instruções como *ADD* e *SHIFT* são consideradas de baixa latência e o

resultado fica disponível já no final do estágio seguinte (*execute*); já as instruções de alta latência – como *DIV*s e acessos à memória – usam uma troca de thread.

3.3.3 Seleção de Threads

A política padrão de seleção de threads é trocar de thread a cada ciclo, dando prioridade a thread menos recentemente utilizada e garantido assim uma divisão justa *(fairness)* no acesso ao pipeline por todas as threads. Mas as threads tornam-se indisponíveis quando sofrem um stall ou executam instruções de longa latência e tem sua prioridade baixada quando executam instruções especulativas (como *loads*, onde um *cache hit* é assumido).

3.3.4 Perspectiva

O processador Niagara encontra-se em fase final de testes e a previsão é que ele entre em produção no primeiro semestre de 2006. Se o Niagara cumprir as expectativas, ele dará um novo fôlego à Sun, que nos últimos anos vem perdendo mercado para servidores Lintel (S.O. Linux executando em hardware Intel X86). E a Sun não pretende parar por aí: a evolução do Niagara será a família de processadores Rock, que adicionará características SMP ao processador.

4.CONCLUSÃO

Os processadores SMT oferecem um alto ganho de performance a um baixo/moderado custo no aumento da área do die [12] e resultados mostram que essa relação é proporcional: o processador Xeon apresenta um ganho médio de 21% ao custo de 5% de aumento [4], enquanto que o Power 5 melhora o desempenho em quase 100% às custas de 24% a mais de área por core [8].

5.REFERÊNCIAS

[1] Eggers, S. J., Emer J. S., Levy, H. M., Lo, J. L., Stamm, R. L., Tullsen, D. M. *Simultaneous Multithreading: A Platform for Next-Generation Processors*. IEEE Micro, (Set/Out 1997), 12-19

[2] Hirata, H. et al.*An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads*. Proc. Int'l Symp. Computer Architecture, Assoc. of Computing Machinery,(1992), 136-145

[3] *Simultaneous Multithreading Project*. University of Washington Computer Science & Engineering, http://www.cs.washington.edu/research/smt/

[4] Marr, D. et al. *Hyper-Threading Technology Architecture and Microarchitecture*. Intel Technology Journal, (Q1/2002), 4-15

[5] Wang, P. H. et al. *Helper Threads via Virtual Multithreading*. IEEE Micro, (Nov/Dez 2004), 74-82

[6] Wang, P.H et al. *Speculative Precomputation: Exploring the Use of Multithreading for Latency*. Intel Technology Journal, (Q1/2002), 28-35

[7] Raasch, S. E., Reinhardt, S. K. *The Impact of Resource Partitioning on SMT Processors*. Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)

[8] Kalla, R., Sinharoy, B., Tendler, J. M. *IBM Power5 Chip: a Dual-Core Multithreaded Processor*. IEEE Micro, (Mar/Abr 2004), 40-47

[9] Clabes J. et al. *Design and Implementation of the POWER5 Microprocessor*. 2004 IEEE International Conference on Integrated Circuit Design and Technology, (2004), 143-145.

[10] Kongetira P., Aingaran K., Olukotun K.*Niagara: A 32-Way Multithreaded Sparc Processor*. IEEE Micro, (Mar/Abr 2005), 21-29

[11] Patterson, D. A., Hennessy J. L. *Computer Organization Design, The Hardware/Software Interface.* Morgan Kaufmann. Third Edition, 2005.

[12] Burns, J., Gaudiot, J.L. *SMT Layout Overhead and Scalability*. IEEE Transactions on Parallel and Distributed Systems, Vol.13 No. 2, (Fev/2002)