Análise Comparativa de Métodos de Coerência de Dados em Memórias Cache

Carlos Eduardo Rodrigues Alves*
Osvaldo Catsumi Imamura**

Instituto de Estudos Avançados (IEAv-CTA)
Rodovia dos Tamoios, Km. 5,5 Caixa Postal 6044
CEP 12231 São José dos Campos - SP
Tel: (0123) 413033 Ramais 405 e 411
e-mail: CTA@BRFAPESP.BITNET

RESUMO

O uso de um sistema de memória compartilhada permite grande iteração entre os processadores de uma máquina MIMD e oferece um paradigma de programação bastante simples. No entanto, a implementação de sistemas de memória compartilhada eficientes apresenta diversos desafios, devido à alta largura de faixa e à baixa latência média requeridos.

Uma solução economicamente viável é associar uma cache a cada processador, armazenando cópias dos dados mais usados nestas caches. Diversas técnicas têm sido estudadas e implementadas para que não ocorram problemas de coerência entre cópias de um mesmo dado guardadas em caches distintas.

Uma técnica pouco explorada é a utilização de caches *write-through* com verificação de consistência para todas as escritas geradas pelos processadores. A impopularidade desta técnica se deve à alta largura de faixa requerida para a memória principal. Este trabalho mostra que, em algumas circunstâncias, o uso de caches *write-through* pode apresentar vantagens sobre o uso de caches *copy back* com protocolos de posse de blocos, apresentando desempenhos menos dependentes da codificação dos programas.

ABSTRACT

A shared-memory system allows a great interation between processors of an MIMD machine and offers a very simple programming model. However, the implementation of efficient shared-memory systems presents several challenges, due to the high bandwidth and low medium latency required.

A cost-effective solution is to associate a cache to each processor, storing copies of the most used data in this caches. Several techniques have been studied and implemented to avoid coherence problems between copies of a single datum stored in distinct caches.

A rarely explored technique is the utilization of write-through caches with verification of all the writes generated by the processors. The unpopularity of this technique is due to the high bandwidth required to the main memory. This work shows that, in some circumstances, the utilization of write-through caches can be advantageous when compared to the use of copy-back caches with block-ownership protocols, with performances that are less dependent on the codification of programs.

---

* Msc, ITA.
** PhD, Universidade de Tóquio.

1. Introdução

Máquinas MIMD de memória compartilhada oferecem uma forma bastante simples de comunicação entre os processadores, feita através de leituras e escritas a variáveis compartilhadas. Isto permite que uma determinada aplicação seja executada de forma paralela através da sua divisão em tarefas que operam sobre os mesmos dados, sem a necessidade de passagens explícitas de mensagens ou de partição dos dados entre os processadores, como ocorre em sistemas distribuídos.

No entanto, a implementação de sistemas de memória compartilhada apresenta diversos problemas, uma vez que estes sistemas precisam atender a acessos de diversos processadores, com velocidades semelhantes às que seriam esperadas de memórias para uniprocessadores. Existe a possibilidade de um mesmo elemento de armazenamento do sistema de memória ser requisitado simultaneamente por mais de um processador, causando conflitos e, conseqüentemente, atrasos na conclusão dos acessos. Estes conflitos tendem a aumentar com o número de processadores, tornando difícil a implementação de sistemas com um número muito elevado (além de poucas dezenas) de processadores. Além disso, o grande espaço de armazenamento exigido por diversas aplicações eleva o custo do sistema, tornando necessária a utilização de dispositivos lentos e relativamente baratos na implementação do sistema de memória.

Uma solução bastante explorada é o uso de caches locais. A cada processador é associada uma memória cache capaz de manter cópias dos dados armazenados na memória compartilhada, permitindo acessos rápidos a essas cópias sem que partes globais do sistema, em geral construídas com dispositivos lentos e sujeitas a conflitos, sejam utilizadas. As caches são implementadas com dispositivos de armazenamento rápido, e sua capacidade de armazenamento é bem menor do que o espaço de endereçamento dos processadores, para manter o custo acessível. A propriedade de **localidade das referências** faz com que as caches possam atender à maior parte dos acessos requeridos pelo processador, apesar da sua pequena capacidade de armazenamento [Smit82].

Entre as decisões a serem tomadas no projeto de sistemas de memória com caches locais, alguns são comuns a sistemas uniprocessadores: qual o tamanho ideal para as caches, a forma de mapeamento a ser usada, a política de reposição etc. Nos sistemas multiprocessadores, o uso de caches locais provoca um problema adicional: a garantia de consistência. Cópias de um mesmo dado podem estar presentes em vários pontos do sistema de memória, e a modificação de uma destas cópias deve se refletir em todas as demais para evitar a presença de múltiplas versões de um mesmo dado no sistema.

Existem diversas técnicas que visam manter a consistência de sistemas de memória com caches locais. Este trabalho compara duas técnicas básicas, o uso de protocolos de posse de blocos de dados e o uso de protocolos *write-through*, quando utilizadas em sistemas de porte modesto (oito processadores). Normalmente, sistemas baseados em protocolos *write-through* são considerados inadequados, por razões que serão apresentadas mais a frente. Os resultados presentes neste trabalho mostram que, em determinadas circunstâncias, o uso de protocolos *write-through* é vantajoso com relação ao uso de protocolos de posse de blocos, mostrando porque estudos anteriores [ArBa86,YaBL89] chegaram a conclusões diferentes.

A próxima seção expõe os protocolos de posse de blocos, usando o protocolo Synapse [Fran84] como exemplo, e os protocolos *write-through*. A seção 3 comenta a metodologia adotada para a realização dos testes e os parâmetros utilizados para os sistemas avaliados. A seção 4 mostra a análise dos resultados dos testes. A seção 5 apresenta as conclusões deste trabalho.

2. Mecanismos de Garantia de Consistência

O método mais simples de garantir a coerência entre as cópias presentes nas caches é armazenar apenas dados não modificáveis ou dados acessíveis a apenas um processador (naturalmente, o código executável se enquadra nesta classificação, desde que não seja modificável). Este esquema, denominado **estático**, não requer

praticamente nenhum *hardware* dedicado além daquele usado em caches para uniprocessadores. Por outro lado, requer que os compiladores sejam capazes de identificar quais os dados permitidos nas caches, identificação esta que freqüentemente é mais conservadora do que o necessário. Além disso, os acessos a dados globais modificáveis representam uma parte significativa dos acessos, e sua exclusão das caches diminui a eficácia destas.

Métodos que permitem a inclusão de dados globais modificáveis nas caches são chamados **dinâmicos**. Para garantir a consistência do sistema, é preciso que diversas verificações sejam feitas a cada acesso. Estas verificações podem ser feitas pela própria cache, por *hardware* ligado à memória global ou por intermédio do próprio processador e *hardware* local (no caso de esquemas orientados por *software*, que não serão cobertos neste trabalho).

2.1. Protocolos de verificação intensiva das escritas (*write-through*)

Uma maneira de garantir a consistência do sistema de memória é monitorar toda e qualquer escrita realizada em qualquer cache, para verificar se não há outras cópias do dado escrito. Se houver, estas cópias precisarão ser atualizadas ou invalidadas, para que outros processadores não leiam dados desatualizados.

A verificação intensiva das escritas deve ser realizada em estruturas ligadas à memória global. Normalmente, são usados barramentos que interligam todas as caches, de forma que todas as escritas podem ser monitoradas por todas as caches, ou são usados diretórios globais que indicam as caches que possuem cópias desatualizadas, permitindo ações seletivas sobre estas caches. Estas implementações são apresentadas na seção 2.3.

É preciso definir, para cada sistema, como serão tratadas as cópias desatualizadas. Basicamente, estas cópias podem ser invalidadas ou atualizadas nas próprias caches. No caso de invalidação, a cópia é simplesmente removida da cache através de uma intervenção de *hardware* ligado à parte global da memória, de forma que se houver uma tentativa de acesso ao dado ocorrerá uma perda e conseqüente busca do dado atualizado na memória global.

No caso de serem usadas atualizações, uma cache com cópia desatualizada sofre uma intervenção de *hardware* externo para ter sua cópia atualizada. Isto pode ser útil se esta cópia for utilizada no futuro, pois evita-se uma perda e conseqüente acesso à memória global. Por outro lado, se esta cópia não for utilizada no futuro, e novas escritas ocorrerem por parte de outros processadores, esta cópia sofrerá atualizações constantes e inúteis, interrompendo com freqüência a operação normal da cache e provocando uma queda no desempenho.

Em caches *write-through*, é possível inexistirem *write-misses* (perdas durante escritas). Neste caso, se o processador envia uma escrita à cache, a presença ou não de cópia anterior do dado a ser escrito é irrelevante. O novo valor é guardado na cache ao mesmo tempo em que é enviado à memória principal. A cache pode continuar operando logo após receber a escrita, mesmo que a memória principal não tenha sido atualizada ou que as demais caches não tenham sido verificadas, o que implica numa baixa latência. No entanto, no caso de um processador enviar uma seqüência rápida de escritas, várias destas escritas podem ficar pendentes simultaneamente no sistema de memória. Estas escritas devem ser resolvidas na ordem de envio para evitar problemas de consistência.

Este esquema é conceitualmente simples, mas apresenta diversas dificuldades de implementação. A maioria destas dificuldades se deve ao fato de que todas as escritas devem ser encaminhadas à memória global (atualização *write-through*) e tratadas por um *hardware* de verificação. Ambos devem possuir uma grande largura de faixa, pois o número de acessos tende a ser alto.

2.2. Protocolos de Posse de Blocos

O principal problema dos protocolos descritos anteriormente é a grande quantidade de acessos enviados à memória global, na sua maior parte representados por escritas. Boa parte dos dados escritos podem ser de uso exclusivo de apenas um processador, ao menos durante parte da execução, de forma que estas escritas não precisariam ser verificadas.

Os protocolos de posse de blocos [Tang76,PaPa84,CeFe78,ArBa86] permitem que boa parte das escritas sejam verificadas localmente, na própria cache em que são realizadas, evitando acessos à memória global. Para que isso possa ser feito, a cada cópia é atribuído um estado, e este estado é guardado na cache para ser verificado a cada acesso. Para reduzir o armazenamento necessário para este estado, bem como reduzir o número de operações de

controle entre caches, as cópias são agrupadas em blocos de palavras contíguas e um estado é associado a todo o bloco. O tamanho do bloco é usualmente igual ao número de palavras trazido da memória principal durante uma perda.

Há diversos protocolos diferentes, cada qual com um conjunto de estados associado. Mas normalmente, existe um estado **inválido**, que indica que o bloco não está presente na cache, um estado **sujo** ou **exclusivo**, que indica que a cópia presente na cache é única e pode ser usada livremente para leitura ou escrita, e outros estados que apresentam restrições aos acessos, particularmente às escritas. Todos estes estados podem ser alterados por influência do processador ligado à cache ou do *hardware* externo.

Um protocolo bastante simples é o **Synapse** [Fran84], que trabalha com invalidações de cópias desatualizadas e atualização da memória via *Write-Back*. Este protocolo possui três estados possíveis para os blocos de cópias: **Inválido**, **Compartilhado** e **Sujo**, que serão denotados apenas **I**, **C** e **S** respectivamente. O estado **I**, como já foi comentado, indica que o bloco não está copiado nesta cache; o estado **C** indica que o bloco está presente, e possivelmente há outras cópias em outras caches; e o estado **S** indica que além da cópia ser única, a memória principal está desatualizada.

Cópias no estado **S** podem ser usadas para escritas ou leituras, enquanto que as cópias no estado **C** permitem apenas leituras. A consistência é garantida fazendo-se com que as diversas cópias de cada bloco estejam todas no estado **C** ou **I**, ou haja apenas uma cópia **S** e todas as demais estejam no estado **I**. Desta forma, a cada instante um determinado bloco é usado somente para leitura (cópias **C**) ou usado como dado local (cópia **S**), o que garante a consistência.

Caso um processador tente um acesso não permitido à cópia da sua cache, o estado desta cópia deve ser alterado antes que o acesso seja realizado. Esta alteração envolve envios de requisições ao restante do sistema de memória e eventuais transferências de dados e intervenções nas cópias das outras caches. As requisições enviadas pelas caches podem ser do tipo **TC** (pedido de transferência de cópia do bloco, a ser usado apenas para leitura), **TS** (pedido de transferência de cópia do bloco, a ser usado para leitura e escrita), e **DS** (declaração de que um bloco já presente no estado **C** deverá ser usado para escrita).

A Figura 2.1 ilustra como o estado de uma certa cópia pode ser alterado segundo os acessos tentados pelo processador e as requisições provenientes de outras caches. Por exemplo, se um bloco está no estado **I** e uma escrita é tentada, a cache envia um **TS** e aguarda a transferência do bloco, para então completar a escrita.

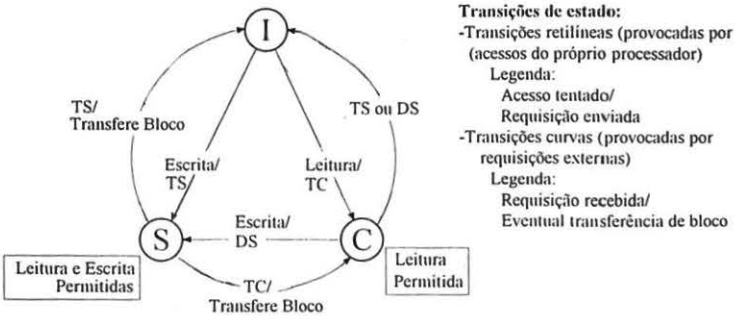


Figura 2.1: Diagrama de estados do protocolo Synapse.

Alguns detalhes foram omitidos da Figura 2.1 para maior simplicidade. Por exemplo, na passagem do estado **I** para o **S** durante uma tentativa de escrita, a escrita não deve ser realizada antes que todas as caches recebam o **TS** e mudem o estado das suas cópias para **I**. Isto implica em introduzir mecanismos de confirmação de

invalidações. Além disso, é possível a ocorrência de conflitos (como, por exemplo, o envio de dois **DS** simultâneos para o mesmo bloco), que devem ser resolvidos por algum *hardware* de arbitragem.

Existem diversos outros protocolos, alguns envolvendo atualizações das cópias, ao invés de invalidações. Alguns deles podem ser encontrados em [YaBL89,ArBa86].

2.3. Implementação dos Protocolos de Consistência com Diretórios Centralizados

Nos esquemas de garantia de consistência baseados em *hardware* os acessos que não podem ser totalmente resolvidos pela cache local causam o envio de requisições para a parte global do sistema de memória e para as demais caches locais. Estas requisições podem ser pedidos de transferência de blocos de dados, pedidos de invalidação, ou escritas (no caso de caches *write-through*).

A distribuição destas requisições exige a utilização de dispositivos ligados a todos os processadores e ao sistema de memória. Os dispositivos mais comuns são os barramentos farejadores e os diretórios centralizados.

Os barramentos farejadores (*Snooping-Buses*) são barramentos que ligam as caches locais à memória global. Além de agir como uma rede de interconexão entre as caches e o sistema de armazenamento principal, estes barramentos podem ser monitorados pelas caches locais, de forma que todas as transações efetivadas pelo barramento podem ser verificadas pelas caches. Caso uma cache detecte uma transação envolvendo dados nela copiados, ela poderá intervir nesta transação ou passar por alguma mudança de estado, conforme o protocolo utilizado. Por exemplo, num sistema com verificação intensiva das escritas e caches *write-through*, se uma cache percebe que outra cache está realizando uma escrita em um dado nela copiado, ela poderá invalidar ou atualizar a cópia, enquanto a escrita é enviada à memória global.

Um problema deste tipo de sistema é que se for utilizado um único barramento farejador, este barramento poderá se tornar o gargalo do sistema, limitando o número de processadores a poucas unidades. Este problema é especialmente sério quando utilizadas caches *write-through*, devido à grande largura de faixa exigida da memória global. Além disso, as requisições externas que eventualmente são geradas são enviadas a **todas** as caches para serem verificadas, e não apenas às que precisariam ser informadas, o que causa muita interferência. Este último problema pode ser reduzido através da utilização de um diretório auxiliar ligado a cada cache local, contendo informações sobre o conteúdo da cache. Este diretório monitora o barramento, enviando à cache apenas as requisições importantes, deixando esta livre para atender aos acessos do processador.

Uma alternativa para o uso dos barramentos farejadores é utilizar um diretório ligado à memória global, contendo informações sobre o conteúdo das caches e o estado das cópias locais [CeFe78,Tang76]. O diretório monitora as requisições enviadas aos módulos, verificando quais caches precisam ser notificadas. Desta forma, as caches só recebem as requisições necessárias, evitando-se interferências inúteis como nos sistemas baseados em barramentos farejadores.

A memória pode ser dividida em diversos módulos, cada qual contendo uma parte do espaço de endereçamento, associando-se um diretório a cada módulo. Cada um destes diretórios monitora os acesso enviados ao módulo de memória a ele associado. Desta forma, diversas verificações podem ser realizadas simultaneamente, com um conseqüente aumento da largura de faixa da memória principal.

3. Sistemas Avaliados

A seguir apresentamos a metodologia empregada e a arquitetura dos sistemas avaliados.

3.1. Métodos de Avaliação de Sistemas de Memória

Existem diversos métodos de avaliação de sistemas computacionais em geral, e de sistemas MIMD fortemente acoplados em especial. Parte destes métodos se baseia no modelamento estatístico do sistema em

questão, e a subseqüente resolução ou simulação destes modelos. Outros se baseiam no teste das próprias máquinas, ou no uso de programas que simulam a operação destas máquinas.

Os métodos baseados em modelamento estatístico são adequados para sistemas simples, como os desprovidos de cache. Para sistemas mais elaborados, o modelamento estatístico tende a encobrir diversos detalhes, fornecendo resultados grosseiros. Por exemplo, é muito difícil modelar o comportamento dos diversos protocolos de consistência que existem na literatura, especialmente durante iniciações, quando as caches estão vazias. Também é difícil analisar o impacto da codificação dos programas em cada tipo de sistema. Esta abordagem foi usada, por exemplo, em [VeLZ88] e [YaBL89] para sistemas com caches locais e diversos protocolos e em [BrDu83] para sistemas com caches locais e controle de consistência estático.

A simulação da operação de sistemas de memória é mais confiável do que o modelamento estatístico, mas pode apresentar uma carga computacional muito maior. Para reduzir a complexidade dos simuladores, é possível substituir alguns dos componentes do sistema de memória por modelos estatísticos, sem sacrificar substancialmente a precisão destes simuladores. Por exemplo, uma rede de interconexão de múltiplos estágios é bastante complexa para ser simulada, mas pode ser substituída por um modelo estatístico para que outros componentes do sistema sejam estudados com mais detalhes.

Para este trabalho, decidiu-se pelo uso da simulação operacional dos sistemas de memória. Como o porte dos sistemas envolvidos é pequeno (até oito processadores), não houve a necessidade de utilização de modelos estatísticos para nenhum dos componentes.

Os métodos baseados em simuladores operacionais se subdividem em duas classes distintas. Uma delas se baseia no uso de simuladores de processadores alimentados por programas paralelos para gerar acessos para o simulador de memória. Também podem ser usados "*traces*" gerados por multiprocessadores reais ou simulados.

A outra classe se baseia no uso de geradores aleatórios de acessos. Estes geradores usam parâmetros como a taxa média de acessos, percentagem de escritas, percentagem de acessos a dados globais, presença de "hot-spots[1]" etc.

Os geradores aleatórios são mais simples do que os baseados em programas paralelos, e a parametrização das correntes de acesso permite análises rápidas para correntes de diversas características. Este tipo de simulação foi usada em [ArBa86] para sistemas com caches locais e diversos protocolos de consistência.

A principal deficiência destes métodos é a dificuldade de caracterizar correntes de acessos reais através de parâmetros simples. Diversos detalhes, como o aproveitamento de linhas de cache e a presença de dependências entre acessos de processadores distintos, costumam ser ignoradas devido à dificuldade de representá-los por parâmetros numéricos. Além disso, a escolha dos valores para os parâmetros deve ser feita de forma cuidadosa.

Estes problemas podem ser resolvidos pelo uso de correntes reais de acesso, ou seja, pelo uso de "traces" de máquinas reais ou simuladores de processadores. No entanto, para gerar as correntes de acesso devem ser usados programas paralelos de características diversas, se possível programas próximos aos que serão executados no sistema a ser implementado. Raramente é possível encontra um conjunto sintético de programas que represente toda a carga da máquina-alvo, e se torna necessário interpretar os resultados obtidos para poder extrapolar estes resultados para programas diferentes.

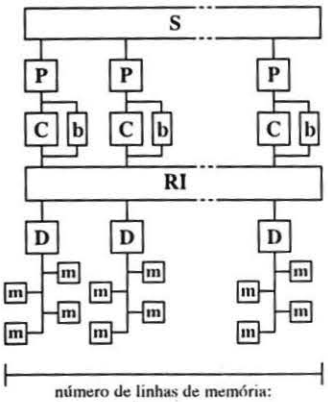
A escolha de um destes modelos de geração de correntes de acesso deve ser baseada no tipo de sistemas a serem simulados. Sistemas em que os acessos já realizados não tem influência sobre os novos acessos podem ser simulados com correntes de acesso aleatórias de modelos bastante simples. É o caso dos sistemas desprovidos de caches. Em sistemas com caches, acessos passados têm muita influência sobra a operação do sistema. Assim, para os estudos realizados neste trabalho, decidiu-se pelo uso de simuladores de processadores.

---

[1]Localidades mais referenciadas do que as demais. Em geral, tratam-se de semáforos ou outras formas de sincronização, e tendem a favorecer o surgimento de conflitos.

3.2. Descrição dos Sistemas Avaliados

Duas arquiteturas foram avaliadas, uma baseada em protocolo de posse de blocos e outra em protocolo *write-through*. Ambas se baseiam no esquema ilustrado na Figura 3.1.



P = Processador (8 ao todo)
C = Cache local
RI = Rede de Interconexão
D = Diretório centralizado
m = elementos de armazenamento (quatro por linha)
b = buffer para escrita na memória principal (usado apenas em sistemas write-through).
S = memória especial para sincronização

Figura 3.1: Esquema básico dos sistemas avaliados.

São usados oito processadores em todos os sistemas. O porte destes sistemas é modesto, de forma que deve-se tomar cuidado ao extrapolar os resultados aqui presentes para sistemas maiores.

Deve-se notar que não é considerado o efeito dos acessos a instruções (busca de operações), apenas o dos acessos a dados. Os acessos a instruções apresentam características bem diferentes dos acessos a dados, sendo, de forma geral, mais adequados ao uso de caches (por não serem modificados durante a execução e por apresentarem maior localidade). Um estudo sobre caches dedicadas a instruções pode ser encontrado em [SmGo85]. Também não são consideradas as operações de entrada e saída de dados.

A memória especial para sincronização, de latência nula, foi introduzida para evitar que os acessos de sincronização interferissem nos resultados. Acessos de sincronização apresentam padrões bastante peculiares (apresentando mais conflitos do que o normal, podendo ocorrer *hot-spots*), merecendo um estudo à parte.

3.2.1. Organização Interna das Caches

As caches são infinitas, o que elimina discussões sobre a sua capacidade, formas de mapeamento ou políticas de reposição. Esta consideração é pouco realista, mas simplifica muito os estudos por isolar os problemas de limitação da capacidade de armazenamento das caches dos problemas mais diretamente relacionados aos protocolos de consistência. Assim, as únicas perdas nas caches que ocorrem se devem a iniciação (as caches são iniciadas vazias) ou a invalidações de cópias, e não a reposições de cópias.

Uma conseqüência negativa da consideração de caches infinitas é que as reposições são muito importantes em sistemas que operam com **protocolos de atualização** de cópias nas caches, ao invés de usarem protocolos de invalidação, pois nestes sistemas as reposições são a única forma de eliminação de cópias inúteis das caches, que podem causar interferências indesejáveis. Devido a este problema, e como boa parte dos sistemas reais se baseia em invalidações de cópias, os estudos aqui apresentados ficaram restritos a este tipo de sistema.

As caches transferem dados em blocos de quatro palavras consecutivas (linhas de dados), a menos das escritas *write-through* (quando utilizadas), que são feitas palavra a palavra e não aproveitam as vantagens do *pipelining* das transferências. As escritas *write-through* são enviadas por um *buffer* à memória principal, e os

processadores remetentes podem continuar operando, sem porém enviar novos acessos até que os anteriores sejam completados (para evitar problemas de consistência).

3.2.2. Organização da Memória Principal

A organização da memória principal é do tipo L-M [BrDu83]. Este tipo de memória está organizado em **l** linhas de **m** (neste caso, quatro) elementos de armazenamento cada, daí o seu nome. Os endereços são entrelaçados de forma que cada linha contenha vários conjuntos de quatro palavras contíguas. Conjuntos contíguos são colocados em linhas diferentes. Assim, os endereços de 0 a 3 estão na linha 1, os endereços de 4 a 7 estão na linha 2 e assim por diante. Transferências de blocos de dados são feitas com quatro palavras consecutivas pertencentes a uma mesma linha de elementos de armazenamento, e podem ser feitas com uso de *pipelining*, com **quatro** ciclos de latência para a primeira palavra e **um** ciclo para cada uma das palavras subseqüentes.

O controle de consistência é realizado por diretórios centralizados, um diretório ligado a cada linha de elementos de armazenamento. Com isto, permite-se que a largura de faixa do *hardware* de verificação cresça junto com a largura de faixa da memória principal, o que não ocorre com barramentos farejadores. Doravante, será chamada **módulo de memória** a associação entre uma linha de quatro elementos de armazenamento e um diretório.

O número de módulos de memória interfere diretamente na largura de faixa da memória principal, sendo um parâmetro de grande importância nas avaliações. Simulações preliminares foram usadas para se encontrar um valor médio para o número de módulos, que permitisse futuras extrapolações dos resultados para valores diferentes. No entanto, notou-se que a variação do desempenho segundo o número de módulos era diferente para cada tipo de sistema, sendo muito mais pronunciado para sistemas com protocolos *write-through*. Como não foi encontrado um valor típico para este parâmetro (a largura de faixa de sistemas reais pode variar muito), a solução encontrada foi realizar simulações com várias quantidades de módulos de memória (um, dois, quatro e oito módulos). Com mais de oito módulos, o desempenho passa a variar muito pouco.

A rede de interconexão não apresenta conflitos, além daqueles causados por acessos enviados a um mesmo módulo ou a um mesmo processador. Com isto, a rede de interconexão se assemelha a uma *crossbar*, com latência de um ciclo para cada travessia, seja em direção aos processadores ou à memória. Estas características foram escolhidas para reduzir a influência da rede de interconexão no desempenho do restante do sistema, e ao mesmo tempo permitir que o tráfego efetivo fosse medido, para que se pudesse obter a largura de faixa necessária. Nas simulações em que era importante limitar a largura de faixa, esta era regulada pelo número de módulos de memória, e não pela rede de interconexão.

3.2.3. Protocolos Simulados

São simulados os protocolos Synapse (SY) e Write-Through com invalidação de cópias desatualizadas (WT). Outros protocolos foram testados, mas os resultados destes foram bastante próximos dos encontrados para os protocolos já citados, de forma que estes dois serão usado como exemplos de protocolos de posse de blocos e protocolos *write-through*.

Um parâmetro particularmente importante é o tamanho das linhas de cache usadas pelos protocolos de consistência, que pode ou não ser igual ao tamanho dos blocos usados nas transferências de dados. No caso dos protocolos de posse de blocos, o tamanho considerado é de quatro palavras, enquanto que no caso dos protocolos *write-through* são usadas linhas de uma palavra apenas. Estes valores se baseiam em sistemas reais, sendo bastante adequados para este estudo. No entanto, é preciso ter em mente a sensibilidade do desempenho dos sistemas a este parâmetro.

4. Resultados

Como exemplo de problema a ser solucionado, será utilizada a decomposição LU de matriz densa. Este problema apresenta diversas aplicações reais, e pode ser paralelizado de diversas formas [Rose92], uma das quais

será mostradas aqui. O mesmo programa básico foi testado em quatro variações, cada qual apresentando um padrão de acessos distinto. Um conjunto mais extenso de simulações pode ser encontrado em [Alve93].

4.1. Descrição dos programas

Dada uma matriz quadrada não singular **A**, a decomposição LU consiste em encontrar uma matriz triangular inferior **L** com diagonal principal unitária e uma matriz triangular superior **U** tais que:

$A = L \times U$

Os métodos aqui apresentados se baseiam na eliminação de Gauss sem pivotamento [Rose92]. Nas simulações, a ordem da matriz **A** foi 20×20.

O programa paralelo utilizado decompõe a matriz **A** em linhas. Este programa, escrito num dialeto de Pascal, está ilustrado no Programa 1. Oito instâncias da rotina LU são executadas em paralelo.

```
program LUpar;

const N = 20;
      NP = 8;
var   A : array [1..N,1..N] of single;
      i,j,k : integer;

procedure LU;

     var  i,j,k : integer;
          InvA,f : single;

     begin
     for k := 1 to N - 1 do
          begin
          InvA := 1 / A [k,k];
          while ### ainda há linhas a serem atualizadas ###
               begin
               ### determina-se a próxima linha (i) ###
               f := A [i,k] * InvA;
               for j := k + 1 to N do
                    A [i,j] := A [i,j] - f * A [k,j];
               A [i,k] := f.
               end;
          ### barreira ###
          end;
     end;

begin
cobegin
LU;LU;LU;LU;LU;LU;LU;LU;
coend
end.
```

Programa 1: Decomposição LU paralela baseada em linhas.

Para melhor compreender este programa é preciso notar algumas das suas características. Cada iteração do laço mais externo, de índice **k**, atualiza uma submatriz de **A**, com base na linha de **A** imediatamente acima desta submatriz, e da coluna de **A** imediatamente à esquerda, como na Figura 4.1

Figura 4.1: Etapa da decomposição LU.

A expressão matricial para cada iteração do laço externo é:

$A \leftarrow A - l \times u$

Onde **A** é a submatriz, **l** é a coluna da esquerda e **u** é a linha superior desta submatriz. Cada processador toma uma linha de índice **i**, e realiza uma composição linear entre esta linha e a linha de índice **k**, colocando o resultado de volta na linha de índice **i**.

Alguns detalhes foram omitidos do Programa 1 e substituídos por um texto que os representa entre '###'s. A parte relativa à implementação das barreiras não é relevante neste trabalho. O que é preciso definir é como as linhas da matriz **A** serão distribuídas entre os processadores. É esta distribuição, juntamente como a forma de armazenamento de matrizes bidimensionais, que leva a padrões de acesso diferentes. Ao todo, são apresentadas quatro versões do Programa 1, denominadas LU1 a LU4.

Nos programas LU1 e LU3, buscou-se uma maior localidade de referências, fazendo com que cada linha da matriz A pudesse ser escrita por apenas um processador. Desta forma, a alocação de linhas pelos processadores era fixa. Nos programas LU2 e LU4 utilizou-se um esquema de alocação automática de linhas entre os processadores, de forma que qualquer processador pudesse atualizar qualquer linha da matriz **A** disponível. Com isto, e com a eliminação da barreira entre iterações do laço mais externo (de índice **k**), obteve-se um melhor balanceamento de carga entre os processadores, com um conseqüente aumento de desempenho. Este aumento de desempenho não é relevante a este estudo, de forma que é preferível, nos gráficos de *Speed-Ups*, prestar mais atenção à comparação entre o desempenho dos sistemas do que ao desempenho bruto de cada um.

O armazenamento da matriz também é relevante ao padrão de acessos. Se este for feito linha por linha, uma cache trará elementos de uma mesma linha a cada perda que ocorrer, e estes elementos serão eventualmente utilizados. Se o armazenamento de **A** for feito coluna por coluna, uma cache trará elementos de uma mesma coluna a cada perda, e estes elementos não só serão inúteis, como também poderão provocar futuras interferências entre caches.

As características dos quatro programas estão sumariadas na Tabela 4.1. É importante notar que todos estes programas apresentam uma taxa de escritas de cerca de 33%, que pode ser considerada alta.

	Matriz armazenada por linhas	Matriz armazenada por colunas
Cada linha da matriz é atualizada sempre por um mesmo processador	LU1	LU3
As linhas da matriz podem ser atualizadas por qualquer processador	LU2	LU4

Tabela 4.1: Características básicas dos programas de decomposição LU.

4.2. Resultados das Simulações

Os resultados estão expostos nesta seção na forma de gráficos. Os gráficos de *Speed-Up* apresentam as curvas de desempenho dos sistemas SY (Synapse) e WT (*Write-Through*) em função do número de módulos de memória utilizados (de um a oito). Nos gráficos de *Speed-Ups* foram adicionadas duas linhas horizontais para indicar o desempenho de um sistema ideal (Super), em que a latência dos acessos é nula, e o de um sistema sem caches, com oito módulos de memória global com tempos de acesso de dois ciclos.

Os demais gráficos indicam o número de eventos que ocorreram **para cada acesso global**. Tratam de eventos relacionados às caches (perdas, invalidações e intervenções externas) e do tráfego na rede de interconexão nos sentidos processadores-memória (P-M) e memória-processadores (M-P). Cada invalidação ou transferência de bloco de dados é contado quatro vezes (uma por palavra). Estes dados são pouco sensíveis ao número de módulos utilizados, de forma que apenas os resultados obtidos com oito módulos são mostrados.

Das Figuras 4.2 e 4.4 (desempenho com os programas LU1 e LU3) nota-se que até o *Speed-Up* do sistema SUPER ficou abaixo de 6, apesar de oito processadores serem usados. Isto se deve ao desbalanceamento de carga provocado pela alocação estática das linhas de A, e também pela introdução da barreira. Como já foi comentado, este desbalanceamento não é relevante a este estudo, mesmo porque para matrizes maiores, este desbalanceamento se tornaria cada vez menor.

A Figura 4.2, relativa ao desempenho do programa LU1, mostra o comportamento esperado de um programa bem implementado. O número de invalidações é nulo, as perdas se devem apenas à iniciação das caches e o *Speed-Up* dos sistemas com caches e oito módulos de memória ficou relativamente próximo do alcançado pelo sistema Super (cerca de 85%). Com um número menor de módulos de memória, o desempenho do sistema SY foi apenas um pouco inferior, enquanto que o do sistema WT caiu sensivelmente. A razão para isto pode ser vista no gráfico de tráfego na rede de interconexão: o tráfego processadores-memória em sistemas WT foi cerca de quatro vezes maior do que o do sistema SY, o que causou um grande número de conflitos no sistema WT de um módulo de memória, devido à sua pequena largura de faixa.

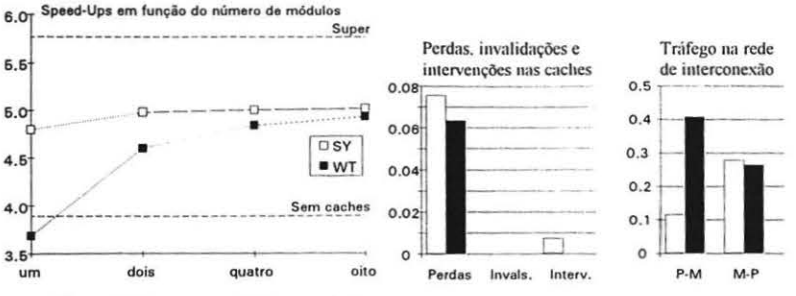


Figura 4.2: Desempenho do programa LU1.

O programa LU2 é bastante semelhante ao programa LU1, mas usa alocação dinâmica das linhas para os processadores. O resultado é uma menor localidade de referências e o compartilhamento de dados modificáveis. No entanto, como o algoritmo é orientado por linhas, e como o armazenamento da matriz A também é feito por linhas, ainda há um bom aproveitamento dos blocos de dados trazidos da memória global para as caches.

Na Figura 4.3, gráfico de *Speed-Up*, pode-se notar um ligeiro aumento de desempenho por parte de todos os sistemas, com relação ao programa LU1. Este aumento se deve ao já comentado balanceamento de carga. Quando comparados ao desempenho do sistema Super, nota-se uma queda do desempenho dos sistemas com cache, devido à menor localidade das referências. Os sistemas com oito módulos apresentaram desempenhos da ordem de

70% do desempenho do sistema Super. Esta queda se deve ao aumento do número de perdas e ao surgimento de invalidações nas caches.

Como era de se esperar, o desempenho do sistema WT com um módulo de memória foi bastante inferior ao do sistema com oito módulos. A queda foi de cerca de 36%, maior do que no programa LU1, devido ao maior número de requisições enviadas à memória principal. O sistema SY de um módulo de memória também apresentou uma queda de desempenho com relação ao sistema de oito módulos (18%), porque o número de requisições enviadas à memória aumentou muito em comparação com o programa LU1.

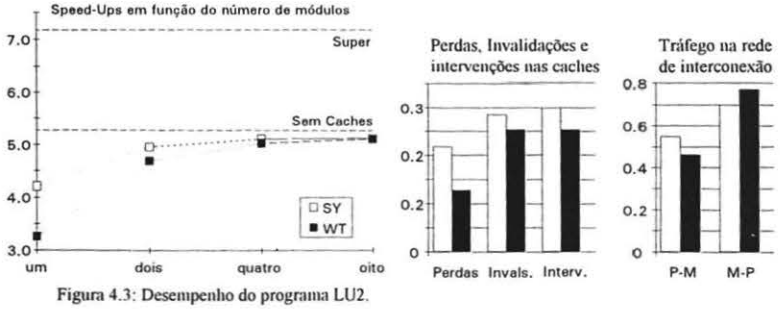


Figura 4.3: Desempenho do programa LU2.

Para estes dois primeiros programas, nota-se que o sistema SY se saiu melhor do que o WT, especialmente quando foram utilizados poucos módulos de memória. Com um número muito grande de módulos de memória, a comparação fica comprometidas pelas imprecisões da simulação.

A situação se altera bastante com o programa LU3. Neste caso, o armazenamento da matriz **A** em colunas fez com que ocorresse um grande número de perdas, invalidações e intervenções nas caches do sistema SY. Isto se deve à forma como é feito o controle de consistência neste tipo de sistema: as alterações de estado das cópias são feitas em blocos de dados, de forma que quando um processador realiza uma escrita, as cópias do dado alterado e dos dados vizinhos são invalidadas. O resultado disto é um grande número de invalidações inúteis, que causam mais perdas nas caches.

Esta invalidação das cópias de dados vizinhos não é necessária do ponto de vista de garantia de consistência, mas deve ser feita por razões práticas relativas à implementação do sistema. Para tornar possível a invalidação de uma única palavra em um bloco, seria necessário associar um estado a cada palavra. Assim, durante uma transferência de bloco de dados, várias ações do protocolo teriam que ser feitas, tornando complexa esta operação. Outra solução seria usar blocos de apenas uma palavra, mas isto tornaria ineficientes as transferências.

O desempenho do sistema WT foi menos afetado pela má distribuição dos dados do que o do sistema SY. Isto ocorre graças às invalidações dado-a-dado deste tipo de sistema, que evita invalidações inúteis e as conseqüentes perdas nas caches. Neste programa em particular, a cada perda que ocorre numa cache, várias palavras de uma mesma coluna da matriz **A** são trazidas. As palavras adicionais não serão utilizadas, podendo vir a sofrer invalidações, mas a palavra efetivamente utilizada será mantida na cache, evitando novas perdas.

A Figura 4.4 ilustra esta situação. Nota-se um número muito elevado de perdas e requisições no sistema SY, e um número alto, mas sensivelmente inferior, de problemas semelhantes no sistema WT. Quando observado o gráfico de *Speed-Up*, nota-se um desempenho do sistema WT de 67% do desempenho do sistema Super, contra os 43% do sistema SY.

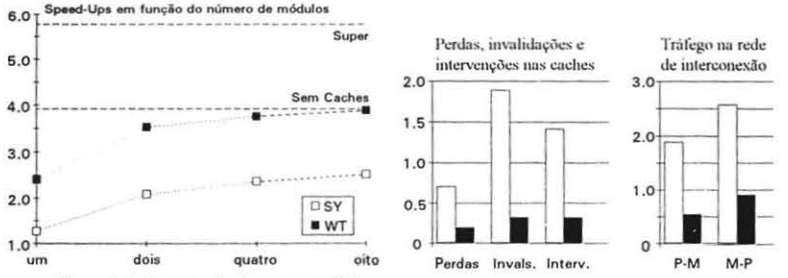


Figura 4.4: Desempenho do programa LU3.

O programa LU4 é uma versão do programa LU2 com armazenamento da matriz **A** feito em colunas, como foi feito para o programa LU3 com relação ao programa LU1. Os resultados representam um caso extremo, em que há bastante paralelismo "disponível" (como pode ser visto do bom desempenho do sistema S) mas a distribuição dos dados não segue a orientação das linhas de cache e nem favorece o reaproveitamento dos dados. Este programa pode ser visto como um exemplo de programa feito para obter paralelismo sem considerar problemas relacionados ao sistema de memória.

A principal diferença deste programa para o anterior é que com a alocação variável das linhas, o sistema WT perdeu a possibilidade de reaproveitamento de parte dos dados. O sistema SY já não podia aproveitar estes dados, devido às já comentadas invalidações em blocos, de forma que o desempenho do sistema WT caiu de forma mais pronunciada do que o do sistema SY. O número de invalidações e intervenções nas caches do sistema WT praticamente triplicou, enquanto que o do sistema SY, que já era bastante elevado, se alterou pouco, como pode ser visto na Figura 4.5. Ainda assim, o desempenho do sistema WT foi superior ao do sistema SY. Os *Speed-Ups* para os sistemas SY e WT de oito módulos são de 44% e 67% do *Speed-Up* do sistema Super, respectivamente.

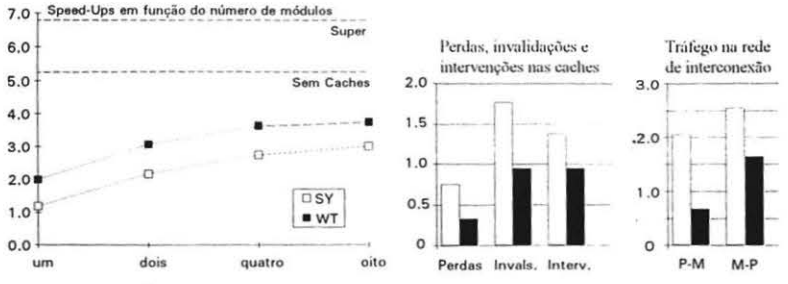


Figura 4.5: Desempenho do programa LU4.

Os resultados apresentados são fortemente dependentes do tamanho dos blocos utilizados. É de se esperar que, quanto maiores os blocos, maior a eficiência das transferências de blocos de dados, e maior a possibilidade de interferência entre caches. Estes fatores implicam numa necessidade maior de se implementar programas que utilizem os blocos de dados de forma conveniente.

Como os sistemas com caches *write-through* fazem o controle de consistência separadamente para cada dado, e não em blocos, a influência do tamanho dos blocos é bem menor do que em protocolos como o SY.

5. Comentários Finais

Em geral, trabalhos que estudam protocolos de consistência [ArBa86,YaBL89,YYFu85] utilizam programas bem implementados, ou utilizam parâmetros otimistas em análises estatísticas (por exemplo, probabilidades de perdas nas caches inferiores a 5% e compartilhamento de dados inferior a 10%). Nestas condições, os protocolos de posse de blocos têm desempenho superior aos *write-through*. Mas é preciso questionar se é possível conseguir programas tão bem implementados para qualquer aplicação, e qual é o custo de desenvolvimento de tais programas.

Os resultados aqui apresentados fazem parte de um estudo maior [Alve93] que envolve alguns protocolos adicionais e outros programas (outras formas de decomposição LU, multiplicação de matrizes, resolução de sistemas tridiagonais etc), buscando analisar o impacto da qualidade da codificação no desempenho dos sistemas. De forma geral, estes resultados reforçam a principal conclusão deste trabalho: a viabilidade de sistemas *write-through*, muito criticados na literatura, por apresentarem desempenhos menos dependentes da codificação de programas do que os sistemas com protocolos de posse de blocos.

Nos programas LU1 e LU2, em que a distribuição dos dados favorece o uso de dados de endereços contíguos, ambos os sistemas apresentaram desempenhos semelhantes. Nos programas LU3 e LU4, em que dados contíguos são usados por processadores distintos, caracterizando uma codificação descuidada, o desempenho do sistema *write-through* foi 54% (LU3) e 25% (LU4) melhor do que o do Synapse.

Boa parte das críticas aos sistemas *write-through* se deve à necessidade de implementação de uma memória principal com largura de faixa mais alta do que a necessária a sistemas com protocolos de posse de blocos. De fato, notou-se nas simulações o baixo desempenho dos sistemas *write-through* com baixa largura de faixa. No entanto, o aumento desta largura de faixa pode ser conseguido a custos aceitáveis, com o uso de múltiplos módulos de memória, associados a diretórios centralizados. Além disso, uma análise do tráfego "instantâneo" na rede de interconexão durante as simulações mostrou que em 95% do tempo não havia mais de duas requisições, ou dados, trafegando na rede num mesmo ciclo, o que mostra que a rede de interconexão não precisa ter uma largura de faixa muito elevada.

Bibliografia

[Agar88] AGARWAL, A.; SIMONI, R.; HENNESSY, J.; HOROWITZ, M.- *An Evaluation of Directory Schemes for Cache Coherence* - Proc. 15th Int. Symp. on Comp. Architecture, CS Press, Los Alamitos, Calif. pp 280-289, Jun. 1988.

[Alve93] ALVES, Carlos E. R. - *Estudo de Arquitetura de Memória para Máquinas MIMD Fortemente Acopladas* - Dissertação de Mestrado, ITA-CTA, 1993.

[ArBa84] ARCHIBALD, James & BAER, Jean-Loup - *An Economical Solution to the Cache Coherence Problem* - Proc. of 11th. Int. Symp. on Computer Architecture, pp 355-361, Jun.1984.

[ArBa86] ARCHIBALD, James & BAER, Jean-Loup - *Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model* - ACM Trans. on Comp. Systems, Vol. 4, no. 4, pp 273-298, Nov. 1986.

[BrDu83] BRIGGS, F. A. & DUBOIS, M. - *Effectiveness of Private Caches in Multiprocessor Systems with Parallel-Pipelined Memories* - IEEE Trans. on Comp., Vol C. 32, no. 1, pp 48-59, Jan. 1983.

[CeFe78] CENSIER, L. M & FEAUTRIER, P. - *A New Solution to Coherence Problems in Multicache Systems* - IEEE Trans. On Comp., Vol C-27, no. 12, pp 1112-1118, Dec. 1978.

[Fran84] FRANK, S.J. - *Tightly Coupled Multiprocessor System Speeds Memory Access Times* - Electronics, Vol. 57, no. 1, pp 164-69, Jan. 1984.

[Ghar90] GHARACHORLOO, K.; LENOSKI, D.; LAUDON, J.; GIBBONS, P.; GUPTA, A.; HENNESSY, J. - *Memory Consistency an Event Ordering in Scalable Shared-Memory Multiprocessors* - Technical Report CSL-TR-89-405, Computer Systems Laboratory, Stanford University, California, Mar.1990.

[Lamp79] LAMPORT, Leslie - *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs* - IEEE Trans. on Comp., Vol C. 28, no. 9, pp 690-691, Set. 1979.

[PaPa84] PAPAMARCOS, M. & PATEL, J. - *A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories* - Proc. of 11th. Int. Symp. on Computer Architecture, pp 348-354, Jun. 1984.

[Rose92] ROSE, Luiz A. - *Introduction to Parallel Numerical Algorythms* Anais da II Jornada EPUSP/IEEE em Sistema de Computação de Alto Desempenho, Vol 3, Maio 1992.

[SmGo85] SMITH, James E. & GOODMAN, James R. - *Instruction Cache Replacement Policies and Organizations* - IEEE Trans. on Comp., Vol C-34, no. 3, pp 234-241, Mar. 1985.

[Smit82] SMITH, Alan Jay - *Cache Memories* - Computing Surveys, Vol.14, No. 8, pp 478-580, Set 1982.

[Tang76] TANG, C.K. - *Cache System Design in the Tightly Coupled Multiprocessor System* - Proc. AFIP Nat. Comput. Conf., Vol. 45, pp 749-753, 1976.

[VeLZ88] VERNON, Mary K.; LAZOWSKA, Edward D.; ZAHORJAN, John - *An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols* - Proc. 15th Int. Conf. on Parallel Processing, CS Press, Los Alamitos, Calif. pp 308-314, Jun.1988.

[YuBL89] YANG, Qing; BHUYAN, Laxmi N.; LIU, Bao-Chyn - *Analysis and Comparison of Cache Coherence Protocols for a Packet-Switched Multiprocessor* - IEEE Trans. on Comp., Vol. C-38,no. 8, pp 1143-1153, Ago. 1989

[YYFu85] YEN, Wei C.; YEN, David W. L.; FU, King-Su - *Data Coherence Problem in a Multicache System* - IEEE Trans. on Comp., Vol C-34, no. 1, pp 56-65, Jan. 1985.