UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

HENRIQUE COTA DE FREITAS

Chip Multithreading: Conceitos, Arquiteturas e Tendências

Trabalho Individual I
TI 1253

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, dezembro de 2006.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ALAT	Advance Load Address Table
BMT	Blocked Multithreading
BP	Branch Prediction
BR	Branch
CISC	Complex Instruction Set Computing
CMP	Chip Multiprocessor
CMT	Chip Multithreading
CR	Computação Reconfigurável
FP	Floating-Point
FX	Fixed-Point
GPP	General-Purpose Processor
IC	Instruction Cache
IF	Instruction Fetch
IFAR	Instruction Fetch Address Register
IMT	Interleaved Multithreading
ISA	Instruction Set Architecture
L1	Cache Level 1
L2	Cache Level 2
L3	Cache Level 3
LD / ST	Load / Store
MCM	Multi-Chip Module
MCV	Memory Control Unit
MMV	Monitor de Máquina Virtual
MPSoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
PC	Program Counter
RISC	Reduced Instruction Set Computing

SMP	Symmetric Multiprocessor
SMT	Simultaneous Multithreading
TLB	Translation Look-Aside Buffer
VLIW	Very Long Instruction Word

LISTA DE FIGURAS

RESUMO

Nos projetos atuais de *chips multithreading* (CMTs) um dos pontos críticos é a definição de qual abordagem *multithreading* adotar. A classificação *Explicit Multithreading* tem sido utilizada largamente e compreende as técnicas para exploração do paralelismo nativo de fluxos de instruções de origens ou programas diferentes, sem o uso de especulação. As principais técnicas são: *Interleaved Multithreading* ou IMT (chaveamento de instruções de *threads* diferentes em cada novo ciclo), *Blocked Multithreading* ou BMT (chaveamento entre *threads* após a ocorrência de um evento de grande latência) e *Simultaneous Multithreading* ou SMT (execução simultânea de *threads* sem necessidade de chaveamento).

Além da abordagem *multithreading*, outro ponto crítico em CMTs é a quantidade e o tipo de núcleos de processamentos (homogêneos ou heterogêneos) internos ao *chip*. Neste caso, o tipo, a quantidade de núcleos e a abordagem *multithreading* devem ser definidos em conjunto, em função das cargas de trabalho típicas do ambiente onde o processador será aplicado. A relação entre múltiplos núcleos e abordagens *multithreading* resulta no objeto principal de estudo deste trabalho, o *Chip Multithreading*. Um *chip multithreading* é um processador com execução de múltiplas *threads* simultâneas. O CMT pode conter um único núcleo SMT ou múltiplos núcleos combinando ou não suporte a múltiplas *threads* com IMT, BMT ou SMT.

Duas novas tendências no projeto de CMTs *multi-core* são as *Networks-on-Chips* (NoCs) e a virtualização. A NoC é capaz de suportar a nova demanda de paralelismo de *threads* e troca de mensagens internas ao *chip* com baixa latência e alta vazão de dados. A virtualização "simula" a existência de processadores virtuais para o suporte a múltiplos sistemas operacionais.

Portanto, os principais problemas associados aos novos projetos das arquiteturas de CMTs são: Quais as mudanças nas abordagens *multithreading* e *multi-core*, levando em consideração o surgimento e adoção crescente das *networks-on-chips* e da virtualização? Qual a melhor combinação de técnicas e arquiteturas para que seja possível obter o maior desempenho do *chip* sem gargalos na comunicação interna?

O objetivo deste trabalho é apresentar uma análise e avaliação crítica dos processadores CMTs e as principais tendências futuras nas arquiteturas destes processadores levando em consideração os conceitos e problemas citados.

Palavras-Chave: *Chip Multithreading, Chip Multiprocessor, Explicit Multithreading, Network-on-Chip*, Virtualização.

Chip Multithreading: Concepts, Architectures and Trends

ABSTRACT

In the current designs of chip multithreading (CMT), one important consideration is the multithreading approach. The Explicit Multithreading classification has been used for many companies and the techniques are based on native parallelism exploration of instruction flows from different sources or different programs, without speculation. The main techniques are: Interleaved Multithreading (IMT), Blocked Multithreading (BMT), Simultaneous Multithreading (SMT).

Besides multithreading approach, another important consideration is the number and type of processing cores (homogenous or heterogeneous) intra chip. In this case, the type, the number of cores and the multithreading approach must be defined together and related to environment typical workloads where the processor will be used. The relation among multiple cores and multithreading approaches results in the main study object of this work, the chip multithreading. A chip multithreading is a processor with simultaneous execution of multiple threads. The CMT can have one SMT core or multiple cores without or with multiple threads IMT, BMT or SMT.

Two new trends in multi-core CMT designs are the Networks-on-Chips (NoCs) and the virtualization. The NoC is capable to support the new demand of thread parallelism and intra-chip message passing with low latency and high throughput. The virtualization "simulates" virtual processors to support multiple operating systems.

Therefore, the main problems related to the new CMT architecture designs are: What are the changing in multithreading and multi-core approaches, considering the appearing and adoption of networks-on-chips and virtualization? What are the better technique and architecture combinations to obtain the high performance of the chip without bottlenecks in internal communication?

The work objective is to present the critical analyses and evaluations of CMT processors and the main future trends in these processor architectures considering the concepts and problems cited before.

Keywords: Chip Multithreading, Chip Multiprocessor, Explicit Multithreading, Network-on-Chip, Virtualization.

1 INTRODUÇÃO

Com a crescente demanda por desempenho computacional, projetos de arquiteturas de processadores com suporte a múltiplas *threads* (SPRACKLEN, 2005) têm aumentado consideravelmente nos últimos anos. Uma *thread* pode ser definida como a execução de um fluxo de instruções de um mesmo programa. Um *chip* de processador pode suportar a execução de múltiplas *threads* simultaneamente ou não. A grande vantagem é que, em ambos os casos, este suporte permite que não se perca tempo com mudanças de contextos entre *threads* ativas.

O suporte a *threads* simultâneas, também conhecida como SMT (*Simultaneous Multithreading*) (UNGERER. 2002) (UNGERER 2003) (BOIVIE 2005) é uma das técnicas utilizadas para paralelizar a execução dos fluxos de instruções. Nesta técnica o processador possui suporte a execução simultânea, ou seja, execução paralela das múltiplas *threads*. As técnicas de *multithreading* foram propostas para que o processador pudesse suportar em *hardware* o paralelismo demandado e suportado pelos *softwares* e sistemas operacionais.

Em um processador tradicional com um único núcleo (*core*), a utilização da técnica SMT possibilita a visão lógica de mais de um núcleo pelo *software*. Isto significa que se o processador suportar dois fluxos de instruções simultâneas, para o software e para os sistemas operacionais, a execução será realizada em um processador com dois núcleos lógicos de processamento. Portanto, duas *threads* serão executadas simultaneamente.

No entanto, uma das formas para melhorar o desempenho em processadores está no aumento do número de núcleos físicos (*Chip Multi-Processor* ou *Chip Multi-Core*) (OLUKOTUN, 1996) (BARROSO, 2000) (KUMAR, 2004) (KUMAR, 2005-a) presentes em sua organização interna. O desenvolvimento de arquiteturas de processadores com vários núcleos de processamento interligados por redes de interconexão (DE ROSE, 2003) (KUMAR, 2005-b) é a alternativa atual de várias empresas. Projetos da Intel (GOCHMAN, 2006) (INTEL, 2006-b), IBM (BEHLING, 2006) e AMD (AMD, 2005) priorizam arquiteturas *multi-core*, através de seus primeiros processadores *dual-core* interconectados por barramentos e chaves *crossbar*. A Sun além da solução *dual-core* (TAKAYANAGI, 2005), já possui uma solução *multi-core* (KONGETIRA, 2005) com oito núcleos e uma chave *crossbar* para interconexão.

Com o aumento dos núcleos em um único *chip* de processador a aplicação das diversas técnicas de *multithreading* se torna ainda mais interessante. Neste caso, é possível usar técnicas para suportar *multithreading* sem paralelismo em um núcleo, mas com paralelismo em *chip* ou com paralelismo em núcleo e *chip*. Como são possíveis

chips multi-core com núcleos heterogêneos (KUMAR, 2004) (KUMAR, 2005-a) as combinações para suporte a *multithreading* são diversas e dependem da especificidade de cada núcleo, ou seja, para qual fim este núcleo foi projetado. Um *chip* com múltiplas *threads* em execução simultaneamente é um *Chip Multithreading* (CMT).

A tendência é que os projetos *multi-core* passem rapidamente da utilização de redes de interconexão para redes de comunicação interna como as *Networks-on-Chips* (NoCs) (BENINI, 2002) (BENINI, 2005) (ZEFERINO, 2004) para suportar melhor o paralelismo demandado pelas aplicações em execução nos diversos núcleos.

Outra tendência que já está sendo adotada pela Intel é o suporte a máquinas virtuais em *hardware*, chamado de tecnologia de virtualização (UHLIG, 2005) (NEIGER, 2006). Neste caso, um único processador "simula" a existência de outro processador para que seja possível suportar mais de um sistema operacional. Na tecnologia tradicional SMT o processador suporta mais de um fluxo de instruções para um mesmo sistema operacional.

Este trabalho procura abordar o assunto *chip multithreading* de maneira bem objetiva, mostrando de forma clara os conceitos, as técnicas de projeto e as tendências para o futuro. Os principais problemas, objetivos e motivações que demandaram o estudo sobre CMT estão descritos a seguir.

1.1 Problemas

É possível apontar como o principal problema em CMT a grande demanda de comunicação que poderá surgir com os novos projetos de processadores *multi-core*. Afinal, serão múltiplas *threads* em execução, simultaneamente, demandando comunicação *intra-chip* e acessos a dispositivos compartilhados. Uma questão em aberto é se haverá mudança ou não nas atuais técnicas de projeto e / ou arquitetura para suporte a *multithreading* com o surgimento no mercado da virtualização e das NoCs.

Portanto, outro problema está relacionado ao desempenho de processadores CMT com o uso de técnicas SMT em cada um dos núcleos. Núcleos SMT irão aumentar o desempenho dos processadores CMT?

Estes problemas ainda estão abertos e não possuem uma resposta objetiva, já que esta área de pesquisa aborda um assunto muito recente do estado da arte de arquitetura de processadores. No entanto, os conceitos, as técnicas e as análises presentes neste trabalho procuram responder ou oferecer uma base para estudos e pesquisas mais aprofundadas sobre o assunto.

1.2 Objetivos

O principal objetivo deste trabalho está no estudo do estado da arte e preparação de um material (tutorial) didático sobre *Chip Multithreading* abordando aspectos referentes às diversas soluções de arquiteturas de processadores com um ou vários núcleos sejam eles homogêneos ou heterogêneos.

Como objetivos adicionais pretende-se analisar e avaliar as soluções e técnicas atuais para que seja possível indicar uma tendência na aplicação do suporte a *multithreading* em tecnologias com virtualização e também em arquiteturas de *chips multi-core* com *networks-on-chips*.

1.3 Motivações

A motivação deste trabalho está relacionada ao surgimento da nova demanda de suporte a *multithreading* em projeto de processadores *multi-core* e suas implicações na adoção de soluções de redes de comunicações *intra-chip* e virtualização.

Como motivação adicional está a preparação de um material em língua portuguesa que possa contribuir como mais uma fonte de estudo complementando os principais documentos, relatórios e artigos sobre o assunto que em sua maior parte estão escritos em língua inglesa.

1.4 Estrutura do Trabalho

O trabalho está dividido em capítulos que procuram apresentar gradativamente os principais conceitos, técnicas de projeto de arquiteturas e tendências que possam servir como uma base de estudos e fonte de disseminação de pesquisas e *surveys* publicados nos importantes periódicos e conferências internacionais. Os capítulos são os seguintes:

2. Conceitos e Técnicas sobre *Multithreading*

   Neste capítulo são apresentados os conceitos sobre *Implicit* e *Explicit Multithreading*, abordando principalmente as técnicas *Interleaved Multithreading*, *Blocked Multithreading* e *Simultaneous Multithreading*.

3. Conceitos sobre Virtualização

   O suporte a máquinas virtuais é apresentado neste capítulo como uma introdução conceitual para que no capítulo 6 seja feito um relacionamento e uma avaliação com as possíveis tendências em CMT.

4. Principais Abordagens de Projeto dos Processadores Atuais

   Os conceitos de superescalaridade, *simultaneous multithreading* e *chip multitprocessor* são abordados de forma a apresentar uma visão inicial e introdutória da arquitetura interna dos processadores. Processadores com vários núcleos de processamento demandam um sistema de comunicação *intra-chip* eficiente de baixa latência e alta vazão de dados. Este capítulo apresenta também as principais arquiteturas dos sistemas de comunicação *intra-chip*.

5. Arquiteturas de Processadores CMT Comerciais

   Neste capítulo alguns processadores comerciais da Intel, IBM e Sun Microsystems têm suas arquiteturas internas apresentadas e relacionadas aos conceitos citados.

6. Avaliações e Tendências em CMT

Por fim, um capítulo específico para avaliar e apontar tendências em CMT com base em todos os conceitos e técnicas de projetos já apresentados.

Os dois últimos capítulos são respectivamente: conclusões e referências.

2 CONCEITOS E TÉCNICAS SOBRE *MULTITHREADING*

Nos projetos atuais de processadores um dos grandes objetivos é a extração ao máximo do desempenho. Uma das formas está na exploração do paralelismo seja na execução das instruções ou dos fluxos de instruções. Neste contexto, podemos considerar que um fluxo de instruções é uma *thread* e que uma *thread* é um processo, ou parte de um programa em execução. Se um processador suporta a execução de múltiplas *threads*, significa que este processador é capaz de executar fluxos de instruções diferentes. Neste caso, cada uma destas *threads*, ou fluxo de instruções, inicia em endereços diferentes de memória.

Alguns estudos e pesquisas exploram o paralelismo no nível de instrução através dos *pipelines* de processadores escalares e superescalares (HENNESSY, 2003) (PATTERSON, 2005). Neste tipo de paralelismo o objetivo é executar o maior número de instruções em paralelo (em um menor tempo) pertencentes a uma mesma *thread*. Nos *pipelines* escalares esta execução se faz através do processamento de uma instrução em estágios. Sendo assim, é possível ter várias instruções em estágios de processamento diferentes, mas cada estágio só possui uma única instrução. O processamento em estágios também é realizado em *pipelines* superescalares (DE ROSE, 2003). A grande diferença é que nesta abordagem é possível que várias instruções sejam executadas ao mesmo tempo, portanto, um único estágio de execução pode conter mais de uma instrução. A técnica de superescalaridade é muito importante e serve como base para processadores SMT (*Simultaneous Multithreading*). O capítulo 3 descreve com mais detalhes esta técnica exemplificando e objetivando a organização interna de processadores.

Esta seção apresenta uma abordagem específica sobre os conceitos de *multithreading*, ressaltando, portanto, o paralelismo em nível de *thread*. O suporte a *multithreading* possui duas abordagens (UNGERER, 2002) (UNGERER, 2003): *Implicit Multithreading* e *Explicit Multithreading*.

- *Implicit Multithreading*: Exploração do paralelismo existente em programas seqüenciais através de especulação no nível de *thread*. Nesta abordagem um processador gera múltiplas *threads* especulativas de um único programa seqüencial, dinamicamente, ou estaticamente com ajuda do compilador, e executa todas concorrentemente.

- *Explicit Multithreading*: Exploração do paralelismo existente entre programas de origens diferentes. As *threads* geradas a partir de cada um destes programas podem ser executadas em um mesmo *pipeline*.

Nos dois casos, cada uma das *threads* possui um banco de registradores e contadores de programa específicos, representando cada um dos múltiplos contextos em atividade no processador. A diferença está na abordagem de execução de *threads* especulativas de um mesmo programa seqüencial ou na execução de *threads* independentes e de programas distintos. As classificações e detalhes técnicos de cada uma das duas abordagens são descritos nas seções seguintes.

2.1 *Implicit Multithreading*

Conforme já mencionado, nesta abordagem a exploração de paralelismo é obtida através da especulação de *threads* (KRISHNAN, 1999) (ARDEVOL, 2004) de um mesmo programa seqüencial. Algumas técnicas são encontradas na literatura e todas se referem a trechos de regiões contíguas de programas que podem ser executadas especulativamente e concorrentemente pelo processador. As abordagens em *Implicit Multithreading* são (UNGERER, 2002): *Multiscalar*, *Trace Processor*, *Superthreaded, Single-Program Speculative Multithreading*, *Dynamic Multithreading*, *Speculative Multithreading*, *MEM-slicing Algorithm* e *Simultaneous Subordinate Microthreading*.

Multiscalar: Nesta arquitetura existem unidades de processamento interconectadas por um anel unidirecional. Cada uma destas unidades recebe uma tarefa a partir da divisão do programa seqüencial. Estas tarefas são subordinadas a um hardware seqüenciador que é responsável pelo controle, verificação, previsão e cancelamento de cada uma das tarefas. No entanto, cada unidade é capaz de buscar e executar as instruções da sua respectiva tarefa. Um processador *multiscalar* suporta especulação por controle (hardware seqüenciador) e por dependência de dados (carga de instruções não depende das instruções em execução).

Trace Processor: Um processador com múltiplos e distintos núcleos (unidades) de processamento onde há a quebra de um programa em traços. Os traços são coletados por uma *cache* especial de instruções que captura as seqüências. Um núcleo executa um traço enquanto outros núcleos executam futuros traços especulativamente.

Superthreaded: Uma técnica similar ao *multiscalar* que permite as *threads* com especulação de dados e controle executarem em paralelo. Como na arquitetura *multiscalar*, o compilador particiona estaticamente um programa em *threads* que são executadas em paralelo pela arquitetura.

Single-Program Speculative Multithreading: Suporta especulações de dependência de dados que não são conhecidas em tempo de compilação. Ao contrário de uma abordagem *superthreaded*, esta verifica as operações de armazenagem de dados. Se nesta verificação ocorrer uma dependência de dados, o processador espera pela armazenagem de dados da *thread* antecessora.

Dynamic Multithreading: Esta abordagem possui um *hardware* específico capaz de criar *threads* durante a análise de trechos de códigos (*loops*). A execução destas *threads* é especulativa e simultânea através de um *pipeline multithreaded*.

Speculative Multithreading: Também utiliza um hardware específico para particionar um programa seqüencial em múltiplas threads que executam sucessivas interações de um mesmo loop.

MEM-slicing Algorithm: Alguns estudos apontaram que instruções de memória são as melhores para iniciar ou quebrar (criação) uma *thread* especulativa. Este algoritmo propõe a geração de *threads* a partir de uma instrução (uma fatia) até um comprimento máximo para a *thread*.

Simultaneous Subordinate Microthreading: Nesta abordagem é feita uma modificação nas arquiteturas superescalares para executar *threads* no nível de microprogramas concorrentemente. Uma *microthread* subordinada poderia, por exemplo, melhorar a predição de desvio de uma *thread* primária ou antecipar a busca de dados.

Todas as abordagens *Implicit Multithreading* são para programas seqüenciais e, portanto, não são adequados em ambientes onde a cargas de trabalhos demandam *threads* de origens diversas. Exemplos são:

- Processadores GPP (*General-Purpose Processor*) em computadores pessoais que são submetidos a múltiplas e diferentes *threads* que podem não possuir relação.

- Processadores de servidores que executam *threads* criadas a partir das solicitações (requisições, transações) recebidas de origens completamente diferentes.

- Processadores de Rede (INTEL, 2001) que também executam *threads* demandadas por pacotes de redes completamente diferentes.

A abordagem *Explicit Multithtreading* que suporta o paralelismo de múltiplas *threads* de múltiplos programas, tem sido adotada nos atuais projetos de processadores e ao longo deste trabalho será dada uma ênfase a esta abordagem.

2.2 *Explicit Multithreading*

Explicit multithreading (UNGERER, 2003) é uma abordagem de execução (simultânea ou não) de vários e diferentes fluxos de instruções (*threads*). Quando a execução não é simultânea, é possível que o processador mude a execução do fluxo de instruções de uma *thread* para outra, fazendo com que os recursos possam ser utilizados de uma maneira otimizada. Neste caso, um processador com suporte a *explicit multithreading* permite a concorrência e execução de *threads* chaveando contextos. Este suporte é obtido através do uso de múltiplos contadores de programa (PCs - *Program Counters*) para cada *thread*, sendo que na maioria dos casos um banco de registradores é responsável pela armazenagem do contexto de cada *thread*. O tempo da troca de contexto é reduzido praticamente a zero, já que na mudança entre *threads*, os registradores de contexto e o contador de programa permanecem intactos, portanto, não há neste caso uma troca de dados entre memória e registradores para mudança do contexto. Estes conjuntos de registradores e PCs são replicados em função da

quantidade de *threads* que são suportadas (Ex. 8 *threads* = 8 PCs). O suporte a *explicit multithreading* pode ser classificado em função da Figura 1.

Várias *threads* podem estar ativas ao mesmo tempo, no entanto, em função do projeto é possível que haja apenas uma *thread* ativa em execução por ciclo. Neste caso é possível utilizar técnicas de projeto para suporte de *multithreading* de grão fino ou de grão grosso, conforme descrito a seguir:

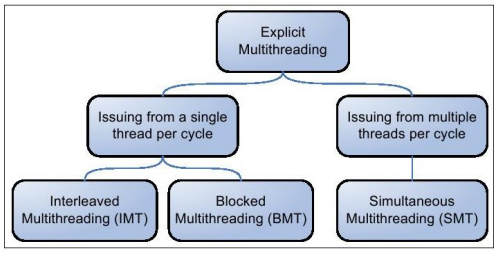

Figura 1. *Explicit Multithreading* (BOIVIE, 2005)

Interleaved multithreading (*fine-grained multithreading*) é o método em que a cada novo ciclo uma nova instrução de uma *thread* diferente é executada. O processador muda de contexto a cada novo ciclo. Neste caso é possível eliminar conflitos de dependências de dados e aumentar o ganho na execução do *pipeline*.

Blocked multithreading (*coarse-grained multithreading*) é o método em que a execução de uma *thread* só é interrompida quando um evento de alta latência ocorre, tal como um acesso à memória. Neste caso, a cada novo ciclo a execução permanece com a mesma *thread* havendo uma mudança de contexto somente quando um acesso à memória, por exemplo, é realizado. Sendo assim, é possível esperar o resultado do acesso á memória executando uma nova *thread*. Como a mudança de contexto pode ser estimulada por eventos, podemos dividir a técnica *Blocked Multithreading* conforme ilustrado pela Figura 3.

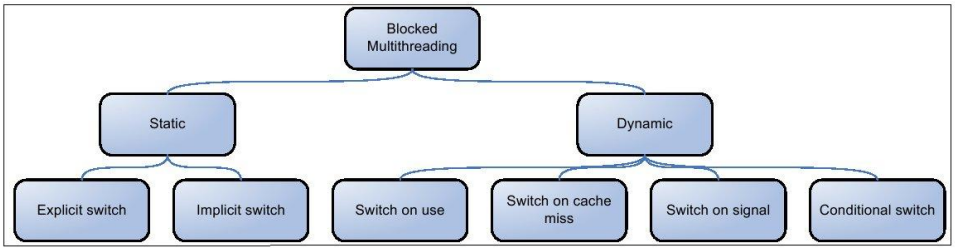

Figura 2. Classificações do *Blocked Multithreading* (BOIVIE, 2005)

- Modelo estático: Este modelo estabelece que uma instrução seja responsável por definir a mudança de contexto. No caso da mudança explícita, existe uma instrução específica que força a mudança de contexto. Ao contrário, no caso da mudança implícita, uma instrução executa um acesso de leitura ou escrita em memória ou um desvio e provoca a mudança de contexto.
- Modelo dinâmico: Neste modelo as mudanças de contexto são definidas através das ocorrências de eventos. Em *switch on cache miss* ocorre uma mudança de contexto quando uma instrução de acesso à memória tenta ler ou escrever ocasionando uma falta de página. Em *switch on signal* a mudança ocorre quando

da existência de uma interrupção ou um envio de sinal. Em *switch on use* uma instrução tenta usar um dado que ainda não foi buscado da memória, ocasionando uma mudança de contexto. Em *Conditional Switch* a mudança ocorre quando uma condição for totalmente verdadeira. Por exemplo, se for definido que a condição é aplicada a um grupo de instruções de *load/store*, ao encontrar um *cache miss* haverá uma mudança de contexto, caso contrário, quando houver um *cache hit* a *thread* permanece em execução.

Na técnica *Simultaneous Multithreading* (SMT) existe efetivamente múltiplas *threads* ativas em execução no mesmo ciclo, portanto, simultaneamente concorrendo por recursos. É uma técnica que aproveita dos recursos disponíveis em processadores superescalares onde ocorre a execução paralela de múltiplas instruções. Com a aplicação da técnica SMT, o processador deixa de apenas executar paralelamente as instruções para executar paralelamente os fluxos de instruções (execução paralela de *threads* no mesmo ciclo de clock). A técnica SMT de certa forma combina as técnicas IMT e BMT sendo mais eficaz do que as duas e proporcionando um melhor desempenho, já que deixamos de ter um *hardware* que suporta uma única *thread* por ciclo para múltiplas *threads* por ciclo.

Basicamente um *chip multithreading* deve suportar a execução de várias *threads* ao mesmo tempo. Portanto, um processador SMT é um processador CMT. No entanto, processadores com núcleos IMT ou BMT são também processadores CMT. Isto se deve à execução conjunta, uma por núcleo, de várias *threads* ao mesmo tempo.

A Figura 3 apresenta um dos núcleos de um processador CMT. Este núcleo suporta até 4 *threads* ativas por vez, mas não a execução de 4 *threads* ao mesmo tempo. Neste caso, há o chaveamento de contexto, demandado por um acesso à memória (evento de grande latência). A princípio esta poderia ser a representação de execução de *threads* em um núcleo IMT.

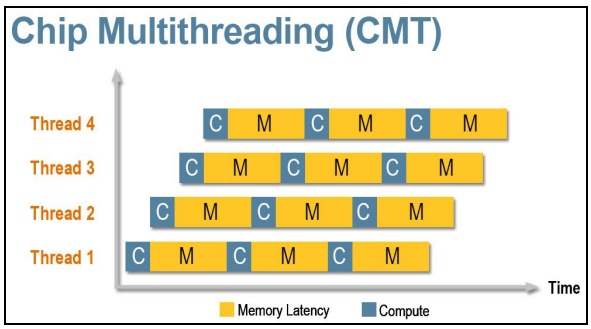

Figura 3. Núcleo de processamento de um *Chip Multithreading* (YEN, 2005)

Como o processador é CMT, ele possui outros núcleos que suportam as mesmas 4 *threads* ativas com chaveamento de execução. A Figura 4 ilustra as múltiplas *threads* em execução no mesmo ciclo, porém, em núcleos diferentes.

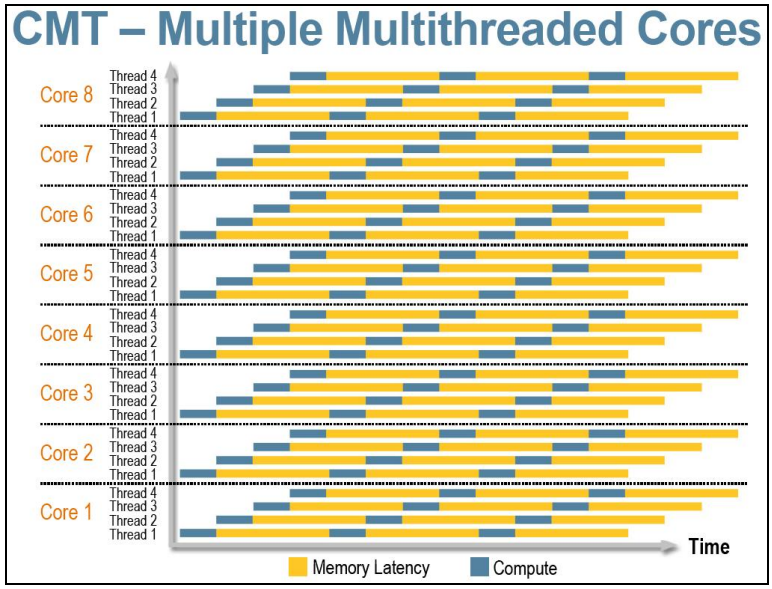

Figura 4. *Chip Multithreading* com oito núcleos (YEN, 2005)

As Figuras 3 e 4 foram retiradas da documentação do UltraSparc-T1, processador da Sun Microsystems, que será descrito na seção 5.3.2 com mais detalhes.

A Figura 5 apresenta uma visão da execução de *threads* em um processador SMT. Neste caso, são 4 *threads* sendo executadas simultaneamente em um processador com capacidade de despachar até 8 instruções no mesmo ciclo de *clock*. Esta capacidade de despacho é referente à arquitetura superescalar, que será descrita com mais detalhes na seção 4.1. A grande diferença do processador SMT para um processador similar ao apresentado nas Figuras 3 e 4, é que apenas um único núcleo suporta 4 *threads* no mesmo ciclo. No exemplo anterior são necessários 4 núcleos de processamento. Existem vantagens e desvantagens na aplicação de técnicas IMT, BMT e SMT para a construção de um *chip multithreading* (CMT). O capítulo 6 avalia as diferentes abordagens e aponta tendências na área de projetos CMT.

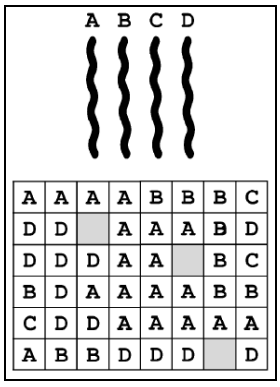

Figura 5. Processador SMT (UNGERER, 2002)

3 CONCEITOS SOBRE VIRTUALIZAÇÃO

Através dos estudos sobre *multithreading*, é possível entender as vantagens na execução paralela de vários fluxos de instruções. Sendo assim, é possível imaginar que o suporte a vários sistemas operacionais em execução simultaneamente também se deve ao suporte a *threads* simultâneas. No entanto, esta não é uma afirmação verdadeira e a execução paralela de vários sistemas operacionais se deve ao suporte de virtualização, que envolve outros conceitos importantes relacionados a *hardware* e *software*.

Antes de iniciar a apresentação dos conceitos de virtualização (UHLIG, 2005) (WHATELY, 2005) (NEIGER, 2006) se faz necessário definir e descrever as diferenças entre o suporte a multiprogramação (em processador *singlethreading*), *multithreading* e virtualização.

- Multiprogramação (*singlethreading*): No estudo de sistemas operacionais aprende-se que várias partes de programas estão em execução (processos) ao mesmo tempo e que o sistema operacional suporta a execução destes processos. No entanto, é sabido que em sistemas operacionais mais simples o que existe é o suporte ao pseudoparalelismo, demandado pelos múltiplos processos que concorrem pelos recursos de *hardware*. Nestes sistemas operacionais o que existe é a troca rápida de contexto entre os processos, com algoritmos de escalonamento eficientes que fazem parecer que tudo está executando em paralelo, quando na verdade não estão. Um sistema operacional em um processador sem suporte a múltiplas *threads* simultâneas (processos no nosso caso), resulta em uma execução pseudoparalela dos processos.

- *Multithreading*: No capítulo 2 foi descrito que um processador pode ser projetado para suportar múltiplas *threads* simultâneas (SMT) ou não (IMT ou BMT). Este suporte facilita a vida do sistema operacional que pode manter vários processos ativos ao mesmo tempo, sem perder tempo no chaveamento de contextos. Neste caso o processador não limita o sistema operacional e o paralelismo de execução de *threads* realmente acontece. Podemos ter pequenas diferenças, já que em processadores SMT a execução é efetivamente paralela, mas em processadores IMT ou BMT existe uma execução chaveada entre contextos ativos. Tanto em multiprogramação (*singlethreading*) quanto em *multithreading* existe o chaveamento, a

diferença é a seguinte: Em multiprogramação (*singlethreading*) somente um contexto está ativo por vez. Isto significa que em cada troca de contexto (escalonamento), perde-se muito tempo em substituição de dados em registradores e contador de programa. Em *Multithreading* vários contextos estão ativos ao mesmo tempo, mas podemos ter apenas um contexto em execução por vez (IMT ou BMT). No entanto, o chaveamento é realizado entre contextos ativos, não há troca de dados entre registradores e contador de programa, porque cada *thread* possui seu próprio banco de registradores e contador de programa. Portanto, não há perda de tempo entre os contextos ativos. O atraso referente à troca de contexto pode acontecer em processadores *multithreading*, mas neste caso todas as *threads* ativas estão bloqueadas e uma nova *thread* ganha o contador de programa e o banco de registradores em substituição a uma *thread* bloqueada.

- Virtualização: Neste caso o processador suporta a execução de mais de um sistema operacional ao mesmo tempo. Neste tipo de abordagem, são utilizados monitores de máquinas virtuais (MMVs) (WHATELY, 2005) que auxiliam o processador no gerenciamento dos sistemas operacionais que compartilham o mesmo hardware. A Figura 6 apresenta uma visão geral do sistema que será descrito com mais detalhes neste capítulo.

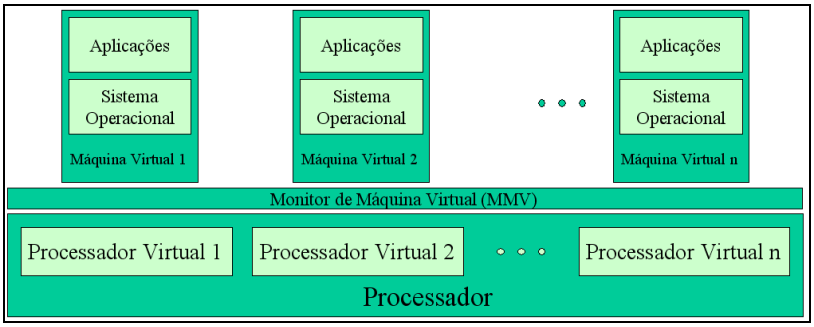

Figura 6. Visão Geral do Sistema de Virtualização

Um monitor de máquina virtual (MMV) permite a execução de múltiplos sistemas operacionais ao mesmo tempo, compartilhando e concorrendo por recursos de hardware. Um MMV também pode ser chamado de *Hypervisor* e nasceu na década de 70 com seu apogeu na então série de *mainframes* IBM 370 (WHATELY, 2005).

O ressurgimento dos MMVs tem coincidido com a evolução e rápido crescimento da Internet. A grande demanda por informação tem provocado o surgimento de grandes centros provedores de dados e conhecimento que devem ser capazes de receber e processar com eficiência e desempenho as solicitações geradas pelos usuários. Neste contexto, surgem grandes problemas da área de redes de computadores, sistemas distribuídos e segurança de dados que podem ser solucionados através de máquinas virtuais gerenciadas por monitores virtuais.

Um exemplo da aplicação de virtualização está em um centro de processamento de dados composto por servidores de aplicação, banco de dados, entre outros. O grande volume de requisições demanda um maior número de servidores para atender. O hardware pode ser um limitante, mas o sistema operacional, o servidor lógico enxergado pelos usuários, pode aumentar em cada uma das máquinas. Neste caso, é possível

instalar MMVs em cada um dos servidores aumentando o número de sistemas operacionais ou servidores lógicos disponíveis no centro de processamento de dados. Neste exemplo é possível enxergar rapidamente dois benefícios que podem ajudar a na solução dos problemas citados anteriormente. São eles:

- Balanceamento de carga: Distribui-se melhor a carga gerada pelos usuários entre os servidores lógicos.

- Segurança de dados: A alteração de um dado em um servidor lógico não é enxergada pelo outro servidor lógico gerenciado pelo mesmo MMV.

Uma máquina virtual pode ser classificada segundo duas abordagens ilustradas pela Figura 7:

Abordagem I (Clássico): O MMV está diretamente sobre o hardware. O MMV possui maior prioridade enquanto as máquinas virtuais (no nosso caso sistemas operacionais) executam em modo usuário sobre o MMV. O MMV deve emular as operações solicitadas pelo sistema convidado ao *hardware*.

Abordagem II (Hospedada): Neste caso existe um sistema operacional hospedeiro onde é feita a instalação da máquina virtual e dos sistemas operacionais convidados. Exemplos são: VirtualPC (Microsoft) e VMware GSX. Neste tipo de máquina virtual os acessos aos recursos de *hardware* são feitos através dos *drivers* do sistema operacional e não pelo MMV.

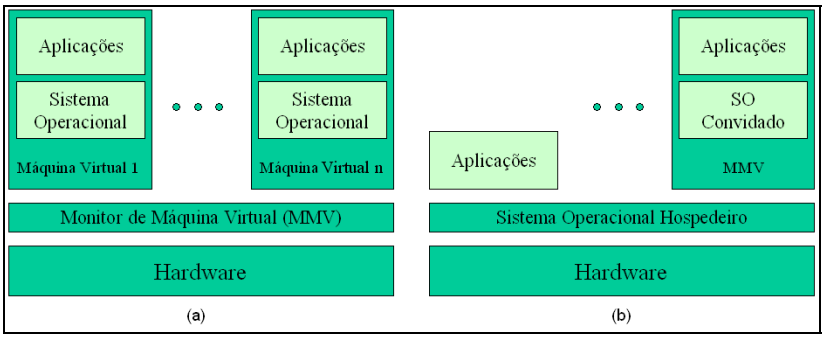

Figura 7. Abordagens de Máquinas Virtuais: (a) Clássico, (b) Hospedada (WHATELY, 2005)

O objetivo da Intel (UHLIG, 2005) (NEIGER, 2006) é levar para a arquitetura do processador a tecnologia de virtualização nativa, passando algumas tarefas do MMV para o próprio *hardware*. A técnica nativa basicamente é uma virtualização assistida por *hardware* que auxilia a virtualização gerenciada por MMVs. Neste caso, permanece o uso do MMV, mas com o auxílio do *hardware* que pode otimizar e aumentar o desempenho de todo o sistema. Basicamente são novas instruções (privilegiadas) que podem controlar a virtualização complementando e simplificando o uso dos monitores MMVs. Duas versões de MMVs que suportam as novas instruções dos processadores Intel são: VMware Workstation 5 e VMware Player.

O suporte em *hardware* é capaz de melhorar o desempenho através do uso de instruções privilegiadas não assistidas no modo virtual, e capaz de suportar as interrupções clássicas ou emuladas. Outra vantagem é poder eliminar a necessidade de

alteração de binários de máquinas virtuais (ex.: sistemas operacionais no modo clássico) gerenciadas por MMVs para que seja permitido o suporte à virtualização e execução direta.

No capítulo 6 é feita uma avaliação e uma relação da tecnologia de virtualização proposta pela Intel com os demais conceitos apresentados neste trabalho.

4 PRINCIPAIS ABORDAGENS DOS PROJETOS DE PROCESSADORES ATUAIS

Nesta seção são apresentadas três abordagens usadas na maioria dos processadores atuais em ordem histórica de surgimento. Basicamente, os principais conceitos a respeito de cada uma delas já foram descritos ao longo deste trabalho, mas nesta seção alguns detalhes do ponto de vista de blocos funcionais e construtivos da arquitetura são apresentados.

4.1 Superescalaridade

A técnica de superescalaridade foi desenvolvida a partir da idéia de um *pipeline* de instruções comum (escalar), no início da década de 90 (JOHNSON, 1991). A partir daí arquiteturas que implementam essa técnica surgiram rapidamente e hoje dominam o mercado dos processadores GPPs. A idéia geral é acelerar e aumentar o desempenho no paralelismo em nível de instrução, através da inclusão de várias unidades funcionais no estágio de execução do *pipeline*. Deste modo, o objetivo principal é identificar e utilizar uma unidade funcional ociosa para que sejam possíveis o processamento e execução de várias instruções ao mesmo tempo.

Diferente do escalar, que busca uma instrução por vez, o *pipeline* superescalar (Figura 8) busca da memória, a cada ciclo, um bloco completo com várias instruções. Essas instruções são então decodificadas em paralelo e enviadas aos estágios de execução (HENNESSY, 2003). As maiores diferenças com relação ao *pipeline* escalar, no entanto, podem ser observadas a partir desse ponto do *pipeline*. É a partir daqui que ocorrem mudanças nos estágios do *pipeline* para suportar a execução fora de ordem (*out-of-order*), típica desta técnica. Como um dos objetivos é o uso de várias unidades funcionais, evitando ao máximo a ociosidade, é comum executar instruções em uma ordem diferente daquela presente no bloco lido da memória. Sendo assim, há um estágio chamado de despacho, responsável pela identificação e tratamento das dependências de dados entre as instruções. Uma das técnicas utilizadas para tratar dependências de dados (falsas e de saída) é a renomeação de registradores. Ao fim das etapas deste estágio as instruções são despachadas para filas de delegação ou remessa. O estágio de delegação é responsável pela remessa das instruções para o estágio de execução propriamente dito. Para isto se faz necessário a verificação da disponibilidade de recursos que serão utilizados por cada instrução, o que inclui a verificação de operandos, unidades funcionais, interconexões, barramentos e portas para acesso ao *buffer* de reordenamento.

O estágio de execução possui várias unidades funcionais capazes de executar várias instruções ao mesmo tempo. Estas unidades recebem as instruções que estão em filas na unidade de delegação. Cada unidade funcional deste estágio é especializada para o tipo de operação que será realizada pela instrução. Por fim, o estágio de graduação ou *commit* é o responsável pela verificação da semântica seqüencial das instruções que foram executadas, além da escrita dos resultados. A diferença básica para uma estrutura com um *pipeline* simples, como mostrado anteriormente, é que na técnica superescalar várias instruções são executadas ao mesmo tempo, e não apenas uma.

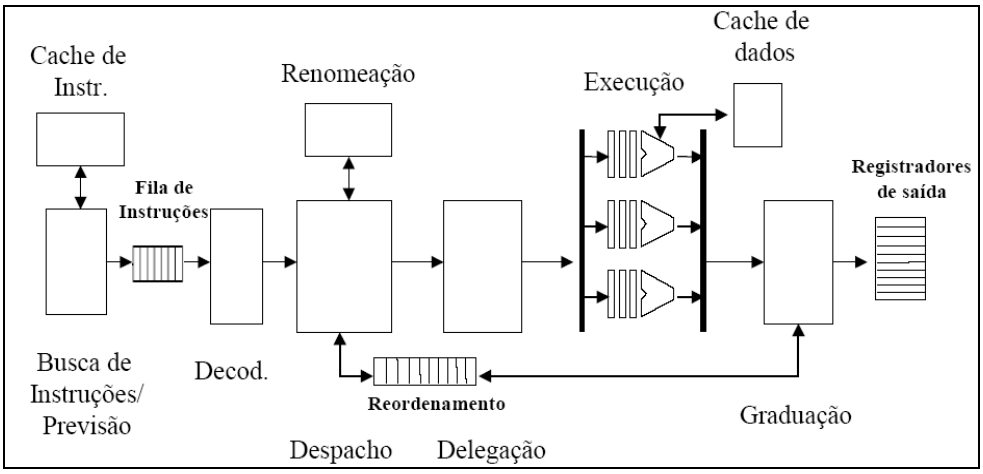

Figura 8. Visão Geral de uma Arquitetura Superescalar (DE ROSE, 2003)

Existem várias técnicas que são utilizadas em arquiteturas superescalares, tais como os seguintes exemplos: predição e predicação de desvios, especulação, reuso de instruções e de traços, além de outros. Referências importantes sobre assunto e que podem ajudar no entendimento de detalhes não descritos neste trabalho são: (MCFARLING, 1993), (YEH, 1994), (EVERS, 1996), (LIPASTI, 1996), (GABBAY, 1998), (SMITH, 1998), (CALDER, 1999), (SANTOS, 2003), (KIN, 2006), (PILLA 2006).

4.2 *Simultaneous Multithreading* (SMT)

Conforme já apresentado na seção 2.1 um processador SMT (EGGERS, 1997) tem a capacidade de executar em um mesmo ciclo de *clock* mais de uma *thread*. Porém, em um *pipeline* escalar cada estágio comporta apenas uma única instrução resultando em apenas uma única instrução executada ao final de cada ciclo. Mas em uma arquitetura superescalar (seção 4.1) existe o suporte a execução de mais de uma instrução por ciclo. Aproveitando o suporte da arquitetura superescalar são feitos os projetos de processadores SMT (TULLSEN, 1996) (GONÇALVES, 2002) (ACOSTA, 2005) (DAL PIZZOL, 2005).

Em uma arquitetura superescalar (Figura 8) espera-se que várias instruções sejam executadas ao mesmo tempo, mas para isto é necessário buscar várias instruções, um bloco de instruções. Dependendo das unidades funcionais livres nos estágios do *pipeline*, é possível, inclusive, executar as instruções fora de ordem, mas sempre verificando as consistências dos resultados encontrados para que dependências de dados, por exemplo, não causem erros de execução / resultado.

Aproveitando as múltiplas unidades funcionais do *pipeline* de instruções, é possível buscar então um conjunto de instruções, mas não de uma mesma seqüência ou de uma mesma *thread*, mas de *threads* ou seqüências (trechos de programas) diferentes. Através de múltiplos contadores de programa (PCs) buscam-se instruções de trechos de programas que iniciam em endereços de memória completamente diferentes.

A Figura 9 apresenta uma arquitetura de processador SMT (UNGERER, 2002) onde é possível verificar que a unidade de busca de instruções recebe os endereços de memória de vários contadores de programa. Neste caso é possível executar várias *threads* ao mesmo tempo no final de cada ciclo, desde que não haja algum conflito ou dependência que cause um atraso na execução. Nesta arquitetura da Figura 9 o processador suporta instruções inteiras e de ponto flutuante, possuindo dois *pipelines* separados para a execução destes dois tipos de instruções. As unidades funcionais de uma arquitetura superescalar podem ser visualizadas nesta figura através da unidade de renomeação de registradores, das filas de instruções (já que a vazão não é ideal em função de conflitos e dependências) e das múltiplas unidades de execução representadas por *floating-point units* e *integer load/store units*.

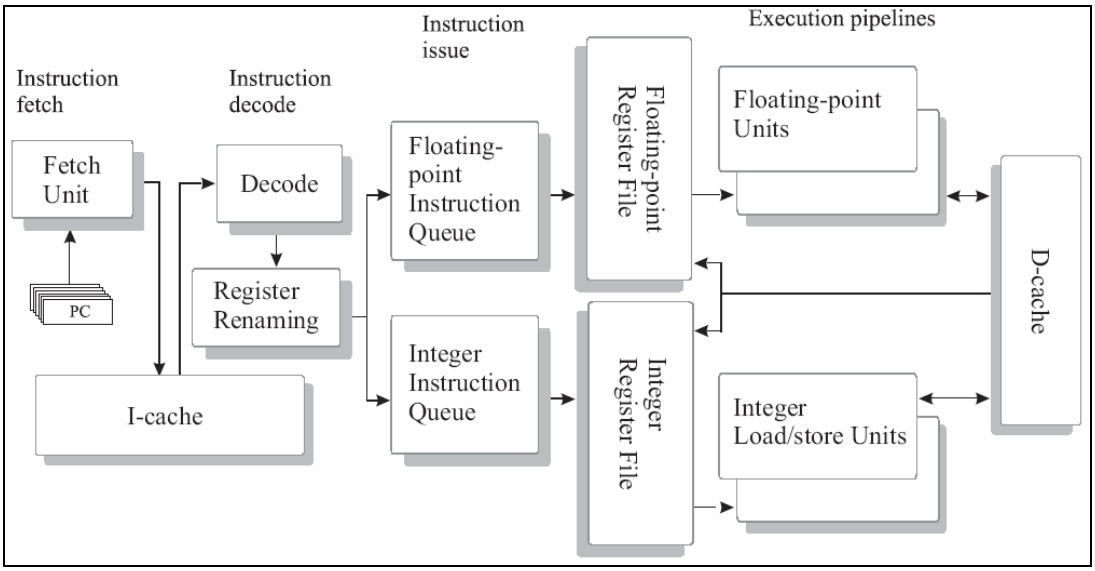

Figura 9. Visão Geral da Arquitetura de um Processador SMT (UNGERER, 2002)

A Figura 10 (EGGERS, 1997) apresenta a execução de 5 *threads* em três tipos de arquiteturas diferentes. No primeiro caso uma arquitetura superescalar, no segundo caso uma arquitetura superescalar com suporte a múltiplas *threads* de grão fino (IMT) e no terceiro caso um processador SMT.

Nesta representação cada linha possui as unidades funcionais do *pipeline* que são utilizadas em apenas um único ciclo. Os espaços (*slots*) em branco são unidades funcionais do *pipeline* que não estão sendo utilizadas por nenhuma das *threads*. Os *slots* brancos podem ocorrer horizontalmente ou verticalmente devido a problemas diferentes:

- Horizontalmente: Neste caso existe um baixo paralelismo de instruções. Por mais que existam quatro instruções disponíveis, uma dependência de dados,

por exemplo, impossibilita a execução e o uso das outras unidades funcionais do *pipeline*.

- Verticalmente: Um evento de grande latência pode obrigar que nenhuma instrução seja executada em determinado ciclo. Um exemplo seria o acesso à memória ou outro periférico.

A execução das *threads* na Figura 10 ocorre da seguinte forma:

- Superescalar: Em um processador superescalar somente uma *thread* é executada por vez. Na Figura 10.a somente a *thread* 1 está sendo executada, mas com a execução de mais de uma instrução da mesma *thread* no mesmo ciclo (mesma linha) em quatro dos nove ciclos apresentados pela figura.

- *Multithreading* IMT: Neste caso é possível executar mais de uma *thread*, mas com chaveamento de contexto. Em todos os nove ciclos as unidades funcionais do *pipeline* são utilizadas. Porém em cada ciclo somente uma *thread* tem acesso a estas unidades. No ciclo seguinte é feito o chaveamento e novas instruções de uma outra *thread* são executadas. No entanto, em um mesmo ciclo mais de uma instrução de uma mesma *thread* pode estar em execução.

- *Simultaneous Multithreading*: Nesta arquitetura é possível ter no máximo quatro *threads* executando simultaneamente. Isto significa que instruções de *threads* diferentes, portanto, de trechos de programas diferentes, estão em execução no mesmo ciclo compartilhando o mesmo pipeline. O segundo ciclo (de cima para baixo) apresenta três *threads* executando simultaneamente e um *slot* vago. Caso todos os *slots* estivessem preenchidos, seria possível que o quarto *slot* fosse de uma nova *thread* ou de uma segunda instrução de uma das três *threads* que já estão em execução.

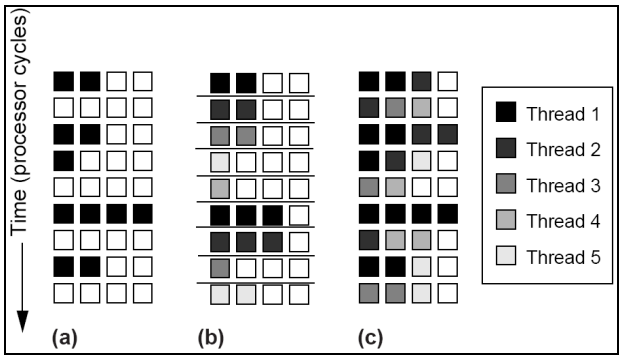

Figura 10. Execução de *Threads*: (a) Processador Superescalar, (b) Processador Superescalar com Suporte a Múltiplas *Threads* de Grão Fino, (c) Processador SMT (EGGERS, 1997)

Comparando o desempenho dos três modelos é possível verificar o ganho do processador SMT em relação aos demais. Enquanto o processador superescalar processou apenas uma única *thread* (1), o processador IMT iniciou o processamento de outras *threads*, mas com a penalidade de não terminar a primeira no mesmo ciclo do superescalar. No entanto, o processador SMT inicia o processamento de várias *threads*

no mesmo ciclo e termina a primeira *thread* no mesmo ciclo do processador superescalar. Fazendo uma projeção futura, o processador SMT terminaria todas as *threads* antes dos outros dois processadores.

A grande vantagem do processador SMT se deve ao fato de haver um melhor aproveitamento das unidades funcionais do *pipeline* pelas *threads*. Em ciclos onde haveria dependência de dados e, portanto, o não paralelismo de instruções de uma mesma *thread*, uma nova instrução de uma outra *thread* é colocada em execução evitando ao máximo a ociosidade do *pipeline*, ocorrendo em uma menor quantidade de *slots* vazios.

4.3 *Chip Multiprocessor* (CMP)

Uma das alternativas para aumentar o desempenho dos processadores atuais é o acréscimo de mais de um núcleo de processamento na arquitetura interna do processador. Na verdade, um único *chip* passa a ter vários processadores conforme ilustrado pela Figura 11.

A grande maioria dos processadores de propósito geral são exemplos de arquiteturas com núcleos homogêneos (iguais) e para um mesmo propósito de funcionamento (aplicações gerais no caso do GPP). No entanto, projetos de processadores *multi-core* para aplicações em sistemas embarcados, freqüentemente possuem núcleos heterogêneos (KUMAR, 2004) (KUMAR, 2005-a). Neste caso, cada núcleo, ou conjunto de núcleos, é responsável por processamentos específicos e distintos dos demais. Uma classe de processadores que representa adequadamente as arquiteturas com núcleos heterogêneos são os processadores de rede (INTEL, 2001) (COMER, 03) (FREITAS, 2003). Nesta classe, podemos encontrar processadores com núcleos responsáveis pelo processamento de pacotes, outros por gerenciamento de tabelas de roteamento, ou até por camadas específicas de rede e qualidade de serviço. Processadores como estes também são conhecidos como MPSoCs (*Multiprocessor System-on-Chip*) (WOLF, 2004) já que parte da arquitetura é composta por um sistema de periféricos internos ao próprio *chip*.

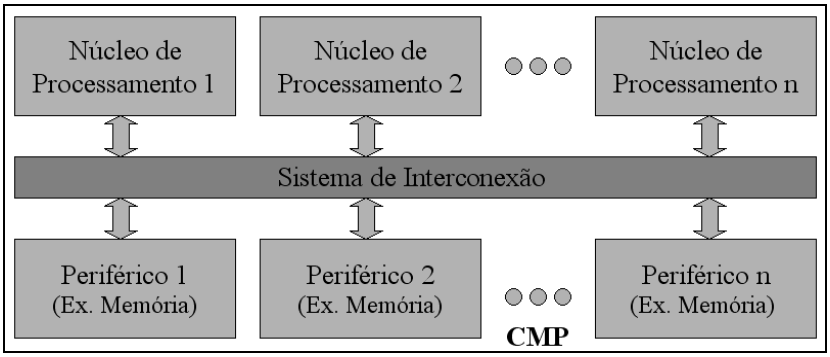

Figura 11. Visão Geral de um *Chip Multiprocessor*

Em um processador *quad core* (quatro núcleos) onde cada núcleo possui um *pipeline* escalar e sem suporte a múltiplas *threads*, podemos ter quatro *threads* simultâneas, uma para cada núcleo. Apesar da simplicidade do núcleo, este tipo de processador é considerado um *chip multithreading*, já que o *chip* suporta mais de uma *thread* em execução no mesmo ciclo.

A Figura 12 apresenta uma comparação entre um processador SMT superescalar de oito vias e um processador *multi-core* onde cada núcleo é um superescalar de duas vias. Ambos os processadores são CMTs que suportam até quatro *threads* simultâneas. No caso do processador SMT instruções de cada uma das *threads* podem ser executadas ao mesmo tempo, já no CMP cada núcleo recebe uma única *thread*, mas duas instruções de cada *thread* podem ser executadas ao mesmo tempo. A diferença é que no caso do SMT é possível ter até oito instruções de uma única *thread* em execução ao mesmo tempo, em detrimento da execução de outras *threads*. Este prejuízo não ocorre no CMP onde cada *thread* tem seu núcleo específico, mas com largura de no máximo duas instruções.

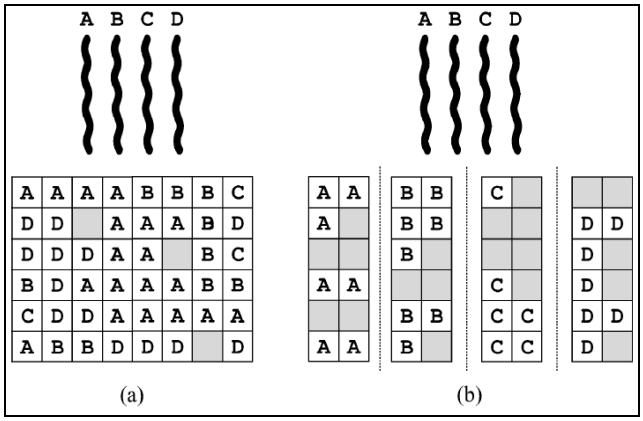

Figura 12. (a) SMT x (b) Múltiplos Núcleos de Processamento Interno (UNGERER, 2002)

Os processadores com arquiteturas *multi-core* estão se tornando a grande aposta para o aumento do desempenho em sistemas computacionais. Sendo assim, diversas pesquisas acadêmicas ou industriais, têm investido nas novas gerações de processadores com múltiplos núcleos. Porém existem duas alternativas para organização interna dos núcleos do *chip*, conforme a seguir:

- Segmentos de processadores onde os núcleos suportam múltiplas *threads* simultâneas (SMT) (KALLA, 2004) (INTEL, 2006-b). Cada núcleo extrai o paralelismo do fluxo de instruções (*threads*) através da arquitetura superescalar em que cada núcleo está baseado.

- Núcleos básicos, sem complexidade na organização interna que dá suporte ao paralelismo (GEPPERT, 2005) (SUN, 2006). Isto significa que, os núcleos não possuem superescalaridade, mas podem suportar as múltiplas *threads* e contextos, porém, de grão fino (IMT), por exemplo.

Decisões de projeto para definição de qual alternativa adotar precisam ter como alvo o conhecimento do ambiente de aplicação do processador e principalmente a carga de trabalho típica que será submetida ao processador. De posse destas informações é possível projetar melhor um processador que se adapte melhor ao tipo de dado / informação que deve ser processada resultando em um melhor desempenho ou menor tempo de processamento. Pesquisas sobre as melhores alternativas de projeto de arquiteturas de processadores têm usado basicamente o estudo de cargas de trabalho para entender melhor o comportamento do processador.

Em Olukotun (1996) foi feito um estudo onde dois tipos de arquiteturas foram expostos a um mesmo tipo de carga de trabalho. Basicamente o estudo procurou definir qual o melhor tipo de arquitetura para cargas onde havia um baixo ou grande paralelismo no nível de *thread*. A Figura 13 apresenta os dois tipos de arquiteturas que foram comparadas.

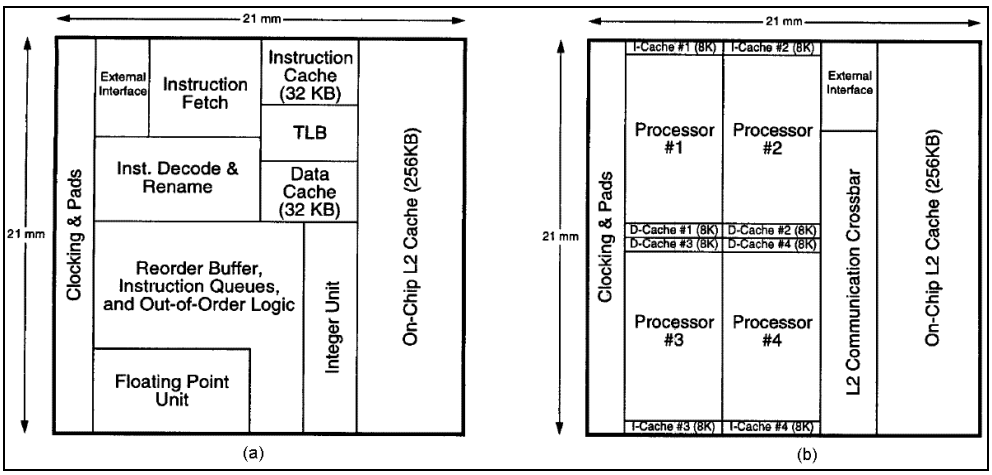

Figura 13. (a) Superescalar x (b) CMP (OLUKOTUN, 1996)

- Arquitetura superescalar: Extensão do processador MIPS R10000 com execução de seis instruções simultâneas.

- Arquitetura CMP: Execução de duas instruções simultâneas por núcleo de processamento (baseado também no MIPS R10000).

Para garantir apenas a influência da carga de trabalho submetida, os dois projetos possuem as mesmas latências de acessos à memória, principalmente o tempo de *cache hit* e a mesma ocupação de área. As mesmas cargas de trabalho foram aplicadas aos dois projetos com as seguintes características: operações de números inteiros, ponto flutuante, e de multiprogramação.

Os resultados demonstraram que para cargas de trabalho onde as aplicações não são paralelizáveis, o ganho é favorável ao processador superescalar em 30%. Neste caso existe uma exploração melhor do paralelismo no nível de instrução. Para aplicações onde existe um baixo paralelismo de *threads*, o ganho ainda é favorável à arquitetura superescalar, mas no máximo de 10%. No entanto, onde há um grande paralelismo no nível de *thread*, o ganho passa a ser da arquitetura CMP variando de 50% a 100% em relação ao superescalar.

As cargas de trabalho com grandes níveis de paralelismo em *thread* executam aplicações independentes com processos independentes. Aplicações de visualização e multimídia, processamento de transações e aplicações científicas de ponto flutuante são exemplos destas cargas de trabalho. Esta pesquisa serviu como base para o processador Hydra CMP (HAMMOND, 2000), que também gerou resultados para o futuro processador UltraSparc-T1 (KONGETIRA, 2005) descrito no capítulo 5.

Para estabelecer a comunicação entre os múltiplos núcleos um sistema de interconexão *intra-chip* deve ser projetado. A maior parte dos processadores utilizam

barramentos ou chaves *crossbar*. No exemplo da Figura 13.b uma chave *crossbar* é utilizada para comunicação entre os núcleos e a *cache* L2. Outro processador com um sistema de interconexão interessante é o Piranha (BARROSO, 2000).

A arquitetura do Piranha é ilustrada pela Figura 14. São núcleos baseados na arquitetura do processador Alpha conectados diretamente às suas respectivas *caches* de dados e instruções. Este CMP possui o seguinte sistema de comunicação *intra-chip*:

- Uma chave *crossbar* (*Intra-Chip Switch*) para interconectar o primeiro nível de *cache* a outros módulos do *chip* (ex. *cache* L2).

- Sistema baseado em um roteador *cut-through*, *buffers* de entrada e saída, e um *packet switch*. Este sistema é responsável pela interface com *chips* externos onde existe a troca de pacotes de dados.

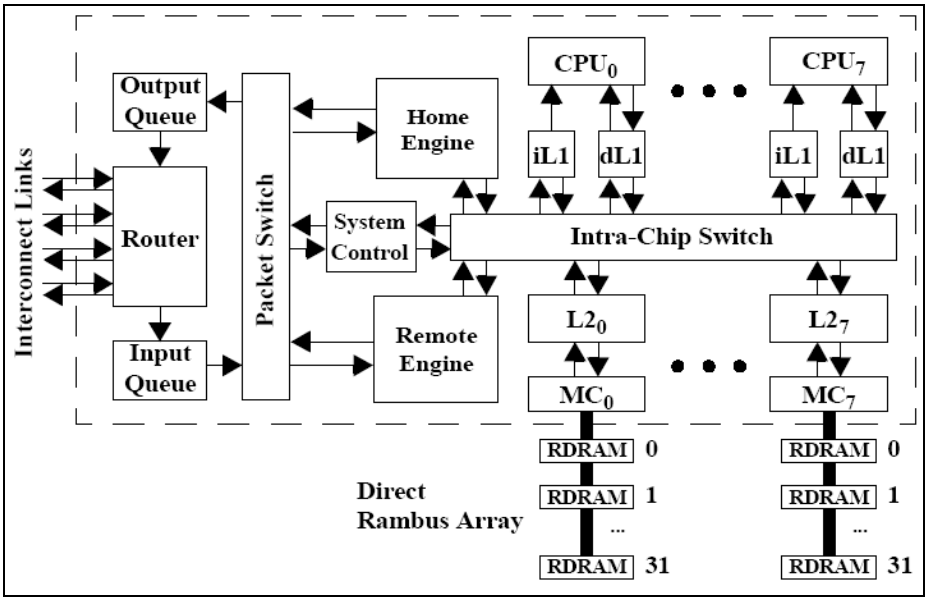

Figura 14. Arquitetura do Processador Piranha (BARROSO, 2000)

Os sistemas de comunicação *intra-chip* têm ganho muita importância nos novos projetos de CMP devido a grande tendência do aumento do número de núcleos. A seção 4.3.1 aborda os principais sistemas com uma ênfase em *Network-on-Chip* (NoC) que têm surgido como uma solução de alto desempenho para chips *multi-core* que devem suportar uma alta demanda de aplicações nativamente paralelas.

O sistema de comunicação adotado no projeto do processador Piranha é parecido como uma NoC (roteadores) dividida em vários processadores Piranha. Talvez este seja um ponto interessante da história para relacionar uma necessidade antiga com a demanda atual de comunicação dos diversos núcleos de processadores através de uma *Network-on-Chip*, conforme será descrito na seção seguinte.

4.3.1 Sistemas de Comunicação *Intra-Chip*

As redes de interconexão surgiram primeiramente para interligar os vários processadores de um multicomputador. Estas mesmas redes estão presentes atualmente

nas arquiteturas multiprocessadas e são responsáveis pelo sistema de comunicação *intra* e *extra chip*.

Entre as redes mais utilizadas e difundidas estão o barramento e a chave *crossbar* devido à boa relação custo / desempenho. As principais características das redes de interconexão podem ser apresentadas através de duas classificações ilustradas pelas Figuras 15 e 16.

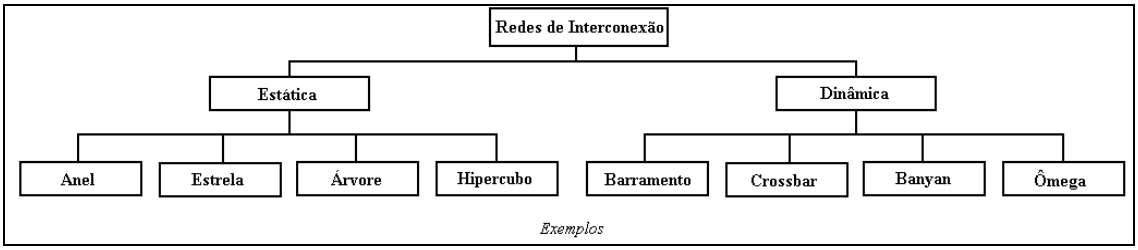

Figura 15. Classificação das Redes de Interconexão (DE ROSE, 2003)

- Rede Estática: Nesta classe as redes possuem interconexões fixas que não mudam ao longo do tempo.

- Rede Dinâmica: Ao longo do tempo as conexões podem mudar de acordo com as necessidades de comunicação. Barramento e chave *crossbar* são redes dinâmicas.

A Figura 16 apresenta uma classificação segundo o tipo de chaveamento de dados (ANDERSON, 1975) (AHMADI, 1989), que pode ser tanto temporal como espacial.

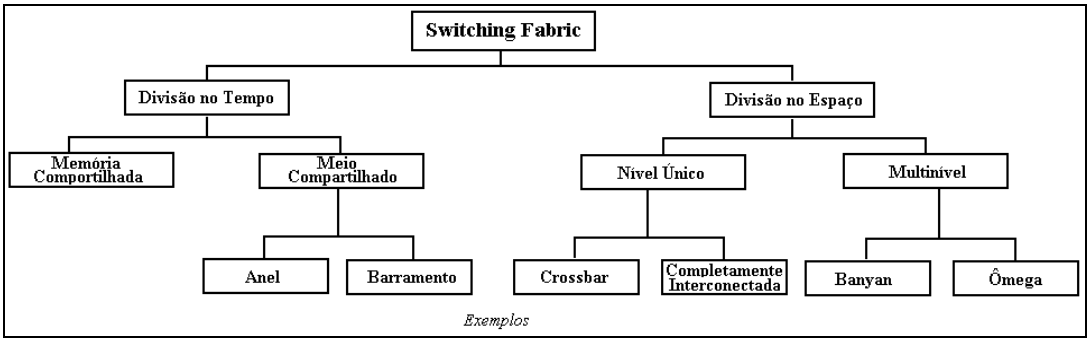

Figura 16. Classificação das Unidades de Chaveamento (*Switching Fabrics*)

- Divisão no Tempo: É utilizado multiplexação no tempo para alcançar o chaveamento de dados. Pode ser feito por memória compartilhada com a obrigatoriedade de uma vazão equivalente ao roteamento de dados. O outro tipo é por meio compartilhado, onde o barramento é um exemplo. O principal problema do barramento é a largura de banda, já que somente uma comunicação pode estar ativa por vez. No entanto, é uma das melhores opções para *broadcast* e *multicast*, já que todos os nós estão conectados em um mesmo ponto.

- Divisão no Espaço: Os caminhos já estão definidos espacialmente. Há uma classificação para o tipo de rede em nível único e multinível. A chave *crossbar* é uma rede de nível único, ou seja, só existe uma unidade de

chaveamento sem a utilização de cascatas destas unidades. A chave *crossbar* é similar a uma matriz de conexões onde cada célula é um ponto de chaveamento entre entradas e saídas. Não existe um compartilhamento de conexões, ou seja, uma mesma entrada não se conecta a múltiplas saídas e vice-e-versa ao mesmo tempo. Em multinível é feito um cascateamento de unidades de chaveamento para evitar ocorrências de conflitos.

A grande vantagem da divisão no espaço refere-se à inexistência do tempo de multiplexação da divisão temporal. No entanto, o custo espacial é maior em função dos pontos de conexão, podendo haver também funções bloqueantes.

A Figura 17 apresenta dois exemplos de aplicações para chave *crossbar* e barramento em um *chip* com quatro núcleos de processamento.

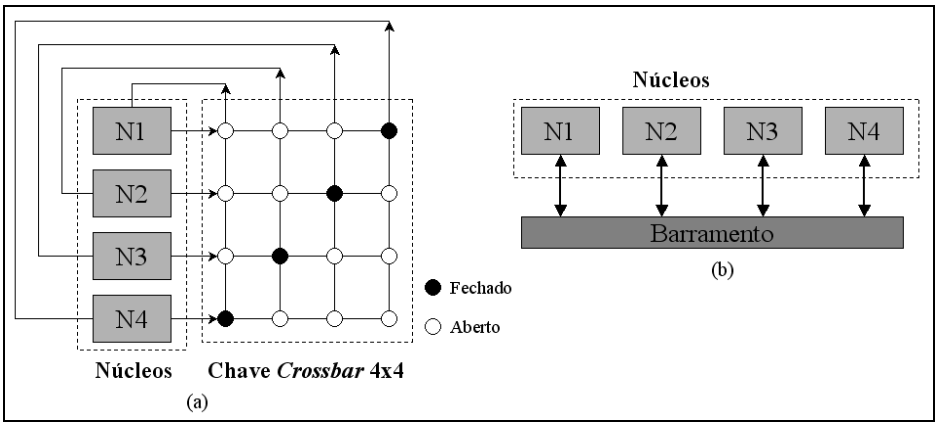

Figura 17.Exemplos de aplicação: (a) Chave *Crossbar*, (b) Barramento

4.3.1.1 Networks-on-Chip (NoCs)

Com o surgimento da abordagem *multi-core* e da necessidade crescente de suportar o paralelismo interno, constatou-se a necessidade de aumentar a taxa de transmissão e recebimento de dados internos. Sendo assim, o conceito *Network-on-Chip* (NoC) (BENINI, 2002) (BENINI, 2005) (ZEFERINO, 2004) surgiu para que fosse agregado todo um sistema de rede de comunicação de dados eficiente para garantir a diminuição da latência de transmissão, o aumento da vazão dados e por conseqüência do processamento.

Como em qualquer rede de comunicação de dados, existem problemas associados ao tempo de transmissão, latência de rede, largura de banda, e vazão dos pacotes. Os conceitos relacionados às redes normalmente são aplicados em ambientes *multi-core*, onde a troca de mensagem se faz necessária a maior parte do tempo. Sendo assim, o fato da rede estar presente internamente ao *chip*, faz com que todos os problemas também sejam repassados para esta abordagem e todos os parâmetros de projeto de uma rede devem ser levados em consideração.

Entre as características principais na arquitetura de uma NoC podemos citar: o *link*, a arquitetura de chaveamento (ou roteadores) e a interface de rede.

- Espera-se que o *link* da rede de interconexão possua uma baixa latência e uma boa largura de banda. Estas duas características são fundamentais para reduzir o tempo médio de transmissão de pacotes.

- Em relação à arquitetura de chaveamento (ou roteadores) espera-se que seja possível a transferência de pacotes entre as portas de entrada e saída com um alto desempenho. Um projeto pode incluir chaves *crossbar* e *buffers* para as portas de entrada e saídas. Neste caso, torna-se possível o uso do algoritmo *store-and-forward*, onde os pacotes que ficam armazenados nos *buffers* são analisados e depois transferidos. O algoritmo *cut-and-through* também pode ser usado, mas neste caso é necessário o uso dos *buffers* apenas para terminar de receber o pacote, porque logo após o último bit recebido o pacote já é direcionado para uma das saídas. É importante ressaltar que, o tempo médio de espera em fila dos pacotes deve permanecer baixo para que o tempo total residente na unidade de chaveamento e por conseqüência na rede de interconexão seja reduzido.

- A interface de rede deve prover acesso padronizado às diversas unidades de processamento, além de outras unidades de chaveamento de pacotes ou redes de interconexão de dados.

A Figura 18 apresenta a ilustração de uma NoC. Nesta figura o roteador é representado pelos blocos menores e cada núcleo de processamento é representado pelos *Processing Units*.

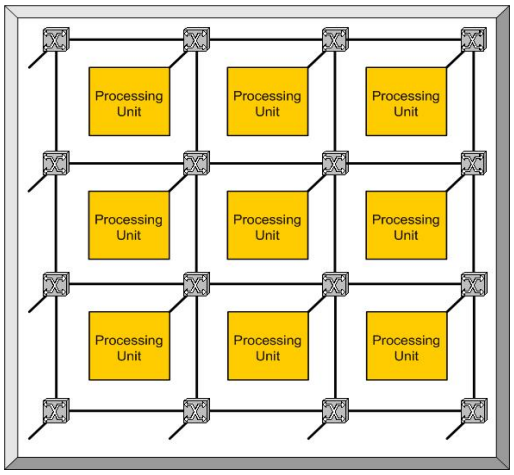

Figura 18. Exemplo de uma *Network-on-Chip* (MAEHLE, 2006)

A rede de interconexão desta NoC é uma *mesh*, sendo que cada nó é representado por um roteador conectado diretamente a um núcleo por um *link* dedicado. Cada roteador está conectado a mais quatro outros roteadores encarregados de estabelecer a melhor rota de comunicação entre cada um dos núcleos de processamento.

No capítulo 6 são feitas avaliações e projeções de tendências em processadores CMT com *networks-on-chip*.

5 ARQUITETURAS DE PROCESSADORES CMT COMERCIAIS

Neste capítulo alguns dos principais processadores CMT comerciais são apresentados. As arquiteturas ilustram a aplicação dos conceitos de superescalaridade, *explicit multithreading* e *chip multiprocessor*. Apesar das diferenças nos projetos, todos os *chips* de processadores suportam múltiplas *threads* simultâneas.

5.1 Processadores Intel

Os processadores da Intel que serão abordados fazem parte de duas classes diferentes: Processadores Heterogêneos e Processadores Homogêneos.

5.1.1 Processador de Rede IXP1200

O Processador de Rede Intel IXP1200 (INTEL, 2001) (Figura 19) possui uma arquitetura multi-processada heterogênea composta de um processador (core) StrongARM e seis *Microengines*. Todos os processadores são baseados no modelo RISC (*Reduced instruction Set Computing*) sendo que as seis *microengines* possuem microarquiteturas idênticas, mas não necessariamente para um mesmo tipo de processamento, havendo uma flexibilidade de operação. O objetivo principal desta arquitetura está associado ao aumento de desempenho através da aplicação de conceitos de *multithreading* apresentado no capítulo 2. Este suporte se dá através das *Microengines* que compõem a arquitetura principal. O StrongARM é um processador antigo, estando a novidade no projeto das *Microengines* e na organização interna do IXP1200 para prover a comunicação entre os diversos núcleos. O processamento nesta arquitetura foi dividido em duas partes, sendo cada parte associada a um tipo de processador:

- StrongARM: Gerenciamento principal, associada à manutenção de tabelas de roteamento e tarefas mais complexas e trabalhosas.

- *Microengine*: Responsáveis pelas tarefas mais simples e relacionadas ao roteamento e processamento de pacotes.

A rede de interconexão (Figuras 19 e 20) utilizada por este processador é o barramento. São barramentos paralelos e independentes para transferência de dados entre o processador StrongARM e as *Microengines*.

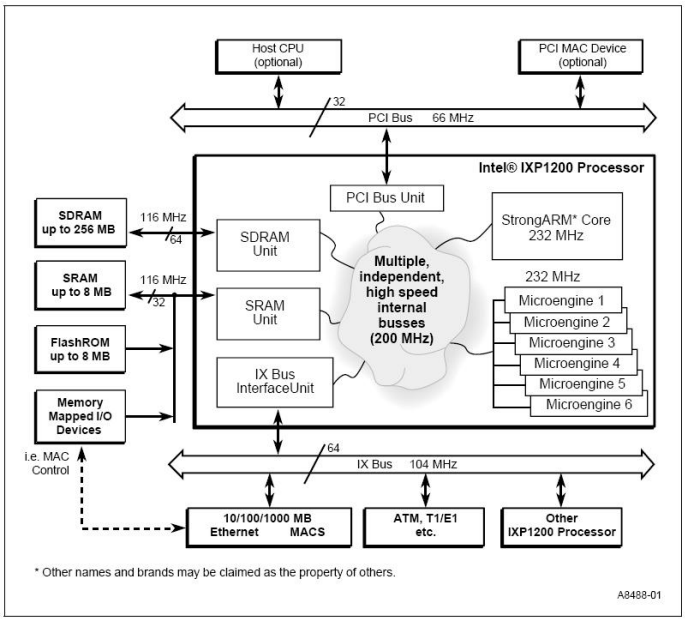

Figura 19. Arquitetura do IXP1200 (INTEL, 2001)

Através da arquitetura da *Microengine* (Figura 20) é possível notar a presença de quatro contadores de programa, ou seja, quatro contextos diferentes. Cada um dos bancos de registradores é dividido em quatro partes dedicadas para cada uma das *threads*. É importante ressaltar que cada *Microengine* consegue executar somente uma única *thread* por ciclo. A troca entre *threads* ocorre, por exemplo, quando há acesso à memória. O tempo é relativamente grande e neste caso é realizada a troca de contexto. De acordo com os conceitos apresentados na seção 2.2 enquadramos esta arquitetura em *Blocked Multithreading* (BMT), já que a troca não é feita no nível de instrução (IMT) e sim de *thread*. Um comando implícito (*Reference*) e uma instrução explícita (ctx_arb) são utilizados, conforme modelo estático BMT.

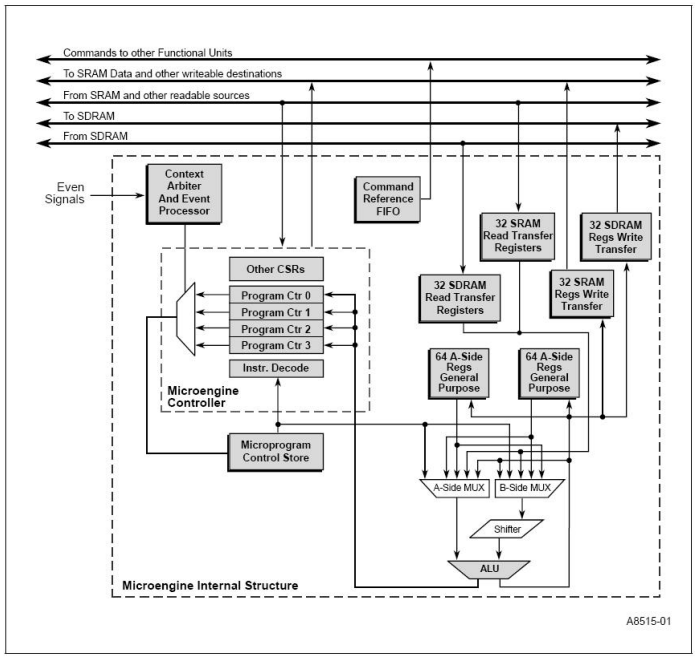

Figura 20. Arquitetura da *Microengine* (INTEL, 2001)

5.1.2 Intel *Dual Cores*

Nesta seção são apresentadas algumas das soluções da Intel para processadores homogêneos *dual core*. Três das arquiteturas ilustradas pela Figura 21 são comparadas a seguir (INTEL, 2006-b):

- O processador Pentium Extreme Edition possui dois núcleos que suportam *hyperthreading*. *Hyperthreading* é o termo da Intel para *Simultaneous Multithreading* (SMT). Sendo assim, nesta arquitetura o *hyperthreading* provê dois caminhos para os fluxos de instruções. Portanto, logicamente este processador possui quatro núcleos.

- Os processadores Pentium D e Core Duo (GOCHMAN, 2006), são compostos por dois núcleos, mas sem suporte a *hyperthreading*. Permanece o suporte a múltiplas *threads* simultâneas, mas em função da execução de duas *threads* em cada um dos dois núcleos. Portanto, fisicamente e logicamente são dois núcleos internos. A diferença entre estes processadores está na comunicação com o segundo nível de *cache*. No processador Pentium D cada núcleo possui sua específica *cache* de nível 2 (L2). No caso do Core Duo a *cache* L2 é compartilhada entre os dois núcleos. A grande vantagem no compartilhamento é o redimensionamento da *cache* utilizada por cada um dos núcleos em função da demanda de acessos à memória. Neste caso, um núcleo pode usar mais espaço da *cache*, mas o outro núcleo pode acessar o bloco compartilhado, aumentando o *cache hit*. Na *cache* separada pode ocorrer um maior acesso à memória devido ao limite de espaço de utilização de um dos núcleos que não tem acesso à outra *cache* L2 e, portanto, maior probabilidade de *cache miss*.

- Nas três arquiteturas o barramento é a escolha para interconectar os núcleos às demais unidades do *chip*.

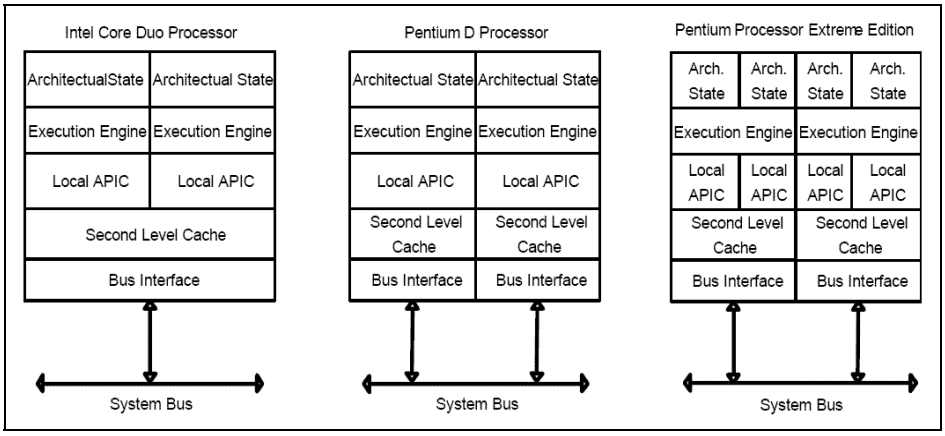

Figura 21. Exemplos de Arquiteturas com Dois Núcleos da Intel (INTEL, 2006-b)

5.1.2.1 Montecito

Montecito (MCNAIRY, 2005) é o codinome do processador Itanium 2 *dual core*, *dual thread* apresentado pela Figura 22. É um processador que possui três níveis de *cache* com uma capacidade total de aproximadamente 13,3Mbytes por núcleo ou 27Mbytes por *chip*. As *caches* L1 e L2 são divididas em dados e instruções enquanto no nível 3 as *caches* de dados e instruções são integradas. Na versão *single-chip* (o Itanium 2), a *cache* L2 também é integrada. Na separação da *cache* L2 o Montecito alcançou uma melhoria de desempenho de até 7% em relação à versão integrada. Na arquitetura, B, I, M e F significam respectivamente as seguintes unidades funcionais: *Branch* (Desvio), Inteiro, Memória e *Floating-Point* (Ponto Flutuante).

Apesar do suporte *dual thread*, o Montecito não é um processador SMT puro. Por suportar mais de uma *thread* no mesmo ciclo o Montecito pode ser considerado um processador CMT. No entanto, a capacidade de suportar duas *threads* simultâneas deve-se a uma arquitetura com dois núcleos Itanium 2. Mas cada um dos núcleos é capaz de suportar duas *threads* através da duplicação de várias unidades funcionais e chaveamento de contexto. A terminologia usada pela Intel é a TMT (*Temporal Multithreading*), mas nada mais é do que o BMT ou *Blocked Multithreading*, segundo a classificação descrita no capítulo 2. Segundo a Intel cada núcleo é capaz de suportar duas *threads* no modo TMT, mas na visão da hierarquia de memória são duas *threads* simultâneas, portanto, SMT. O Montecito é um modelo híbrido de suporte a múltiplas *threads* (BMT e SMT). O chaveamento de *threads* é resultado da ocorrência de cinco eventos, conforme descrito a seguir:

- *L3 cache miss*: ocorrência de alta latência por falta de bloco de dados em *cache* L3

- *Timeout*: Existem *quantums* ou fatias de tempo para a execução de cada *thread*. Quando este tempo ultrapassa um certo limite há uma troca de *threads*.

- *ALAT (Advance Load Address Table) invalidation*: Em alguns momentos uma *thread* pode ceder recursos de *pipeline* para outra *thread* aumentando o desempenho global. Mas havendo acesso externo invalidando uma entrada

da tabela de endereços, um chaveamento ocorre voltando os recursos para a *thread* cedente.

- *Switch hint*: Uma "dica" para chaveamento. Uma instrução no programa em execução para chavear entre a *thread* corrente e uma *thread* em *background*.

- *Low-power mode*: Se uma *thread* provocar uma dissipação de calor acima do modo de baixa potência, ocorre um chaveamento.

A comunicação entre os núcleos é suportada pela unidade *Arbiter* que provê uma baixa latência de comunicação em resposta aos eventos do sistema. Uma topologia de interconexão duplicada baseada em barramento é utilizada pelo *Arbiter*.

O suporte a virtualização é uma mistura de *hardware* e *firmware* que provê implementações de MMVs mais otimizadas. O esquema de virtualização possui um mecanismo para chaveamento rápido entre os sistemas operacionais com um sistema de proteção e isolamento do espaço de endereços. As características combinadas são capazes de deixar o MMV abstrair o número de processadores para os sistemas operacionais de forma que um único processador execute múltiplos SOs enquanto cada um dos SOs acredite ter o controle total sobre o processador.

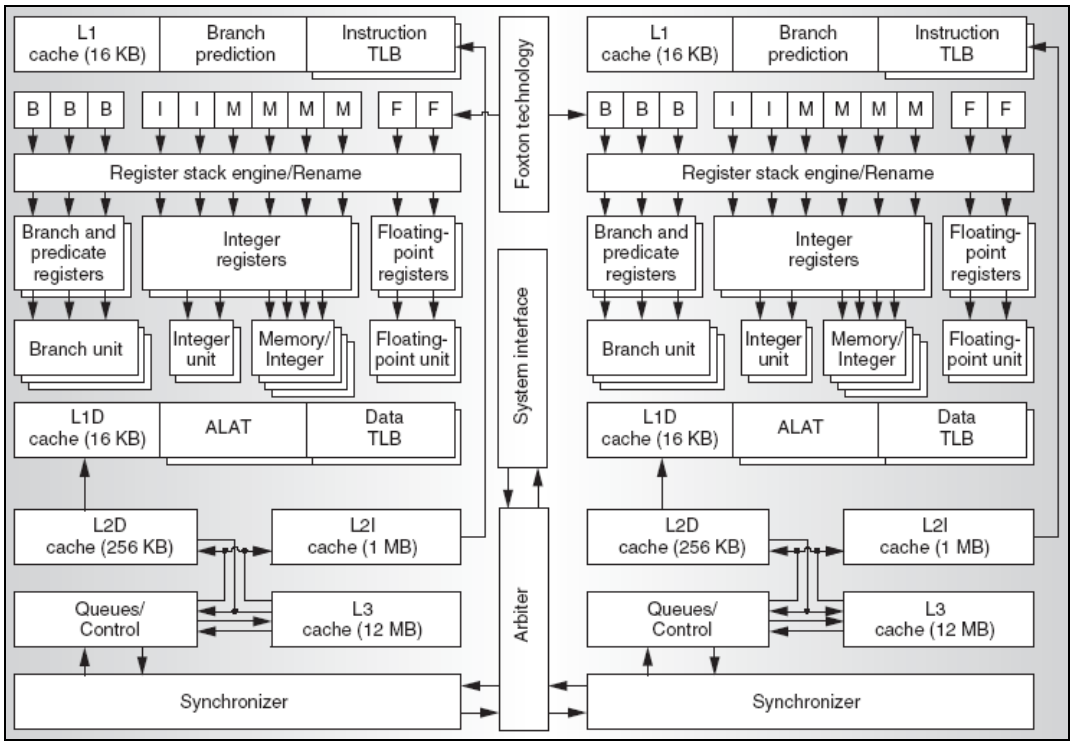

Figura 22. Arquitetura do Processador Montecito (MCNAIRY, 2005)

5.2 Processadores IBM

Esta seção apresenta duas gerações de processadores *dual core* da IBM, os processadores Power4 e Power5.

5.2.1 Power4

O processador Power4 (BEHLING, 2006) é um processador com dois núcleos de processamento com largura de palavra de 64 bits. A arquitetura é baseada em uma máquina superescalar especulativa com execução de até oito instruções em paralelo e fora de ordem. Os núcleos do Power4 não suportam *threads* simultâneas (SMT), mas em função dos dois núcleos é possível ter duas *threads* em execução no mesmo ciclo, o que caracteriza o Power4 como um processador CMT.

A arquitetura ilustrada pela Figura 23 apresenta os dois núcleos e as *caches* L2 interconectados por uma unidade de chaveamento chamada de *Core Interface Unit* (CIU). Esta unidade conecta as três controladoras das *caches* L2 a cada núcleo de processamento através de portas separadas para dados e instruções. Para reduzir a latência de acesso à memória, além da *cache* L2 o terceiro nível de *cache* (L3) também está presente na arquitetura do Power4.

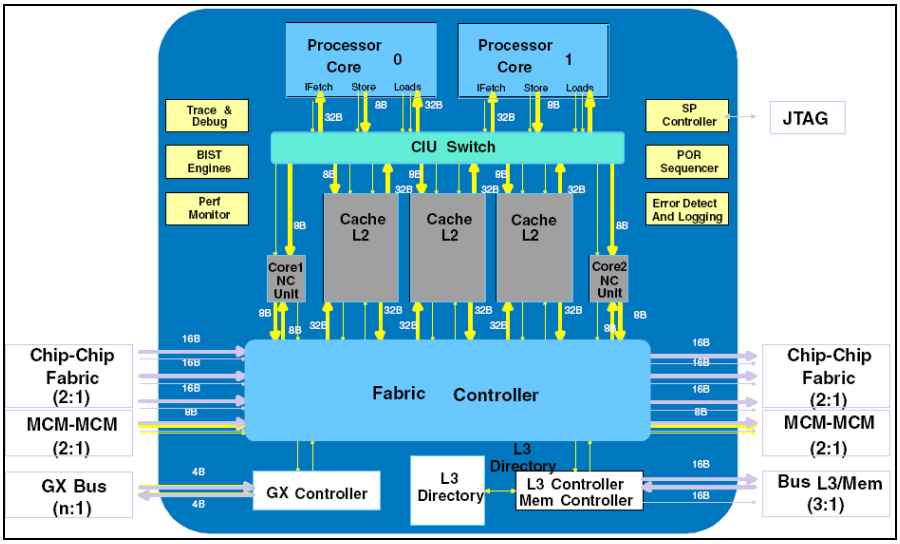

Figura 23. Arquitetura do Processador Power4 (BEHLING, 2006)

A *Fabric Controller* é uma unidade que controla uma rede de barramentos para comunicação entre as *caches* de nível 2 e 3, dos diversos módulos que podem ser conectados ao processador, além de outros Power4. Através do MCM (*Multi-Chip Module*) é possível interconectar até quatro Power4 para suporte a servidores SMP (*Symmetric Multiprocessor*).

O *pipeline* do Power4 está ilustrado pela Figura 24. A arquitetura é superescalar com quatro *pipelines* separados para a execução fora de ordem para instruções das seguintes classes: Desvio (BR: *Branch*), *Load / Store* (LD/ST), Ponto Fixo (FX: *Fixed-Point*) e Ponto Flutuante (FP: *Floating-Point*). No entanto, todas as instruções possuem um *pipeline* compartilhado, referente aos estágios de busca de instruções e formação de grupo, mas terminam novamente em um estágio compartilhado.

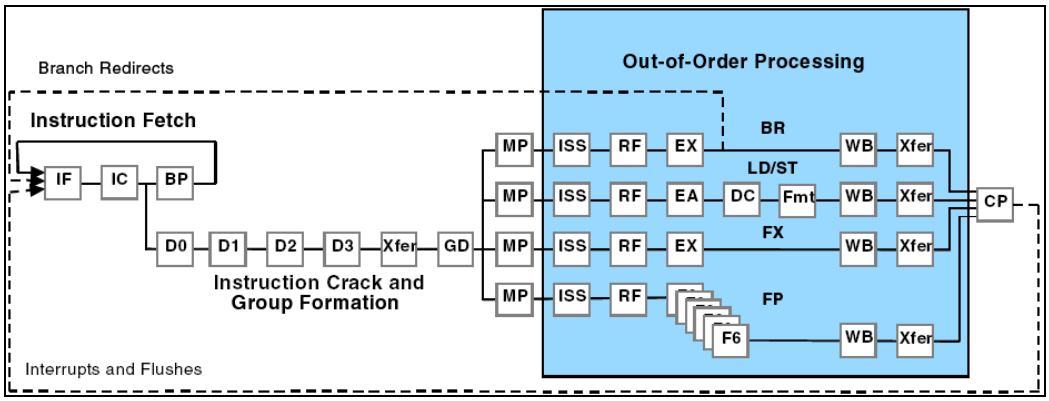

Figura 24. *Pipeline* do Processador Power4 (BEHLING, 2006)

Até oito instruções podem ser buscadas (IF: *Instruction Fetch*, IC: *Instruction Cache* e BP: *Branch Prediction*) da memória ao mesmo tempo, de acordo com os endereços no registrador IFAR (*Instruction Fetch Address Register*). O estágio de formação de grupo além de decodificar a instrução precisa "quebrar" e converter instruções complexas em instruções mais simples, algo parecido como uma conversão de instruções CISC para RISC, para que seja feito um agrupamento para posterior envio aos *pipelines* específicos.

Nos estágios de execução algumas instruções especulativas podem ser executadas. Isto significa que instruções ainda não necessárias podem ser executadas para evitar bolhas no *pipeline* e prejuízo de desempenho, já que unidades ociosas são utilizadas. Isto significa que as instruções no *pipeline* estão em ordem diferente da seqüência original do programa e por este motivo os resultados precisam ser validados para que a execução seja completada. Uma instrução especulativa somente tem sua execução completada quando o seu resultado pode ser entregue ao programa em execução. Isto significa que terminar a execução de uma instrução especulativa não significa completar a execução da instrução, que pode ter seu resultado invalidado e ter que ser executada novamente.

Outra técnica utilizada pelo Power4 é a previsão de desvios. Todos os desvios são previstos pela arquitetura e instruções são especuladas, buscadas e executadas com base nas previsões. Se a previsão estiver correta a execução das instruções segue normalmente, caso contrário, todas as instruções especuladas são descartadas do *pipeline* havendo neste caso uma perda de 12 ciclos de *clock* no desempenho do processador. Para evitar esta perda, a arquitetura do Power4 trabalha com duas técnicas:

- Tabela de históricos de desvios, onde cada entrada corresponde à informação de desvio tomado ou não,
- Informação da direção do desvio, ou seja, a informação do caminho de execução que foi tomado pelo desvio.

O Processador Power4 foi o primeiro processador *dual core* da IBM, mas o Power5 já se encontra no mercado e alguns detalhes da sua arquitetura estão descritos na seção seguinte.

5.2.2 Power5

O processador Power5 (KALLA, 2004), que tem sua arquitetura apresentada na Figura 25.b, possui códigos binários e estrutura compatível com o Power4 (Figura 25.a), mas tem como grande diferença o suporte a múltiplas *threads* simultâneas (SMT). Este suporte é referente a cada um dos dois núcleos fazendo com que o Power5 possua dois núcleos físicos, mas quatro núcleos lógicos de processamento. Este é um processador CMT com quatro *threads* simultâneas.

O suporte SMT de cada núcleo é de duas vias, ou seja, duas *threads* simultâneas. Embora com um nível maior de suporte a *threads*, as simulações da IBM verificaram que o aumento de complexidade do núcleo não foi justificado. Adicionar SMT em um núcleo físico fez com que o desempenho caísse. Esta redução de desempenho pode ser explicada através do uso das *caches* pelas *threads*: Um determinado dado na *cache* pode ser substituído pela *thread* 0, mas a *thread* 1 precisa do dado anterior, portanto, neste caso ocorre um *cache miss*. No entanto, o Power5 também suporta a execução de apenas uma *thread* por núcleo.

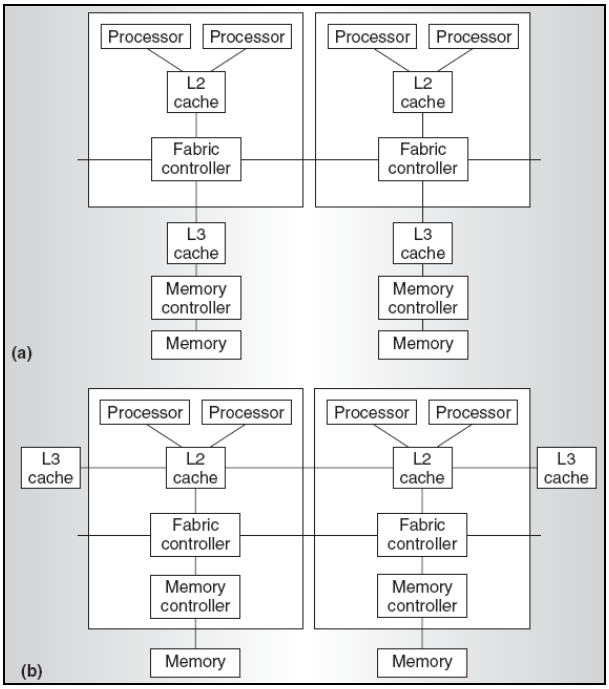

Figura 25. Arquiteturas: (a) Power4, (b) Power5 (KALLA, 2004)

Uma modificação feita em relação ao projeto anterior e que pode ser verificada através da Figura 25 é o posicionamento da *cache* L3. No Power4 a *cache* L3 precisava do *Fabric Controler* para se interconectar com a *cache* L2. Conforme citado na seção 5.2.1 o MMC é capaz de suportar um sistema SMP (*Symmetric Multiprocessor*) para servidores que possuam múltiplos processadores na arquitetura. Em relação ao Power4 este sistema pode conter até 32 processadores. No caso do Power5 pode chegar até 64 processadores. Manter a *cache* L3 conectada a esta unidade de comunicação faz com que a mesma concorra pela utilização com todos os outros módulos conectados. No projeto do Power5 a *cache* L3 se conecta a *cache* L2 diretamente sem a necessidade de usar a *Fabric Controller*.

O *pipeline* do Power5 é idêntico ao *pipeline* do Power4 apresentado pela Figura 24, exceto pelo fato de haver duas unidades de buscas de instruções e contadores de programa para suportar as duas *threads* simultâneas. No caso de usar apenas o modo *single-thread*, o Power5 passa a utilizar apenas um contador de programa.

A Figura 26 ilustra o fluxo de instruções do processador Power5 através dos blocos funcionais do *pipeline*. É importante ressaltar que a figura apresenta os blocos compartilhados ou não pelas *threads* e que exatamente no início e fim do *pipeline* onde havia compartilhamento de recursos passa a ter agora a duplicação de recursos para o suporta a duas *threads*. A comparação entre o que é compartilhado ou não pode ser feito observando-se a Figura 24.

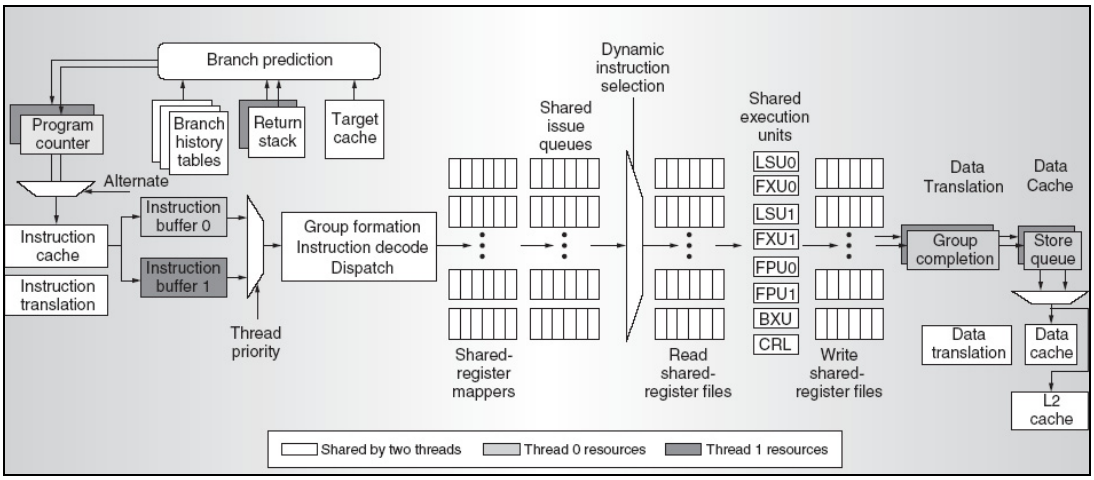

Figura 26. Fluxo de Instruções do Processador Power5 (KALLA, 2004)

A informação mais interessante no projeto do Power5 é justamente o fato do desempenho não melhorar com o suporte SMT por núcleo. Os principais motivos são:

- O número limitado de unidades de execução (o mesmo do Power4) que são compartilhados entre as duas *threads*.

- O alto consumo da largura de banda de memória pelas duas *threads*.

Esta é a principal razão para que o Power5 também suporte apenas uma *thread* por núcleo.

5.3 Processadores Sun Microsystems

A Sun Microsystems foi a primeira empresa a lançar no mercado um processador com oito núcleos homogêneos para processamento de aplicações com alta vazão de dados (CHAUDHRY, 2005) (SUN, 2006). Seu codinome é Niagara. Mas antes será descrito a arquitetura do UltraSparc IV, o primeiro *dual core* da SUN.

5.3.1 UltraSparc IV

O UltraSparc IV (SUN, 2004) (TAKAYANAGI, 2005) é o primeiro processador da Sun que segue a linha de computação de alta vazão. É um processador *dual core*,

conforme ilustrado pela Figura 27, que mantém a compatibilidade com as versões anteriores por utilizar a arquitetura de conjunto de instruções SparcV9.

O suporte às *threads* no mesmo ciclo de execução ocorre devido à execução simultânea de duas *threads*, uma em cada núcleo do UltraSparc IV. Portanto, o UltraSparc IV é um processador CMT.

A arquitetura do UltraSparc IV possui dois núcleos baseados no *pipeline* de 14 estágios do UltraSparc III. As principais características são:

- Superescalar com especulação e previsão de desvios.
- Quatro instruções podem ser despachadas de um *buffer* de instruções de 16 entradas.
- Seis instruções podem ser executadas em paralelo sendo duas de ponto fixo, uma de desvio, uma de *load*/*store* e duas de ponto flutuante (uma multiplicação/divisão e uma adição/subtração).

As duas *threads* em execução compartilham os seguintes blocos funcionais do *chip*:

- Um barramento de dados e endereços para acessar a *cache* L2.
- Uma unidade de controle de memória (MCU: *Memory Control Unit*).
- A porta de interconexão SUN Fireplane.

Apesar da *cache* L2 ser logicamente separada para a execução simultânea das duas *threads*, fisicamente as duas *caches* se encontram em um único módulo de memória.

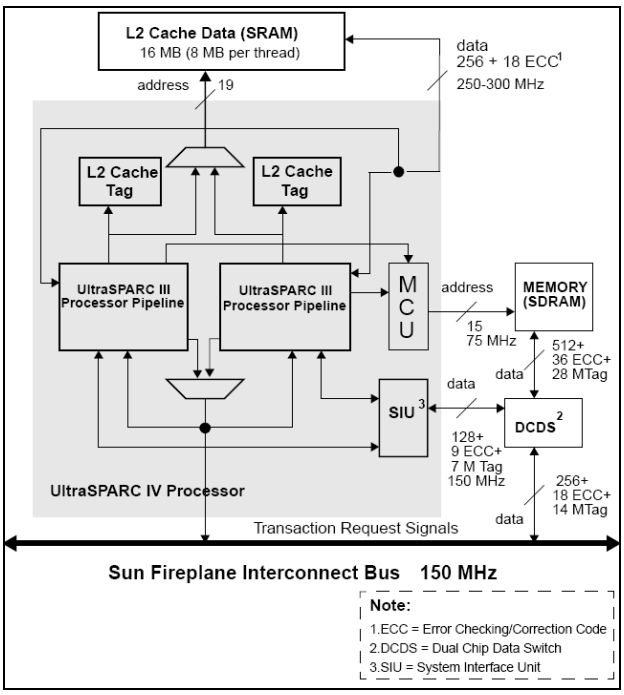

Figura 27. Arquitetura do Processador UltraSparc IV (SUN, 2004)

A linha de computação de alta vazão é destinada a ambientes de processamento de dados que possuam aplicações para servidores *web* e banco de dados ou que possuam cargas de trabalho tipicamente de computação científica e de alto desempenho. Nesta linha a SUN lançou no final de 2005 o processador Niagara, que veio a se chamar UltraSparc-T1, que possui um maior poder de computação e que será descrito na seção seguinte.

5.3.2 UltraSparc-T1 (Niagara)

O processador UltraSparc-T1 (GEPPERT, 2005) (KONGETIRA, 2005) é a mais nova geração de processadores da família UltraSparc. Sua arquitetura, apresentada pela Figura 28, contém oito núcleos de processadores Sparc-V9, que suportam até quatro *threads* através da abordagem IMT. Ao todo é possível ter oito *threads* executando simultaneamente e 32 contextos ativos de *threads* concorrendo pelas unidades funcionais dos *pipelines* de cada um dos núcleos Sparc. Portanto, o UltraSparc-T1 é um processador CMT.

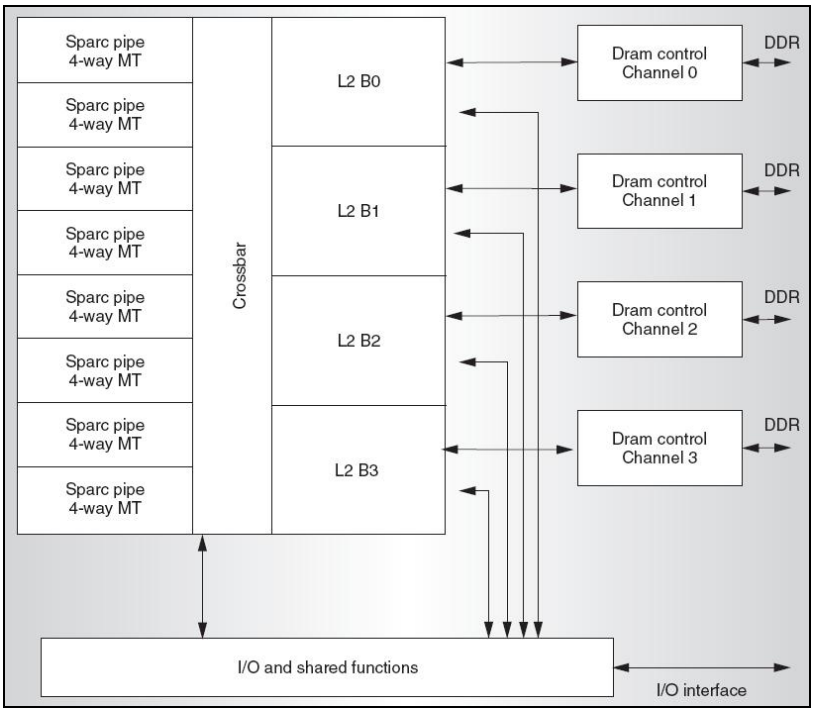

Figura 28. Arquitetura do Processador UltraSparc-T1 (KONGETIRA, 2005)

A comunicação *intra-chip* é feita através de uma chave *crossbar* que é responsável por interconectar todos os núcleos às quatro memórias *caches* de nível 2 e também às demais unidades de entrada de saída e módulos compartilhados.

As principais características do UltraSparc-T1 são:

- Arquitetura do conjunto de instruções Sparc-V9.
- Oito *threads* simultâneas e 32 contextos ativos.
- Rede de interconexão baseada em chave *crossbar* com largura de banda de 134,4GB/s.

- *Cache* L2 associativa por conjunto 12 vias de 3Mbytes.
- Quatro canais DDR2 com largura de banda de 23GB/s.
- Dissipação de potência menor do que 80W.

Basicamente cada núcleo efetua a mudança de *threads* em execução com base em eventos de grande latência e que são apresentados pela Figura 29. Os quatro tipos de eventos são estímulos de entrada para o bloco lógico de seleção de *threads* e podem ser descritos como:

- Tipo de instrução: instruções de alta latência como *load*, desvios, multiplicação e divisão.
- *Misses*: falta de um bloco na *cache*, provoca um acesso ao seguinte nível na hierarquia de memória e que possui uma maior latência.
- Interrupções: interrupções podem causar bolhas ou paradas no *pipeline*.
- Conflito de recursos: recursos compartilhados e que não podem ser utilizados ao mesmo tempo também podem provocar bolhas ou paradas no *pipeline*.

A Figura 29 também apresenta a organização interna do Sparc-V9 em função dos estágios de *pipeline*. É importante ressaltar que o UltraSparc-T1 possui núcleos baseados no ISA (*Instruction Set Architecture*) do Sparc-V9, portanto, o número de estágios de *pipeline* e organização interna podem variar em função dos projetos onde são aplicados o ISA do Sparc-V9.

Nesta arquitetura o Sparc-V9 possui um *pipeline* escalar de 6 estágios, onde a grande novidade é o segundo estágio que é responsável pela seleção da *thread*. São quatro contadores de programa que auxiliam na escolha das instruções que serão buscadas da memória. As instruções de cada *cache* são armazenadas em uma *cache* de instruções e despachadas para um *buffer* de instruções que mantém os quatro contextos ativos. O mesmo bloco funcional que seleciona a *thread* a ser lida da memória, também seleciona a *thread* que entrará em execução. A partir deste momento a instrução da *thread* escolhida é decodificada e enviada para os demais estágios do *pipeline* que seguem um padrão tradicional de execução, acesso à memória e escrita em banco de registradores.

Além do estágio de seleção de *thread* e dos quatro contadores de programa, é importante ressaltar o acréscimo de quatro bancos de registradores para que seja possível suportar os quatro contextos ativos das *threads*. No estágio de acesso à memória, também aparece a interface da chave *crossbar*, que é utilizada para acesso às *caches* de dados compartilhadas, além das tabelas de páginas TLB (*Translation Look-Aside Buffer*).

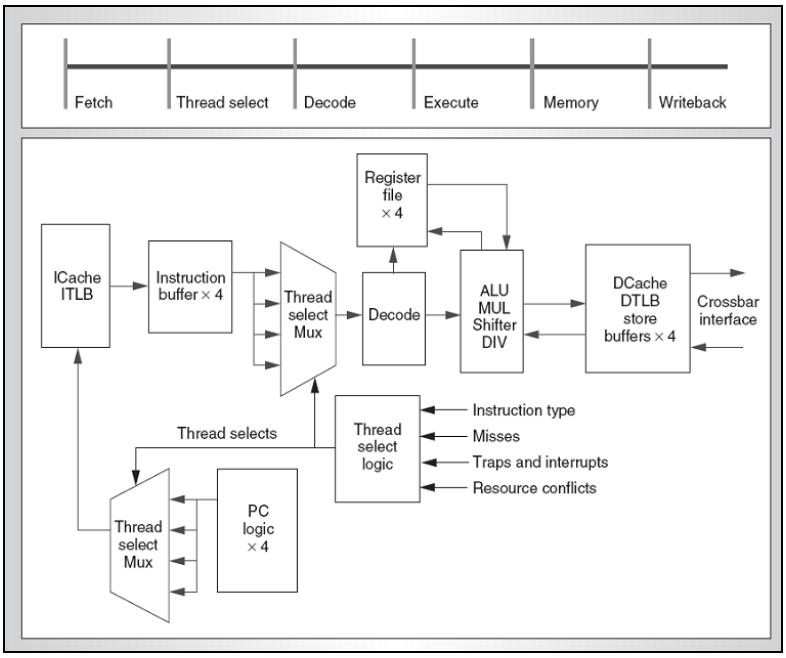

Figura 29. *Pipeline* do Núcleo Sparc-V9 (KONGETIRA, 2005)

A Figura 30 apresenta a execução de instruções no *pipeline* da Figura 29.

Existem duas situações que podem diferenciar o tipo de chaveamento entre as *threads*: quando todas as *threads* estão disponíveis (Figura 30.a) ou quando pelo menos uma das *threads* não está disponível (Figura 30.b). Com base nestas duas situações a execução de instruções se comporta da seguinte forma:

- Situação 1 (Figura 30.a): O mecanismo de seleção e execução de *threads* é bem simples. No primeiro ciclo, quando a instrução da *thread* 0 é buscada, seleciona-se uma instrução da *thread* 1 para decodificação e envio para os demais estágios do *pipeline*. No segundo ciclo, busca-se uma instrução da *thread* 1 e decodifica-se da *thread* 2. Este processo se repete até que a quarta *thread* tenha sua instrução selecionada para execução para que no próximo ciclo o mesmo seja feito para a *thread* 0, depois *thread* 1 e assim sucessivamente. Este processo é baseado no chaveamento de instruções de *threads* diferentes a cada novo ciclo ou *Interleaved Multithreading* (IMT).

- Situação 2 (Figura 30.b): Neste caso apenas duas *threads* estão disponíveis (t0 e t1). Inicia-se o processo de busca e execução normalmente, conforme descrito pela situação 1. No entanto, no terceiro ciclo há uma instrução *load* da *thread* 0 em execução. Neste ciclo uma instrução da *thread* 1 está em decodificação e o normal seria a seleção de uma instrução da *thread* 0 novamente, já que as *threads* 2 e 3 não estão disponíveis. Embora isto seja o óbvio, selecionar uma instrução da *thread* 0 pode causar uma bolha futura no *pipeline*, já que a instrução *load* é de grande latência. Desta forma uma nova instrução da mesma *thread* 1 é selecionada para decodificação e posterior execução. Como a busca de uma nova instrução t1 resultou também em um *load*, no próximo ciclo volta-se a selecionar uma instrução da *thread* 0 para execução. Este processo se repete em função das *threads* disponíveis e dos eventos de grande latência que podem obrigar a

permanência de uma *thread* no *pipeline* sem que haja uma troca de contexto no ciclo seguinte.

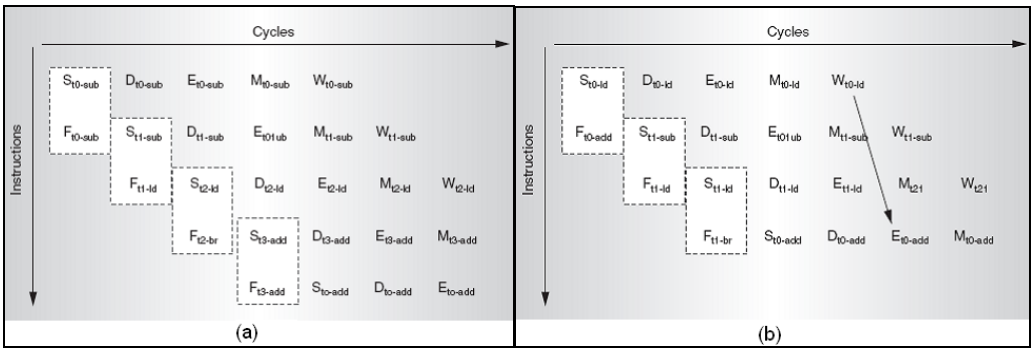

Figura 30. Execução de Instruções no *Pipeline*: (a) Todas as *threads* disponíveis, (b) Duas *threads* disponíveis (KONGETIRA, 2005)

O UltraSparc-T1 é atualmente o processador de propósito geral com maior quantidade de núcleos homogêneos disponível no mercado. Soluções com quantidade maior estão em desenvolvimento e as novas tendências em *chip multithreading* serão discutidas no próximo capítulo.

6 AVALIAÇÕES E TENDÊNCIAS EM CMT

Ao longo deste trabalho vários conceitos, técnicas e arquiteturas de processadores foram apresentadas enfatizando as características dos *chips* de processadores atuais, que são os diversos suportes às múltiplas *threads* simultâneas, resultando nos processadores CMT (*Chip Multithreading*).

Através do estudo foi possível constatar que a grande tendência são os projetos de processadores com múltiplos núcleos. Os *Chip Multiprocessors* são a alternativa atual para extrair o máximo de paralelismo das aplicações, possibilitando suportar múltiplas *threads* através dos múltiplos núcleos. Apesar dos múltiplos núcleos terem se mostrados eficientes neste suporte, as múltiplas *threads* podem ser executadas no mesmo ciclo bastando que o projeto do processador com apenas um núcleo seja baseado nos conceitos SMT ou *simultaneous multithreading*.

Portanto, um processador SMT é um processador CMT. No entanto, um processador CMP onde cada núcleo suporta apenas uma única *thread* ou múltiplas *threads* (IMT, BMT ou SMT), também é um processador CMT.

A pergunta que fica é qual a melhor combinação para a arquitetura de processadores CMTs? Os projetos de processadores comerciais e os resultados encontrados pelos respectivos fabricantes indicam uma tendência: o projeto de processadores CMP com núcleos simples e suporte a IMT ou BMT. A extração de paralelismo das *threads* em cada núcleo deve evitar a concorrência por recursos limitados do *pipeline* superescalar (KALLA, 2004), e a melhor opção é aumentar a quantidade de núcleos para aumentar a quantidade de *threads* suportadas pelo processador. A princípio uma pergunta ainda difícil de responder é se a abordagem SMT e os *pipelines* superescalares permanecerão nos projetos dos núcleos de processamento. Um processador superescalar perde em desempenho onde existe um paralelismo grande no número de *threads*, mas ganha quando este paralelismo é baixo. A abordagem SMT pode não ganhar em desempenho na utilização da superescalaridade. Portanto, é melhor ter vários núcleos com ou sem superescalaridade?

Esta pergunta é facilmente respondida da seguinte forma: tudo vai depender das cargas de trabalho dos futuros processadores. Isto implica em uma mudança na forma de programar e, portanto, na geração da própria carga de trabalho. É bem provável que no futuro não se pense em outro tipo de aplicação que não seja nativamente paralela, ou seja, tenha sido desenvolvida através de modelos de programação paralela e distribuída

e que, talvez, demande uma intensa comunicação e troca de mensagens internas ao *chip*. Haverá a possibilidade de processadores GPPs trabalharem com cargas de trabalho completamente diferentes das quais hoje existem e os computadores pessoais estarão mais parecidos com os servidores no que diz respeito ao comportamento da carga de trabalho e por conseqüência na própria arquitetura do processador.

A demanda por maior desempenho computacional tem dado um grande impulso nas pesquisas sobre virtualização. Isto se deve a grande necessidade de aumentar a capacidade de processamento, balancear carga e manter seguros diversos dados. Para solucionar esta demanda os monitores de máquinas virtuais estão sendo utilizados para gerenciar e "convidar" sistemas operacionais virtuais a participar e aumentar a quantidade de servidores lógicos de um parque computacional. Este é um caminho sem volta. Os processadores atuais devem suportar a virtualização através de instruções especiais que transfiram parte da responsabilidade de gerenciamento para o próprio processador, otimizando o processo de virtualização e simplificado a tarefa encarregada pelos softwares ou monitores de máquinas virtuais. Este processo de suporte em *hardware* é uma tendência presente nos atuais projetos da Intel e já ganha força nos novos monitores disponibilizados no mercado. Mas uma pergunta em aberto é a seguinte: Como ficam os processadores virtuais ou núcleos virtuais com o acréscimo de núcleos físicos internos ao *chip* no caso dos CMPs?

Como ficam as *threads* neste novo cenário de virtualização? Ao longo do tempo será necessário, talvez, caracterizar um conjunto de *threads* em função de uma determinada máquina virtual, principalmente quando no CMP houver uma quantidade muito grande de núcleos interconectados por uma rede *intra-chip*.

Arquiteturas de processadores *multi-core* são uma realidade e novas soluções para aumentar o desempenho são necessárias e compreendem desde a melhoria ou otimização da organização interna dos núcleos de processamento, a rede interna de comunicação (*Networks-on-Chips*?), até códigos otimizados pelos compiladores. Neste contexto o processador Piranha (BARROSO, 2000) aparece com uma característica interessante: existe um sistema de interconexão dos núcleos baseado em chave *crossbar* e outro sistema baseado em um roteador para comunicação externa. A princípio poderia se formar uma NoC, mas os roteadores estariam embutidos nos CMPs. Mas para CMPs com vários núcleos, este sistema baseado em roteador deve substituir o sistema mais simples baseado em chave *crossbar*, dando origem às NoCs para CMPs homogêneos, conforme está descrito a seguir.

Os processadores *quad core* da Intel (INTEL, 2006-c) começam a se tornar realidade, mas segundo a Intel uma nova geração de processadores *multi-core* irão surgir nos próximos anos. Conforme a Figura 31, oitenta núcleos (INTEL, 2006-a) não se comunicarão mais por barramentos e chaves *crossbar* simples, mas por uma *network-on-chip*. A princípio cada núcleo terá seu próprio roteador, seguindo as arquiteturas atuais de NoCs. A pergunta em relação às NoCs fica, no atual momento, em relação à topologia. Uma malha? Uma toróide? Uma hipercubo? É fácil responder esta pergunta caso o processador fosse projetado para um propósito específico e a carga de trabalho fosse conhecida. A resposta seria: depende da carga de trabalho. No entanto, para uma classe de processadores homogêneos e de propósito geral ainda é cedo para determinar uma tendência em se tratando de topologia da NoC.

Um problema já conhecido por pesquisadores da área de alto desempenho e processamento paralelo e distribuído está associado à quantidade de roteadores, processadores ou saltos necessários entre uma comunicação da origem ao destino. Um pacote de dados que trafega por um caminho muito extenso, que possui muitos elementos de processamento entre a origem e destino, tais como os roteadores de uma NoC, tem seu tempo médio de transmissão aumentado. Este é o principal problema em uma rede de comunicação de dados ou uma NoC o tempo médio de transmissão.

Uma alternativa para aumentar o desempenho das NoCs é a aplicação dos conceitos de Computação Reconfigurável (CR) (COMPTON, 2002) (MARTINS, 2003). Através da CR é possível aliar a flexibilidade ao desempenho. Algumas pesquisas utilizam FPGAs (*Field Programmable Gate Array*) para reconfigurar as implementações, arquiteturas e topologias de NoCs e com isto ajustar a própria NoC a uma nova demanda da arquitetura do CMP ou da comunicação interna (BREBNER, 2003) (CHING, 2004) (BARTIC, 2005) (KIM, 2005) (FREITAS, 2006).

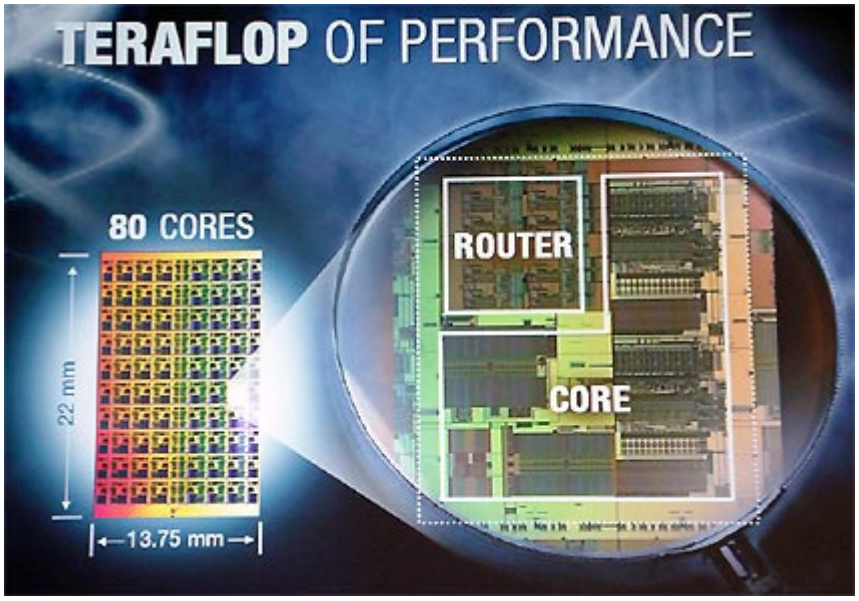

Figura 31. Projeção de *Chip Multi-Core* da Intel (INTEL, 2006-a)

Talvez a melhor solução para solucionar este problema, seria alocar corretamente o conjunto de *threads* de um determinado programa, ou carga de trabalho, ou de uma máquina virtual. Alocar corretamente seria manter estas *threads* em um sub-conjunto de núcleos próximos do *chip*, para evitar uma comunicação intensiva entre núcleos distantes e que, por conseqüência, um alto tempo de transmissão associada também a uma maior latência de rede. A resposta pode ser o compilador.

Pesquisas envolvendo alto desempenho procuram direcionar esforços em soluções de hardware, tais como os *pipelines* e mecanismos de superescalaridade, mas parte da extração do paralelismo, contudo, pode ser feita eficientemente em tempo de compilação. A grande vantagem da exploração de paralelismo pelo compilador (TSAI, 1999) (FRANKE, 2003) (CHEN, 2005) (LI, 2005) (SONG, 2005) (AKHTER, 2006) (AMARASINGHE, 2006) (LIU, 2006) está na possível redução e simplificação do *hardware*, que muitas vezes resulta em um maior desempenho de execução de instruções e *threads*, além da diminuição do gasto de energia (HARI. 2006).

Algumas das soluções mais conhecidas de otimização de códigos estão relacionadas com as arquiteturas VLIW (*Very Long Instruction Word*) (NAKRA, 1999) (UNGERER, 2003) e com suporte a múltiplas *threads*. No entanto, com o surgimento cada vez mais presente das arquiteturas *multi-core*, pesquisas envolvendo otimizações em compiladores tem aumentado consideravelmente. Uma solução necessária, já que a arquitetura *multi-core* é nativamente paralela e demanda geração de código paralelo.

O momento atual é parecido com o surgimento do conceito RISC (*Reduced Instruction Set Computing*) onde havia dúvidas sobre as aplicações e alternativas de substituição em relação ao conceito anterior CISC (*Complex Instruction Set Computing*). Não se tem certeza se soluções com núcleos de complexidade reduzida irão substituir os núcleos com maior complexidade nos processadores *multi-core*. Talvez os compiladores sejam os principais responsáveis pela mudança no futuro das organizações internas destes núcleos e do próprio *chip*.

7 CONCLUSÕES

Este trabalho apresentou os principais conceitos e arquiteturas relacionadas a *chip multithreading*, iniciando pelas abordagens de *threads* implícitas e enfatizando as *threads* explicitas, nas quais os principais processadores comerciais atualmente são baseados.

Entre os principais problemas apresentados, foi verificado através do estudo da literatura que a abordagem SMT pode não ter um bom desempenho no uso conjunto com a abordagem CMP. No entanto, a superescalaridade ainda é uma questão em aberto.

O certo é que os atuais projetos devem aumentar consideravelmente o número de núcleos internos ao *chip*, suportando cada vez mais a virtualização, além de uma rede de comunicação *intra-chip* (NoC) com alta largura de banda e baixa de latência.

A principal contribuição do trabalho é referente ao material produzido que apresentou a relação entre os principais conceitos e arquiteturas em *chip multitrheading* e as tendências futuras na área, atingindo os objetivos definidos na introdução.

Como trabalhos futuros, as sugestões são:

- Estudos sobre compiladores, os principais conceitos e técnicas de otimização para geração de códigos paralelos atendendo a demanda dos futuros CMTs com NoCs.
- Estudo de novas arquiteturas de conjunto de instruções para suporte a virtualização.
- Estudo das relações entre *threads*, máquinas virtuais e *chips* com quantidades elevadas de núcleos de processamento.
- Estudo das arquiteturas e topologias para *Networks-on-Chips*.
- Estudo do consumo de potência e energia em processadores CMT.

REFERÊNCIAS

ACOSTA, C., et al. A Complexity-Effective Simultaneous Multithreading Architecture, **International Conference on Parallel Processing**, p. 157-164, June 2005.

AHMADI, H., DENZEL, W. E., A Survey of Modern High-Performance Switching Techniques, **IEEE Journal on Selected Areas in Communications**, v. 7, n. 7, September 1989

AKHTER, S., ROBERTS J., Multi-Core Programming, Increasing Performance through Software Multi-threading, **Intel Press**, 336p., April 2006.

AMARASINGHE, S, Multicores from the compiler's perspective: a blessing or a curse?, **IEEE International Symposium on Code Generation and Optimization**, p. 137, March 2006.

AMD, **The AMD Opteron Processor for Servers and Workstations**, AMD Opteron Processor Overview, 2005

ANDERSON, G. A., JENSEN, E. D., Computer Interconnection Structures: Taxonomy, Characteristics, and Examples, **ACM Computing Surveys**, v. 7, n. 4, December 1975

ARDEVOL, J. R., **Chip Multiprocessors with Speculative Multithreading: Design for Performance and Energy Efficiency**, Tese de Doutorado em Ciência da Computação, University of illinois at Urbana-Champaign, 2004.

BARROSO, L. A., et al., Piranha: a scalable architecture based on single-chip multiprocessing, **27th Annual International Symposium on Computer Architecture**, p. 282-293, 2000.

BARTIC, T. A., et al., Topology adaptive network-on-chip design and implementation, **IEE Proc. Comput. Digit. Tech.**, v. 152, n. 4, p. 467-472, July 2005.

BEHLING, S., et al., The Power4 Processor Introduction and Tuning Guide, IBM Redbooks, Disponível em: http://www.ibm.com/redbooks, Acesso em: nov. 2006.

BENINI, L., MICHELI, G. D., Network-on-chip architectures and design methods, **IEE Proceedings Computers & Digital Techniques**, v. 152, Issue 2, p. 261-272, March 2005.

BENINI, L., MICHELI, G. D., Networks on chips: a new SoC paradigm, **IEEE Computer**, v. 1, p. 70-78, January 2002.

BOIVIE, V., **Network Specific Multithreading Tradeoffs**. Tese de Doutorado, Linköpings Universitet, Sweden, 2005.

BREBNER, G., Levi, D., Networking on Chip with Platform FPGAs, **IEEE International Conference on Field-Programmable Technology**, p. 13-20, 2003.

CALDER, B., Reinman, G., Tullsen, M., Selective value prediction, 26th Annual international Symposium on Computer Architecture. **IEEE Computer Society**, Los Alamitos, CA, USA, p. 64-74, 1999.

COMER, D. E., **Network Systems Design Using Network Processors**, Prentice Hall, 2003.

COMPTON, K., HAUCK, S., Reconfigurable Computing: A Survey of Systems and Software, **ACM Computing Surveys**, v. 34, n. 2, p. 171-210, June 2002.

CHAUDHRY, S., et al., High-performance throughput computing**, IEEE MICRO**, Volume 25, Issue 3, p. 32-45, May-June 2005.

CHEN, M. K., et al., Shangri-La: achieving high performance from compiled network applications while enabling ease of programming, **Conference on Programming Language Design and Implementation**, p. 224-236, 2005.

CHING, D., SCHAUMONT, P., VERBAUWHEDE, I., Integrated Modeling and Generation of a Reconfigurable Network-on-Chip, **18th International Parallel and Distributed Processing Symposium**, p. 139-145, 2004.

DAL PIZZOL, G., et al., Branch prediction topologies for SMT architectures, **International Symposium on Computer Architecture and High Performance Processing**, p. 118-125, October 2005.

DE ROSE, C., NAVAUX, P. O. A., **Arquiteturas Paralelas**, Editora Sagra Luzzatto, 2003.

EGGERS, S., et al., Simultaneous Multithreading: A Platform for Next Generation Processors, IEEE MICRO, p. 12-19, 1997

EVERS, M., CHANG, P.-Y, PATT, Y., Using Hybrid Branch Predictors to Improve Branch Prediction, **23rd International Symposium on Computer Architecture**, p. 3-11, May 1996.

FRANKE, B., O'BOYLE, M. F. P., Compiler parallelization of C programs for multi-core DSPs with multiple address spaces, **1st IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis**, p. 219-224, 2003.

FREITAS H. C., et al., Reconfigurable Crossbar Switch Architecture for Network Processors, **IEEE International Symposium on Circuits and Systems**, p. 4042-4045, May 2006.

FREITAS, H. C., MARTINS, C. A. P. S., Processadores de Rede: Conceitos, Arquiteturas e Aplicações **III Escola Regional de Informática RJ/ES**, Vitória, p. 127-166, 07 de outubro, 2003.

GABBAY, F., MENDELSON, A. Using Value Prediction to Increase the Power of Speculative Execution Hardware, **ACM Transactions on Computer Systems**. v. 16, n. 3, p. 234-270. ACM, New York, 1998.

GEPPERT, L., Sun's big splash [Niagara microprocessor chip], **IEEE Spectrum**, v. 42, Issue 1, p. 56-60, January 2005.

GOCHMAN, S., et al., Introduction to Intel Core Duo Processor Architecture, **Intel Technology Journal**, v. 10, Issue 2, p. 89-98, March 15, 2006.

GONÇALVEZ, R., NAVAUX, P. O. A., Improving SMT performance scheduling processes, Euromicro, **Workshop on Parallel, Distributed and Network-based Processing**, p. 327-334, January 2002.

HAMMOND, L, et al., The Stanford Hydra CMP, **IEEE MICRO**, v. 20, Issue 2, p. 71-84, March-April 2000.

HARI, S, et al. Compiler-Directed Power Density Reduction in NoC-Based Multi-Core Designs, **IEEE International Symposium on Quality Electronic Design**, p. 570-575, March 2006.

HENNESSY, J. L., PATTERSON, D. A., **Arquitetura de Computadores Uma Abordagem Quantitativa**, Editora Campus, 3ª edição, 2003.

INTEL, 80 FPU Cores on a Chip for Teraflop Performance, **Tom's Hardware**, Disponível em: http://www.tomshardware.com/2006/09/27/idf_fall_2006/page2.html, Acesso em: nov. 2006

INTEL, **IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture**, March 2006.

INTEL, **Intel Core 2 Duo Desktop Processor**, Product Brief, 2006.

INTEL**, IXP1200 Network Processor Family**, Intel Manual, 2001.

JOHNSON, M., **Superscalar Microprocessor Design**, Englewood Cliffs: Prentice Hall, 1991.

KALLA, R., SINHAROY, B., TENDLER, J. M., IBM Power5 Chip: A Dual-Core Multithreaded Processor, **IEEE MICRO**, v. 24, Issue 2, p. 40-47, 2004

KIM, D., LEE, K., LEE, S., YOO, H., A Reconfigurable Crossbar Switch with Adaptive Bandwidth Control for Networks-on-Chip, **IEEE International Symposium on Circuits and Systems**, p. 2369-2372, 2005.

KIM, H., et al., Wish Branches: Enabling Adaptive and Aggressive Predicated Execution**, IEEE MICRO**, v. 26, Issue 1, p.48-58, Jan-Feb 2006.

KONGETIRA, P., et al., Niagara: a 32-way multithreaded Sparc processor, **IEEE MICRO**, v. 25, Issue 2, p. 21-29, March-April 2005.

KRISHNAN, V., TORRELAS, J., A Chip-Multiprocessor Architecture with Speculative Multithreading, **IEEE Transactions on Computers**, v. 48, n. 9, p. 866-880, September 1999.

KUMAR, R., et al., Heterogeneous chip multiprocessors, **IEEE Computer**, v. 38, Issue 11, p. 32-38, November 2005.

KUMAR, R., et al., Single-ISA heterogeneous multi-core architectures for multithreaded workload performance, **31st International Symposium on Computer Architecture**, p. 64-75, June 2004.

KUMAR, R., Zyuban, V., Tullsen, D.M., Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling, **32nd International Symposium on Computer Architecture**, p. 408-419, June 2005.

LI, F., et al., Compiler-directed proactive power management for networks, **International Conference on Compilers, Architecture and Synthesis for Embedded Systems**, p.137-146, 2005.

LIPASTI, M., Wilkerson, C., Shen, J., Value locality and load value prediction, **ACM SIGOPS Operating Systems Review**. v. 30, n. 5, p. 138-147, ACM, New York, December, 1996.

LIU, W., et al., POSH: a TLS compiler that exploits program structure, **ACM Symposium on Principles and Practice of Parallel Programming**, p. 158-167, 2006.

MAEHLE, E., Network-on-Chip, **Institut für Technische Informatik**, Disponível em: http://www.iti.mu-luebeck.de/Research/PFTC/NPU/. Acesso em: nov. de 2006.

MARTINS, C. A. P. S., et al., Computação Reconfigurável: conceitos, tendências e aplicações, **Jornada de Informática**, Congresso da Sociedade Brasileira de Computação, Capítulo 8, 2003

MCFARLING, S., Combining Branch Predictors, **DEC Western Res. Lab.**, Palo Alto, CA, Tech. Rep. DEC WRL Tech. Note TN-36, 1993.

MCNAIRY, C., BHATIA, R., Montecito, A Dual-Core, Dual-Thread Itanium Processor, **IEEE MICRO**, v. 25, Issue 2, p. 10-20, March-April, 2005

NAKRA, T., GUPTA, R., SOFFA, M. L., Value Prediction in VLIW Machines, **26th Annual International Symposium on Computer Architecture**, p. 258-269, 1999.

NEIGER, G., et al., Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization, **Intel Technology Journal**, v. 10, Issue 3, p. 167-177, August 10, 2006.

OLUKOTUN, K., et al., The Case for a Single-Chip Multiprocessor, **7th International Conference on Architectural Support for Programming Languages and Operating Systems**, p. 2-11, 1996.

PATTERSON, D. A., HENNESSY, J. L., **Organização de Computadores a Interface Hardware/Software**, Campus, terceira edição, 2005.

PILLA, M. L., et al., A Speculative Trace Reuse Architecture with Reduced Hardware Requirements, **18th International Symposium on Computer Architecture and High Performance Computing**, Ouro Preto, p. 47-54, 2006.

SANTOS, R. R., et al., Complex branch profiling for dynamic conditional execution, **International Symposium on Computer Architecture and High Performance Processing**, p. 28-35, November 2003.

SMITH, J., A study of branch prediction strategies, **International Symposium on Computer Architecture**. ACM Press, New York, NY, p. 202-215, 1998.

SONG Y., et al., Design and implementation of a compiler framework for helper threading on multi-core processors, **IEEE International Conference on Parallel Architectures and Compilation Techniques**, p. 99-109, September 2005.

SPRACKLEN, L., ABRAHAM, S.G., Chip Multithreading: Opportunities and Challenges, **International Symposium on High-Performance Computer Architecture**, p. 248-252, February 2005.

SUN Microsystems, Developing Scalable Applications for Throughput Computing, **White Paper Sun Microsystems**, Disponível em: http://www.sun.com/servers/coolthreads/coolthreads_whitepaper.pdf, Acesso em: nov. 2006.

SUN Microsystems, **UltraSPARC IV Processor Architecture Overview**, Technical White Paper, version 1.0, February, 2004.

TAKAYANAGI,T., et al., A Dual-Core 64-bit UltraSPARC Microprocessor for Dense Server Applications, **IEEE Journal of Solid-State Circuits**, v. 40, N. 1, p. 7-18, January 2005.

TSAI, J. Y., et al., Compiler Techniques for the Superthreaded Architectures, **International Journal of Parallel Programming**, v. 27, n. 1, p. 1-19, 1999.

TULLSEN, D. M., et al., Exploiting Choice: Instruction Fetch and Issue on a Implementable Simultaneous Multithreading Processor, **23rd Annual International Symposium on Computer Architecture**, p. 191-202, 1996.

UHLIG, R., et al., Intel Virtualization Technology, **IEEE Computer**, p. 48-56, 2005

UNGERER, T, et al., Multithreaded Processors, **The Computer Journal**, British Computer Society, v. 45, n. 3, p. 320-348, 2002

UNGERER, T., et al., A Survey of Processors with Explicit Multithreading, **ACM Computing Surveys**, v. 35, Issue 1, p. 29-63, March 2003.

WHATELY, L. A., AMORIM, C. L., Sistemas de Computação Baseados em Máquinas Virtuais, Minicurso, **Workshop em Sistemas Computacionais de Alto Desempenho**, 22p., 2005

WOLF, W., The Future of Multiprocessor System-on-Chip, **Proceedings of the DAC**, June, 2004

YEH, T.-Y, Patt, Y., The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited, **27th International Symposium on Microarchitecture**, p. 228-232, San Jose, California, November, 1994.

YEN, D. W., Scalable Sparc Systems Next-Generation Computing, Scalable Systems Group, **Sun Microsystems**, 2005.

ZEFERINO, C. A., SANTO, F. G. M. E., SUSIN, A. A., ParIS: A Parameterizable Interconnected Switch of Networks-on-Chip, **ACM Symposium on Integrated Circuits and Systems Design**, p. 204-209, 2004.