Pontifícia Universidade Católica do Rio Grande do Sul

Faculdade de Engenharia – Faculdade de Informática

Curso de Engenharia de Computação

Memória Cache em uma Plataforma Multiprocessada (MPSoC)

Proposta de Trabalho de Conclusão

Autores
Antonio A. de Alecrim Jr.
Rafael Fraga Garibotti

Orientador
Prof. Dr. Fernando Gehm Moraes

Porto Alegre, março de 2007.

Índice

Índice de Figuras

Índice de Tabelas

1 Introdução

Um MPSoC (Multi-processor system-on-chip) consiste de uma arquitetura composta por diversos recursos, incluindo múltiplos processadores embarcados, módulos de hardware dedicados, memórias e um meio de interconexão.

Um número crescente de aplicações embarcadas possui requisitos estritos de desempenho, consumo de potência e custo. Exemplos dessas aplicações são encontrados nas áreas de telecomunicações, multimídia, redes e entretenimento. O atendimento de cada um desses requisitos separadamente é uma tarefa difícil e a combinação deles constitui um problema extremamente desafiador. Contudo, MPSoCs possibilitam a adaptação da arquitetura do sistema de forma a melhor atender os requisitos da aplicação: a atribuição de poder computacional onde é necessário tende a atender aos requisitos de desempenho; a remoção de elementos desnecessários reduz o custo e o consumo de potência. Isso mostra que MPSoCs não são apenas chips multiprocessadores, os quais apenas utilizam da vantagem da alta densidade de transistores para colocar mais processadores em um único chip, e não tentam atender as necessidades da aplicação. Ao contrário, MPSoCs são arquiteturas personalizadas que fazem um balanço entre as restrições da tecnologia VLSI com os requisitos da aplicação.

MPSoCs, além de processadores, contém um meio de interconexão. Diferentes meios de interconexão são utilizados: conexão ponto a ponto, barramento único, barramento hierárquico e redes intra-chip. A arquitetura mais utilizada é o barramento, contudo, não é escalável, e suporta poucas dezenas de núcleos. A mais nova forma de interconexão são as redes intra-chips, também denominadas NoCs (do inglês, Network on Chip) [MOR04]. Essas redes vêm sendo pesquisadas com o intuito de resolver problemas relacionados à comunicação de dados entre os componentes do sistema. Dentre esses problemas, encontra-se a baixa escalabilidade, e a ausência de paralelismo, uma vez que a interconexão, através de um barramento compartilhado, permite que apenas uma comunicação possa ser estabelecida em um dado momento.

Tais sistemas multiprocessados contêm diversos processadores com memórias locais, os quais podem acessar uma memória compartilhada através de um subsistema de comunicação. A memória compartilhada, que normalmente contém múltiplos módulos de memória, permite compartilhamento de código e dados entre processadores, a baixo custo e de maneira eficiente.

As maiores vantagens de se ter MPSoCs com memória compartilhada é o custo que a implementação representa, e o modelo de programação simples apresentado (que permite fácil desenvolvimento de software paralelo). Entretanto, devido a competição pelo uso dos recursos compartilhados, a latência média de acesso à memória compartilhada tende a ser maior, o que degrada a performance. A maneira usual de reduzir o acesso à memória comum é a utilização da **Memória Cache**. Existem dois tipos de memória cache em sistemas multiprocessados: Caches Compartilhadas e Caches Privadas.

Apesar da utilização de caches compartilhadas reduzir a média de tempo de acesso a memória, o seu uso é limitado, pois elas não diminuem a disputa por recursos compartilhados. A introdução de caches privadas é a maneira mais correta de lidar com esse problema. Uma cache privada associada ao processador é capaz de satisfazer a maioria de referências à memória, localmente, reduzindo a necessidade de acesso à memória compartilhada.

No entanto, a existência de caches privadas introduz um novo problema. Múltiplas cópias do mesmo dado podem existir em diferentes caches simultaneamente e, se os processadores estão habilitados a atualizar suas próprias cópias, uma visão de memória inconsistente ocorrerá, levando ao mau funcionamento do programa. Conhecido como **Problema de Coerência de Cache**, as principais razões para a ocorrência de inconsistência de dados incluem o compartilhamento de dados, que podem ser modificados, processos de migração, entre outros. De acordo com modelos, um sistema é dito ser coerente se todas as leituras, por qualquer processador, retornam o valor produzido pela última operação de escrita, sem importar qual processador realizou esta operação [STA03]. Para manter esta coerência, o sistema deve incorporar algum tipo de método que defina o completo e consistente conjunto de operações de acesso a memória compartilhada, prevenindo a inconsistência entre caches, e garantindo a execução do programa com dados corretos.

1.1 Motivação

A necessidade de assegurar o funcionamento de sistemas de computação tem exigido esforços crescentes no sentido de evitar ocorrências de inconsistência da memória, ou seja, o uso de um dado não atualizado.

A motivação do presente trabalho reside na importância das arquiteturas MPSoCs para os atuais sistemas embarcados, e na necessidade de desenvolver arquiteturas escaláveis. Assim, a implementação de um sistema com estas características permitirá mais opções aos desenvolvedores, podendo, estes, escolher entre usar trocas de mensagens entre tarefas, ou a utilização de uma memória compartilhada para tal finalidade, além de dispor de um mecanismo coerente de memória cache.

O desenvolvimento de arquiteturas paralelas, usando memórias caches, abre novas perspectivas relacionadas ao desempenho.

1.2 Objetivos

Este tratado estende o trabalho de mestrado de Cristiane Raquel Woszezenki [WOZ06], o qual propôs a criação de uma plataforma MPSoC no nível físico (VHDL nível RTL), com uma infra-estrutura de hardware e uma infra-estrutura de software capaz de gerenciar a execução de tarefas no sistema. O presente trabalho acrescentará novos módulos à plataforma, inserindo, primeiramente, uma memória compartilhada para, em seguida, caches, proporcionando um melhor desempenho do conjunto.

O objetivo deste trabalho é aplicar os conhecimentos adquiridos durante o curso, desenvolvendo uma estrutura multi-processada com suporte a caches.

1.3 Resultados Esperados

Ao final do trabalho de conclusão espera-se que o sistema desenvolvido:

- Possa ter suporte a um módulo "memória", onde estaria uma memória compartilhada entre os processadores conectados;
- Tenha caches, para obter um melhor desempenho desta memória compartilhada;
- Tenha incluído pelo menos um mecanismo de coerência de cache.

2 Referencial Teórico

Uma das técnicas mais importantes que se cristalizaram ao longo do tempo em sistemas de memória é a utilização de memórias cache. A utilização de memórias cache permite acesso mais rápido a informações armazenadas num nível lento, e com maior capacidade, em uma hierarquia de memória.

O conceito fundamental num sistema de cache é que, dispondo de um nível intermediário mais rápido, porém limitado em tamanho (devido ao custo mais elevado que o primeiro, por exemplo), e valendo o princípio da localidade (ele consiste no fato de que as referências à memória feitas por um processo não são aleatórias, dado um instante de execução). É muito mais provável que itens de memória que serão acessados, dentro de um período razoável de tempo, sejam vizinhos dos itens acessados recentemente, chamada de localidade espacial, e que esses itens previamente acessados sejam acessados novamente, chamada de localidade temporal, podendo manter nesse nível intermediário, cópias das vizinhanças das informações acessadas no nível mais lento. Devido ao princípio da localidade, podemos trabalhar com estas cópias durante um bom tempo, sem ter que acessar o sistema mais lento. Isto leva à situação em que o programa enxerga o espaço de endereçamento do sistema grande, através dos tempos de acesso do sistema rápido.

Para que o sistema funcione, é necessário que o conjunto das vizinhanças acessadas se insira no sistema intermediário por um tempo satisfatório, então, o processador busca os itens de dados requisitados na cache e, se o dado referenciado encontra-se lá, diz-se que ocorreu um cache hit e o acesso é completado em apenas um ciclo de clock. Mas, se o item não está presente na cache, diz-se que ocorreu um cache miss. Quando acontece um cache miss, um bloco (unidade de informação da cache), contendo o dado referenciado e os dados armazenados em sua vizinhança, são copiados da memória principal para a memória cache. Após a transferência deste bloco de dados para a memória cache, o processador reinicia a execução da instrução, gerando uma nova busca na cache, onde o dado agora será encontrado. Quando ocorre um cache miss, o acesso consome vários ciclos de clock para ser completado.

2.1 Mapeamentos de Memória Cache

As memórias *cache* podem ser estruturadas de formas diferentes. A memória *cache* é formada por uma memória RAM e pelo seu **diretório**. A memória RAM é organizada em **blocos** de dados, proveniente da memória principal. Elas podem ser **mapeadas diretamente**, **ter mapeamento totalmente associativo**, **mapeamento associativo por conjunto** ou, **mapeado por setor**.

A organização mais simples é a da *cache* **mapeada diretamente** (Figura 1). Neste tipo de *cache* temos acesso simultâneo ao *tag* (nome dado a um dos campos do diretório) e aos dados. Não é necessária memória associativa, e o algoritmo de substituição é trivial. A taxa de acerto é baixa, se 2 (ou mais) blocos mapeados no mesmo slot são utilizados alternadamente, o reduzindo-se assim o desempenho.

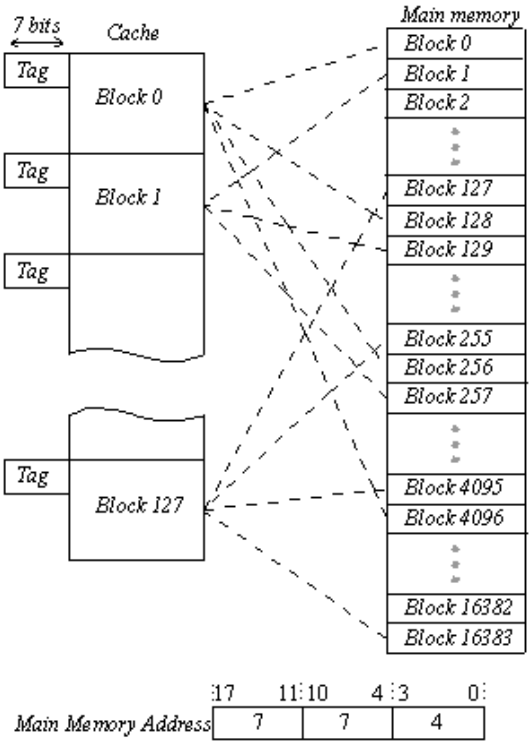

Figura 1 – Memória cache com mapeamento direto.

Na cache mapeada diretamente, uma determinada palavra só pode ocupar uma única posição na cache, havendo ainda uma tag separada para cada palavra. No exemplo, mostrado na Figura 1, cada bloco tem 16 words (4 bits). Os bits de tag [17..11] são usados para comparação (*cache hit*, *cache miss*) e os bits [10 .. 4] representam o índice do bloco. Para saber se um dado está na cache, basta comparar a tag. A tag contém informações sobre um endereço de memória, que permite identificar se a informação desejada está ou não na cache. Ele só precisa conter a parte superior do endereço, correspondente aos bits que não estão sendo usados, como índice da cache. Os índices são usados para selecionar a única entrada da cache que corresponde ao endereço fornecido. Além disso, existe um bit de validade, para indicar se uma entrada contém ou não um endereço válido.

As *caches* com **mapeamento totalmente associativa** (Figura 2), tem alto desempenho e alto custo de hardware, porém a flexibilidade de mapeamento permite implementar diversos algoritmos de substituição. A substituição de um bloco só é necessário quando a cache está cheia. A contenção de blocos é pequena, e é utilizado na TLB (*translation-lookaside buffer*), do sistema de memória virtual.

Na cache com mapeamento totalmente associativo, uma determinada palavra pode ocupar qualquer posição na cache. Como visto no exemplo mostrado na Figura 2, cada bloco tem 16 words (4 bits). Os bits de tag [17..4] são usados para comparação. Neste caso, para saber se um bloco está na cache, é necessário pesquisar todos os blocos. Se o bloco referenciado não for encontrado (ou seja, ocorreu um *cache miss*), a lógica de controle da memória cache se encarrega de copiar o bloco apropriado a partir da memória principal em um bloco livre. Mas se a cache estiver cheia, é necessária a substituição de um bloco.

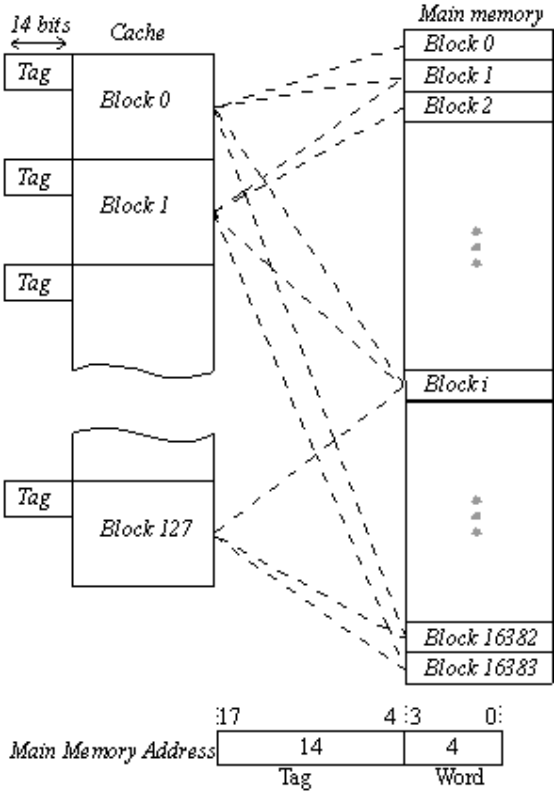

Figura 2 – Memória cache com mapeamento totalmente associativo.

A questão central na substituição é a escolha do bloco a ser descartado. A maneira mais simples seria escolher aleatoriamente qualquer um dos blocos do conjunto selecionado. A escolha aleatória é atrativa devido à sua simplicidade de implementação, porém, esta política aumenta a chance de ser escolhido um bloco que será referenciado em breve. Em resumo: com a escolha aleatória existe o risco de um aumento excessivo no número de *cache miss*.

Em geral, é adotada uma política de escolha mais segura, denominada LRU (*Least Recently Used*). A política LRU diz que o bloco a ser substituído é aquele que não é referenciado há mais tempo. Este critério de escolha se baseia no princípio da localidade.

Caches com **mapeamento associativo por conjunto** (Figura 3), tem compromisso entre mapeamento direto e totalmente associativo. Ele reduz o tamanho e o custo da comparação associativa, e é usado pela maioria das CPUs atuais. É importante ressaltar que o tamanho do bloco, o número de blocos por conjunto (Slot), e o número de conjuntos, podem variar, resultando em configurações diferentes.

A memória cache associativa por conjunto, ao receber o endereço de uma locação de memória, interpreta o endereço da seguinte forma. O endereço é, logicamente, dividido em três campos: *word*, formado pelos bits menos significativos; *conjunto* (set), formado pelos n bits seguintes; e *tag*, composto pelos m bits mais significativos do endereço. A memória cache usa os bits do campo *word* para selecionar um byte específico dentro de um bloco. O campo *set* é usado para selecionar um dos conjuntos, enquanto o campo *tag* é utilizado para verificar se o dado referenciado se encontra em algum dos blocos do conjunto selecionado.

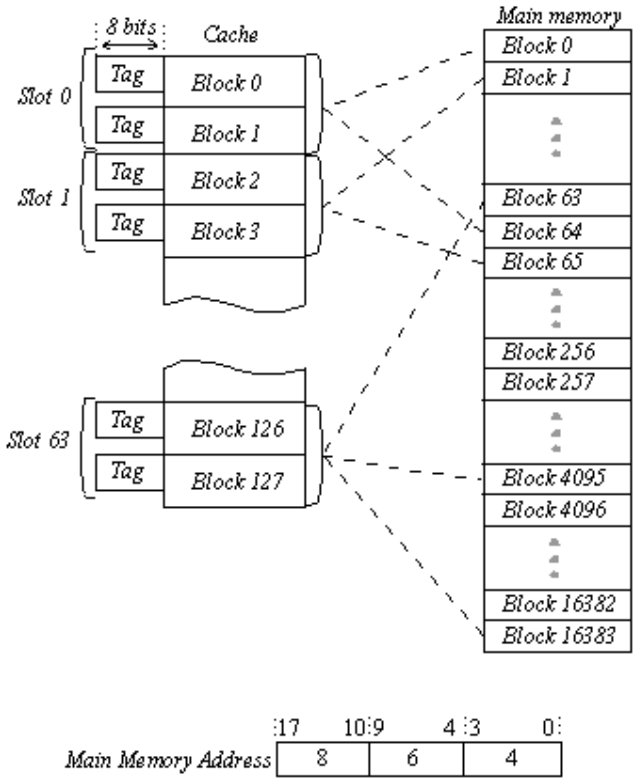

Figura 3 – Memória cache com mapeamento associativo por conjunto.

Em um acesso, a memória cache seleciona primeiro o conjunto e, em seguida, compara o tag do endereço recebido com os *tags* armazenados nas entradas de diretório do conjunto selecionado. O bloco pode ser colocado em qualquer elemento deste conjunto. Se o tag no endereço coincide com algum dos *tags* no diretório, isto significa que o bloco com o byte referenciado encontra-se no bloco associado à entrada do diretório que contém o tag coincidente. Este bloco é, então, selecionado, e o byte dentro deste bloco é finalmente acessado.

2.2 Política de Atualização da Memória Cache

Para manter coerência entre cache e memória principal, políticas de escrita, como *write-through* ou *write-back,* devem ser utilizadas para atualizações. A política *write-through* realiza para todo ciclo de trabalho uma escrita na cache e na memória principal, simultaneamente. Sua desvantagem é que o ciclo de escrita passa a ser mais lento que o ciclo de leitura. Já a política de *write-back* realiza a escrita apenas na cache. O bloco onde a palavra foi escrita é marcada como *alterada*. Quando o bloco for removido da cache, todo ele é atualizado na memória principal. Sua desvantagem é um maior tráfego entre memória principal e cache.

Nas políticas de alocação em escrita, se o item sendo escrito não está na cache, é porque existem duas possibilidades: ou o item é trazido para a cache para então ser atualizado (seguindo a política adotada para atualização), ou o item é atualizado apenas na memória principal, não sendo trazido para a cache.

2.3 Memória Virtual

A memória principal disponível em um computador é, em geral, bem menor do que o tamanho máximo de memória permitido pelo processador. O esquema de memória virtual foi criado para permitir a execução de programas cujas exigências, quanto ao tamanho da memória, sejam maiores do que a capacidade de memória instalada no sistema.

Em um sistema sem memória virtual, o endereço gerado pelo programa em execução é o próprio endereço usado para acessar a memória principal. O mesmo não acontece em um sistema com memória virtual. O endereço gerado pelo programa, ou **endereço virtual**, é diferente do **endereço real,** usado para acessar a memória principal. Os possíveis endereços virtuais que podem ser gerados pelo programa formam o **espaço de endereçamento virtual**, enquanto os endereços na memória principal constituem o **espaço de endereçamento real**. Sob o ponto de vista de um programa, a memória disponível é aquela representada pelo espaço de endereçamento virtual. O espaço de endereçamento virtual visto e utilizado pelo programa, pode ser bem maior do que o espaço de endereçamento real, retirando do programa as limitações impostas pela capacidade da memória física de fato existente no sistema.

É importante perceber que o espaço de endereçamento virtual é uma abstração. Embora, sob o ponto de vista do programa, as instruções e dados estejam armazenados dentro do espaço de endereçamento virtual, na realidade eles continuam na memória principal, representada pelo espaço de endereçamento real. Esta distinção, entre endereços e espaços de endereçamento, exige um mecanismo que faça a correspondência entre o endereço virtual, gerado pelo programa, e o endereço real, que é usado para acessar a memória principal.

A técnica de memória virtual permite, ainda, que as instruções e os dados do programa que se encontram no espaço virtual, não estejam necessariamente presentes na memória principal, no momento em que são referenciados. Assim, além do mapeamento anteriormente mencionado, é necessário um mecanismo para o carregamento automático na memória principal das instruções e dados, que são referenciados pelo programa dentro da sua memória virtual, e que não se encontram presentes na memória física.

O mapeamento entre endereços virtuais e reais é realizado por um componente do sub-sistema de memória, denominado **tradutor dinâmico de endereços**, ou DAT (*Dynamic Address Translator*). Para realizar este trabalho, o DAT utiliza uma **tabela de mapeamento,** localizada na memória principal. Esta tabela de mapeamento permanece na memória principal, durante a execução do programa. Ao receber um endereço virtual, o DAT usa este endereço para indexar a tabela de mapeamento. A entrada indexada contém o endereço real, correspondente ao endereço virtual.

Este tipo de mapeamento não é feito no nível de cada locação de memória, pois isto exigiria uma tabela de mapeamento com um número de entradas igual ao tamanho do espaço de endereçamento virtual. Para manter um tamanho de tabela aceitável, o mapeamento é feito no nível de blocos. Cada entrada na tabela de mapeamento contém o endereço-base de um bloco, ou seja, o

endereço real, a partir do qual o bloco está armazenado na memória principal. Em uma **memória virtual paginada**, os blocos são chamados **páginas** e possuem **tamanho fixo**. Em uma **memória virtual segmentada**, os blocos são chamados **segmentos**, e podem ter **tamanhos variáveis**.

Na paginação (Figura 4), o espaço de endereçamento lógico de um processo pode ser *não contínuo*, alocando-se blocos para a memória física, sempre que existir espaço disponível. A paginação divide a memória física em partes de tamanho fixo, chamadas de *blocos* (*frames*), e divide a memória lógica em partes do mesmo tamanho, chamadas de *páginas*. A paginação mantém controle de todos os *blocos* livres e, para executar um programa com n *páginas*, necessitam encontrar n *blocos* livres para carregar o programa.

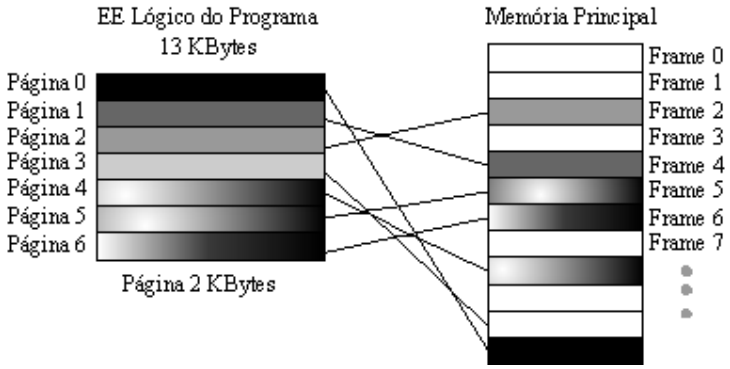

Figura 4 – Memória Paginada.

Define-se uma Tabela de Páginas (page table) para traduzir o endereço lógico em físico, e sua fragmentação é interna. A Figura 5 mostra um esquema de tradução de endereço. O endereço gerado pela CPU é divido em:

- Número da Página (p) – usada como um índice em uma tabela de páginas que contém o endereço base de cada página na memória física;
- Posição na Página (d) – combinado com o endereço base para definir o endereço de memória que é enviado a unidade de memória.

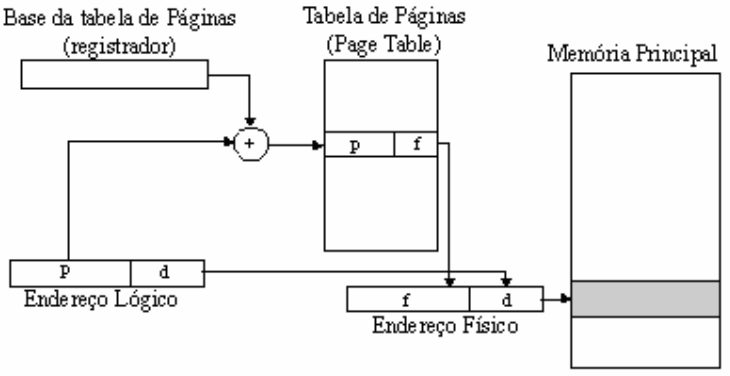

Figura 5 – Conversão de endereços na paginação.

A tabela de páginas contém uma proteção de memória que é implementada através de bits de proteção, associados a cada bloco. Se o bit está "válido", indica que a página associada está no espaço de endereçamento lógico do processo e, portanto, o acesso é legal, mas se está "inválido", isto indica que a página não está no espaço de endereçamento lógico do processo.

Na segmentação (Figura 6), um programa é uma coleção de segmentos, e um segmento é uma unidade lógica, como, por exemplo, programa principal, função, variáveis locais, variáveis globais, etc...

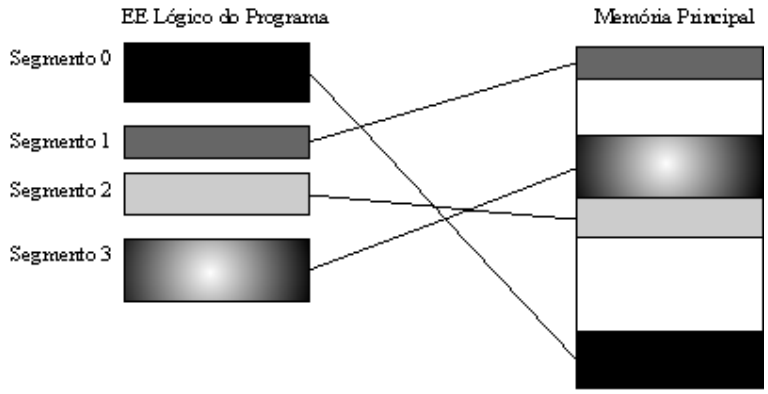

Figura 6 – Memória Segmentada.

A Tabela de Segmentos (segment table), mapeia endereços físicos bidimensionais. Cada entrada na tabela possui uma base que contem o endereço físico inicial, no qual o segmento reside na memória e, o limite,onde é especificado o tamanho do segmento. A Figura 7 mostra um esquema de tradução de endereço. O endereço gerado pela CPU é divido em:

- Número do Segmento (s) – usada como um índice em uma tabela de segmentos que encontram a base de um segmento na memória e verificam se o deslocamento requisitado se encontra dentro do limite máximo do segmento;
- Posição no Segmento (d) – usado para definir o endereço de memória que é enviado a unidade de memória.

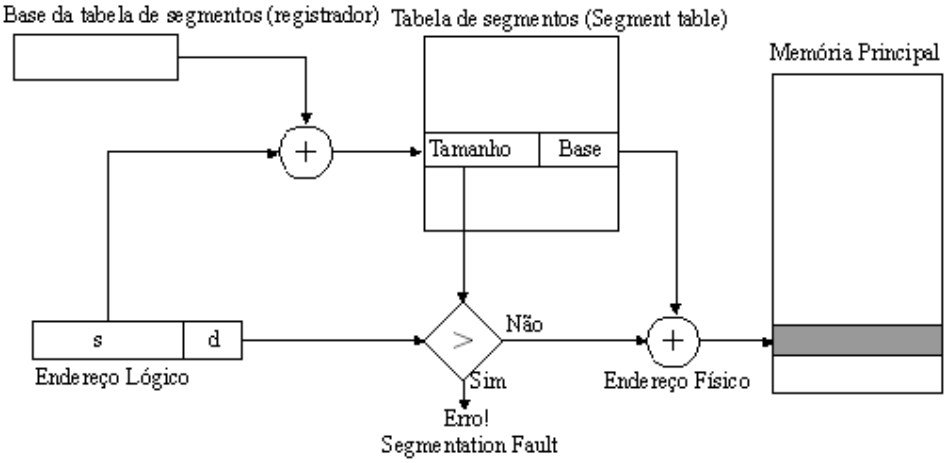

Figura 7 – Conversão de endereços na segmentação.

A realocação é dinâmica, e por tabela de segmento. Contém segmentos compartilhados com o mesmo número de segmento. Possui fragmentação externa. Ela tem proteção, e com cada entrada na tabela de segmento é associado um bit de validação, e tem privilégios de leitura/escrita/execução. Os bits de proteção são associados com segmentos, isto é, o compartilhamento de código ocorre em nível de segmento. Uma vez que segmentos variam em tamanho, a alocação de memória é um problema dinâmico.

3 Protocolos de Coerência de Cache

As soluções para o problema de coerência de cache podem ser divididas em duas categorias: *software* e *hardware*.

As soluções por software geralmente se baseiam em uma análise dos códigos dos programas pelo compilador. O compilador separa os itens de dados em dois grupos: *cachable* (que podem ser armazenados em cache), e *non-cachable* (não podem ser armazenados em cache). O sistema operacional tenta evitar que os dados *non-cachable* sejam copiados para a cache, mantendo-os sempre na memória principal. Isso pode levar a uma utilização ineficiente da cache, visto que alguns dados compartilhados podem ser de uso exclusivo de um processador, em algum intervalo de tempo. Outras técnicas permitem uma análise mais detalhada dos códigos, para determinar períodos livres para variáveis compartilhadas. O compilador, então, insere instruções ao código gerado para forçar a coerência de cache em períodos críticos [GOO89]. As abordagens em software tentam evitar a necessidade de um *hardware* adicional, deixando para o compilador ou o sistema operacional tratar o problema, mas, por serem efetuadas em tempo de compilação, tendem a tomar decisões que não aproveitam a cache da melhor maneira possível [STA03].

As soluções em hardware são as mais utilizadas em sistemas multiprocessadores comerciais, e são conhecidas como **Protocolos de Coerência de Cache.** Estes devem ser capazes de detectar acessos incoerentes à memória, e garantir a coerência dos blocos da cache (invalidando ou atualizando o bloco), em tempo de execução [LIL93]. Os protocolos de coerência de cache se diferem, principalmente, na maneira com que eles distribuem informações a respeito de escritas da memória aos outros processadores, no sistema. Segundo essas características, os esquemas de coerência de cache por hardware podem ser divididos em dois grupos: ***Directory Protocols*** e ***Snoopy Protocols***. Os Directory Protocols mantém as informações de cópias e compartilhamento dos blocos da cache de todos os processadores centralizado em um local, chamado diretório. Já os Snoopy Protocols são implementados nos controladores de cada cache, e ficam monitorando o barramento, para saber se eles possuem alguma cópia do bloco solicitado. Assim, as informações de compartilhamento do bloco são mantidas em cada cache.

Neste Capítulo, apresentam-se os conceitos básicos de alguns protocolos de coerência de cache.

3.1 Directory Protocols

Estes protocolos armazenam informações a respeito de onde as cópias dos blocos estão localizadas. A responsabilidade para manutenção da coerência neste grupo de protocolos é predominantemente delegada a um controlador central, que é normalmente uma parte do controlador da memória principal. Um **diretório**, geralmente localizado na memória principal, armazena informações do estado global com relação ao conteúdo das várias caches locais. Quando

ocorre uma requisição dos controladores da cache local, o controlador central analisa o diretório e fornece os comandos necessários para transferir dados entre memória e caches, ou entre caches. Ele é também responsável por manter o estado da informação atualizado, e cada ação local que possa afetar o estado global de um bloco deve ser relatada ao controlador central.

Os esquemas baseados em diretório se apresentam, basicamente, em três categorias possíveis: diretórios com mapeamento completo, diretórios limitados e, diretórios encadeados. Nas soluções clássicas (diretórios com mapeamento completo), cada entrada de diretório tem um bit por processador, mais um bit de dirty. Um exemplo deste esquema de diretório pode ser encontrado em [GOO89]. Em diretórios limitados, existe um número fixo de bits para indicar cópias em processadores, restringindo o número de cópias simultâneas de qualquer bloco. Em diretórios encadeados, um mapeamento completo é emulado, distribuindo o diretório entre as caches [CHA90].

O diretório pode ser centralizado, juntamente com uma memória compartilhada centralizada, ou também pode ser distribuído, assim como a memória. Neste caso, o diretório não se torna um gargalo, permitindo que acessos a diferentes entradas de diretórios possam ser realizados de forma distribuída [HEN03]. Esses diretórios distribuídos garantem escalabilidade. Entradas de tal diretório são organizadas na forma de listas simples, ou duplamente ligadas, onde todas as caches que compartilham o mesmo bloco são acrescentadas a uma lista. Comandos do controlador da cache, fornecidos ao início da lista, percorrem a mesma por meio de ponteiros.

Em [HEN03], um exemplo simples de protocolo de diretório armazenado de forma distribuída é apresentado, o qual implementa as duas operações citadas anteriormente: falha de leitura da cache e gravação de um bloco da cache. Para implementar este protocolo, são mantidos três estados para cada bloco da cache:

- *Compartilhado* - o bloco na memória está atualizado e um ou mais processadores possuem em sua cache tal bloco;
- *Não inserido na cache* - nenhum processador tem uma cópia do bloco da cache;
- *Exclusivo* - o bloco na memória está desatualizado e somente um processador tem a cópia do bloco na cache (proprietário).

Quando o bloco está no estado compartilhado e ocorrer uma falha de leitura, o processador solicitante recebe os dados da memória e é adicionado ao conjunto de compartilhadores do bloco. Se houver uma falha de escrita, todos os processadores recebem mensagens para invalidarem seus blocos de cache, e o estado do bloco se torna exclusivo, indicando no conjunto de compartilhadores que o único processador a possuir uma cópia foi o que realizou a solicitação.

Quando um bloco está no estado não inserido na cache e ocorrer uma falha de leitura, o estado do bloco se torna compartilhado, e a cache local recebe os dados da memória. Caso ocorra uma falha de escrita, o processador solicitante se torna proprietário, definindo o estado do bloco como exclusivo. O conjunto de compartilhadores do bloco reflete esse estado, mantendo indicação de cópia de bloco somente para esse processador.

Quando o bloco está no estado exclusivo e ocorrer uma falha de leitura, o processador proprietário recebe uma mensagem de busca de dados, mudando o estado do bloco para compartilhado, e fornecendo ao solicitante o bloco desejado. É adicionado ao conjunto de compartilhadores o processador solicitante.

Ao substituir um bloco, o processador proprietário deverá realizar write-back de dados, atualizando a cópia em memória, e removendo o processador do conjunto de compartilhadores de tal bloco. Em falhas de escrita, uma mensagem é enviada ao antigo proprietário, invalidando seu bloco na cache. Em seguida, o conjunto de compartilhadores mantém somente o processador solicitante, mantendo o estado do bloco como exclusivo para ele.

Os *directory protocols* apresentam as desvantagens causadas pelo uso de um ponto central e pelo *overhead* de comunicação entre os vários controladores de cache e controlador central. Entretanto, esses sistemas são mais escaláveis, suportando um número maior de processadores, e envolvendo barramentos múltiplos ou redes de interconexão.

3.2 Snoopy Protocols

Diferente do caso anterior, que é caracterizado por um método centralizado, os *snoopy protocols* utilizam distribuição para tratar coerência de cache. Eles são baseados nas ações de controladores de cache local, e suas informações de estado local sobre o dado na cache. Subseqüentemente, todas as ações com o atual bloco compartilhado devem ser anunciadas a todas as outras caches através de *broadcast*. Controladores de cache local são capazes de "espionar" a rede, e são capazes de reconhecer as ações e condições que determinem alguma reação, de acordo com o protocolo utilizado para preservar a coerência.

Snoopy protocols são ideais para multiprocessadores baseados em barramento, pelo fato de que o barramento compartilhado favorece o *broadcast*. Entretanto, as transmissões de mensagens para coerência no barramento compartilhado causam um tráfego adicional. Conseqüentemente, apenas um sistema com um número médio ou pequeno de multiprocessadores pode ser suportado com *snoopy protocols*.

Existem dois protocolos de monitoração básicos, chamados ***write-invalidate*** e ***write-update*** (ou *write-broadcast*). No protocolo *Write-invalidate* (Invalidação), a cada escrita realizada em um bloco de uma cache de um processador, uma mensagem é enviada pelo barramento, com o objetivo de invalidar todas as cópias presentes nos demais processadores. Assim, esse processador fica com acesso exclusivo ao bloco da cache para efetuar operações de escrita, dando a idéia de leitores-escritor, ou seja, podem haver vários leitores para um bloco da cache, mas apenas um escritor. No caso do protocolo *Write-update* (Atualização), ao ser realizada uma escrita em um bloco da cache por um processador, ela é atualizada em todas as cópias presentes nas caches dos outros processadores.

Em geral, o mais eficiente desses protocolos depende do padrão de leituras e escritas da

memória [STA03]. A abordagem de invalidação é a mais usada nos sistemas multiprocessados atuais, pois sugere menor tráfego no barramento. Mas nenhum dos dois métodos é capaz de melhorar a performance para todos os tipos de cargas de trabalho. Por esse motivo, alguns protocolos propostos combinam as duas políticas. Eles iniciam com *broadcast* de escritas, mas quando uma longa seqüência de escritas locais é encontrada ou prevista, o sinal de invalidação do bloco é enviado. Estas soluções procuram adaptar os esquemas de coerência de maneira a melhorar o desempenho.

3.2.1 Write-Once

Historicamente é o primeiro protocolo de invalidação que foi proposto na literatura [GOO83] (Figura 8). Nesse esquema, os blocos em uma cache podem estar em um de quatro estados: *Invalid*, *Valid,* Reserved e Dirty.

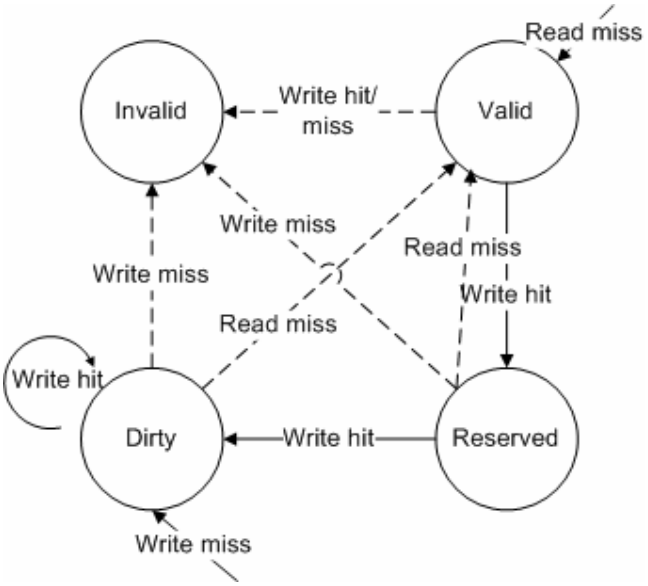

Figura 8 – Protocolo Write-Once.

Em um read miss, se a informação solicitada está em um bloco em outro processador que esteja no estado Dirty, este processador fornece o bloco solicitado, realizando um write-back para a memória. Caso nenhum bloco esteja no estado Dirty, então o bloco vem da memória principal. Todas as caches com uma cópia do bloco definem seu estado como *Valid*.

Em um write hit, caso o estado do bloco seja Dirty, a escrita é realizada sem alteração de estado. Se o estado for Reserved, então o estado do bloco é mudado para Dirty. Se o estado for *Valid*, o bloco é gravado na memória por write-through e seu estado mudado para Reserved. O estado *Valid* quer dizer que é um dado compartilhado, e esta informação deve ser colocado no barramento para as outras cópias serem invalidadas.

Em um write miss, o bloco é carregado da memória ou do bloco de outra cache cujo estado seja Dirty. Uma vez que o bloco foi carregado, a escrita é realizada, e o estado é modificado para

Dirty. Todas as suas cópias nas outras caches vão para o estado *Invalid*.

3.2.2 Berkeley

Desenvolvido pela Universidade da Califórnia para ser aplicado em uma máquina RISC multiprocessada, esse esquema (Figura 9), usa transferências diretas entre caches [KAT85]. Este protocolo de coerência de cache usa a idéia de linha de cache proprietária. Em qualquer momento, um bloco pode estar de forma exclusiva em apenas uma das caches, ou na memória. Existem quatro estados: *Invalid*, *Valid*, *Shared Dirty* e *Private Dirty*. Quando um bloco é compartilhado, apenas o proprietário contém o bloco com a linha na cache no estado *Shared Dirty*; todos os outros têm a linha no estado *Valid*. Portanto, uma linha de cache pode ser *Shared Dirty* ou *Private Dirty* em apenas uma cache (a proprietária). Blocos no estado Shared Dirty e Private Dirty, quando escolhidos para substituição na cache, são gravados na memória principal por write-back.

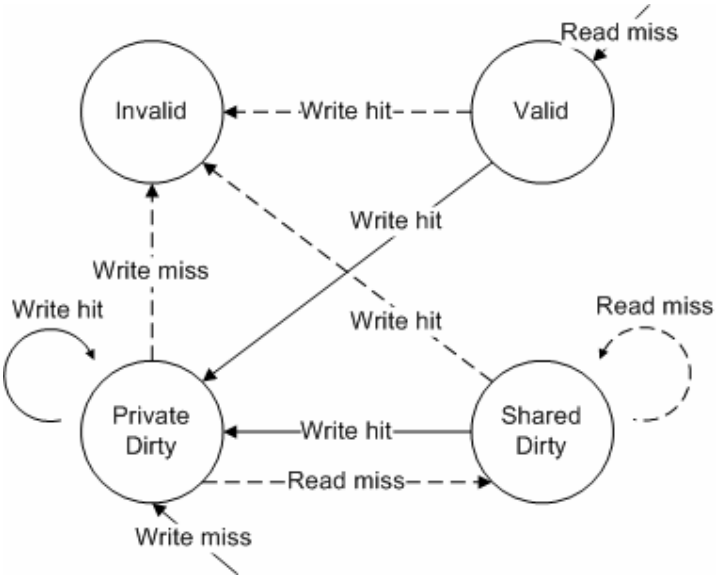

Figura 9 – Protocolo Berkeley.

Em um read *miss*, a cache que tem o bloco nos estados *Shared Dirty* ou *Private Dirty* transfere diretamente o bloco para a cache que fez o pedido, mudando seu estado para Shared Dirty. Se nenhuma cache possui o bloco ou ele está em qualquer outro estado, então o bloco vem da memória principal. O bloco da cache que fez o pedido tem seu estado alterado para *Valid*.

Em um *write hit*, se o estado do bloco é Private Dirty, então o estado não é alterado. Se o estado é *Valid* ou Shared Dirty, o estado é alterado para Private Dirty e um broadcast é feito pelo barramento para que todas as outras caches possam invalidar suas cópias.

Em um *write miss*, o bloco é colocado no estado Private Dirty e se torna a proprietária, e todas as cópias em outras caches são *Invalid*.

3.2.3 Illinois

Este método [PAP84] (Figura 10), se baseia no fato de que falhas de acesso a blocos podem ser tratadas procurando por blocos em outras caches, ou na memória principal, e também assume que a cache que requisitou pode determinar a origem do bloco. A melhora de desempenho proposta por esse método é que ele percebe que invalidações para acertos de escrita não são necessárias em blocos privados não modificados, pois é possível determinar, a partir dos estados, se o bloco é ou não compartilhado. Assim como os anteriores, também são usados quatro estados: *Invalid*, *Read Private (Exclusive)*, *Shared* e *Dirty*. Também é utilizada a abordagem de write-back, mas somente quando o bloco está no estado Dirty.

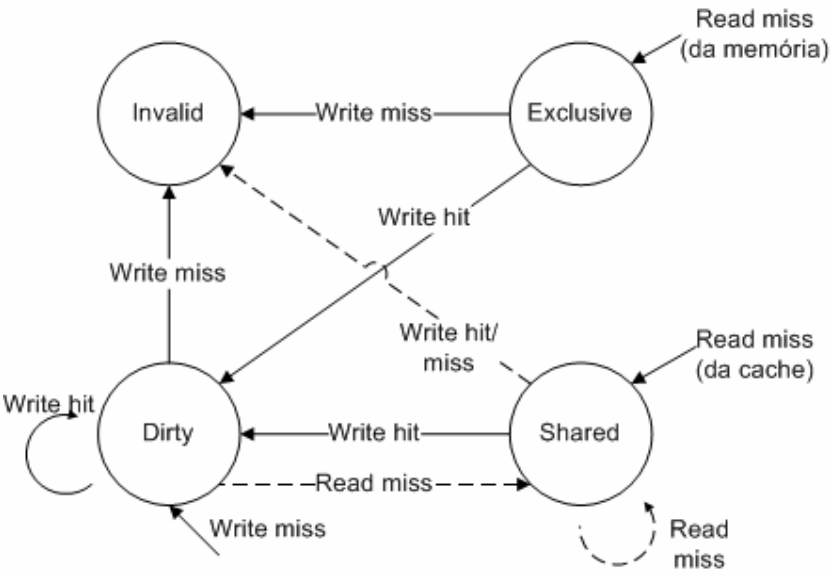

Figura 10 – Protocolo Illinois.

Em um *read miss*, a cache com maior prioridade que tenha uma cópia, vai colocar o bloco no barramento, e gravar o bloco na memória principal, se o estado for Dirty. Todas as caches que possuem uma cópia do bloco mudam seu estado para *Shared*. Se o bloco vir da memória principal, então nenhuma outra cache tem o bloco, e o estado é definido como *Exclusive*.

Em um *write hit*, um bloco Dirty é escrito sem transições, e um bloco *Exclusive* é mudado para Dirty. Se o bloco é *Shared*, então é necessário invalidar todas as cópias existentes, e o bloco gravado também é definido como Dirty.

Em um *write miss*, todas as caches com cópias vão para *Invalid,* e o bloco é carregado como Dirty.

3.2.4 Firefly

É o esquema usado no workstation Firefly [THA84]. Este protocolo (Figura 11) apresenta quatro estados, mas utiliza apenas três: *Read Private (Exclusive), Shared,* e *Dirty*. O estado *Invalid* é utilizado apenas para uma condição inicial para a linha da cache. Por ser um protocolo de

atualização, e não de invalidação, como os anteriores, escritas na cache geram *broadcasts* e escritas à memória, ou seja, todas as outras caches compartilhando o bloco "espionam" o barramento e numa escrita atualizam suas cópias, ao invés de invalidá-las. Dessa forma, nenhuma linha da cache será inválida depois de ser carregada. Existe ainda uma linha de barramento especial chamada *SharedLine,* que é utilizada para indicar que outras caches estão compartilhando o bloco.

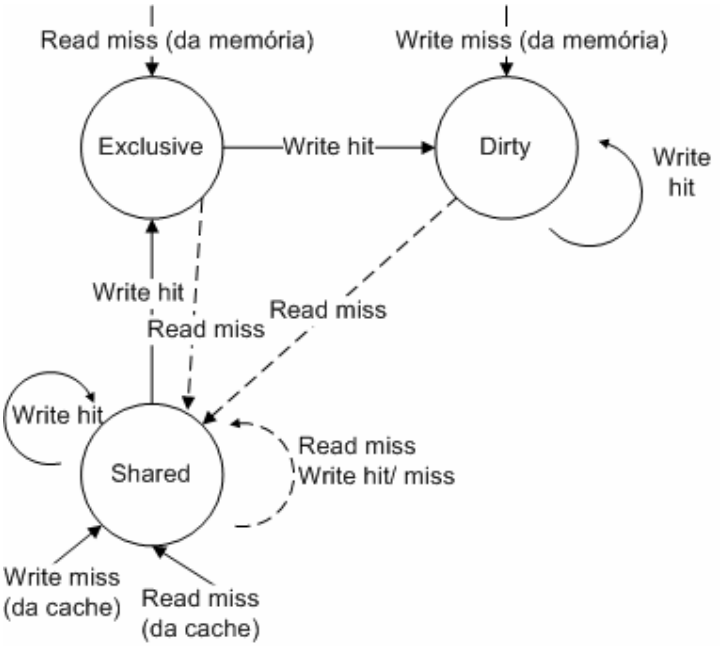

Figura 11 – Protocolo Firefly.

Em um *read miss*, se qualquer outra cache tem uma cópia do bloco, ela comunica a cache requisitante pela *SharedLine* e seus estados passam a ser *Shared* e, caso o bloco seja *Dirty,* ele é gravado na memória principal. Se nenhuma cache possui o bloco, este vem da memória principal e é definido para ficar no estado *Exclusive*.

Em um *write hit*, nenhuma alteração de estado é realizada caso o estado seja Dirty. Se o estado for *Exclusive*, então é modificado para Dirty. Quando o estado for *Shared* é que aparece a principal modificação desse protocolo: como pode haver cópias em outras caches e não há invalidações, então, todas as outras caches compartilhando o dado, "pegam" o dado do barramento e atualizam suas cópias. Além disso, as caches ativam o SharedLine, para que a cache requisitante verifique se ainda existem caches compartilhando o bloco: se essa linha estiver com sinal alto, o compartilhamento existe, e então o estado é mantido em *Shared*. Caso contrário, não há cópias, e o estado pode ser definido como *Exclusive*.

Em um *write miss*, se nenhuma cache possui o bloco, este é carregado da memória no estado Dirty, e depois escrito. Se alguma cache o possui, então o bloco é carregado no estado *Shared,* e a palavra é escrita na memória, sendo que os blocos que possuem uma cópia são atualizados.

3.2.5 MESI

No protocolo MESI [STA03] (Figura 12), largamente utilizado em sistemas multiprocessadores comerciais, tais como Pentium e PowerPC, um bloco pode ter quatro estados que são chamados de protocolos MESI, devido aos seus estados: *Modified, Exclusive, Shared e Invalid*.

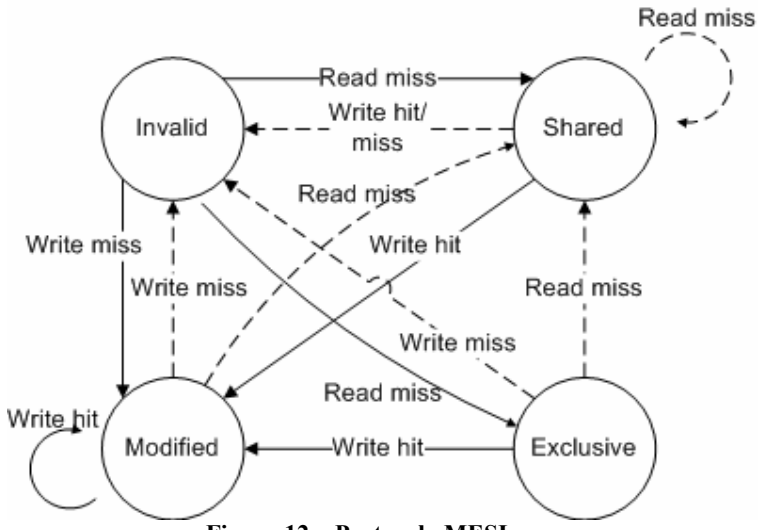

Figura 12 – Protocolo MESI.

Em um *read miss*, se alguma cache tem o bloco no estado *Modified*, então ela fornece o bloco para a cache que o requisitou, e ambos definem seu estado para *Shared*. Caso alguma cache tenha o bloco no estado *Shared* ou *Exclusive*, então a cache que fez o pedido faz a leitura do bloco da memória principal, e todas as caches com cópias mudam seu status para *Shared*. Se nenhuma cache possui o bloco requisitado, então o processador carrega o bloco da memória principal, e define seu estado como *Exclusive*.

Em um *write hit*, caso o estado do bloco seja *Modified*, nenhuma transição é necessária. Se o estado for *Exclusive*, então é alterado para *Modified*. Caso o estado do bloco seja *Shared*, é necessário invalidar outras cópias existentes. O bloco passa de *Shared* para *Modified*.

Em um *write miss*, o bloco é carregado da memória principal, caso não existam cópias, e marcado como *Modified,* e logo após é realizada a escrita. Se alguma cache possui uma cópia do bloco, ela escreve no barramento, e marca seu estado para *Invalid*. O processador da cache que fez a requisição lê esta linha, realiza sua escrita, e marca como Modified. Todas as outras cópias existentes são invalidadas.

4 Arquitetura Proposta

A Figura 13 apresenta a infra-estrutura de hardware originalmente integrada no trabalho de [WOZ06]. Esta infra-estrutura possui os seguintes componentes:

1. **Rede intra-chip (NoC) HERMES** [MOR04]: realiza a interconexão dos núcleos (Processadores Plasma) e o roteamento de pacotes entre os mesmos. A rede é representada pelo conjunto de roteadores (círculos com a letra *R* na Figura).

2. **Plasma**: nodo processador que executa as aplicações do MPSoC. O nodo *Plasma* possui uma CPU com arquitetura compatível com a arquitetura MIPS. Cada CPU possui uma memória local, a qual não é acessível aos outros processadores. O *Plasma mestre* tem por função principal ler da memória *repositório de tarefas* as aplicações a serem executadas, e transferi-las para os processadores *Plasma escravos.*

3. **NI (Network Interface)**: faz a interface entre o processador e a rede.

4. **DMA (Direct Memory Access)**: transfere para a memória do processador o código-objeto das tarefas enviadas por um nodo mestre.

Os dois últimos componentes, NI e DMA, encontram-se dentro do módulo Plasma (Figura 14).

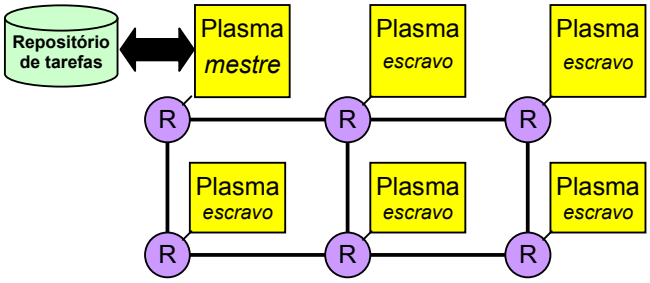

Figura 13 – Arquitetura Original.

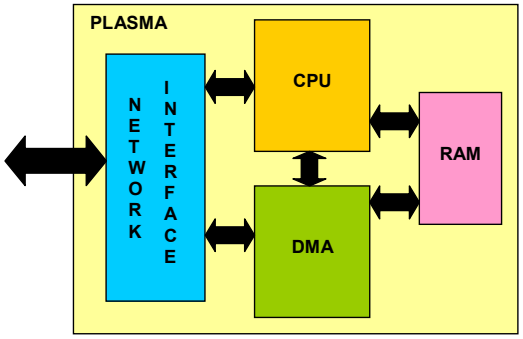

Figura 14 – Visão interna do nodo processador Plasma.

O presente Trabalho de Conclusão de Curso tem por ***objetivo*** estender a arquitetura original, adicionando memórias cache locais nos processadores Plasma e uma memória central compartilhada. A Figura 15 apresenta a arquitetura modificada, havendo a substituição de um processador plasma por uma memória central (módulo *Memória* na Figura).

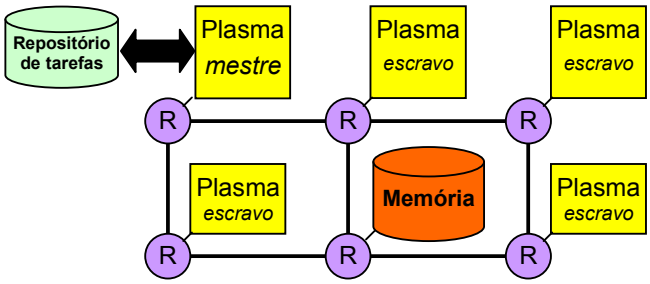

Figura 15 – Arquitetura Proposta.

O desenvolvimento desta plataforma é divido em projeto de hardware e software.

O projeto de hardware compreende reuso de propriedade intelectual (processador Plasma e o MPSoC), desenvolvimento de uma memória compartilhada (Figura 16) e inclusão da memória cache e seu controlador no nodo de processamento Plasma (Figura 17). O nodo com memória compartilhada (Figura 16) deverá ter implementado em seu controlador as políticas de coerência e diretórios. Planeja-se utilizar um controlador de cache centralizado, baseado nos protocolos de diretórios, descrito em VHDL.

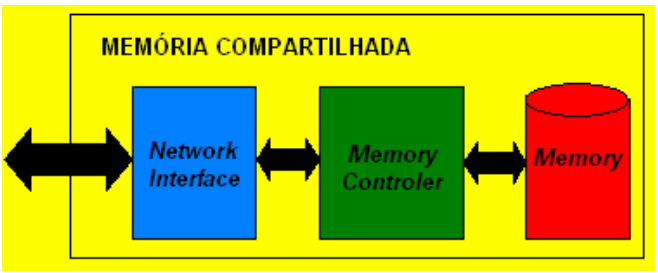

Figura 16 – Arquitetura da Memória Compartilhada.

A Figura 17 apresenta o módulo Plasma modificado, com inclusão do controlador de memória cache e a memória cache.

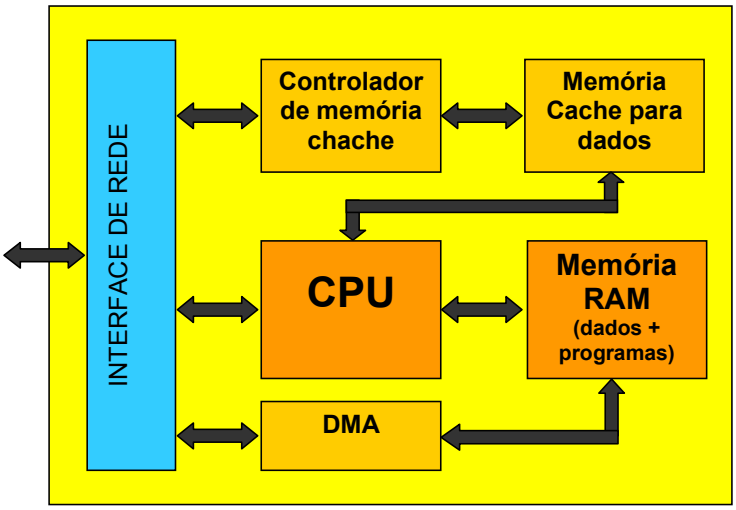

Figura 17 – Arquitetura Plasma com Memória Cache.

Dada a exigüidade do prazo para a realização do trabalho de conclusão, o projeto não será prototipado em *FPGA* (*Field-Programmable Gate Array*), mas sim executadas simulações sobre a descrição VHDL do sistema em programas adequados, a fim de validar a arquitetura.

No projeto de software serão desenvolvidos os programas que estarão sendo executados nos processadores Plasma, bem como drivers, para configuração. O processador Plasma Mestre, estará executando algoritmos diferentes dos outros, pois este tem a responsabilidade de gerenciar a rede, enquanto que os outros terão a função de processar as tarefas.

Há a possibilidade do controlador de cache ter implementações em software, dependendo este, da implementação escolhida, mas, com certeza, terão que ser acrescentados comandos no micro-kernel, para suportar a memória compartilhada e também as memórias caches.

5 Cronograma de Atividades

A Tabela 1 apresenta o cronograma das atividades que serão desenvolvidas durante o presente projeto de pesquisa.

Tabela 1 – Cronograma das atividades.

Atv.	Tarefa a ser desenvolvida	Março				Abril					Maio				Junho				Julho	
		1	2	3	4	1	2	3	4	5	1	2	3	4	1	2	3	4	1	2
1	Escrita da proposta de TC																			
2	Estudo do MPSoC																			
3	Software/hardware para acesso à MP																			
4	Inserção da memória compartilhada na arquitetura MPSoC																			
5	Inserção da memória de dados local ao processador																			
6	Controlador da memória cache																			
7	Avaliação do controlador de cache																			
8	Coerência de cache																			
9	Escrita do volume final de TC																			
10	Apresentação do TC																			

Atividade 1:

- Escrita da proposta de TC. Escrita da presente proposta de trabalho de conclusão.

  Resultados esperados: Ao final desta atividade obtém-se o texto a ser entregue ao avaliador.

Atividade 2:

- Estudo do MPSoC desenvolvido por Cristiane Raquel Woszezenki [WOS06]. Nesta atividade buscar-se-á compreender os passos de instalação do ambiente do software, necessário à execução do MPSoC, análise de simulações e sinais internos, assim como uma compreensão geral do sistema.

  Resultados esperados: Ao final desta atividade, espera-se obter conhecimento, tanto da estrutura do MPSoC, como suas interfaces, para realizar o trabalho proposto.

Atividade 3:

- Inclusão de mecanismos de software e hardware para acesso a uma memória externa, denominada Memória Principal (MP). Deverão ser implementadas as rotinas read_global e write_global, as quais se comunicarão com o wrapper do processador para a comunicação com a NoC através de pacotes.

  Resultados esperados: observação dos pacotes de leitura/escrita, chegando no nodo da rede que conterá a MP.

Atividade 4:

- Inserção da memória compartilhada na arquitetura MPSoC. O wrapper desta memória deve interpretar os pacotes oriundos da rede, conforme definidos na tarefa precedente. No caso de

escrita, deve-se escrever na memória os dados contidos no pacote de escrita e, no caso de leitura, deve-se enviar ao processador que realizou a requisição um pacote com os dados lidos a partir do endereço solicitado, na quantidade também definida no pacote de leitura.

Resultados esperados: Ao final desta atividade, espera-se obter um sistema com suporte à uma memória compartilhada, necessário para realizar o trabalho proposto.

Atividade 5:

- Inclusão de uma memória local de dados no processador. Deverão ser implementadas as rotinas read_local e write_local, permitindo ao processador ler e escrever nesta memória local. Esta atividade é simplificada, dado que na arquitetura desenvolvida em [WOS06] o processador mestre já possui uma instância de memória local.

  Resultados esperados: Comunicação entre o processador e a memória local.

Atividade 6:

- Controlador de cache. Adaptação da memória de dados desenvolvida na etapa anterior para memória cache. Deverão ser implementadas as tabelas de tradução de endereços, política de mapeamento, política de escrita, interface com a NoC, dentre outros módulos. O controlador de memória cache será implementado integralmente em hardware (VHDL), dado o desempenho requerido (leitura em pelo menos 2 ciclos de clock).

  Resultados esperados: validação funcional simples das ações de read miss, read hit, write miss (escrita em um dado cujo bloco não esteja na cache), e write hit (escrita em dado presente na cache com a correspondente política de escrita).

Atividade 7:

- Avaliação do controlador de cache. Avaliar com diferentes programas o desempenho da cache (porcentagens de hit/miss). Opcionalmente, avaliar diferentes estratégias de mapeamento e escrita. O tamanho da cache e dos blocos deve ser um parâmetro a ser incluído na avaliação.

  Resultados esperados: dados quantitativos referentes ao desempenho do sistema.

Atividade 8:

- Coerência de cache. Implantação da política de coerência de cache, baseada em diretórios. A escolha desta política deve-se à natureza distribuída dos processadores.

  Resultados esperados: escritas coerentes de dados quando os mesmos estiverem distribuídos entre diversas caches.

Atividade 9:

- Escrita do volume final: escrita do volume que será entregue à banca de avaliação do trabalho de conclusão.

  Resultados esperados: Ao final desta atividade espera-se obter o volume final a ser entregue à banca avaliadora.

Atividade 10:

- Apresentação do TC: apresentação do trabalho de conclusão desenvolvido ao decorrer do semestre.

  Resultados esperados: aprovação no TC.

6 Recursos Necessários

Todos os recursos necessários existem no GAPH (Grupo de Apoio ao Projeto de Hardware), prédio 32, salas 726/727.

6.1 Recursos de Hardware

- Dois computadores Pentium 4 2.4 Ghz (compatível ou superior);
- Estação de trabalho.

6.2 Recursos de Software

- Sistema Operacional Windows XP;
- Sistema Operacional Linux;
- Active-HDL;
- ModelSim;
- Leonardo Spectrum;
- ISE 8.1;
- Cygwin.

6.3 Fontes de Pesquisa

- Biblioteca Central;
- Internet (acesso ao portal de períodos CAPES).

7 Referências Bibliográficas

[CHA90] Chaiken, D.; Fields, C.; Kurihara, K.; Agarwal, A. "Directory-based cache coherence in large-scale multiprocessors". Computer, vol. 23, issue 6, Jun 1990, pp.49-58.

[DAE06] Daewook, K.; Manho, K.; Sobelman, G. E. "DCOS: Cache embedded switch architecture for distributed shared memory multiprocessor SoCs". In ISCA '06, 2006, pp. 979-982.

[GOO83] Goodman, J. R. "Using cache memory to reduce processor-memory traffic". In ISCA '83: *Proceeding of the 10th annual international symposium on Computer Architecture*, 1983, pp. 124-131.

[GOO89] Goor, A. J. V. de. "Computer Architecture and Design". Addison-Wesley Publishers, 1989.

[HEN03] Hennessy, J. L.; Patterson, A. "Arquitetura de Computadores uma Abordagem Quantitativa". Editora Campus, 2003.

[KAT85] Katz, R. H.; Eggers, S. J.; Wood, D. A.; Perkins, C. L.; Sheldon, R. G. "Implementing a cache consistency protocol". In ISCA '85: *Proceeding of the 12th annual international symposium on Computer Architecture*, 1985, pp. 276-283.

[LIL93] Lilja, D. J. "Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons". ACM Comput. Surv., , vol. 25, issue 3, Set. 1993, pp.303-338.

[MON06] Monchiero, M.; Palermo, G.; Silvano, C.; Villa, O. "Efficient Synchronization for Embedded On-Chip Multiprocessors". *Very Large Scale Integration (VLSI) Systems*, vol.14, issue 10, Out. 2006, pp. 1049-1062.

[MOR04] Moraes, F. et al. "Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip". Integration the VLSI Journal, 38(1), October 2004. pp. 69-93.

[PAP84] Papamarcos, M. S.; Patel, J. H. "A low-overhead coherence solution for multiprocessors with private cache memories". In ISCA '84: *Proceeding of the 11th annual international symposium on Computer Architecture*, 1984, pp. 348-354.

[PET06] Pétrot, F.; Greiner, A.; Gomez, P. "On cache coherency and memory consistency issues in NoC based shared memory multiprocessor SoC architectures". In DSD'06: *Conference on Digital System Design – 9th EUROMICRO*, 2006, pp. 53-60.

[STA03] Stallings, W. "Arquitetura e Organização de Computadores". Prentice Hall, 2003.

[TAE05] Taeweon, S.; Daehyun, K; Lee, H.-H. S. "Cache coherence support for non-shared bus architecture on heterogeneous MPSoCs". DAC, Jun. 2005, pp. 553-558.

[THA84] Thacker, C. "Private Communication". Jul.1984.

[WOS06] Woszezenki, C. "Alocação de Tarefas e Comunicação Entre Tarefas em MPSoCs". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, Dez. 2006.