Comparativo entre Diferentes Estratégias de Multithreading

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

fernando.kronbauer@students.ic.unicamp.br

RESUMO

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

Categorias e Descritores de Assunto

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

Termos Gerais

Performance, Design.

Palavras-Chave

Multithreading, Hyper-Threading, UltraSPARC T1.

1.INTRODUÇÃO

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

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

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

2.A TECNOLOGIA HYPER-THREADING DA INTEL

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

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

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

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

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

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

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

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

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

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

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

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

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

3.NIAGARA

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.COMPARATIVO

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

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

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

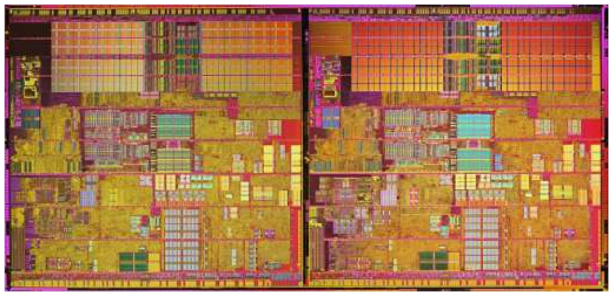
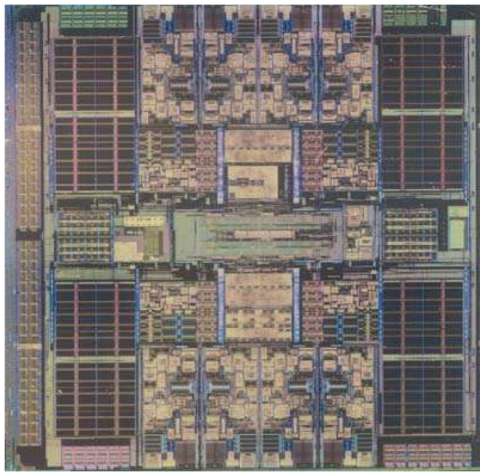


Figura 1. Fotografia do *die* do Niagara



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

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

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

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

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

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

4.CONCLUSÕES

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

5.REFERÊNCIAS

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

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

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

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

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

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

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