SISTEMA DE MEMÓRIAS MULTICACHE PARA UMA MÁQUINA PARALELA MIMD

PROJETO MULTIPLUS

Ageu C. Pacheco Jr.[*§]
NÚCLEO DE COMPUTAÇÃO ELETRÔNICA
UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

Alexandre M. Meslin[*‡]
NÚCLEO DE COMPUTAÇÃO ELETRÔNICA
UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

RESUMO

Memórias *cache* são hoje dispositivos essenciais em arquiteturas modernas de computadores. Especialmente em sistemas multiprocessados, onde os meios de comunicação entre processadores e os módulos de memória estão sujeitos a sobrecargas excessivas de transferências, a especificação cuidadosa de um sistema distribuído de *caches* é de fundamental importância para o desempenho final da máquina.

O presente trabalho consiste do estudo e análise de várias opções de projeto de sistemas *multicache* para imediata aplicação no projeto de um computador paralelo *MIMD* de alto desempenho do NCE/UFRJ denominado Projeto MULTIPLUS.

Inicialmente é feita uma breve abordagem dos aspectos clássicos envolvidos em projetos de memória *cache* com especial ênfase à discussão do problema da manutenção da coerência da informação. Em seguida a arquitetura do sistema MULTIPLUS é apresentada. O restante do trabalho é então dedicado à análise e discussão das diversas alternativas possíveis dentro do contexto da arquitetura MULTIPLUS, justificando aquelas adotadas.

ABSTRACT

Today, cache memories are essential devices in many modern computer architectures. Specially in multiprocessor systems, where the communication media between the processors and the memory modules are more susceptible to transfer overloads, a careful specification of a multicache system is of upmost importance to attain the desired levels of performance.

This work presents a study of various design alternatives for a multicache system to be used in the development of the NCE/UFRJ's high performance MIMD parallel computer named Project MULTIPLUS.

In the first part, a brief presentation of the classical aspects involved in cache design is given, with special emphasis on the data coherence problem. The architecture of the MULTIPLUS system is shown afterwards. The remaining of the work is then devoted to the discussion and analysis of the design alternatives under the MULTIPLUS perspective.

---

* Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, CP 2324 - CEP 20001 - Rio de Janeiro - RJ, tel.: 290-3212 R. 328

§ PhD, Analista de Sistema do NCE/UFRJ, Professor Assistente do Departamento de Informática da UFRJ. Principais áreas de interesse: hierarquia de memórias, memórias cache e arquiteturas paralelas em geral.

‡ Engenheiro Eletrônico pela UFRJ, Analista de Sistema do NCE/UFRJ. Principais áreas de interesse: microeletrônica, memórias cache e arquiteturas paralelas.

1 - INTRODUÇÃO

Visando aumentar a capacidade de processamento dos computadores (ainda monoprocessados), observou-se que, mesmo com a evolução da tecnologia, as unidades de processamento (*PE's - Processing Element*) necessitam informações a uma taxa (ciclo de *CPU*) que cresce muito mais rapidamente do que a das memórias convencionais de fornecer estas informações (ciclo de memória). Em consequência, a unidade de processamento permanece muito tempo parada (*idle*). Nestas condições, a memória do sistema é chamada de "gargalo de von Newmann" [STON87].

- Observou-se também que a maioria dos programas passa grande parte do seu tempo de execução em trechos de tamanhos reduzidos (quando comparados ao tamanho do programa principal)[1]. Na tentativa de se aproveitar esta particularidade dos programas, foi introduzido um novo nível de memória, denominada *cache*[2], localizada entre a *CPU* e a memória principal, contendo cópias de algumas posições desta. Suas principais características são: capacidade de armazenamento reduzida (em relação à memória principal) e alta velocidade (5 a 10 vezes mais rápida). O ganho em desempenho é obtido trazendo-se para a *cache* as regiões correntemente acessadas e/ou buscando objetos antes de sua demanda pelo processador. Juntamente com os objetos, são armazenados, em campos distintos, seus identificadores (*tags*) e estados (*flags*) utilizados pelo controlador de *cache*.

Dependendo do número de dispositivos que podem acessar a memória principal através do barramento, o "gargalo", que antes era apenas o tempo de acesso da memória principal, passa a incluir a disputa pelo barramento entre a unidade de processamento e os controladores de entrada e saída. Assim, uma nova justificativa para as memórias *cache* é a redução da taxa de acessos ao barramento pelo processador.

Com o aparecimento dos computadores multiprocessados, o uso de *caches* tornou-se indispensável. Sua importância maior não reside mais no fato de tentar manter a unidade de processamento com taxa de ocupação total, e sim em possibilitar que, com uma taxa de acerto (*hit rate*) a mais próxima possível de 100%, um maior número de *PE's* possam compartilhar de um mesmo barramento. Modernamente, mesmo uma memória *cache* com tempo de acesso da mesma ordem de grandeza da memória principal é justificável, pois funcionaria apenas como uma espécie de *buffer* redutor do tráfego entre processadores e a memória principal [KAPL73].

Este trabalho analisa diversas opções de arquitetura de memórias *caches* para o Sistema MULTIPLUS [AUDE90] e também suas implicações quanto ao tipo de protocolo de barramento e outros aspectos de arquitetura e sistema operacional. A escolha da arquitetura de *cache* é de fundamental importância na arquitetura geral do sistema, principalmente no esquema proposto de barramento duplo.

2 - ALTERNATIVAS DE ARQUITETURAS DE "CACHES" PARA MULTIPROCESSAMENTO

Dada a importância assumida pela utilização de memórias *caches* em sistemas multiprocessados, é lícito hoje afirmar que o desempenho de tais sistemas está intimamente ligado ao bom projeto do sistema de memória *cache*. Idealmente, o projeto deve considerar o tipo de aplicação à qual a máquina

---

[1] Princípio da localidade de referência

[2] Neste trabalho, quando *cache* for referida no feminino, como **a** *cache*, estará se referindo à memória *cache*, e no masculino, como **o** *cache*, ao controlador de memória *cache*.

se destina, para que possa ser identificada a sua carga de trabalho (*workload*) típica. Por outro lado, não é solução aceitável uma *cache* de tamanho muito grande que eleva demais o custo do projeto, além de aumentar a sua complexidade a ponto de torná-lo comercialmente inviável.

Variações mínimas da taxa de acerto da *cache* podem comprometer seriamente o desempenho do sistema. Seja por exemplo, um sistema cujo tempo de acesso à memória principal é 10 vezes maior do que o de acesso à *cache*. Considerando-se apenas ciclos de leitura, o tempo médio de ciclo de *CPU* pode ser expresso pela fórmula:

$$t_{eff} = h * t_{cache} + (1 - h) * t_{main}$$

onde: t_{eff} = tempo eficaz médio de ciclo de *CPU*,
h = taxa de acerto na *cache*,
t_{cache} = tempo de acesso à *cache*,
t_{main} = tempo de acesso à memória principal.

Assim, se houver uma variação da taxa de acerto de 99% para 98% (variação relativa menor do que 1%), o tempo médio de ciclo aumenta em 10%.

Existem dois tipos de configurações básicas de memória *cache* em sistemas multiprocessados: a *cache* privada (*private cache*) (figura 1.a) e a *cache* compartilhada (*shared cache*) [DUBO82] (figura 1.b). A *cache* compartilhada pode ser acessada por mais de um processador. É geralmente um *buffer* de alta velocidade de uma determinada parte da memória. A memória *cache* privada é dedicada a um único processador e pode armazenar blocos de diversas unidades de memória. A desvantagem da *cache* compartilhada em relação à privada é que a primeira não reduz o tráfego nas vias de comunicação entre *CPU* e os módulos de memória [KAPL73].

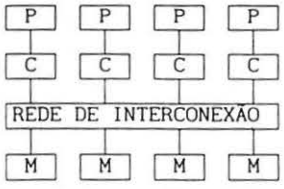


figura 1.a: *cache* privada

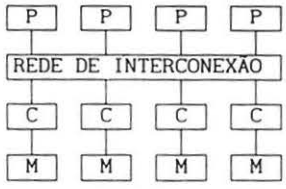


figura 1.b: *cache* compartilhada

2.1 - ARQUITETURAS

Nesta seção são apresentadas algumas características e parâmetros de arquiteturas de controladores e memórias *cache*.

2.1.1 - "CACHE" VIRTUAL X FÍSICO

A arquitetura de gerência de memória possibilita duas alternativas de arquitetura de *cache* com relação ao endereço: *cache* virtual e físico.

Cache virtual: o endereço que o controlador de *cache* recebe é o endereço (virtual) gerado pelo processador. Neste caso não existe o atraso correspondente ao processamento do endereço pela gerência de memória, possibilitando ao controlador de *cache* respostas mais rápidas. Por outro lado, a cada troca de contexto, quando as tabelas de gerência de memória são alteradas, há a necessidade de invalidar a *cache* toda, uma vez que o novo processo poderá gerar endereços virtuais que estejam presentes na *cache* mas não correspondam ao mesmo endereço físico. Por isso mesmo, alguns

controladores de *cache* armazenam o endereço físico juntamente com os dados para evitar a necessidade de invalidar a *cache* a cada troca de processo. Para este tipo de controlador é interessante que a gerência processe o endereço virtual em paralelo com a busca na *cache* e forneça o endereço físico em tempo menor do que o ciclo de *cache*.

Cache Físico: a memória *cache* de endereço físico (real) ou simplesmente *cache* físico recebe o endereço já processado pela gerência de memória.

Para minimizar os efeitos da gerência nas arquiteturas descritas acima, geralmente aproveita-se o fato da gerência de memória não utilizar os *bits* de menor ordem da palavra de endereço e utilizam-se estes para endereçar a *cache*. Em paralelo, a gerência de memória converte o endereço virtual em real em um tempo compatível com o de acesso ao rótulo de endereço na *cache*, ficando ambos os endereços disponíveis, quase que simultaneamente, para comparação pelo controlador de *cache* [HENN86]. O principal problema deste esquema está na limitação do número de conjuntos (*set's*) da *cache* pelo tamanho da página de memória virtual porque somente os *bits* de *offset* dentro da página são utilizados para endereçar a *cache*.

2.1.2 - Tamanho do Bloco

Definição de Bloco: é a menor unidade de informação que o controla dor de *cache* pode ler ou escrever da memória principal. Geralmente possui número de *bits* múltiplo da largura do barramento de dados. Um bloco pode possuir mais de uma palavra do processador. Alguns autores preferem dar ao bloco o nome de linha devido à forma com que os dados são normalmente organizados na memória *cache*.

O tamanho do bloco é um importante parâmetro para o desempenho da *cache*, visto que estudos sobre taxas de acerto de programas monoprocessados tem mostrado que, para um *cache* de tamanho fixo, a taxa de falhas inicialmente diminui com o aumento do tamanho do bloco [EGGE89] porque programas tendem a referenciar instruções e dados na vizinhaça dos recentemente referenciados [KAPL73]. A medida que o tamanho do bloco cresce e se aproxima do tamanho da *cache*, a taxa de falhas começa a crescer. Tal fenômeno é explicado pelo fato de que a medida que a distância de um certo objeto pertinente ao bloco aumenta com relação ao objeto originariamente referenciado, sua chance de vir a ser referenciado diminui (*memory pollution* [SMIT87]).

2.1.3 - Associatividade

Um conjunto associativo consiste de um certo número de posições de blocos da *cache* que são referenciados pela mesma parte baixa do endereço (*cache index*) e que por esta razão podem armazenar blocos congruentes da memória principal. Quando um bloco é lido da memória principal para a *cache*, um algoritmo de reposição deve selecionar qual bloco dentro de um mesmo conjunto deve ceder seu lugar. Em geral, são algoritmos do tipo *LRU*, *FIFO* ou de base aleatória.

2.1.4 - "Cache" de Instruções e dados

O comportamento da demanda de dados e instruções de um programa não é idêntico, o que leva a necessidade de políticas de manutenção diferentes [SMIT83]. Enquanto que a demanda de instruções de um programa é sequêncial e

com repetições destas sequências através de *loops*, dados são geralmente acessados por uma complexa lei de formação ou puramente sequênciais quando se trata de vetores.

Sendo o código de um programa de apenas leitura, as invalidações na *cache* somente ocorrerão ao término deste (quando posições da memória principal residentes na *cache* forem carregadas com novos itens), o que torna a coerência por *software* fácil de ser implementada, visto que o sistema operacional sabe quais áreas vão ser liberadas. Em sistemas em que a *cache* é acessada através de endereços virtuais, há necessidade de invalidação também quando houver troca de contexto. Neste mesmo caso estarão os dados que apenas serão lidos ou utilizados por somente uma *CPU*. O problema de coerência se restringe aos dados compartilhados por mais de uma *CPU* e que sejam de leitura e/ou escrita.

2.2 - Métodos para Manutenção da Coerência em Sistemas "Multicache"

Esquemas de coerência de memória *cache* para sistemas multiprocessados interligados por um barramento comum a todas as unidades que podem acessar a memória principal já foram analisados em diversos trabalhos ao longo do tempo [WILK65] [LEE69] [GOOD83]. A característica básica de um esquema de coerência consiste na monitoração do barramento (*snoop*) e/ou na troca de informações entre cada unidade que possua *cache* para detectar mudanças de conteúdo em objetos que estejam mapeados em sua *cache* de forma que esta possa ser atualizada ou invalidada. Durante ciclos de leitura da memória principal, a monitoração é importante somente em alguns esquemas de *cache* que não atualizam a memória principal a cada ciclo de escrita da *CPU*. Neste caso, o valor contido em alguma *cache* será o atualizado, necessitando ser enviado para a unidade que o requisitou.

Em sistemas *multicache* conectados por redes, que não interliguem todas as unidades, não há como monitorar a ação de unidades distantes (que não estão conectadas diretamente). Uma análise deste caso é feita na seção 5.

2.2.1 - "Write Through"

Este esquema destaca-se pela sua simplicidade e reduzido número de estados (dois - *Invalid* e *Valid*) para blocos de informação residentes. Outra grande vantagem está no fato que praticamente qualquer tipo de protocolo de barramento suporta esta política de controle de *cache*.

Neste esquema, toda escrita realizada pela *CPU* passa automaticamente para a memória principal. Em caso de *hit* na escrita (na *cache* privada), o dado da *cache* também é atualizado. Se em uma operação de leitura da *CPU* houver ausência na *cache* (*miss* ou *invalid*), o bloco inteiro que contém o dado faltoso é lido da memória principal e armazenado na *cache* como válido. Subsequentes leituras deste bloco são atendidas pela *cache*. Variações deste esquema que incluem também as escritas da *CPU* na formação de localidade de referência podem alocar um bloco na *cache* nos ciclos de escrita.

O método é também amplamente disponível em controladores de *cache* integrados fabricados comercialmente. Como exemplos, pode-se citar o Cypress CY7C604 e CY7C605 [CYPR90], o Motorola 88200 [MOTO88], ambos para suas respectivas linhas de processadores RISC, e o Austek Microsystems A38152 [AUST87] para sistemas baseados em microprocessadores 80386 da Intel.

2.2.2 - "WRITE BACK"

O esquema *write-through* de atualização da memória principal acaba gerando tráfego desnecessário no barramento quando uma posição de memória é seguidamente acessada em escritas. Um esquema alternativo de atualização, chamado de *copy-back* ou *write-back*, não escreve necessariamente na memória principal a cada escrita da *CPU*. A *cache* possui, não mais apenas dois estados como no esquema de *write-through* e sim quatro que são: *Invalid* (posição ausente ou não atualizada nesta *cache*), *Private* (conteúdo válido e presente apenas nesta *cache*. A memória principal está atualizada), *Modified* (o conteúdo da *cache* está atualizado, presente somente nesta *cache* e a memória principal não está atualizada), *Shared* (o conteúdo da *cache* está válido assim como a memória principal e pode existir uma ou mais cópias em outras *caches*).

Este esquema é utilizado no sistema SEQUENT SYMMETRY [MANU87] e também pela CYPRESS que possui um controlador de *cache* (CY7C604 e CY7C605) [CYPR90] para sua linha *SPARC*.

Quando um determinado processador **i** requisita um dado que não está presente na sua memória *cache*, este será lido da memória principal. Caso haja outra cópia deste mesmo dado em outra *cache* **j** ($j \neq i$), o dado será armazenado como *shared*. Caso não exista nenhuma outra cópia, o dado será armazenado como *private*.

Em qualquer dos casos anteriores, um acesso do processador **i** a este mesmo dado ocorre sem que o barramento externo seja utilizado, ou seja, um acesso a um dado presente na *cache* não gera acesso ao barramento, com exceção de uma escrita a um dado compartilhado (*shared*). Neste caso, o acesso é necessário somente para notificar aos outros controladores que o dado mudou de estado (de *shared* para *modified*).

Escrita a um bloco marcado como *private* apenas muda o seu estado para *modified* enquanto que em blocos no estado *shared* gera acesso ao barramento para que os outros controladores de *cache* possam invalidar o bloco, como já foi dito anteriormente.

Quando um dado presente na *cache* **i** é acessado na memória principal por um outro processador **j**, o seu estado muda para *invalid* se o acesso for de escrita, ou para *shared* se for de leitura. Caso o estado inicial seja *modified*, duas ações podem ocorrer: ou ciclo corrente é interrompido, o controlador **i** atualiza a memória principal e o processador **j** repete o ciclo interrompido; ou o controlador de cache **i** inibe a memória principal e fornece o dado diretamente para o outro *cache* (*direct data intervention*), atualizando ou não a memória principal.

O meio utilizado para interligar as diversas unidades que compartilham a memória deve prever no seu protocolo a possibilidade de interrupção de ciclo de barramento em leituras e em escritas e/ou a desabilitação da memória por outro dispositivo.

2.2.3 - "WRITE ONCE"

Este esquema é uma modificação, proposta por Goodman [GOOD83], do esquema de *write-back* convencional. O reduzido tráfego no barramento é o grande destaque deste método de *cache* de quatro estados. Este esquema é utilizado pela MOTOROLA em sua linha RISC 88000 [MOTO88].

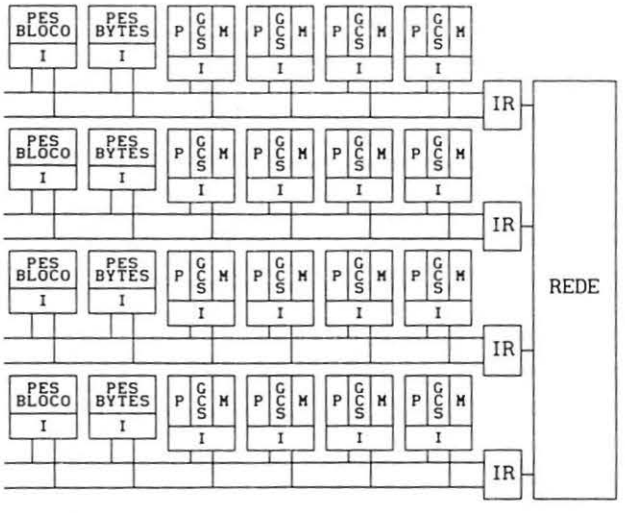
Inicialmente o método de *write-through* é utilizado, e na escrita de um bloco todos os outros controladores de *cache* invalidam cópias deste eventualmente presentes em suas *caches* de forma tal que a única cópia do bloco garantidamente pertence a uma única *cache*, que é marcada como *private*. Subsequentes escritas neste mesmo bloco não necessitam atualização imediata

da memória principal; o bloco é marcado como *modified* e o método de *write-back* é então utilizado. O *snoop* verifica, a cada operação no barramento, se o endereço está na *cache* local. Se houver *hit* no *snoop* em uma operação de escrita, o bloco é marcado como *invalid*. Se coincidir em uma operação de leitura, nada será feito, a menos que o bloco tenha sido modificado (*modified*). Se o estado é apenas *private*, este é trocado para *shared*. Se é *modified*, a memória principal será inibida pela interface de *cache* que atualizará o dado nela. No mesmo ciclo de barramento (ou imediatamente após) o dado poderá ser lido da memória principal. O estado do dado na *cache* local será então trocado para *shared*.

3 - Descrição da Arquitetura do MULTIPLUS

O MULTIPLUS é um sistema multiprocessado paralelo constituído por uma série de módulos de processamento baseados em microprocessadores RISC da linha CY7C60X da Cypress de arquitetura *SPARC* conectados através de uma rede de chaveamento e barramentos.

A figura 2 uma configuração com quatro *clusters* da arquitetura do sistema MULTIPLUS, incluindo os PES e a rede de interconexão com dois níveis de chaves. A figura 3 detalha o NP.



Legenda:

P - Processador
C - Cache
G - Gerência
S - Snoop
M - Memória
I - Interface de barramento
IR - Interface de Rede
PES - Processador de Entrada e Saída (orientado a bloco/byte)

figura 2: diagrama em blocos do Sistema **MULTIPLUS**.

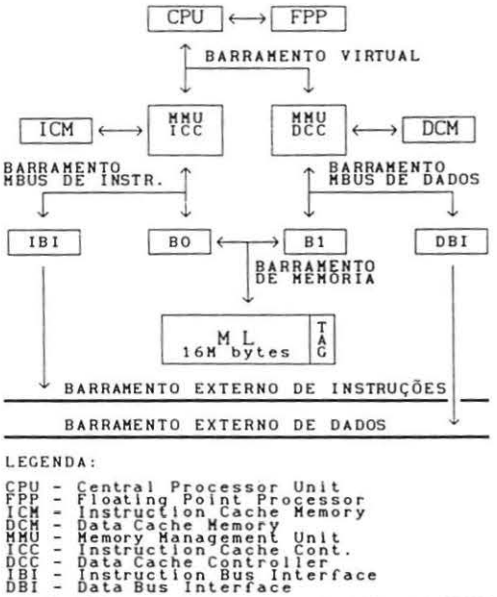


figura 3: diagrama em blocos do NP do MULTIPLUS

Cada módulo de processamento, ou Nó de Processamento (NP), possui uma memória local (partição da memória global), uma *cache* privada, barramento interno virtual interligando processadores e controladores de *cache*, além de dois barramentos internos físicos (**Mbus**). Os NP's se conectam a um barramento paralelo duplo que, teoricamente, pode suportar até oito processadores, formando um *cluster*. Os barramentos conectam os processadores às interfaces da rede de chaveamento (IR) que pode ter até oito níveis de chaves.

A E/S será feita por unidades com processamento próprio chamados de Processadores de Entrada e Saída (PES) orientados a blocos (discos, fitas, etc.) ou a *byte* (terminais e impressoras).

4 - BARRAMENTO

Um dos grandes "gargalos" de sistemas multiprocessados está no trá fego do barramento. Diversas soluções já foram apresentadas para minimizar o seu efeito como por exemplo a inclusão de memórias *cache* [LIPT68] [TANG76] [GOOD83], a utilização de redes de interconexão [GOTT83], a utilização de memórias privadas [SEIT85] ou novos protocolos de barramento mais rápidos.

No projeto do Sistema MULTIPLUS procurou-se diminuir o "gargalo" através da utilização de um barramento duplo, uma vez que simulações preliminares mostraram que, mesmo utilizando-se diversos recursos como os já citados anteriormente, o número de processadores ficaria limitado a aproximadamente quatro. Entretanto, esta arquitetura de barramentos múltiplos aumenta a complexidade do circuito de controle de *cache*, principalmente no que diz respeito ao esquema de *snoop* que agora precisa monitorar ambos os barramentos.

Primeiramente, pensou-se em dois barramentos idênticos que poderiam ser usados em operações de escrita ou leitura e em acessos às instruções ou dados; ou seja, quando um processador necessitasse acessar um objeto através

do barramento, ele poderia utilizar qualquer um dos dois barramentos. O problema e que a política *write-back* obriga o controlador de *cache* a monitorar o barramento em acessos de leitura e escrita. O controlador de *cache* utilizado (CY7C605) possui capacidade apenas para monitorar um barramento. Para que fosse mantida a coerência dos objetos armazenados em *cache*, seria necessário que as ações nos barramentos externos ao NP fossem "multiplexadas no tempo" para que o *snoop* do controlador de *cache* possa acompanhar todas as transações, o que certamente levaria esta solução de dois barramentos idênticos a um desempenho semelhante ao de um barramento simples.

Alguns resultados do trabalho de Smith [SMIT82] sugerem que uma boa aproximação para a taxa de acessos à memória é: 50% de busca de instruções, 35% de leitura de dados e 15% de escrita de dados. No sistema MULTIPLUS, é esperado, no pior caso, que a taxa de falhas na *cache* de código seja de no máximo 10%. Foi também estimada a taxa de utilização de objetos remotos em 20%. A partir destes dados, deduz-se que apenas 1% das buscas de instruções usarão o barramento. Nos acessos de leitura de dados, considerando uma taxa de acerto na *cache* de 80%, cerca de 2.5% utilizarão o barramento, enquanto que as escritas necessitarão quase 1.5%(chegando a necessitar perto de 3% no caso de utilização de política de *write-through*).

A partir destes resultados pode-se deduzir que uma especialização de barramentos utilizando-se um deles para escrita e o outro para leitura iria dividir bem a sua demanda.

O problema de *snoop* descrito no item anterior poderia ser resolvido utilizando-se uma das facilidades oferecidas pelo controlador de *cache* que é a possibilidade de escolha entre dois esquemas de manutenção de coerência: *copy-back* ou *write-through*. Este segundo esquema necessita de *snoop* somente nos ciclos de escrita.

A especialização dos barramentos descrita anteriormente limita a utilização do esquema de manutenção de coerência, impossibilitando o uso de *copy-back*, que poderia diminuir a taxa de ocupação no barramento. Uma opção mais natural seria a utilização de um barramento somente para leitura de instruções. Neste barramento não haveria necessidade de *snoop* uma vez que, por definição do Sistema, as intruções não são modificáveis, deixando todas as transações de dado, com a utilização de *snoop*, para o segundo barramento. O controlador de *snoop* ficaria dedicado a monitorar apenas o barramento de dados.

5 - Opções de Memória "Cache"

Devido à defasagem tecnologica que os Centros de Pesquisa e Desen volvimento vivem hoje no Brasil, é impossível desenvolver de imediato todo o conjunto de circuitos integrados necessários ao projeto. Optou-se, em uma primeira fase, pela utilização da unidade de processamento de arquitetura *SPARC* da Cypress Semiconductor Corporation [CYPR90]. Em relação à *cache*, existem duas alternativas que são: a utilização do controlador da Cypress, que se interliga diretamente com o resto dos circuitos integrados, ou a implementação de um controlador discreto. Esta segunda opção foi abandonada devido à complexidade, custo e área de circuito que demandaria.

O controlador de *cache* escolhido (CY7C605) integra, em um único encapsulamento de 244 pinos, uma Unidade de Gerência de Memória (*MMU - Memory Management Unit*) e um Controlador de *Cache* (*CC - Cache Controller*), além de possuir internamente todas as *tags* e *flags* necessárias para manutenção de coerência e reposição de blocos, sendo totalmente compatível com a arquitetura *SPARC*.

A gerência possui uma *TLB* (*Translation Lookaside Buffer*) de 64

entradas com procura por *hardware* em caso de falha, suporta 4096 contextos e possui proteção a nível de páginas, que podem ser declaradas como *cachable* ou *not cachable*, e podem ser de 4K, 256K, 16M ou 4G *bytes*.

O controlador de *cache* suporta políticas de manutenção de coerência do tipo *write-through* sem alocação por escritas (*without write allocate*) ou *copy-back* com alocação por escritas (*with write allocate*), com possibilidade de troca direta de dados entre *caches* (*direct data intervention*), atualizando ou não a memória principal (*reflectivity*). Cada controlador pode gerenciar até 64K *bytes* de memória *cache* em 2048 blocos de 32 *bytes* mapeados diretamente (utilizando duas pastilhas de memória do tipo CY7C157). Podem ser agrupados em até 4 (8 como poderá ser visto a seguir) unidades que somam suas caracteristicas, fornecendo uma memória *cache* de 256K (512K) *bytes* de capacidade de armazenamento com mapeamento direto.

Um sistema de *cache* mínimo Cypress é constituido por um controlador de *cache* e duas memórias. Um *cache* único diminuiria o custo e a área necessária para sua implementação. Outra grande vantagem estaria na simplicidade de interfaceamento entre processador e *cache*. Por outro lado, um *cache* único para instruções e dados poderia gerar conflito de endereços entre instruções e dados, principalmente no caso do controlador da CYPRESS CY7C605 que só possui modo de mapeamento direto, ocasionando um possível grande número de invalidações. Particularmente em relação à arquitetura do MULTIPLUS, embora a ligação entre processador e *cache* seja mais simples porque só existe uma via de conexão, a interface entre *cache* e a memória local seria mais complexa devido a existência de dois barramentos. Além disso, a existência de um único controlador de *cache* para instruções e dados também obrigaria a uma serialização das ações dos dois barramentos para fins de *snoop* de forma similar ao caso de barramentos não especializados. A separação dos *caches* tende a diminuir também o número de invalidações devido a possíveis conflitos entre endereços de código e dado entre tarefas, o que possivelmente ameniza o efeito do *cache* ser de mapeamento direto.

A arquitetura da unidade de processamento da Cypress (*IU + FPP + CC/MMU*) não prevê a separação do *cache* em dado e instrução. Para contornar este problema, pode-se utilizar um artifício, anulando ciclos de busca de instrução para o *cache* de dados e de busca de dados para o *cache* de instruções. Este arranjo descrito tem ainda a vantagem de permitir expandir a memória *cache* dos 64K *bytes* iniciais para até 256K *bytes* para instruções e dados.

6 - Consistência

O problema de consistência em memórias *cache* aparece em multiproces sadores com *caches* privados [ARCH84]. Nestes sistemas, cada processador tem associado a ele uma *cache* para armazenar o conteúdo das localizações de memória acessadas mais recentemente. Se mais de uma *cache* pode ter uma cópia da mesma localização de memória e não existe nenhum meio de evitar que os processadores modifiquem simultaneamente suas respectivas cópias, poderá haver inconsistência nas diversas cópias, isto é, a **base de dados** poderá se tornar incoerente.

Com o aparecimento de sistemas multiprocessados e *caches* privados distribuidos, houve um aumento significativo na dificuldade de se manter a consistência dos objetos armazenados nas *caches*.

A política de manutenção de coerência passou a ter importância primordial em relação a outros parâmetros clássicos tais como o algoritmo de reposição, grau de associatividade, etc. O desempenho do sistema multiprocessado está intimamente relacionado ao esquema de coerência implementado porque todos os controladores de *cache* que podem compartilhar

objetos devem poder se comunicar para que mudanças no estado de objetos compartilhados sejam sinalizadas.

Independentemente do método utilizado, um sistema multiprocessado tem uma *cache* coerente se um acesso de leitura a qualquer bloco sempre retorna o valor escrito mais atualizado.

6.1 - Manutenção da Coerência por "Hardware"

O controlador de *cache* utilizado possui um circuito de *snoop* que monitora continuamente ações no seu barramento **Mbus** (de endereços físicos) mantendo o estado dos objetos contidos na *cache* sempre atualizados. Para que os controladores de *cache* possam monitorar os seus respectivos barramentos externos ao NP, os *buffers* da interface devem estar sempre habilitados levando o endereço para dentro do NP.

As operações entre processador e memória local não podem ser monitoradas por outros controladores de *cache* a não ser que sejam realizadas via barramento externo. Em *write-back* todas as operações necessitam ser monitoradas, mas estas só acarretam mudanças no estado daqueles objetos que estiverem armazenados também em outros *caches*. Para poder distinguir os acessos que certamente não irão mudar o estado das outras *caches* dos que podem mudar, foi incluído um sinalizador em cada bloco de *cache* na memória principal local indicando se aquele bloco já foi utilizada por algum outro *cache* no mesmo *cluster*. Acessos não armazenáveis na *cache* não mudam o estado deste sinalizador.

Inicialmente o estado do sinalizador está **DESLIGADO**, simbolizando que o bloco não está presente em nenhuma outra *cache* do mesmo *cluster*. Leituras ou escritas no modo coerente[3] (do tipo *COHERENT READ* e *COHERENT WRITE*) originadas por um controlador de *cache* remoto e do mesmo *cluster* levam o estado do sinalizador do bloco em questão para **LIGADO**. O estado do sinalizador de compartilhamento volta a **DESLIGADO** quando o controlador de *cache* local endereçar este bloco com o comando de invalidação (*COHERENT INVALIDATE*) em conjunto ou não com leitura ou escrita.

Todos os endereçamentos dirigidos à memória local poderão ser executados sem a necessidade de utilizar o barramento externo, com exceção de acessos no modo coerente e cujo bloco, na memória local, esteja com o sinalizador de compartilhamento **LIGADO**.

6.2 - Manutenção da Coerência por "Software"

Como a memória global do sistema (*backing memory*) está distribuida pelos diversos *clusters* e nós de processamento, e pode ser acessada por qualquer processador homogeneamente, com custo variável, um esquema para manter a coerência baseada em *snoop* torna-se inviável, pois para cada transação haveria a necessidade de ser enviada uma mensagem do tipo *broadcast* por toda a rede de interconexão, o que seria muito custoso.

A maior dificuldade de manter a coerência em sistemas "fracamente acoplados" está no custo da troca de informações entre controladores de *cache* quando ocorre mudança nos estados dos dados armazenados nelas.

Existe uma alternativa proposta por Stenström [STEN89] para manter a coerência das *caches* mesmo através da rede. Este método permite que existam quantas cópias o sistema necessitar de cada dado. O método propõe a inclusão de um campo de *flags* em cada *cache* local (que teria 256 *bits* no caso do

---

[3] Acessos no modo coerente são acessos que tratam um bloco da *cache* como um único objeto, ou seja, referem-se sempre a oito *words* (256 *bits*).

MULTIPLUS), o que o torna inviável.

Modificação mais viável deste método, para a arquitetura do MULTIPLUS, transfere as *flags* das *caches* locais para a memória principal com a inclusão de um *bit* de validade. Neste caso, quando um NP acessar um dado, este passa a pertencer ao seu *cluster*. O número do *cluster* é escrito no lugar do dado e o *bit* de validade é apagado. Outros NP's do mesmo *cluster* podem armazenar este dado em sua *cache* local. Quando outro NP de outro *cluster* tentar acessar este dado na memória principal (este dado, por definição, estará invalido em sua *cache* local), encontrará o *bit* de validade desligado e a própria chave re-roteará o acesso para o *cluster* **DONO DO DADO**. Este *cluster* invalidará o dado, caso seja uma escrita, ou apenas o transmitirá em caso de leitura. Caso o dado não seja invalidado, este não poderá ser armazenado na *cache* do NP destino. Entretanto, neste esquema, somente um único *cluster* poderá ter cópia do dado.

Esta alternativa adaptada da proposta de Stenström envolve muito *hardware* o que se traduz em aumento de custo e maior complexidade de projeto, implementação e depuração. Uma solução que simplifica o problema em questão seria a da não inclusão na *cache* de dados compartilhados de leitura e escrita que somente podem ser acessados via rede de interconexão. A perda de eficiência resultante desta simplificação é minimizada com a exclusão de todas as instruções, das variáveis de apenas leitura e, sob controle do *software*, de variáveis compartilhadas de leitura e escrita momentaneamente privadas (variáveis migratórias) deste conjunto de objetos não armazenáveis em *cache*,porque estes tipos de objetos não necessitam de *snoop*.

Fica a cargo do núcleo do sistema operacional programar corretamente as tabelas de gerência, avisando ao controlador quais as páginas que podem ser armazenadas em *cache* ou não, e de manter a exclusão mútua das variáveis migratórias.

6.3 - Manutenção da Coerência Durante E/S

Os esquemas de manutenção de coerência discutidos anteriormente só garantem a consistência dos dados armazenados em *cache* para transações entre NP's. Será discutida agora a manutenção de coerência em relação à E/S.

A E/S no sistema MULTIPLUS, por definição de sua arquitetura, está distribuida pelos diversos *clusters*, cada um contendo um processador de E/S conectado a dispositivos de armazenamento de massa e outro a terminais, impressoras e rede de alta velocidade.

A dificuldade de manter a coerência durante E/S reside na possibilidade da substituição de uma página de memória por outra de outro tipo, como, por exemplo, uma página de código de uma tarefa que já foi terminada ser substituida por outra página de dados de uma nova tarefa a executar.

Uma outra substituição problemática ocorre quando uma página de dados de leitura e escrita é trocada por uma página de instruções utilizando o barramento de instruções para carga do novo bloco de instruções. O *snoop* do controlador de *cache* de dados de um NP remoto, mas do mesmo *cluster*, não perceberia a transação e não invalidaria blocos da sua *cache* que por ventura ainda contivessem algum dado referente àquela página. Embora o processador deste controlador de *cache* não mais vá se referir a esta página como dado, uma ação de *write-back* poderia ocorrer para a substituição de um bloco com um dado na *cache* no estado *modified* por um bloco de alguma outra página válida de dado. Este *write-back* escreveria de volta na memória principal dados antigos por cima de uma área de instruções atualmente válida.

Para contornar este tipo de problema, o núcleo do sistema operacional deverá forçar um *write-back* através do comando de *flush* de

página para cada página de dados ou instruções liberada na memória principal. Após esta operação, toda transferência de E/S nunca levará a um acerto do *snoop* do *cache*, possibilitando assim que as transferências de E/S sejam realizadas através de qualquer um dos barramentos, independentemente de serem blocos de dados ou instruções.

Com esta liberdade, o sistema operacional e os Processadores de E/S (PES) podem ser simplificados, tratando toda E/S como sendo transferência de blocos de dados, eliminando assim a necessidade, para o sistema operacional, de discriminar se o bloco é de dado ou instrução e, para o PES, de possuir duas interfaces com os barramentos do *cluster*.

Em resumo, toda E/S será feita pelo barramento de dados externo e a cada liberação de uma página pelo sistema operacional, o seu núcleo se encarregará de enviar um comando de *flush* para o controlador de *cache* invalidando todas as posições na *cache* ocupadas por objetos das páginas liberadas.

7 - Operações Indivisíveis

A Unidade Inteira (*IU* CY7C601) realiza dois tipos de operações indivisíveis, uma para ler ou escrever dados de 64 *bits* (*double word*) e outra para ler, alterar e escrever um *byte* ou uma *word*. Este último tipo de instrução é particularmente utilizada em sicronização de processos de forma a garantir exclusão mútua em recursos que não podem ser compartilhados simultaneamente. No sistema MULTIPLUS, ela será substituida por instrução do tipo *fetch and increment* (*F&I*).

Por definição de arquitetura do Sistema Operacional do MULTIPLUS, as sincronizações serão feitas por uma fila de processadores com um ponteiro para a próxima posição vaga. Cada processador que necessitar um determinado recurso deverá ler o valor atual do ponteiro da fila e incrementá-lo para que este possa apontar para a próxima posição vaga. Na fila, o processador apenas escreverá o seu número na posição que foi lida do apontador.

Quando o processador que estiver utilizando atualmente o recurso for liberá-lo, deverá, primeiramente, consultar a fila para descobrir qual é o próximo processador e avisá-lo. Este método elimina o chamado *hot-spot* que tende a se formar na variável de sincronização, distribuindo-o para as memórias locais dos processadores.

7.1 - Coerência em Operações Indivisíveis

A indivisibilidade das operações atômicas, que é garantida pelo processador (este não aceita interrupções e pedidos de barramento), deve ser garantida também na memória principal através do seu árbitro que deve assegurar a posse do barramento ao processador até o fim do ciclo. Se dois processadores tentarem executar ao mesmo tempo instruções indivisíveis na mesma palavra de memória, o árbitro deve providenciar para que tudo se passe como se um processador executasse a instrução e somente depois de concluí-la, o outro a iniciasse, ou seja, o resultado deverá ser o mesmo para execuções em "paralelo" ou puramente sequenciais. Este tipo de tarefa não é de díficil realização, uma vez que todas estas ações necessitam de pelo menos um caminho em comum que será mutuamente exclusivo.

No caso dos processadores possuirem *cache* local, como no MULTIPLUS, esta última afirmação não é verdade sempre, caso a palavra na qual a operação se realize esteja presente em pelo menos uma das *caches* de um dos processadores. Neste caso, enquanto um estiver lendo a palavra de sua própria *cache*, o outro a estaria lendo da memória principal (ou da sua *cache*

também). Ambos iriam incrementar o mesmo valor e escrevê-lo de volta. Embora os controladores de *cache* garantam a manutenção da coerência do dado, esta não implica na manutenção da "sequencialidade" que por sua vez garante a exclusão mútua.

Uma solução possível seria a alocação das variáveis de sicronização em páginas *not caching*. Embora viável, é de difícil implementação inicial porque em uma primeira etapa deve ser utilizado o compilador de alguma máquina comercial baseada na arquitetura *SPARC* e muito provavelmente este compilador não deverá fornecer a facilidade de declarar variáveis como sendo do tipo *not caching*.

7.2 - Operações Indivisíveis na Memória Local

A natureza do algoritmo de sincronização não favorece a um acerto da variável de sicronização na *cache*, necessitando que esta seja trazida da memória principal na maioria das vezes. A baixa reutilização e o alto custo da inclusão de um bloco na *cache* leva a opção de separar as variáveis de sincronização como não armazenáveis em *cache*, mesmo quando utilizadas somente dentro do mesmo *cluster*. Trazer um bloco da memória principal para a *cache* requer pelo menos quatro acessos à memória. Mesmo utilizando-se o modo de troca de dados entre *caches* sem utilizar a memória principal, o tempo gasto não compensa, no caso, devido a pouca possibilidade de reutilização deste tipo de bloco.

8 - Estágio Atual e Perspectivas Futuras

Embora a alocação de variáveis de sincronização em páginas do tipo *not cachable* seja um tanto complexa, este esquema será utilizado, nesta primeira fase, para lidar com o problema das operações indivisíveis.

Com relação ao problema analisado anteriormente, de variáveis compartilhadas entre NP's de *clusters* diferentes, optou-se por alocá-las em páginas *not cachable* eliminando-se assim a necessidade de aplicação de esquemas de manutenção de coerência de alta complexidade. Esta solução não deverá comprometer o desempenho do sistema pois é esperado um tráfego de informações reduzido na rede de interconexão devido a precisa especificação da hierarquia de memória.

Os próximos passos serão a implementação de um simulador para o NP visando uma melhor quantificação da relação custo versus benefício de acordo com a política de manutenção de coerência de *cache* comparando *write-through*, com menor desempenho e menos *hardware*, com *write-back*, com melhor desempenho e demandando mais *hardware*.

9 - Agradecimentos

Os autores agradecem ao CNPq e à FINEP o apoio ao desenvolvimento deste projeto.

10 - Referências

[ARCH84] J. ARCHIBALD e outro, "An Economical Solution to the Cache Coherence Problem", ACM SIGARCH Computer Architecture News, vol. 12(3), pp. 355-362

[AUDE90] J. S. AUDE e outros, "MULTIPLUS: Um Multiprocessador de Alto

Desempenho", Anais do 10º Congresso da SBC, Vitória, pp. 93-105, jul/1990

[AUST87] AUSTEK MICROSYSTEMS, "A38152 Microcache™ for Intel 80386-based Microprocessor", External Reference Specification, jun/1987

[CYPR90] "SPARC RISC USER'S GUIDE", Cypress Semiconductor Corporation, segunda edição, fev/1990

[DUBO82] M. DUBOIS e outro, "Effects of Cache Coherence in Multiprocessors", IEEE Transactions on Computers, vol. C-31(11), pp. 1083-1099, nov/1982

[EGGE89] S. J. EGGERS e outro, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs", ACM SIGARCH Computer Architecture News, vol. 17(2), pp. 257-270, abr/1989

[GOOD83] J. R. GOODMAN, "Using Cache Memory to Reduce Processor-Memory Traffic", ACM Computer Architecture News, vol. 17(3), pp. 124-137, jun/1983

[GOOT83] A. GOTTLIEB e outros, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", IEEE Transactions on Computers, vol. C-32(2), pp. 175-189, fev/1983

[HENN86] J. L. HENNESSY e outro, "An Overview of the MIPS-X-MP Project", Computer Systems Laboratory, Stanford University - Stanford - CA 94305-2192, Technical Report No. 86-300, abr/1986

[KAPL73] K. R. KAPLAN e outro, "Cache-based Computer Systems", IEEE Computer, vol. 6, pp. 30-36, mar/1973

[LEE69] F. F. LEE, "Study of 'Look-Aside' Memory", IEEE Transactions on Computers, vol. C18, pp. 1062-1064, nov/1969

[LIPT68] J. S. LIPTAY, "Structural Aspects of the System/360 Model 85-II - The Cache", IBM System Journal, vol. 7(1), pp. 15-21, jul/1968

[MANU87] T. MANUEL, "How Sequent's New Model Outruns Most Mainframes", Electronics, Mai/1987

[MOTO88] "MC88200 Cache/Memory Management Unit User's Manual", Motorola Inc., primeira edição, dez/1988

[SEIT85] C. L. SEITZ, "The Cosmic Cube", Communications of the ACM, jan/1985, vol. 28(1), pp. 22-33

[SMIT82] A. J. SMITH, "Cache Memories", ACM Computer Surveys, vol. 14(3), pp. 473-530, set/1982

[SMIT83] J. E. SMITH e outro, "A Study of Cache Organization and Replacement Policies", ACM Computer Architecture News, vol. 17(3), pp. 132-137, jun/1983

[SMIT87] A. J. SMITH, "Line (Block) Size Choice for CPU Cache Memories", IEEE Transactions on Computers, vol. C-36(9), pp. 1063-1075, set/1987

[STEN89] P. STENSTRÖM, "A Cache Consistency Protocol for Multiprocessors with Multistage Networks", ACM Computer Architecture News, vol. 17(3), pp. 407-415, jun/1989

[STON87] H. S. STONE, "High-Performance Computer Architecture", Addison-Wesley Publishing Company, 1987

[TANG76] C. K. TANG, "Cache System Design in the Tightly Coupled Multiprocessor System", AFIPS Conference Proceedings, National Computer Conference, NY, vol. 45, pp. 749-753, jun/1976

[WILK65] M. V. WILKES, "Slave Memories and Dynamic Storage Allocation", IEEE Transactions on Electronic Computers, vol. EC-14, págs. 270-271, abr/1965