A TÉCNICA *LOCK ACQUIRER PREDICTION* E A SUA APLICAÇÃO EM SISTEMAS DE MEMÓRIA COMPARTILHADA DISTRIBUÍDA

Cristiana Bentes Seidel

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Cláudio Luis de Amorim, Ph.D.

Prof. Ricardo Bianchini, Ph.D.

Prof. Edil Severiano Tavares Fernandes, Ph.D.

Prof. Orlando Gomes Loques Filho, Ph.D.

Prof. Sérgio Takeo Kofuji, D.Sc.

Prof. Wagner Meira Junior, Ph.D.

Rio de Janeiro, RJ - Brasil

SETEMBRO DE 1998

SEIDEL, CRISTIANA BENTES

A Técnica *Lock Acquirer Prediction* e a sua Aplicação em Sistemas de Memória Compartilhada Distribuída [Rio de Janeiro] 1998

XV, 141 p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 1998)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Sistemas Operacionais

2 - Memória Compartilhada Distribuída

I. COPPE/UFRJ II. Título (série)

A meu pai,

José Flávio.

Agradecimentos

Primeiramente, gostaria de agradecer aos meus orientadores Cláudio Amorim e Ricardo Bianchini, pela paciência, pelo constante estímulo e pela confiança em meu trabalho.

Agradeço à Universidade do Estado do Rio de Janeiro, e em particular aos Profs Adalbert Pfeiffer, Orlando Bernardo Filho e Nival Nunes de Almeida, pelo apoio, compreensão e pela liberação concedida para que eu pudesse continuar com tranquilidade o meu trabalho de doutorado.

Ao Núcleo de Computação Eletrônica agradeço pelo uso do sistema SP-2. Em especial a Sérgio Guedes, cuja presteza e eficiência na gerência do sistema facilitaram muito o desenvolvimento e a análise dos meus experimentos.

A Luis Rodolpho Monnerat, que também trilhou os difíceis caminhos de TreadMarks e do SP-2, agradeço pelas aplicações de migrações sísmicas e por toda a ajuda com detalhes de TreadMarks e do SP-2.

A todos os participantes das "reuniões de quarta-feira do Ricardo", Luiz Favre, Vinicio, Sílvio, Eduardo, Carla, Raquel, Lauro e Rodrigo. Suas sugestões e comentários tiveram grande influência nos rumos e no desenvolvimento da minha tese e nossas reuniões tiveram contribuição fundamental na minha formação como pesquisadora.

Às minhas companheiras de sala e sofrimento, Paula, Clícia e Carla, pela amizade, pelas longas conversas e pelos momentos de desabafo. Tirar "bugs" dos meus sistemas foi mais agradável (se é que isto é possível!) com a companhia de vocês.

Aos amigos Malena, Carol, Raquel, Lauro e Rodrigo, agradeço pelas discussões, comentários e sugestões no meu trabalho e pricipalmente pelos momentos de descontração. Nossos almoços, as comemorações intermináveis do aniversário da Malena, os jogos de tênis e nossas viagens para o SBAC tornaram os meus anos de COPPE bem mais divertidos.

A Ricardo devo ainda um agradecimento especial, por ter sido muito mais do que um grande orientador, mas um grande amigo e companheiro, sempre presente nas horas mais importantes, sejam elas técnicas ou não.

A toda minha família pelo apoio e principalmente pelo incentivo. Um agradecimento especial à minha mãe Vera e à minha irmã Isabela, meu filho não poderia ter tido melhores "mães" durante a minha ausência. Sem o apoio e a ajuda das duas, eu jamais poderia ter me dedicado tanto a este doutorado.

Aos dois homens da minha vida, Paulo e João Pedro, que aguentaram juntos com muita compreensão e bom-humor todas as minhas ausências. Aos dois eu agradeço principalmente por me fazerem esquecer, por vários momentos, da existência de *software DSMs* e protocolos de coerência.

A Paulo, meu marido e companheiro, agradeço por estar ao meu lado em todos os momentos difíceis e decisivos da minha vida. Seu amor e carinho fazem com que isso tudo valha a pena.

E finalmente, um agradecimento muito especial àquele que foi o meu maior incentivador. Àquele que sempre me apoiou incondicionalmente e que me deu exemplo de coragem e determinação, a quem eu dedico essa tese, meu pai José Flávio.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc)

A Técnica *Lock Acquirer Prediction* e a sua Aplicação em Sistemas de Memória Compartilhada Distribuída

Cristiana Bentes Seidel

Setembro/1998

Orientadores: Cláudio Luis de Amorim e Ricardo Bianchini

Programa: Engenharia de Sistemas e Computação

Esta tese apresenta a técnica *Lock Acquirer Prediction* (LAP), que prevê dinamicamente a ordem de transferência dos *locks* numa aplicação paralela, e um conjunto de três sistemas *software Distributed Shared Memory (software DSM)* desenvolvidos para explorar as vantagens oferecidas por LAP. Os três sistemas propostos exploram LAP de forma diferente e, no paradigma de memória compartilhada, utilizam modelos de programação distintos. Tais modelos têm implicações no uso de LAP, no desempenho e na complexidade dos sistemas. Nossos resultados mostram que LAP consegue acertar suas previsões com alta precisão e permite sensível redução no *overhead* de busca de dados compartilhados acessados dentro de uma seção crítica. Mostramos também que o aumento da complexidade do modelo de programação permite que LAP seja usada de forma agressiva e simplifica o protocolo de coerência empregado. Nossos resultados mostram ainda que o aumento da complexidade do modelo não necessariamente leva a ganhos de desempenho, o que é diferente do esperado intuitivamente. Concluímos que LAP merece a atenção dos projetistas de sistemas *software DSM*, uma vez que pode alcançar bom desempenho sem a necessidade de hardware específico ou suporte de compiladores. Além disso, concluímos que, dentre os modelos de programação estudados, somente o mais complexo alcança desempenho que justifique a sua complexidade de programação.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

The Lock Acquirer Prediction Technique and its Use in Distributed Shared-Memory Systems

Cristiana Bentes Seidel

September, 1998

Thesis Supervisors: Cláudio Luis de Amorim and Ricardo Bianchini

Department: Computing and Systems Engineering

In this thesis we propose the Lock Acquirer Prediction (LAP) technique for software-based distributed shared-memory systems (software DSMs). LAP dynamically predicts the order of lock transfers in parallel applications and thus allows software DSMs to update the data accessed in critical sections even before the corresponding locks are requested. We also propose three software DSMs that exploit LAP assuming programming models with varying degrees of complexity. We study the impact of these models on the usefulness of LAP and on the performance and complexity of our software DSMs. Our results show that LAP is very successful at its predictions, leading to significant data access overhead reductions for our applications. Our results also demonstrate that increasing the complexity of the programming model simplifies the software DSMs, while allowing them to exploit LAP more aggressively. However, increases in programming model complexity do not always lead to performance improvements, which counters the common wisdom in the research community. Based on these results, we conclude that the LAP technique deserves the attention of software DSMs designers, as it can achieve good performance without any hardware or compiler support. Furthermore, we conclude that only the most complex programming model we study achieves performance that justifies its programming complexity.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A programação paralela não é uma tarefa trivial. O programador, em geral, deve se preocupar tanto em explorar o paralelismo potencial da aplicação como em estabelecer a interação entre os processos paralelos. O paradigma de programação paralela determina como essas tarefas são realizadas. Os dois principais paradigmas de programação paralela diferem na forma como os processos interagem: passagem de mensagens ou memória compartilhada.

No paradigma de programação por passagem de mensagens, a interação entre os processos é realizada de forma explícita através de primitivas de envio e recebimento de mensagens. O paradigma de programação com memória compartilhada assume a existência de um espaço de endereçamento global e a interação entre os processos é realizada de forma implícita através de leituras e escritas em estruturas de dados compartilhadas.

A programação no paradigma por memória compartilhada é considerada mais simples porque evita que o programador se preocupe em orquestrar a comunicação entre os processos com passagem de mensagens explícitas. Considere como exemplo uma aplicação simples e regular, conhecida como SOR (*Sucessive Over Relaxation*), que simula o resfriamento de chapas de metal, em que cada elemento da chapa é atualizado de acordo com seus vizinhos ao norte, sul, leste e oeste. Em SOR, uma matriz é dividida em blocos de linha (submatrizes) e cada processo p_i computa a

Passagem de Mensagens

```
for num_iterações
    if (pid é par)
        if (pid ≠ 0)
            envia primeira linha para pid-1;
            recebe limite superior de pid-1;
        envia última linha para pid+1;
        recebe limite inferior de pid+1;
    else
        if (pid ≠ P-1)
            envia última linha para pid+1;
            recebe limite inferior de pid+1;
        envia última linha para pid-1;
        recebe limite inferior de pid-1;
    for num_linhas
        computa;
```

Memória Compartilhada

```
for num_iterações
    for num_linhas
        computa;
    barreira;
```

Figura 1.1: Aplicação SOR para os dois paradigmas de programação.

sua submatriz, utilizando os valores das bordas das submatrizes dos processos vizinhos p_{i-1} e p_{i+1}. SOR é uma aplicação facilmente implementável no paradigma de passagem de mensagens, já que é extremamente regular e são conhecidos estaticamente: (i) quais processos devem enviar e receber cada mensagem (troca ocorre entre vizinhos); (ii) exatamente quais dados devem ser trocados entre os processos (bordas das matrizes); (iii) e quando a troca de mensagens deve ocorrer (antes da computação).

A figura 1.1 mostra o código dessa aplicação para um número par de processadores segundo o paradigma de programação com passagem bloqueante de mensagens à esquerda e segundo o paradigma de memória compartilhada à direita. Conforme podemos observar, o paradigma de passagem de mensagens exige que antes da computação de cada submatriz, cada processo envie as suas bordas e receba as bordas dos vizinhos. Em contraste, no paradigma de memória compartilhada, o programador se preocupa basicamente com a computação das submatrizes, dado que os valores das bordas podem ser acessados diretamente através de operações de leitura e escrita. A única preocupação se torna, então, evitar condições de corrida através da correta sincronização dos processos, através de *locks* ou barreiras por exemplo [8], nos acessos à memória compartilhada. Note, entretanto, que na programação com

passagem de mensagens também existe a necessidade de sincronização, mas ela é atingida implicitamente através das próprias primitivas de comunicação. O programa SOR, por exemplo, tem implementação bastante diferente quando as primitivas de comunicação não são bloqueantes.

SOR ilustra que, mesmo para aplicações ideais no paradigma de passagem de mensagens, programas baseados em memória compartilhada são consideravelmente mais simples. Aplicações não-regulares ou com padrões de compartilhamento não estaticamente definidas tornam a programação com passagem de mensagens ainda mais complexa, mas não afetam a programação com compartilhamento de memória. O avanço da tecnologia de compiladores vai eventualmente simplificar a programação nos dois paradigmas, mas isso é ainda bastante prematuro.

Durante muito tempo, o uso do paradigma de programação com memória compartilhada esteve associado a multiprocessadores com memória centralizada/única. Entretanto, arquiteturas desse tipo apresentam contenção no acesso ao barramento de memória, limitando a sua escalabilidade. Sistemas com memória compartilhada distribuída (figura 1.2), ou *DSM (Distributed Shared Memory)*, unem a facilidade de programação do paradigma de memória compartilhada à escalabilidade de ambientes distribuídos. Esses sistemas assumem normalmente a existência de um processo por processador e, por este motivo, no restante do texto (a menos que explicitamente mencionado) vamos utilizar os dois termos indistintamente.

Em sistemas DSM, a "memória compartilhada" é implementada por mecanismos de hardware e/ou software que transformam, de modo transparente ao usuário, os acessos às memórias remotas em mensagens pela rede. Devido ao alto custo de um acesso remoto e à grande quantidade de acessos remotos gerados pelas aplicações, sistemas DSM mantém os dados compartilhados replicados nas memórias privativas dos diversos processadores. Essa replicação, entretanto, implica na necessidade de se manter a coerência das diversas cópias. A coerência é mantida através de um

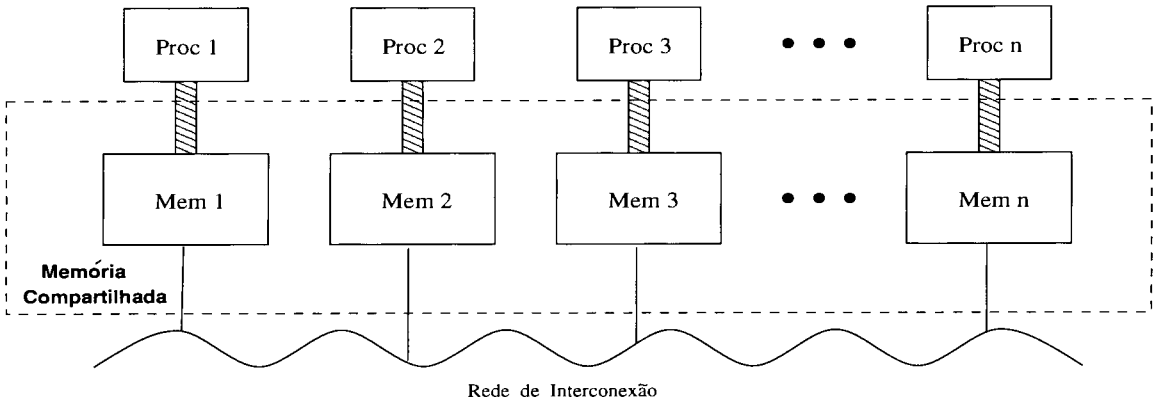

Figura 1.2: Sistema de memória compartilhada distribuída.

protocolo próprio, chamado protocolo de coerência.

Sistemas *hardware DSM* (como por exemplo, Alewife [5], Origin [52] e Exemplar [24]) provêem a abstração da memória compartilhada através de mecanismos de hardware altamente especializados. Esses mecanismos permitem grande eficiência, mas tornam o sistema complexo, o que implica em um tempo de projeto bastante grande. Por esse motivo, são sistemas extremamente caros e que, em muito casos, devido ao longo tempo de projeto, apresentam componentes tecnologicamente ultrapassados.

Sistemas *software DSM* são alternativa de baixo custo para o suporte à memória compartilhada. O grande atrativo desses sistemas está na possibilidade de aproveitar os benefícios de uma arquitetura simples e largamente difundida, como a de uma rede de estações de trabalho com um sistema operacional padrão. A memória compartilhada é implementada através de mecanismos de software, podendo envolver o sistema operacional, a biblioteca do sistema, ou suporte de compilação. Apesar do seu baixo custo, sistemas *software DSM* apresentam desempenho limitado, devido principalmente ao alto custo de comunicação da arquitetura. Trabalhos como [12, 15, 57, 38, 76, 37] mostram que, para uma grande quantidade de aplicações, os diversos *overheads* desses sistemas dominam o tempo de execução.

Nos últimos anos, muito se tem pesquisado para melhorar o desempenho de

sistemas *software DSM* (e.g., [47, 45, 57, 11, 15, 63]). Grande parte dessas pesquisas se concentra na tentativa de atrasar e/ou restringir ao máximo a comunicação gerada pelo protocolo utilizado para manter as memórias coerentes.

Independente da técnica usada para atrasar ou restringir as trocas de mensagens, a propagação das modificações realizadas no dado compartilhado tem que ser realizada para os processos que possuem cópia do mesmo. Não há um consenso, entretanto, na melhor forma de realizar essa propagação. Alguns sistemas propagam as modificações nos dados compartilhados de forma otimista, antes que eles sejam efetivamente acessados. Outros sistemas preferem esperar que os dados compartilhados sejam realmente acessados, para então requisitar suas modificações. Ambas as formas têm suas desvantagens. Na primeira, ocorrem várias propagações desnecessárias, porque elas devem ser enviadas para todos os processadores que possuem cópia do dado. Na segunda, o *overhead* da propagação dos dados é exposto, porque o processador deve esperar que as modificações sejam recebidas quando deseja acessar os dados.

Esses problemas poderiam ser aliviados se, ao modificar um dado compartilhado, o processador soubesse de antemão qual o próximo processador a acessá-lo. Essa informação, porém, não está disponível para o sistema no paradigma de memória compartilhada. A sincronização pode dar uma boa indicação, principalmente quando ela é realizada através do uso de seções críticas delimitadas por operações de *lock* e *unlock*. Ou melhor, se pudermos prever qual é o próximo processador a adquirir uma seção crítica, podemos enviar-lhe antecipadamente as modificações realizadas no dado, eliminando assim, o tempo de espera pelo dado sem o custo adicional de sobrecarregar a rede com grande quantidade de mensagens.

Embora a ocorrência dos eventos de sincronização seja totalmente dependente do comportamento dinâmico da aplicação, apresentamos nessa tese uma técnica chamada LAP (*Lock Acquirer Prediction*) que prevê dinamicamente a ordem de

transferência de *locks* e permite o envio seletivo de atualizações. LAP utiliza um histórico das transferências anteriores e consegue determinar, com alta precisão, os processadores que serão os prováveis próximos donos de uma seção crítica. Como LAP está associada ao uso de *locks*, a extensão com que essas operações são utilizadas numa aplicação tem impacto direto na utilidade da técnica LAP e consequentemente na concepção do sistema *DSM* e no seu desempenho final. O uso de operações de *lock* é regido pelo modelo de programação, o qual especifica as características de paralelismo, comunicação, distribuição dos dados e tarefas paralelas, sincronização e compartilhamento nas aplicações.

Nosso objetivo nessa tese é estudar as vantagens oferecidas por LAP e as implicações dos aspectos de sincronização e compartilhamento do modelo de programação no desempenho e na complexidade de sistemas *software DSM*. Para tal, desenvolvemos três sistemas *software DSM* diferentes: AEC, AEC-light e AEC-bind. Os três sistemas exploram LAP de forma diferente e, dentro do paradigma de memória compartilhada, utilizam modelos de programação distintos. Tais modelos, embora mais complexos que o modelo de programação tradicional no paradigma de memória compartilhada, são ainda muito mais simples do que os modelos de programação no paradigma de passagem de mensagens.

Mais detalhadamente, os diferentes modelos que estudamos utilizam sincronização por *locks* de forma distinta: o modelo de AEC envolve *locks* apenas para definir seções críticas (conforme o uso tradicional desse tipo de primitiva); o modelo de AEC-light requer que todos os acessos a dados compartilhados sejam feitos sob *locks* e, desta forma, envolve tanto os *locks* tradicionais quanto *locks* que não forçam a exclusão mútua; o modelo de AEC-bind é semelhante ao de AEC-light, mas ainda requer a associação explícita dos dados compartilhados a variáveis de sincronização. A técnica LAP foi inicialmente concebida para *locks* tradicionais, mas também pode se beneficiar dos outros tipos de *lock*, permitindo assim uma redução mais efetiva

do *overhead* de acessos a dados compartilhados. Os modelos de AEC-light e AEC-bind simplificam os protocolos uma vez que estes não precisam incluir código para o tratamento de acessos realizados fora de seções críticas. O modelo de AEC-bind simplifica ainda mais o protocolo, uma vez que não é necessária a determinação dinâmica das associações de dados a variáveis de sincronização.

Essa tese mostra os resultados da implementação de LAP no contexto dos nossos três sistemas em um multicomputador IBM SP-2 com oito nós de processamento. O desempenho dos sistemas é comparado com o do sistema TreadMarks [45], o qual é o *software DSM* mais difundido hoje em dia. Nossos resultados mostram que a técnica LAP pode alcançar altas taxas de acerto nas suas previsões. Essas altas taxas permitem uma grande redução no *overhead* de busca de dados compartilhados acessados dentro de uma seção crítica e têm efeito direto no tempo de espera pela seção crítica e no *overhead* de barreiras. Tais reduções levam os sistemas AEC, AEC-light e AEC-bind a obterem ganhos de até 200% no *speedup* das aplicações em relação a TreadMarks.

Nossos resultados mostram ainda que o modelo de programação também tem grande efeito no desempenho dos sistemas, mas esse efeito é diferente do esperado intuitivamente. O aumento da complexidade do modelo não necessariamente leva a ganhos de desempenho. AEC-light nem sempre alcança melhor desempenho que AEC, enquanto que AEC-bind sempre alcança desempenho bastante superior a AEC-light.

Baseados nesses resultados, concluímos que a técnica LAP merece a atenção dos projetistas de sistemas *software DSM*, uma vez que pode alcançar bom desempenho sem a necessidade de hardware específico ou suporte de compiladores. Em relação ao modelo de programação, concluímos que a melhor escolha de modelo depende do tempo disponível para o desenvolvimento do sistema *software DSM* e da experiência dos programadores das aplicações. Sistemas mais difíceis de programar são mais

simples e, dependendo do modelo empregado, têm melhor desempenho final. Na verdade, acreditamos que, na maioria dos casos, a complexidade de programação adicional imposta pelo modelo só é justificada pelos correspondentes ganhos de desempenho no caso de AEC-bind.

1.1 Contribuições da Tese

Sumariamente, as principais contribuições dessa tese são:

- Proposta e avaliação da técnica LAP para previsão dinâmica da ordem de transferência de *locks*;
- Proposta e avaliação de três sistemas *software DSM* diferentes: AEC, AEC-light e AEC-bind; e
- Avaliação do impacto do modelo de programação na complexidade e no desempenho dos sistemas.

1.2 Organização da Tese

O restante da tese é organizado da seguinte forma. Nos Capítulos 2 e 3 apresentamos os tópicos básicos na área de sistemas de memória compartilhada. No Capítulo 2 discutimos como os acessos aos dados compartilhados são observados pelos processadores do sistema e no Capítulo 3 apresentamos questões específicas de sistemas *software DSM*, como o tipo de protocolo e a unidade de coerência empregados. No Capítulo 4 apresentamos a técnica LAP para previsão dinâmica da ordem de transferência de *locks*. Em seguida, nos Capítulos 5, 6 e 7 apresentamos os três sistemas *software DSM* desenvolvidos: AEC, AEC-light e AEC-bind. Em cada um desses capítulos são apresentadas as características mais importantes do sistema, juntamente com o modelo de programação empregado. O Capítulo 8 descreve o ambiente e o conjunto de aplicações utilizados em nossos experimentos. O Capítulo 9 mostra

uma avaliação das previsões realizadas por LAP e uma avaliação do desempenho dos três sistemas propostos. O Capítulo 10 apresenta os trabalhos relacionados a esta tese. Finalmente, no Capítulo 11 apresentamos nossas conclusões e trabalhos futuros.

Capítulo 2

Modelos de Consistência

Para escrever um programa correto e eficiente no modelo de memória compartilhada, o programador deve ter de uma noção precisa de como o sistema de memória se comporta no que diz respeito às operações de leitura e escrita nos dados compartilhados, realizadas por diferentes processadores. Considere, por exemplo, os segmentos de código da figura 2.1. Nessa figura apresentamos uma implementação de um algoritmo de Dekker para exclusão mútua de seções críticas, onde dois processos P_0 e P_1 executam em processadores diferentes. Se as escritas sempre são imediatamente vistas pelos outros processadores, então é impossível que os testes dos comandos $L0$ e $L1$ sejam verdadeiros ao mesmo tempo. Entretanto, se o efeito da escrita for atrasado e durante esse atraso, o processador puder continuar o processamento normal, então ambos os processadores podem chegar no comando `if` antes de terem visto as escrita às variáveis A e B e, portanto, os testes de $L0$ e $L1$ podem ser verdadeiros ao mesmo tempo.

P_0	P_1
A = 0;	B = 0;
⋮	⋮
A = 1;	B = 1;
L0: if (B == 0) ...	L1: if (A == 0) ...

Figura 2.1: Segmentos de código dos processos P_0 e P_1 que compartilham A e B.

Um modelo de consistência de memória provê uma especificação formal da ordem em que as escritas à memória compartilhada realizadas por um processador são observados pelos outros processadores do sistema. Em outras palavras, um modelo de consistência responde a seguinte pergunta: **quando** um processador deve "ver" um valor que foi modificado por outro processador?

Nesse Capítulo apresentamos uma descrição de alguns modelos de consistência de memória, partindo do modelo mais restritivo para o mais relaxado. Nossa preocupação aqui não é apresentar os formalismos da definição de cada modelo, mas sim dar uma noção intuitiva do funcionamento do modelo num sistema DSM e suas implicações no modelo de programação. Definições formais podem ser encontradas em [3, 2, 4, 31].

2.1 Modelo de Consistência Seqüencial

O modelo mais simples e intuitivo de consistência é o chamado consistência seqüencial, ou SC (*Sequential Consistency*). Ele foi formalizado por Lamport [51]: "Um sistema é seqüencialmente consistente se o resultado de qualquer execução é o mesmo que se as operações de todos os processadores fossem executadas em alguma ordem seqüencial, e as operações de cada processador na ordem descrita pelo programa". Em outras palavras, em termos de ordenação dos acessos a dados compartilhados, uma máquina paralela seqüencialmente consistente funciona como se fosse um único processador multiprogramado.

No exemplo da figura 2.1, o uso do modelo SC garante que os dois comandos $L0$ e $L1$ jamais poderiam retornar verdadeiro ao mesmo tempo. Em todas as possíveis intercalações dos comandos de P_0 e P_1, as leituras dos valores de A e B completam uma antes da outra, assim a segunda a completar já viu necessariamente a escrita de 1 na variável.

A forma mais simples de se implementar o modelo SC é garantir que um acesso a

um dado compartilhado só pode ser realizado se o acesso anterior tiver sido observado por todos os outros processadores, i.e., cada acesso realizado no dado compartilhado deve estar imediatamente e globalmente coerente. Apesar da consistência seqüencial apresentar um modelo de memória simples, ela é bastante restritiva, reduzindo o paralelismo potencial do sistema. Além disso, este modelo apresenta alguns obstáculos à implantação de mecanismos de hardware usados para fins de otimização, como por exemplo, *pipelines* de escrita, *write-buffers* ou *bypasses* de acessos à memória [2].

2.2 Modelos de Consistência Relaxados

Garantir a consistência da memória empregando um modelo menos restritivo pode, portanto, ter grande influência no desempenho do sistema. Uma série de modelos de consistência de memória foram propostos para relaxar as condições impostas pelo modelo SC e viabilizar o uso de algumas otimizações no acesso à memória compartilhada. A seguir, vamos apresentar os modelos relaxados mais utilizados, sendo que vamos descrever em mais detalhes apenas os modelos *Release Consistency* e *Entry Consistency* que são os mais utilizados em sistemas *software DSM*.

O modelo *Processor Consistency* (PC) foi proposto por Goodman [32]. PC permite que acessos de leitura ultrapassem acessos de escrita (desde que para posições de memória diferentes) e permite que escritas realizadas por diferentes processadores sejam observadas em dois processadores p_i e p_j em ordens diferentes (desde que as escritas de um mesmo processador sejam sempre observadas na ordem em que ele as executou).

É possível relaxar ainda mais as condições de consistência, se considerarmos que a grande maioria das aplicações paralelas já têm definidas em alto nível suas necessidades de coerência. Isto é, o programador deixa explícito no programa, através de sincronização, os pontos em que a coerência dos dados é necessária. Por exemplo, considere os segmentos de código de P_0 e P_1 da figura 2.2. Estamos utilizando sincro-

nização de uma forma bem genérica através de acessos de leitura e escrita à variável de sincronização s. A sincronização nesse exemplo é usada pelo programador para garantir que os acessos de leitura a A e a B em P_1 retornam necessariamente os valores escritos por P_0.

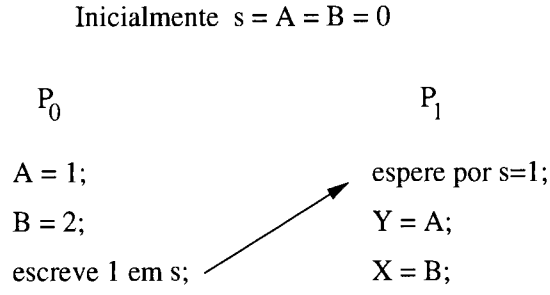

Figura 2.2: Segmentos de código dos processos P_0 e P_1 que utilizam sincronização para garantir a coerência de A e B.

Aplicações paralelas utilizam acessos a variáveis de sincronização para evitar efeitos de condições de corrida e conseqüentemente resultados não-determinísticos. O primeiro modelo relaxado de consistência de memória proposto para tirar proveito da sincronização foi o modelo *Weak Consistency* (WC) [4]. O modelo WC estabelece que acessos a dados compartilhados realizados entre dois acessos de sincronização podem ser observados por outros processadores em qualquer ordem. Isto é, acessos de sincronização funcionam como cercas (*fences*). Quando um acesso de sincronização é realizado, os acessos anteriores devem ter sido observados por todos os processadores.

2.2.1 O Modelo *Release Consistency*

O modelo *Release Consistency* (RC) [31] classifica os acessos a variáveis de sincronização em *acquires* e *releases*. Um *acquire* indica que o processador está iniciando uma operação que pode depender de valores gerados por outro processador. A execução de um *release* indica que o processador está terminando uma operação que gerou valores dos quais outro processador pode depender. Por exemplo, na figura 2.2 o

comando "`escreve 1 em s`" é classificado como `release(s)` e o comando "`espere por s=1`" é classificado como `acquire(s)`.

Para garantir a coerência dos dados compartilhados segundo o modelo RC, a aplicação deve, portanto, utilizar sincronização explícita através de primitivas do sistema. O uso de espera ocupada em um *flag*, por exemplo, não permite que o sistema detecte a existência de operações de sincronização e as classifique como *acquires* ou *releases*. As primitivas mais comuns fornecidas pelos sistemas DSM para sincronização são: *locks* e *unlocks* para delimitar seções críticas; e `barreiras` para sincronização global. Operações de *lock* são classificadas como *acquires* e operações de *unlock* são classificadas como *releases*. Uma operação de barreira é classificada como um *release* seguido de um *acquire*.

Informalmente, as condições para que um sistema esteja coerente segundo o modelo RC são:

- Antes que uma operação *release* tenha sido observada por qualquer processador, todos os acessos a dados compartilhados anteriores devem ter sido observados por esse processador.

- Acessos que seguem uma operação de *acquire* numa variável de sincronização s devem esperar que o *acquire* tenha terminado.

- Acessos a variáveis de sincronização devem ser observados segundo os modelos SC ou PC.

Vários sistemas utilizam o modelo RC, entre eles DASH [53], Munin [22], Quarks [73] e Shasta [63].

Em relação ao modelo SC, RC apresenta uma sensível redução na latência de acesso à memória compartilhada: um processador fica paralisado esperando pela coerência de dados somente em operações de *release*. Resta saber se RC oferece

também um modelo de memória tão simples e intuitivo como o oferecido por SC. O trabalho de Gharachorloo *et al* [31] apresenta uma prova formal de que a execução no modelo RC fornece os mesmos resultados de ordenação dos acessos que a execução no modelo SC, se a aplicação for "propriamente-sincronizada". A definição de um programa propriamente sincronizado é a seguinte:

- Sejam u e v dois acessos a um mesmo dado compartilhado, u é realizado no processador p_u e v em p_v. Para que em qualquer execução do programa, v seja visto pelos outros processadores antes de u, deve haver em p_u uma leitura de uma variável de sincronização e em p_v uma escrita na mesma variável, tal que p_u leia o valor escrito em p_v. Um programa é propriamente sincronizado se essa condição vale para todos os possíveis pares de u e v.

De uma forma simplificada, uma aplicação é propriamente-sincronizada se contém sincronização suficiente para evitar condições de corrida.

O trabalho de Keleher *et al* [44] apresenta uma versão "preguiçosa" do modelo *Release* chamada de *Lazy Release Consistency* (LRC). LRC relaxa as condições de RC, porque não requer que numa operação *release* os acessos anteriores estejam globalmente visíveis. LRC requer que os acessos anteriores ao *release* estejam visíveis somente no processador que vai executar o *acquire* subsequente. Em outras palavras, quando p executa um *acquire* em s, antes que o *acquire* tenha terminado, todos os acessos realizados até o último *release* em s devem estar visíveis em p.

Para determinar sobre quais modificações um processador deve estar ciente no momento de um *acquire*, LRC estabelece uma ordem parcial, chamada *happened-before-1* (*hb*1), dos acessos a dados compartilhados. A ordem parcial *hb*1 [3] é baseada na ordem sequencial de execução em um processador e no encadeamento das operações *acquire* e *release* realizadas em processadores diferentes, mas sob a mesma variável de sincronização. Dois acessos à memória compartilhada a_1 e a_2 são

ordenados por $hb1$, denotado por $a_1 \overset{hb1}{\rightarrow} a_2$, se:

- a_1 e a_2 são acessos do mesmo processador e a_1 ocorre antes de a_2;
- a_1 é um *release* no processador P_1, a_2 é um *acquire* na mesma variável de sincronização em P_2 e a_2 retorna o valor escrito por a_1.

A ordem $hb1$ é definida basicamente pelo fecho transitivo (se $a_1 \overset{hb1}{\rightarrow} a_2$ e $a_2 \overset{hb1}{\rightarrow} a_3$, então $a_1 \overset{hb1}{\rightarrow} a_3$) da ordem de execução de um processador com a ordem das operações de sincronização (um *acquire* é ordenado depois do último *release* anterior na mesma variável de sincronização). Por exemplo, considere os segmentos de código da figura 2.3. Quando P_2 executa o *acquire* na variável de sincronização c, segundo a ordem parcial $hb1$ (ilustrada pelas setas da figura), ele deve receber as modificações realizadas em x_1, x_2 e x_3 de P_0 e as modificações realizadas em x_4, x_5 e x_3 de P_1.

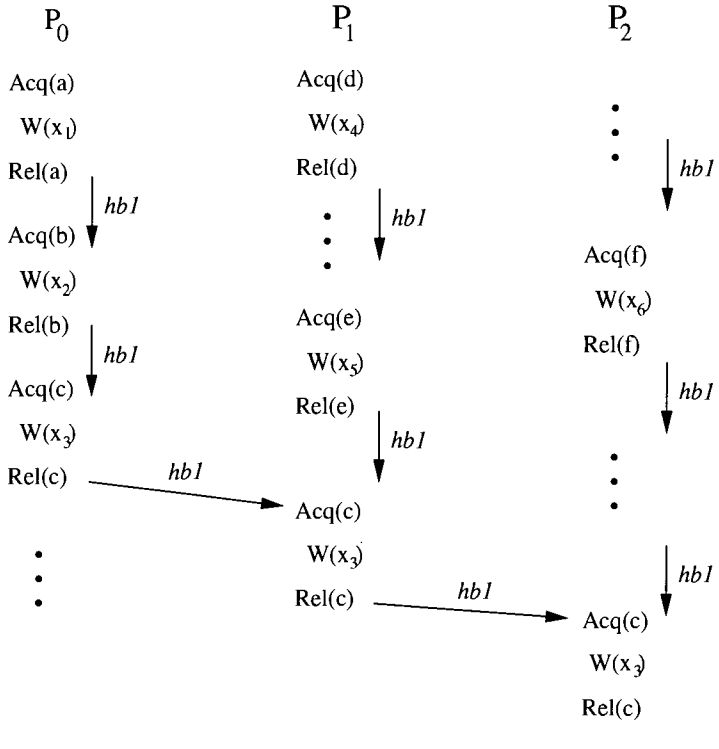

Figura 2.3: Ordem parcial estabelecida pelas operações de sincronização em LRC.

LRC mantém, tal qual RC, o comportamento do modelo seqüencial se o programa for propriamente-sincronizado. Os sistemas TreadMarks [45], HLRC [76], AURC [36] e ADSM [57] são baseados em LRC.

2.2.2 O Modelo *Entry Consistency*

O modelo *Entry Consistency* (EC) foi desenvolvido por Bershad *et al* [10]. Esse modelo relaxa ainda mais as regras que determinam quando um processador deve observar as modificações realizadas nos dados compartilhados. Ele explora a relação entre variável de sincronização e o dado compartilhado protegido por ela. EC garante que o dado compartilhado está coerente num processador, apenas quando este adquire a variável de sincronização que o protege.

Sendo s uma variável de sincronização que protege o dado D_s, informalmente, a condição para que um sistema esteja consistente segundo o modelo EC é:

- Antes que uma operação *acquire* em s tenha terminado em p, todas as atualizações em D_s devem ter sido observadas por p.

Com relação à forma como o dado compartilhado deve ser associado à variável de sincronização, é possível fazê-la de forma explícita ou de forma implícita. A associação explícita é realizada pelo programador (ou pelo compilador se for possível). Através de uma operação especial, o programador indica para cada variável de sincronização quais os dados compartilhados protegidos por ela. A associação implícita deixa a cargo do sistema DSM estabelecer dinamicamente essa relação.

O modelo EC foi proposto em [10] com associação explícita de dado compartilhado com variável de sincronização. O trabalho de Iftode *et al* [39], entretanto, propõe um sistema, chamado *Scope Consistency* (ScC), que estabelece as mesmas condições de consistência do modelo EC, mas realiza a associação de dado compartilhado com variável de sincronização dinamicamente, de forma implícita. Estamos considerando nessa tese que o modelo proposto por Iftode *et al* é simplesmente uma implementação diferenciada do modelo EC, mas não um novo modelo de consistência, conforme descrito no artigo. Assim, utilizaremos daqui para frente o termo EC com e sem associação explícita, para indicar as duas formas de associação.

Como exemplo de sistemas que utilizam EC temos: Midway [11], Brazos [71] e o próprio sistema ScC.

Entry Consistency x Lazy Release Consistency

No modelo de consistência EC o ponto de espera pela coerência ocorre, tal qual em LRC, nas operações de *acquire*. A grande diferença entre os dois modelos está em determinar sobre **quais** modificações o processador deve ser notificado. Em LRC, ele deve ser notificado sobre todas as modificações anteriores segundo a ordem parcial $hb1$. Já o modelo EC se baseia na idéia de que se a aplicação requer sincronização para evitar efeitos de condições de corrida em seus resultados, então essa sincronização delimita não só quando os dados devem estar coerentes, mas também quais dados devem estar coerentes naquele momento. Em outras palavras, se o programador (ou o compilador) estabeleceu uma determinada seção crítica s para acessar o dado D_s, então no momento do *acquire* em s, o processador deve estar ciente apenas das modificações em D_s.

Vamos tomar a figura 2.3 como exemplo novamente. No modelo EC, P_2 no momento do *acquire* em c só deve receber informações sobre as modificações de x_3 realizadas anteriormente. Nesse caso, a determinação do que foi realizado anteriormente segue a ordem total imposta pelo encadeamento de operações *acquire/release* em c.

Modelo de Programação

Embora seja o modelo mais relaxado e, portanto, o que envolve menor comunicação, o grande alvo de críticas a EC está na necessidade da associação entre dado compartilhado e variável de sincronização. Essa associação implica no uso de um modelo de programação diferenciado, próprio para o modelo EC.

Para manter o mesmo comportamento do modelo seqüencial, o modelo de programação utilizado por EC requer não só que a aplicação seja propriamente-

sincronizada, mas também que os acessos aos dados compartilhados se deêm conforme a associação estabelecida. Assim, o programador deve considerar que um dado compartilhado só está visível num processador quando este executa uma operação *acquire* na variável de sincronização que o protege. Por exemplo, considere os segmentos de código de P_0 e P_1 da figura 2.4. A escrita em x por P_0 está separada da leitura de x em P_1 pelas operações de sincronização $rel(a)$ e $acq(a)$ executadas em P_0 e P_1 respectivamente. Se estes códigos executam em um sistema que emprega os modelos RC ou LRC, a leitura de x por P_1 retornará sempre o valor 2. Num sistema que emprega EC, entretanto, a leitura de x por P_1 pode ou não retornar 2. Isso porque os acessos a x não estão protegidos por nenhuma variável de sincronização específica. Para que P_0 e P_1 executem corretamente no modelo EC, deve ser inserido em P_0 uma operação de *release* numa variável de sincronização e em P_1 uma operação de *acquire* na mesma variável.

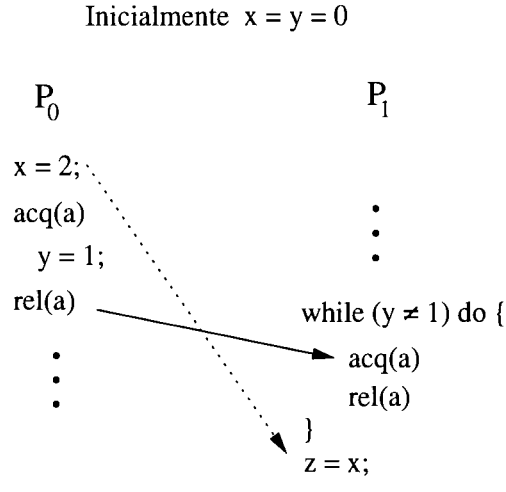

Figura 2.4: Exemplo de código que executa deterministicamente em LRC, mas não em EC.

A grande maioria das aplicações, felizmente, utiliza sincronização específica para proteger os dados compartilhados e não requer modificações para executar corretamente no modelo EC.

Capítulo 3

Sistemas *Software DSM*

Nesse capítulo abordamos alguns conceitos básicos sobre sistemas *software DSM* que serão utilizados ao longo dessa tese. Discutimos a respeito do tipo de protocolo, da unidade de coerência e do suporte para a existência de múltiplos escritores. No fim do capítulo apresentamos o sistema TreadMarks. TreadMarks é o sistema *software DSM* mais utilizado nos dias de hoje e, portanto, é o sistema base utilizado em nossas comparações.

3.1 Protocolo de Atualização x Invalidação

O protocolo de coerência utilizado em um sistema *DSM* define como um processador deve tomar conhecimento das modificações realizadas nos dados compartilhados por outro processador. Existem duas formas diferentes de se notificar um processador sobre modificações realizadas no dado compartilhado: por protocolo de invalidação ou por protocolo de atualização.

Num protocolo de invalidação, um processador é informado a respeito das modificações em um dado compartilhado através de mensagem de invalidação. Ao receber a mensagem o sistema invalida a unidade de coerência que contém o dado. A transferência das modificações realizadas no dado se dá efetivamente no próximo acesso realizado ao dado. Num protocolo de atualização um processador informa a outro processador a respeito de modificações no dado compartilhado enviando-lhe o

próprio dado modificado. O uso de protocolo de atualização evita a ocorrência de falhas de acesso, mas aumenta bastante a quantidade de tráfego desnecessário na rede (muitas vezes o processador recebe modificações que não vai utilizar [13]), em relação ao protocolo de invalidações.

3.2 Unidade de Coerência

Sistemas *software DSM* normalmente utilizam uma página ou um objeto como unidade de coerência. No entanto, alguns sistemas *software DSM*, como Shasta [63] e Blizzard-S [64], permitem o emprego de uma granularidade mais fina como unidade de coerência, já que assumem o uso de sistemas de comunicação de altíssimo desempenho.

O uso da página como unidade de coerência é suportado pelo hardware de memória virtual, presente em qualquer arquitetura convencional. Uma página protegida contra leitura está com conteúdo inválido e o primeiro acesso a essa página causa uma `falha de acesso`. O tratamento dessa falha é realizado pelo sistema *software DSM*, que se encarrega de produzir uma versão atualizada da página. A detecção de escritas realizadas em uma página é feita de forma simples através do mecanismo de proteção de páginas. As páginas são inicialmente protegidas contra escrita. Na primeira falha de escrita, o sistema identifica que a página será alterada, guarda essa informação e desprotege a página para escritas posteriores.

O uso de objeto como unidade de coerência é suportado diretamente pelo programador ou pelo compilador. Para propagar as modificações realizadas no dado compartilhado, o protocolo de coerência tem que ser capaz de detectar quando um objeto é modificado. A detecção de escritas no objeto pode ser realizada de diversas formas. Alguns sistemas (e.g.,[11]) requerem suporte do compilador para inserir instruções extras a cada acesso de escrita ao objeto. Essas instruções servem para ativar *flags* de modificação. Esse tipo de instrumentação apresenta *overhead* signi-

ficativo, podendo duplicar a quantidade de escritas da aplicação. Outros sistemas (e.g.,[64],[63]) se baseiam em alterações no código objeto da aplicação, permitindo algumas verificações nos acessos à memória compartilhada, para invocar ou não (baseado no resultado da verificação) ações do protocolo de coerência. Uma terceira alternativa para detecção de escritas no objeto é utilizar o mecanismo de proteção das páginas onde o objeto está alocado.

3.3 Suporte a Múltiplos Escritores

Devido ao tamanho considerável das unidades de coerência empregadas em certos sistemas *software DSM*, os efeitos do falso compartilhamento podem levar a sérios problemas de desempenho. Considerando a página como unidade de coerência, o falso compartilhamento ocorre quando processadores diferentes escrevem em dados logicamente não relacionados, mas que estão localizados numa mesma página. Se o acesso de escrita à página for exclusivo, então pode ocorrer o efeito "ping-pong" de uma página quando processadores requisitam alternadamente o acesso exclusivo a ela.

Para evitar o efeito ping-pong de uma página, alguns sistemas *software DSM* permitem que vários processadores estejam escrevendo ao mesmo tempo na mesma página, desde que em dados diferentes. O principal problema ao se permitir a existência de múltiplos escritores na mesma página está em combinar todas as escritas realizadas na página para formar uma versão coerente da mesma.

O método mais utilizado para essa "combinação" é a criação de *diffs*. Antes de começar a atualizar uma página o sistema cria uma cópia gêmea da mesma, chamada *twin*. As escritas são realizadas livremente na página e o *twin* mantém a versão original. Quando as escritas locais têm que ser propagadas, o processador compara a cópia corrente da página com o seu *twin*, gerando, em uma estrutura chamada *diff*, as diferenças entre eles. A combinação dos *diffs* gerados por todos os processadores

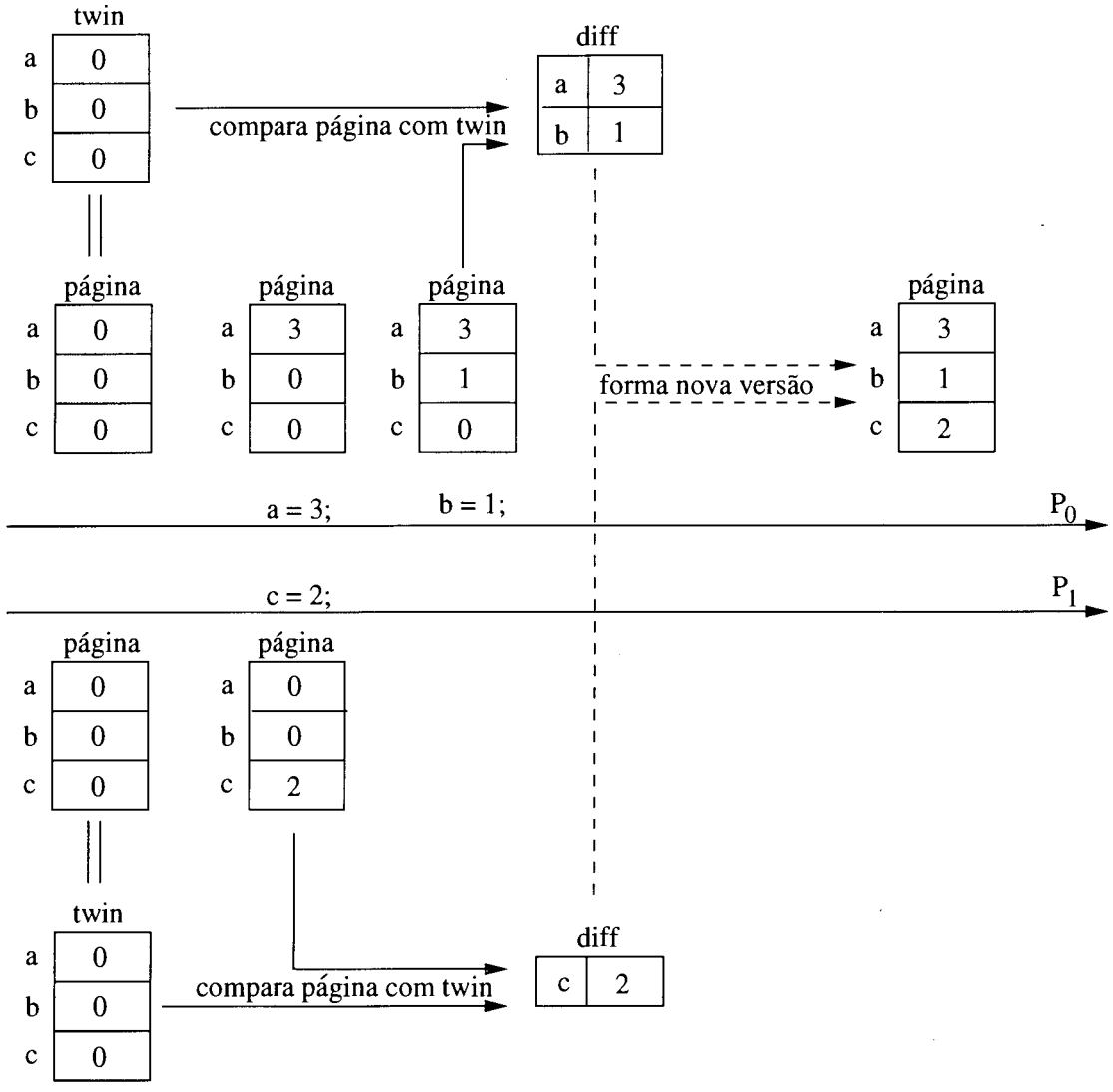

Figura 3.1: Suporte a múltiplos escritores com esquema de criação de *diffs*.

que atualizaram a página permite a formação de uma versão atualizada da mesma.

A figura 3.1 mostra um exemplo em que dois processadores, P_0 e P_1, escrevem na mesma página ao mesmo tempo, mas em dados diferentes. P_0 escreve em a e b e P_1 escreve em c. Antes das escritas, os processadores criam *twins* para as páginas correspondentes e em seguida escrevem livremente na página. Quando uma versão atualizada da página tem que ser determinada, cada processador cria os *diffs* correspondentes às suas escritas e a combinação dos dois *diffs* criados forma a nova versão da página.

3.4 Sistema TreadMarks

TreadMarks [45] é um sistema *software DSM* que emprega a página como unidade de coerência, utiliza protocolo de invalidação para manutenção da coerência dos dados compartilhados e provê suporte a múltiplos escritores através de mecanismo de *diffs*. O modelo de consistência empregado é o LRC e o sistema divide a execução do programa em intervalos para determinar quais modificações no dado compartilhado um processador deve observar no momento da execução de um *acquire*.

Intervalos são segmentos de tempo na execução de um processador. Um novo intervalo é iniciado cada vez que uma operação de sincronização é executada. TreadMarks utiliza a ordem parcial $hb1$ para ordenar os intervalos de diferentes processadores. Através de $hb1$ é possível determinar quais intervalos de outros processadores precedem o intervalo corrente de um processador p. As modificações realizadas nos dados compartilhados são associadas aos intervalos em que elas ocorreram e, assim, numa operação de *acquire* ocorrida no intervalo i, o processador deve ser notificado sobre todas as modificações associadas a intervalos "anteriores" a i segundo a ordem parcial $hb1$.

A notificação sobre as modificações ocorridas nos dados compartilhados é realizada através de *write-notices* (avisos de escrita). Um *write-notice* indica que uma determinada página foi modificada. Cada intervalo contém um *write-notice* para cada página modificada no segmento de tempo correspondente. Quando o processador p executa uma operação de *acquire* num intervalo i ele deve receber os *write-notices* correspondentes a todos os intervalos anteriores a i segundo $hb1$. Ao receber um *write-notice*, o processador invalida a página correspondente. Os *diffs* relativos às modificações em questão só são recebidos na próxima falha de acesso a cada página.

3.4.1 Falhas de Acesso

Inicialmente todas as páginas do sistema estão válidas no processador 0 e inválidas nos outros processadores. Na primeira falha de acesso a uma página k o processador p ($p \neq 0$) deve requisitar uma cópia de k ao processador 0.

Uma falha de acesso a uma página válida é tratada de modo bem simples. Se a falha ocorreu por causa de um acesso de escrita a uma página que ainda não possui *twin*, TreadMarks deve simplesmente criar o *twin* correspondente e desproteger a página contra escrita.

Se a falha ocorreu em uma página k inválida, então o sistema tem que trazer os *diffs* relativos às modificações realizadas em k por outros processadores. Os *diffs* devem ser buscados dos processadores que enviaram *write-notices* para k. Uma característica muito importante de TreadMarks é a criação de *diffs* sob demanda. Em TreadMarks, um processador só cria o *diff* relativo às modificações realizadas em uma determinada página quando chega uma requisição para esse *diff*. Para minimizar o número de mensagens necessárias para a busca de *diffs*, TreadMarks utiliza esquema de "dominância" de intervalos. Um intervalo i domina um intervalo j, se j precede i na ordem parcial $hb1$. Assim as mensagens devem ser enviadas apenas aos processadores cujos intervalos mais recentes não são dominados pelos intervalos mais recentes de outros processadores. Considere, por exemplo, os segmentos de código da figura 3.2. Quando P_2 executa `lock(c)`, ele deve requisitar *diffs* apenas do processador P_1, porque o intervalo I_2 de P_1 domina os intervalos I_0 e I_1 de P_0 e I_0 de P_1.

3.4.2 Operações de Sincronização

A interface de programação oferecida por TreadMarks provê dois tipos de primitivas de sincronização: *lock/unlock* e barreiras. As primitivas *lock/unlock* são utilizadas para delimitar seções críticas e a primitiva barreira é utilizada para sincronização

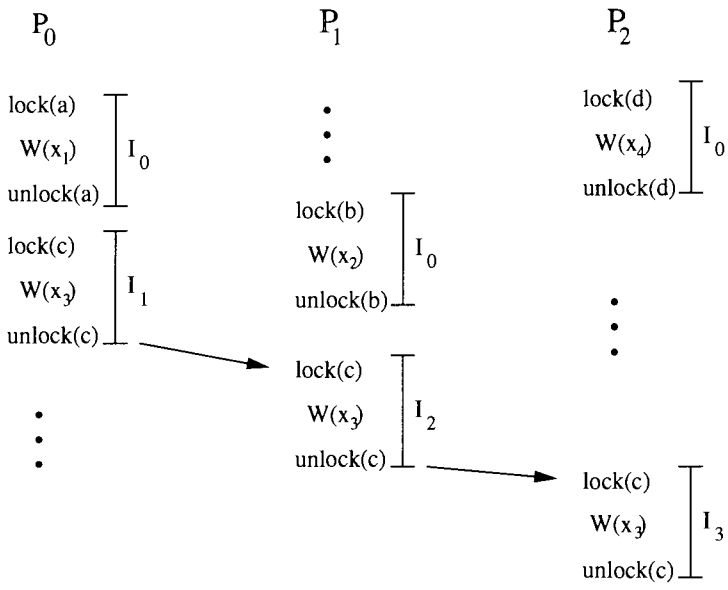

Figura 3.2: Exemplo de dominância de intervalos.

global.

Há um processador gerente, determinado estaticamente, para cada variável de sincronização. Numa operação *lock* em uma variável de sincronização s, o processador p envia uma mensagem ao processador gerente de s (p não conhece a identificação do último processador a acessar s). O processador gerente, então, avança a mensagem para o último *releaser* de s. Este compara seus intervalos com os de p e envia para p os *write-notices* relativos aos intervalos que p ainda não viu. Na chegada da mensagem com os *write-notices*, p invalida as páginas correspondentes.

Numa operação de barreira, cada processador envia para o processador gerente da barreira uma mensagem contendo todos os seus intervalos. O processador gerente "incorpora"todos os intervalos recebidos. Depois que todos os processadores chegaram à barreira, o gerente envia uma mensagem de volta para cada processador p contendo os *write-notices* relativos aos intervalos dos outros processadores que p ainda não viu. Mais uma vez, na chegada da mensagem com os *write-notices*, o processador invalida as páginas correspondentes.

Capítulo 4

A Técnica *Lock Acquirer Prediction*

Tendo apresentado os conceitos básicos de modelos de consistência de memória e de sistemas *software DSM*, apresentamos neste capítulo a técnica *Lock Acquirer Prediction* (LAP) para sistemas *software DSM*. LAP realiza a previsão dinâmica do próximo dono de uma seção crítica e permite o envio de atualizações seletivas para o próximo dono da seção. Primeiramente mostramos as técnicas empregadas na previsão. Em seguida, apresentamos as implicações do uso de LAP em sistemas *software DSM*.

4.1 Técnicas de Previsão

Desenvolvemos duas técnicas de previsão diferentes de acordo com o modelo de programação empregado. A primeira delas considera um modelo de programação convencional para memória compartilhada: *locks/unlocks* protegem seções críticas de acesso exclusivo. A segunda considera um modelo de programação diferenciado: todos os acessos a dados compartilhados são realizados com *locks* e há diferentes tipos de *locks*, dependendo do tipo de acesso realizado no dado.

4.1.1 Modelo de Programação Convencional

Considerando um modelo de programação "convencional" em que a sincronização entre dois processadores se dá através de operações *lock/unlock* para delimitar seções

críticas de acesso exclusivo, desenvolvemos duas técnicas diferentes para previsão do próximo dono de uma determinada seção crítica. Na primeira delas, estamos considerando um sistema em que um processador que executa uma operação *lock* em uma seção crítica ocupada é bloqueado, numa fila FIFO, até que a seção crítica seja liberada por um outro processador. Quando essa fila existe, nossa técnica de previsão, chamada de `fila de espera`, determina que o próximo dono da seção crítica é exatamente o primeiro processador da fila.

Aplicações paralelas, entretanto, nem sempre apresentam fila de espera. Em conseqüência disso, desenvolvemos uma segunda técnica de previsão chamada de `afinidade local`. A técnica de afinidade local tenta descobrir o próximo dono de uma seção crítica baseada em um histórico das transferências de seções críticas anteriores. Através de um grande número de experimentos que discutiremos em capítulos posteriores, observamos que, durante a execução de aplicações paralelas, se um processador p executa um *lock* em s, o próximo dono de s está em um conjunto bastante limitado dos processadores do sistema, chamado de conjunto de afinidade de p. O conjunto de afinidade de p em geral varia muito pouco durante a execução das aplicações, o que significa que ele pode ser facilmente determinado em tempo de execução.

Para determinar o conjunto de afinidade de um processador em relação a uma variável de sincronização s, utilizamos uma matriz A para cada variável de sincronização, onde A_{ij} representa o número de vezes que o processador j foi o próximo dono de s após o processador i. Utilizamos uma heurística simples, baseada somente na frequência das transferências para analisar o passado, já que ela se mostrou bastante satisfatória em nossas avaliações.

A previsão do próximo dono de uma seção crítica s é baseada na determinação do `conjunto de atualização` do processador. O conjunto de atualização de um processador p em relação a s, $U_p(s)$, contém os processadores que são os mais prováveis

próximos donos de s depois de p e que, portanto, deverão receber as atualizações seletivas de p. O número máximo de elementos do conjunto de atualização de um processador, z, é determinado pelo usuário. O algoritmo abaixo mostra como esse conjunto é formado para um processador p em relação a uma seção crítica s.

1. Se há fila de espera: $U_p(s)$ = primeiro processador da fila de espera de s;Fim.

2. Senão, inserir em $U_p(s)$ os processadores j tal que $A_{pj} > 0$ e A_{pj} é maior que $T\%$ do total de operações de *lock* em s, em ordem decrescente de afinidade, até que $U_p(s)$ esteja completo; Fim.

O limite T é utilizado para que a técnica de afinidade não considere o envio de atualizações para processadores cuja afinidade com o processador que está liberando o *lock* seja muito pequena em relação ao total das passagens de *lock*.

4.1.2 Outros Modelos de Programação

As técnicas de previsão de LAP podem ser ainda estendidas se considerarmos o padrão de compartilhamento produtor-consumidor de determinadas aplicações: um processador p "produz" o dado compartilhado (escreve no dado) até uma sincronização global (ou barreira), depois que todos os processadores se sincronizam, um conjunto de processadores vai então "consumir" (acessar para leitura ou escrita) o dado produzido por p. Considerando um modelo de programação em que todos os acessos a dados compartilhados são protegidos por operações *lock/unlock*, podemos explorar o padrão de compartilhamento produtor-consumidor através de LAP. Nesse caso, o processador produtor pode utilizar, além da operação de *lock* tradicional, um tipo de *lock* especial, de escrita única, para escrever no dado.

Desenvolvemos a técnica de `afinidade global` para realizar previsões na transferência de *locks* convencionais ou de escrita única para um outro tipo de *lock* qualquer. Como essa transferência só ocorre depois de uma operação de barreira, a

técnica de afinidade global tenta prever, após a barreira, qual o próximo processador a requisitar um *lock* que foi acessado para escrita antes da barreira. Tal qual a técnica de afinidade local, a afinidade global realiza suas previsões através de um histórico das transferências desse tipo anteriores. Cada processador p mantém um vetor B para cada variável de sincronização s, onde B_i representa o número de vezes que o processador i requisitou um *lock* s após uma barreira e que foi liberado por p após um *lock* convencional ou um *lock* de escrita única em s antes da barreira. O `conjunto de afinidade global` de p é formado pelos processadores j, tal que B_j é maior que $L\%$ da quantidade de vezes que ocorreu passagem de *locks* tipo produtor-consumidor. Tal qual o limite T utilizado na técnica de afinidade local, o limite L impede que um processador p envie atualizações para processadores com afinidade muito baixa em relação ao total de passagens de *lock* de escrita após uma barreira.

A previsão do próximo dono de uma seção crítica s é bem simples. O `conjunto de atualização global` do processador p em relação à s, $UG_p(s)$, contém os processadores do conjunto de afinidade global de p. O número máximo de elementos do conjunto de atualização global, w, também é determinado pelo usuário. O algoritmo para formar $UG_p(s)$ é descrito em um único passo:

1. $UG_p(s) = w$ primeiros processadores (na ordem crescente de identificação dos processadores) do conjunto de afinidade global de p; Fim.

4.1.3 LAP e as Características das Aplicações

Conforme descrito nas subseções anteriores, as previsões realizadas por LAP dependem fundamentalmente do comportamento dinâmico das aplicações no que diz respeito aos seus padrões de sincronização. Entretanto, determinadas características de aplicações paralelas podem gerar um cenário favorável ou desfavorável ao emprego de LAP. Aplicações SPMD que apresentam um padrão de sincronização repetitivo

(como no caso de aplicações regulares iterativas) ou pré-estabelecido (como no caso de um *pipeline* de sincronização) são exemplos de aplicações em que LAP encontra um cenário favorável. Em contraste, aplicações MPMD (paralelismo de tarefas) ou aplicações SPMD completamente irregulares exemplificam o cenário desfavorável a LAP.

4.2 Implicações para Sistemas *Software DSM*

LAP é uma técnica geral que pode ser explorada tanto em sistemas *hardware DSM* [68], como em sistemas *software DSM* [66, 67, 68, 69]. Como esta tese enfoca sistemas *software DSM*, vamos analisar as implicações do uso de LAP na estrutura desses sistemas. A primeira implicação está relacionada com a implementação das técnicas de previsão, porque o sistema deve possuir infra-estrutura suficiente para manter as seguintes informações:

- uma fila FIFO contendo a identificação dos processadores que estão esperando pela seção crítica;
- a identificação do último processador a liberar a seção crítica;
- as matrizes A_{ij};
- os conjuntos de atualização de cada processador.

Num sistema *software DSM*, armazenar esse tipo de informação não é trivial, uma vez que não há uma estrutura centralizada representando cada *lock*. Nesses sistemas, a forma mais simples de manter essas informações é utilizar um processador gerente para cada *lock*. Todos os pedidos para as seções críticas protegidas por um certo *lock* devem ser direcionados ao respectivo gerente, que pode então facilmente implementar a fila de espera e realizar os cálculos para formar o conjunto de atualização de cada processador.

É interessante notar que nem todos os sistemas *software DSM* possuem gerentes de *locks*, como por exemplo o sistema Midway [11], que utiliza algoritmo de filas distribuído para localizar o último dono de uma seção crítica. Já os sistemas TreadMarks e ADSM [57] utilizam gerente, com a fila de espera mantida de forma distribuída. Entretanto, a comunicação envolvida em operações de *lock* nesses sistemas não permite que um processador conheça o seu conjunto de atualização ao entrar numa seção crítica, dado que o pedido de *lock* é avançado diretamente para o último dono da seção e o processador não recebe mensagem de volta do gerente.

A segunda implicação está diretamente relacionada com a implementação das atualizações seletivas. Nem todos os sistemas DSM envolvem protocolo de atualização de dados. Na verdade, a maior parte dos sistemas DSM são baseados exclusivamente em protocolo de invalidações. Em sistemas *software DSM*, o tratamento de atualizações pode envolver alterações nas estruturas de dados do protocolo.

Além de permitir atualizações, o sistema deve ser capaz de detectar quais dados são acessados dentro de seções críticas e quais modificações são realizadas nos seus dados. Num sistema *software DSM*, a forma mais simples de detectar quais dados são acessados dentro de uma seção crítica é através da associação de variável de sincronização com os dados compartilhados protegidos por ela. Por esse motivo, o modelo de consistência EC (com associação implícita ou explícita) é o modelo mais adequado para o emprego de LAP.

A detecção das escritas realizadas dentro da seção crítica pode ser feita de diversas formas. A forma mais simples, que aliás é a utilizada no nosso trabalho, é através do mecanismo de *diffs*. Existem outras possíveis implementações da detecção das escritas, no entanto, tais implementações envolvem instrumentação especial inserida no código fonte (ou no próprio código objeto) da aplicação.

O conjunto de implicações que acabamos de mencionar mostra que a implementação de LAP em sistemas já existentes pode não ser uma tarefa simples, modificando

quase totalmente a estrutura e o código do sistema. Por essa razão, desenvolvemos novos sistemas *software DSM*, os quais são adequados à implementação de LAP. Tais sistemas são descritos em detalhes nos próximos capítulos.

Capítulo 5

O Sistema AEC

Nesse capítulo apresentamos o sistema *software DSM* AEC (*Affinity Entry Consistency*) desenvolvido para explorar as vantagens oferecidas por LAP. Conforme o próprio nome já diz, o sistema AEC é baseado no modelo EC de consistência de memória. Conforme já mencionado, EC é o modelo mais relaxado de todos os modelos de consistência de memória e se adequa perfeitamente às otimizações propostas por LAP.

A associação de variáveis de sincronização com dados compartilhados, imposta pelo modelo EC, é realizada em AEC de forma implícita. O próprio protocolo estabelece essa relação dinamicamente durante a execução da aplicação. Dessa forma, AEC apresenta ao programador um modelo de programação simples, bastante parecido com o utilizado em sistemas baseados em outros modelos relaxados de consistência de memória como RC ou LRC.

A seção 5.1 descreve em maiores detalhes o modelo de programação de memória compartilhada empregada em AEC e suas restrições. A seção 5.2 mostra uma visão geral do protocolo de coerência utilizado em AEC e nas duas seções seguintes apresentamos as funcões básicas do protocolo nas operações de sincronização e nas falhas de acesso. A última seção apresenta a interface oferecida ao usuário e alguns aspectos da implementação de AEC próprios do ambiente Unix utilizado.

5.1 Modelo de Programação

AEC procura obter as vantagens do modelo de consistência EC sem a necessidade da associação explícita de variável de sincronização com dado compartilhado. Na maioria dos casos, programas que utilizam modelo de programação com memória compartilhada e sincronização explícita, com operações de *lock* para sincronização local e barreiras para sincronização global, executam corretamente em AEC. Entretanto, o uso do modelo EC implica em algumas restrições adicionais ao modelo de programação, conforme visto no Capítulo 2. Traduzindo as restrições gerais, expostas na seção 2.2.2, para um modelo de programação com sincronização por *locks* e barreiras, temos as seguintes restrições:

- modificações no dado compartilhado realizadas sob uma determinada seção crítica só são visíveis em um processador p quando p adquire a seção crítica correspondente;
- modificações no dado compartilhado que não são protegidas por seções críticas não são visíveis em p antes da próxima barreira.

Para entender melhor como uma aplicação pode executar corretamente em RC ou LRC e não atender aos requerimentos acima, vejamos alguns exemplos. Considere uma aplicação que utiliza filas de tarefas para distribuir a computação. Somente a retirada de uma tarefa da fila é realizada sob seção crítica. Os acessos compartilhados realizados pela tarefa, porém, não utilizam seções críticas. O programador pode assumir que, quando uma tarefa é selecionada da fila, estão visíveis as modificações realizadas pelas tarefas anteriores já completadas (porque senão a tarefa corrente não teria sido criada). Essa premissa, entretanto, só é verdadeira para os modelos RC e LRC, mas não para EC. Para executar corretamente em EC essa aplicação deveria expandir as seções críticas para que elas pudessem englobar as modificações

realizadas pelas tarefas.

Como um outro exemplo, vamos extrapolar os segmentos de código da figura 2.4 da seção 2.2.2 para uma aplicação real. Considere uma aplicação em que, antes de computar os valores de um nó de uma árvore, um processo tem que esperar que os valores de todos os filhos do nó tenham sido computados. Quando um filho está pronto, ele avisa ao pai ativando um *flag* compartilhado por eles. O acesso ao *flag* é realizado com operações *lock/unlock*. Em EC, o fato de receber o valor do *flag* do filho não significa que o processo recebeu também as modificações que o filho fez no seu nó, fora de seção crítica. Mais uma vez, a seção crítica deveria ter sido expandida para englobar as modificações realizadas no conteúdo do nó.

5.2 Protocolo de Coerência

O protocolo de coerência utilizado no sistema AEC utiliza a página como unidade de coerência e, para aliviar o problema de falso compartilhamento, permite a existência de múltiplos escritores numa mesma página. AEC utiliza mecanismo de criação de *diffs* para suportar a existência de múltiplos escritores.

A idéia básica do protocolo utilizado em AEC é dar tratamento diferenciado para dados acessados dentro e fora de seções críticas. Para os dados compartilhados acessados dentro de seções críticas, a técnica LAP viabiliza o uso de protocolo de atualização para manutenção da coerência, porque permite que as atualizações sejam enviadas de forma seletiva. Para dados compartilhados acessados fora de seções críticas, entretanto, AEC mantém a coerência através de protocolo de invalidação para não sobrecarregar a rede com grande quantidade de mensagens.

5.3 Sincronização Local e a Utilização de LAP

O protocolo de atualização empregado em AEC para manter a coerência dos dados modificados sob seções críticas apresenta o seguinte funcionamento geral: o pro-

cessador p, que está saindo de uma seção crítica s, utiliza a técnica LAP e envia antecipadamente para o conjunto $U_p(s)$ todas as modificações já realizadas dentro da seção crítica s. Quando LAP acerta sua previsão, o próximo processador a utilizar a seção crítica já está com os dados coerentes e tem que esperar somente pela sincronização. Já no caso de LAP errar a previsão, o processador entra na seção crítica sem ter recebido o conjunto de modificações, na primeira falha de acesso ocorrida, o processador, então, requisita as modificações para o último dono da seção crítica.

A figura 5.1 mostra um exemplo do funcionamento de LAP no protocolo de atualização de AEC. As setas contínuas representam envios de atualizações ou requisições de dados e as setas pontilhadas representam o encadeamento das operações de sincronização. Inicialmente, P_0 modifica a e por LAP envia a modificação para P_1. A previsão foi acertada porque P_1 requisita o *lock* logo em seguida, atualiza b e envia as modificações de a e b para P_0. Novamente LAP acertou a previsão e P_0 já entra na seção crítica com os dados coerentes. Ao sair da seção crítica, P_0 envia acertadamente as modificações em a, b e c para P_1. Este modifica novamente a e envia as modificações para P_0, mas dessa vez LAP errou a previsão, porque o próximo processador a requisitar o *lock* é P_2. P_2 entra na seção crítica e no primeiro acesso ao dado b, gera uma falha de acesso e busca as modificações de P_1 que foi o último dono do *lock*. Somente depois de receber os dados, é que o processador P_2 escreve efetivamente em b.

Todas as modificações realizadas sob um determinado *lock* são enviadas na forma de *diffs* e são acumuladas de forma combinada (*merged*). Como é comum que numa seção crítica se acesse um conjunto bastante limitado de dados, o tamanho desse conjunto de *diffs* não é um problema para o nosso protocolo.

A maneira de se combinar os *diffs* associados a uma variável de sincronização em AEC, porém, é um tanto quanto diferente de uma operação *merge* tradicional. Um processador ao receber o conjunto de *diffs* de outro processador, cria os *twins* das

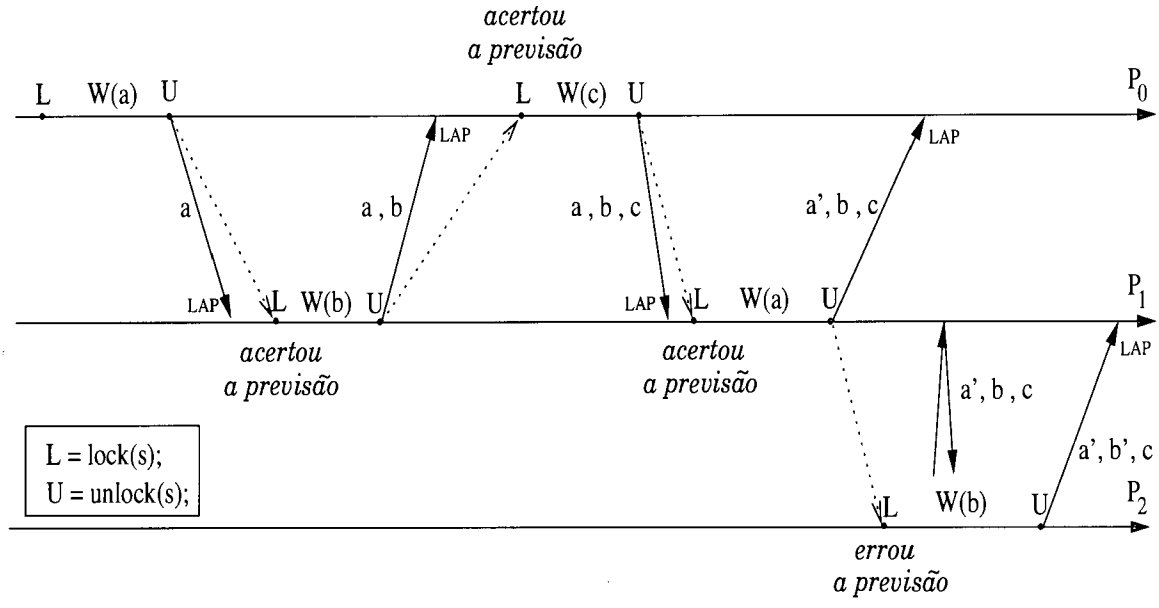

Figura 5.1: O uso de LAP no protocolo de coerência de AEC.

páginas correspondentes, antes de aplicá-los. Criando-se os *twins* antes da aplicação dos *diffs* é possível combinar, na próxima criação de *diffs*, as modificações recebidas do último dono da seção crítica, com as novas modificações realizadas sob a mesma seção crítica, conforme mostra a figura 5.2. Nessa figura apresentamos um exemplo em que um processador recebe um *diff* contendo as modificações nas posições `a` e `b` de uma página. Antes de aplicar o *diff* à página, o processador cria o *twin* com a cópia antiga da página. Em seguida o *diff* é aplicado e a página recebe os valores `<3>` e `<1>` contidos no *diff*. Se durante a execução da seção crítica o processador altera o conteúdo da posição `b` (escrevendo `<2>`), então ao sair da seção crítica o *diff* criado contém apenas os valores `<3>` e `<2>`, que representam a combinação de `<3><1>` com `<2>`.

Uma outra característica importante do protocolo de atualização utilizado em AEC é o aproveitamento do tempo de espera pela sincronização para mascarar alguns *overheads* de coerência, como a aplicação de *diffs* recebidos com antecedência e a criação de *diffs* relativos a modificações realizadas fora de seções críticas. O *overhead* de criação dos *diffs* relativos a modificações realizadas dentro de seções

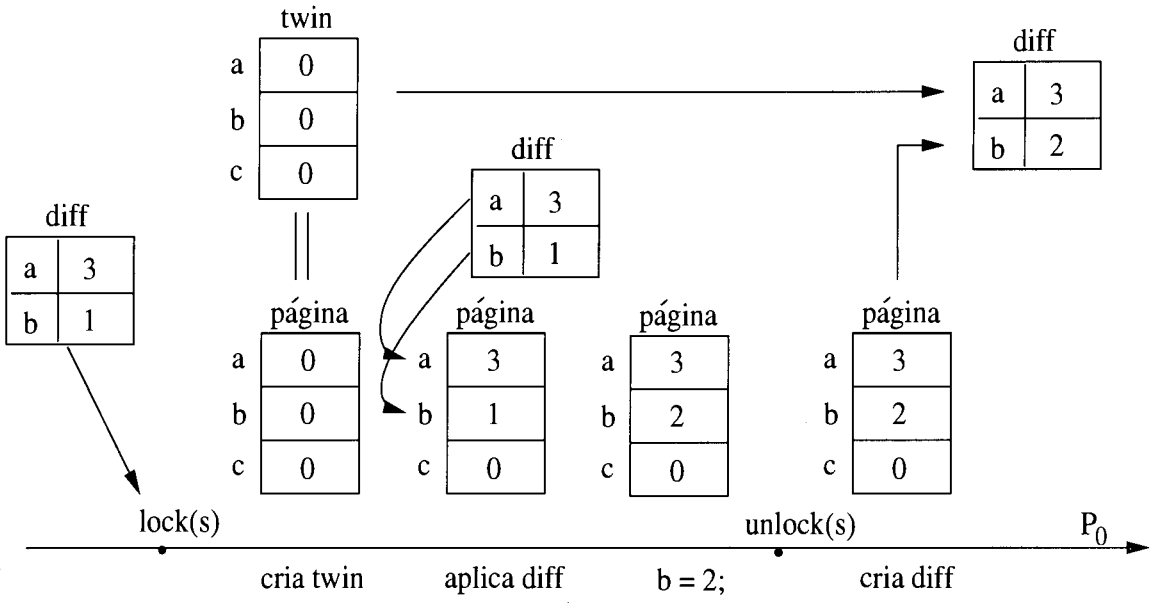

Figura 5.2: Combinação de *diffs* com criação antecipada de *twins*.

críticas, entretanto, não pode ser mascarado.

5.3.1 Os Algoritmos de Lock/Unlock

A implementação das operações *lock* e *unlock* requer a determinação de um processador gerente para cada variável de sincronização s. O processador gerente, G_s é determinado, estaticamente, por $G_s = smodnum.processadores$. G_s é responsável por implementar LAP e por manter um conjunto, chamado m_s, com todas as páginas que foram modificadas sob s.

Os passos realizados pelo protocolo de coerência de AEC no processador p, durante a execução de uma operação *lock* em uma variável de sincronização s, estão ilustrados na figura 5.3 e descritos abaixo:

- p envia mensagem a G_s requisitando o acesso à seção crítica protegida por s;
- enquanto espera pela resposta do gerente:
  - aplica *diffs* (com criação antecipada de *twins*, conforme explicado anteriormente) recebidos por estar em $U_q(s)$ de qualquer outro processador q, *diffs* só são aplicados às páginas válidas;

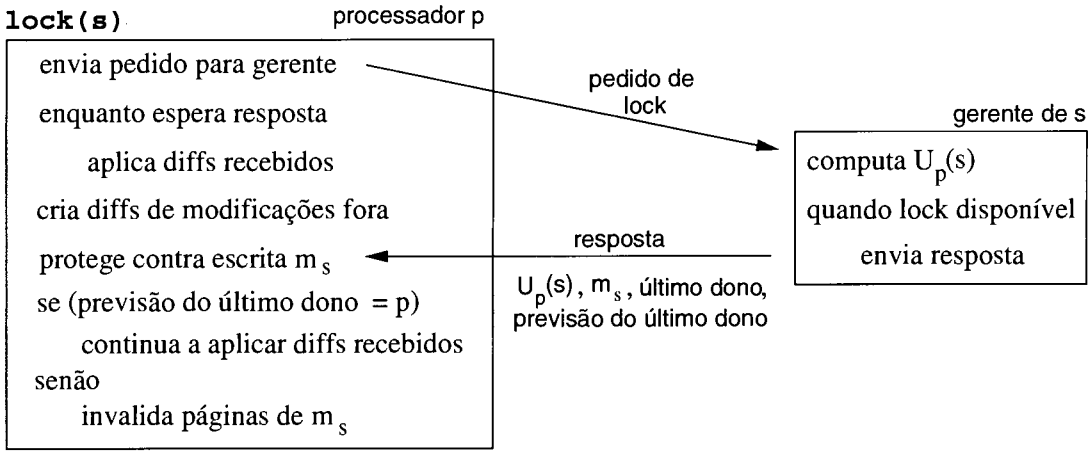

Figura 5.3: Operação *lock* no protocolo de coerência de AEC.

  - cria *diffs* relativos às modificações realizadas fora de seções críticas (esses *diffs* só serão realmente necessários se a página que foi alterada fora de seção crítica, for alterada dentro da seção crítica s, se esse não for o caso, basta manter o *twin* da página durante a execução da seção crítica);

- ao receber a resposta do gerente, o processador recebe também m_s; a identificação do último dono de s, l; o seu conjunto de atualização, $U_p(s)$; e a indicação se $p \in U_l(s)$;

- protege contra escrita todas as páginas de m_s, para poder detectar as modificações realizadas sob s e assim implementar a associação implícita de variável de sincronização com dado compartilhado (no primeiro acesso à seção crítica todas as páginas da aplicação são protegidas);

- se $p \in U_l s$, então o processador continua a aplicar os *diffs* recebidos de l;

- caso contrário, invalida todas as páginas de m_s.

Obviamente, uma operação *lock* é muito mais simples quando o último dono da seção crítica é o próprio processador que está requisitando o *lock*, porque não há necessidade de aplicação de *diffs* ou invalidação de páginas.

Os passos realizados no processador p, durante a execução de uma operação *unlock* em uma variável de sincronização s, estão ilustrados na figura 5.4 e descritos abaixo:

- envia mensagem para G_s, informando que a seção crítica está livre e qual o novo conjunto m_s;
- cria os *diffs* relativos a todas as modificações realizadas sob s;
- associa os *diffs* criados à variável s e protege contra escrita novamente todas as páginas modificadas;
- envia para os processadores de $U_p(s)$ todos os *diffs* ligados a s.

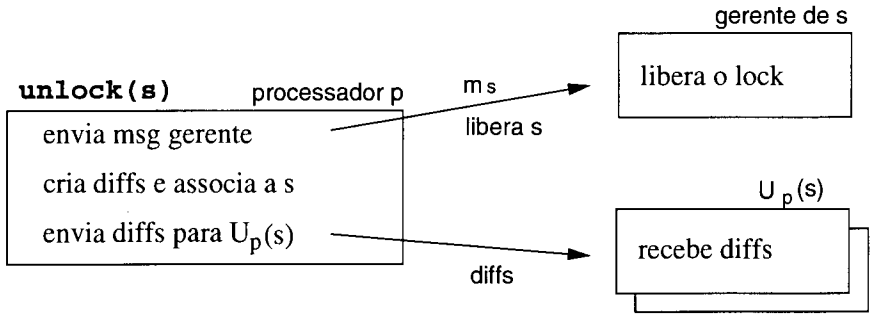

Figura 5.4: Operação *unlock* no protocolo de coerência de AEC.

Todas as operações de *lock* em uma mesma variável de sincronização s podem ser totalmente ordenadas através do uso de um contador, c_s, incrementado a cada nova operação *lock*. As mensagens enviadas para $U_p(s)$ devem conter o valor de c_s para que os processadores que recebem conjuntos de *diffs* de s de processadores diferentes, possam determinar qual conjunto é o mais recente. Suponha um caso em que um processo p, ao liberar a seção crítica s, envia o conjunto de *diffs* para q, mas o próximo dono da seção crítica é r que também envia seu conjunto de *diffs* para q. Como q recebe dois conjuntos diferentes de *diffs*, ele deve estar pronto para descartar o menos atualizado.

5.4 Sincronização Global

Barreiras dividem a execução da aplicação em `fases`. Após a execução de uma barreira, uma nova fase é iniciada e todos os processadores devem ter a mesma visão da memória compartilhada. Assim, numa barreira, AEC tem que garantir a visibilidade de todas as modificações realizadas na fase anterior.

Essa visibilidade é garantida pelo processador gerente da barreira. O gerente recebe de todos os processadores informações sobre as modificações realizadas nas páginas compartilhadas na fase anterior e "avisa" a cada processador sobre as escritas realizadas por outros processadores. Para tal, o gerente utiliza dois tipos de avisos de escrita: `write-notice-out` e `write-notice-in`. Um `write-notice-out` indica que uma página foi alterada fora de seção crítica, mas não contém as alterações realizadas na página. Ele contém apenas a identificação do processador que alterou a página e em qual fase se deu essa alteração. Um `write-notice-in` indica que a página foi alterada em fase anterior sob determinada seção crítica. Ele contém a identificação da seção crítica, do seu último dono e em qual fase se deu a alteração.

O uso de dois tipos de avisos de escrita está relacionado ao tratamento diferenciado de dados compartilhados acessados dentro e fora de seções críticas. Embora as modificações realizadas dentro de seções críticas sejam mantidas coerentes por atualizações (conforme explicado anteriormente), o gerente da barreira deve garantir que elas estejam visíveis também para acessos realizados fora de seções críticas em fases seguintes.

5.4.1 O Algoritmo de Barreira

Tal qual para as variáveis de *lock*, há um processador gerente (determinado estaticamente) para cada variável de barreira. A determinação do gerente de barreira é dada por $G_b = bmodnum.processadores$. Os passos realizados pelo processador p

numa operação de barreira estão ilustrados na figura 5.5 e descritos abaixo:

- ao chegar a uma barreira b, um processador envia para o processador gerente a identificação das páginas alteradas fora de seções críticas e de cada seção crítica s utilizada, juntamente com m_s;
- logo em seguida, o processador pode começar a criar os *diffs* relativos aos acessos realizados fora de seções críticas (dado que normalmente os processadores chegam na barreira em tempos diferentes, essa criação de *diffs* consegue se sobrepor quase totalmente com o tempo de espera na barreira);
- o processador gerente coleta as informações de todos os processadores e determina sobre quais modificações cada processador deve ficar ciente; o gerente deve informar sobre as modificações realizadas dentro de uma seção crítica s aos processadores p, tal que $p \neq l$ (onde l é o último dono de s) e $p \notin U_l(s)$ (os processadores de $U_l(s)$ já receberam as últimas atualizações realizadas sob s);
- em seguida, o gerente envia `write-notices-out` e `write-notices-in` para os respectivos processadores.

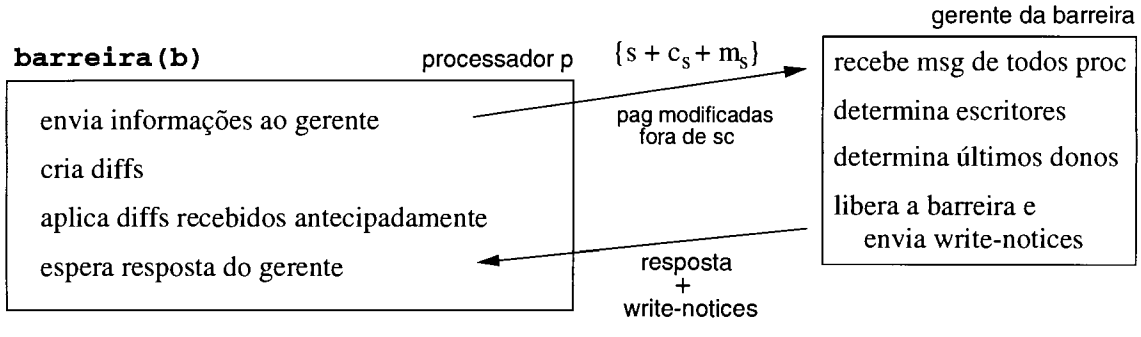

Figura 5.5: Operação de barreira no protocolo de coerência de AEC.

Ao receber um `write-notice-out` e/ou um `write-notice-in`, o processador invalida a página correspondente. Note que, para uma mesma página um processador pode receber diversos avisos de escrita.

5.5 Falhas de Acesso

O algoritmo utilizado pelo protocolo de coerência de AEC para o tratamento de uma falha de acesso está ilustrado na figura 5.6 e descrito abaixo:

- se a falha ocorreu dentro de uma seção crítica s, então o processador deve verificar se está no conjunto de atualização do último dono de s:
  - em caso positivo, ele aplica os *diffs* associados a s que não conseguiu aplicar na operação de *lock* porque a página estava inválida;
  - em caso negativo, ele requisita ao último dono de s todos os *diffs* associados a s;
- se a falha ocorreu fora de uma seção crítica, o processador deve verificar o conjunto de avisos de escrita recebidos para a página e verificar, também, se recebeu antecipadamente, do último dono de alguma seção crítica na fase anterior, *diffs* referentes à página em questão. Se esse for o caso, o processador:
  - aplica os *diffs* recebidos antecipadamente em ordem;
  - busca os *diffs* referentes aos `write-notices-in` e `write-notices-out`, combinando mensagens sempre que possível;
  - aplica os *diffs* recebidos de acordo com a fase em que cada modificação foi realizada.

O protocolo de coerência de AEC apresenta, ainda, uma característica especial nas falhas de acesso. Toda vez que um processador passa mais de duas fases da computação sem acessar uma página, na primeira falha de acesso ocorrida, o processador requisita uma cópia da página ao processador `home`. Há um processador `home` para cada página, que é determinado a cada fase da computação pelo gerente da barreira.

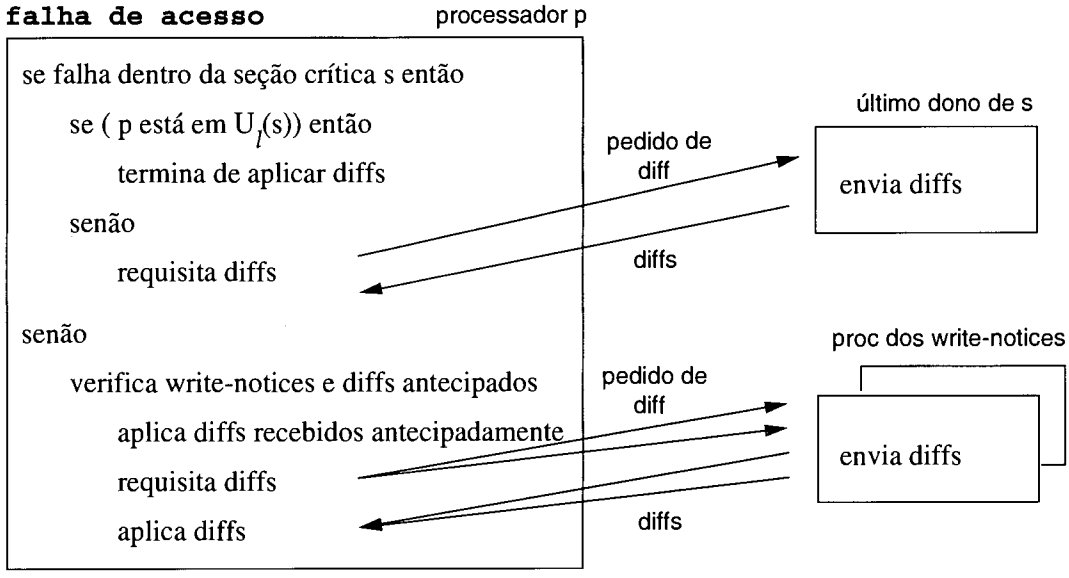

Figura 5.6: Operação de falha de acesso no protocolo de coerência de AEC.

O `home` é escolhido preferencialmente do conjunto dos processadores que têm cópia válida da página. Essa característica do protocolo de AEC foi implementada com o objetivo de evitar que um processador, que passou um período sem usar a página, perca muito tempo coletando *diffs* de um grande conjunto de processadores. O `home` já coletou e aplicou boa parte desses *diffs*. O processador deverá buscar, no máximo, os *diffs* relativos aos avisos de escrita presentes no `home`.

Note que, embora estejamos utilizando um processador `home` para cada página, não consideramos que o protocolo de AEC seja baseado em *home-nodes*. Isso porque o uso de `homes` em AEC ocorre de forma bastante restrita, somente para casos muito específicos de processadores que passam períodos sem acessar determinadas páginas.

5.6 Aspectos de Implementação

5.6.1 Interface de Programação

O sistema AEC é completamente implementado como uma biblioteca C, utilizando uma interface idêntica a do sistema TreadMarks para suportar a criação e a comunicação entre processos. As primitivas fornecidas por AEC são utilizadas pela aplicação como chamadas de procedimento.

Programas em AEC seguem o estilo convencional SPMD de programação no modelo de memória compartilhada, utilizando processos para expressar o paralelismo e primitivas de *lock* e barreira para sincronização. As primitivas `Aec_lock` e `Aec_unlock` são utilizadas para implementar as operações *lock* e *unlock*. A primitiva `Aec_barrier` é utilizada para implementar uma barreira. Variáveis de sincronização são definidas como números inteiros positivos.

A inicialização dos processos paralelos é realizada por um único processo (`processo pai`), através da primitiva `Aec_startup`. É inicializado um único processo em cada processador.

Cada processo paralelo, então, inicializa as estruturas internas do sistema AEC e aloca uma área de 32M de sua memória privativa para armazenar os dados compartilhados. O `processo pai` é responsável por alocar cada estrutura compartilhada da aplicação nessa área, através da primitiva `Aec_malloc`. Em seguida, ele distribui para os outros processadores os endereços das estruturas alocadas, através da primitiva `Aec_distribute`.

Depois que a distribuição dos endereços está completa, todos os processos (inclusive o `processo pai`) executam o mesmo código, realizando sua parcela do trabalho e se comunicando através das primitivas de sincronização (`Aec_lock`, `Aec_unlock` e `Aec_barrier`).

A figura 5.7 mostra um exemplo simples de um programa escrito segundo a interface fornecida por AEC. Este programa executa em n processadores e é composto de 3 fases distintas. Na primeira fase, o processador 0 inicializa um vetor compartilhado. Na segunda fase, todos os processadores incrementam o conteúdo de cada posição do vetor com o seu *pid*. Na terceira fase, os processadores devem ler o vetor compartilhado, para imprimir o conteúdo de cada posição. O programa está corretamente sincronizado e, para n processadores, o valor de todos os elementos do vetor deve ser $\frac{n(n-1)}{2}$.

5.6.2 Características da Implementação

Um programa em AEC consiste de um conjunto de processos que se comunicam através de *sockets*, utilizando protocolo UDP/IP de comunicação. A primitiva `Aec_startup` é responsável pela criação e conexão dos *sockets* entre os processos. Há dois *sockets* diferentes ligando cada par de processos, um para comunicação com interrupção na chegada de cada mensagem e outro para comunicação sem interrupção.

Tanto para a implementação da comunicação entre os processos como para o tratamento de falhas de acesso no protocolo de coerência de AEC, utilizamos as facilidades de tratamento de sinais do sistema Unix. Em Unix, sinais são utilizados para notificar a ocorrência de certos eventos. Quando um sinal é ativado, o sistema desvia para o procedimento de tratamento do sinal, que pode ser definido a nível de usuário, ou através de rotinas-padrão do sistema operacional. O protocolo de AEC possui rotinas próprias para o tratamento dos seguintes sinais:

- SIGIO, que indica que uma operação de E/S é possível em um descritor de arquivos. Como a comunicação em *sockets* segue o padrão de leituras e escritas em arquivos, o tratamento deste sinal é utilizado para permitir a comunicação entre processos.

- SIGALARM, que é utilizado para temporização, em especial para a implementação de mecanismos de *time-out*.

- SIGSEGV, que indica que um processo tentou acessar um endereço para o qual ele não tem permissão de acesso condizente, representa uma falha de acesso no endereço. A rotina de tratamento deste sinal é responsável por realizar as ações de coerência necessárias para manter uma página válida, conforme descrito anteriormente.

Finalmente, é importante notar que, na implementação de AEC e dos outros sistemas que propomos nesta tese, não estamos considerando aspectos de *overhead* de memória (i.e., *garbage collection*), uma vez que em geral a quantidade de memória das *workstations* é suficiente para executar as nossas aplicações com tamanhos de entrada usuais.

```
#include "Aec.h"
int  *shared_array, arrayDim = 10 ;
main(int argc, char **argv)
{
    int start, end, i, p, c, b, s;

    Aec_startup(argc, argv);
    a = 1; s = 2;
    /* Aloca o vetor compartilhado e distribui enderecos */
    if (Aec_proc_id == 0) {
        shared_array = (int *) Aec_malloc(arrayDim * sizeof(int));
        Aec_distribute(&array, sizeof(array)); }

    Aec_barrier(b);

    /* Primeira fase - inicializa vetor compartilhado */
    if (Aec_proc_id == 0) {
     for (i = 0; i < arrayDim; i++)
         shared_array[i] = 0; }
    Aec_barrier(b);

    /* Segunda fase - escreve pid no vetor */
    Aec_lock(s);
    for (i = 0; i < arrayDim; i++)
        shared_array[i] += Aec_proc_id;
    Aec_unlock(s);
    Aec_barrier(b);

    /* Terceira fase - imprime conteudo do vetor */
    for (i = 0; i < arrayDim; i++)
        printf(" %d", shared_array[i]);
    Aec_exit(0);
}
```

Figura 5.7: Exemplo simples da utilização da interface de programação de AEC.

Capítulo 6

O Sistema AEC-light

Nesse capítulo apresentamos o sistema *Affinity Entry Consistency-Light* (AEC-light) desenvolvido a partir do sistema AEC, mas considerando um modelo de programação mais elaborado, em que todos os acessos a dados compartilhados são realizados sob *locks*. O sistema AEC-light foi proposto como uma simplificação de AEC, portanto, é baseado no mesmo modelo de consistência (EC) e realiza a associação de variável de sincronização com dado compartilhado dinamicamente, de forma transparente à aplicação.

É sabido que utilizar somente seções críticas para proteger os acessos aos dados compartilhados leva a um modelo de programação um pouco mais complexo. Em compensação, permite que a técnica LAP seja explorada de forma mais agressiva, dado que é possível utilizar, além das técnicas de fila de espera e afinidade local, a técnica de afinidade global.

Todos os aspectos de implementação, descritos na seção 5.6 para AEC, valem para AEC-light. A seguir, apresentamos as diferenças principais de AEC-light em relação a AEC no que diz respeito ao modelo de programação, ao protocolo de coerência empregado e à interface de programação.

6.1 Modelo de Programação

O modelo de programação utilizado pelo protocolo de AEC-light apresenta, tal qual o modelo de programação utilizado por AEC, interface com operações tipo *lock/unlock* e barreira. Entretanto, para uma aplicação executar corretamente em AEC-light, ela deve atender a algumas restrições extras:

- para cada estrutura de dados compartilhada deve existir uma variável de sincronização correspondente;
- todos os acessos a dados compartilhados devem ser feitos sob *locks*;
- operações de barreira servem somente para sincronização global, não para coerência.

A primeira restrição impõe que o programador estabeleça uma variável de sincronização para cada dado compartilhado, um mesmo dado compartilhado deve ser sempre acessado sob o mesmo *lock*. A segunda restrição é a base do modelo utilizado por AEC-light, todos os acessos a dados compartilhados, sejam eles de leitura ou de escrita, devem ser realizados dentro da seção crítica correspondente. Já a terceira restrição é uma consequência direta da segunda restrição, se todos os acessos são protegidos por *locks*, então não há necessidade de se garantir coerência também na barreira.

Uma aplicação que atende às três restrições acima executa corretamente no protocolo de AEC-light. No entanto, para evitar problemas de desempenho causados pelo excesso de sincronização, AEC-light oferece dois pares de primitivas de *lock* que não requerem nenhum tipo de sincronização, *lock-reader/unlock-reader* e *lock-alone/unlock-alone*, além das primitivas *lock/unlock* padrão. As primitivas *lock-reader/unlock-reader* delimitam seções críticas de acesso não exclusivo, devendo ser utilizadas quando, numa determinada fase da computação, os processadores vão

realizar apenas acessos de leitura aos dados protegidos pelas seções. As primitivas *lock-alone/unlock-alone* delimitam seções críticas de escrita por um único processador, devendo ser utilizadas quando, numa determinada fase da computação, um único processador escreve no dado compartilhado. Note que, as primitivas para escrita exclusiva assumem que não há leitores concorrentes, uma vez que esse tipo de compartilhamento necessitaria de primitivas de *lock* padrão.

6.2 Protocolo de Coerência

Tal qual o protocolo empregado em AEC, o protocolo de coerência de AEC-light utiliza a página como unidade de coerência, permite a existência de múltiplos escritores numa mesma página e utiliza *twins* e *diffs* para detectar as modificações realizadas numa determinada página.

Entretanto, o protocolo de coerência de AEC-light é bem mais simples do que o protocolo de AEC. O principal motivo dessa simplicidade é que AEC-light não precisa tratar falhas de acesso ocorridas fora de seções críticas. Em conseqüência disso, não há necessidade de `write-notices-in` e nem de `write-notices-out`; as operações de coerência ocorrem somente num pedido de *lock* ou numa falha de acesso ocorrida dentro de seção crítica (quando LAP não acerta sua previsão). Mesmo no tratamento de falhas ocorridas dentro de seções críticas, em AEC-light não existe a necessidade de se coletar *diffs* de diferentes processadores, já que o último dono do *lock* possui todos os *diffs*. Finalmente, as operações de coerência que ocorrem nas barreiras de AEC não são mais necessárias em AEC-light. Podemos ilustrar essa simplificação a grosso modo através do número de linhas de código gasto na implementação dos três modulos mais importantes em qualquer sistema software DSM: falhas de acesso, *locks* e barreiras. AEC-light reduz os números de linhas dos módulos de falhas e barreiras em 51% e 38%, respectivamente, enquanto que o código de *locks* aumenta em somente 15%.

A técnica LAP é utilizada da mesma forma que em AEC para acessos realizados em seções críticas protegidas por operação de *lock* padrão. O uso de *lock-alone* e *lock-reader* no modelo de programação permite que o protocolo de coerência de AEC-light empregue, numa operação de barreira, a técnica de previsão de LAP de afinidade global.

A figura 6.1 mostra um exemplo do funcionamento da técnica de afinidade global de LAP no protocolo de AEC-light. As setas contínuas representam envios de atualizações e as setas pontilhadas representam uma sincronização global. Inicialmente, P_0 modifica a, b e c que são protegidas por um *lock-alone*. Na chegada da barreira, P_0 através de LAP global envia as modificações para P_1. A previsão foi acertada porque P_1 executa um *lock-alone* na mesma variável de sincronização na fase seguinte. P_1 atualiza a e c e na chegada da barreira envia as modificações para P_0. Mas neste caso LAP global errou a previsão, porque na fase seguinte é P_2 quem utiliza a seção crítica s, através de um *lock-reader*. Na execução do *lock-reader*, então, P_2 busca as modificações de P_1.

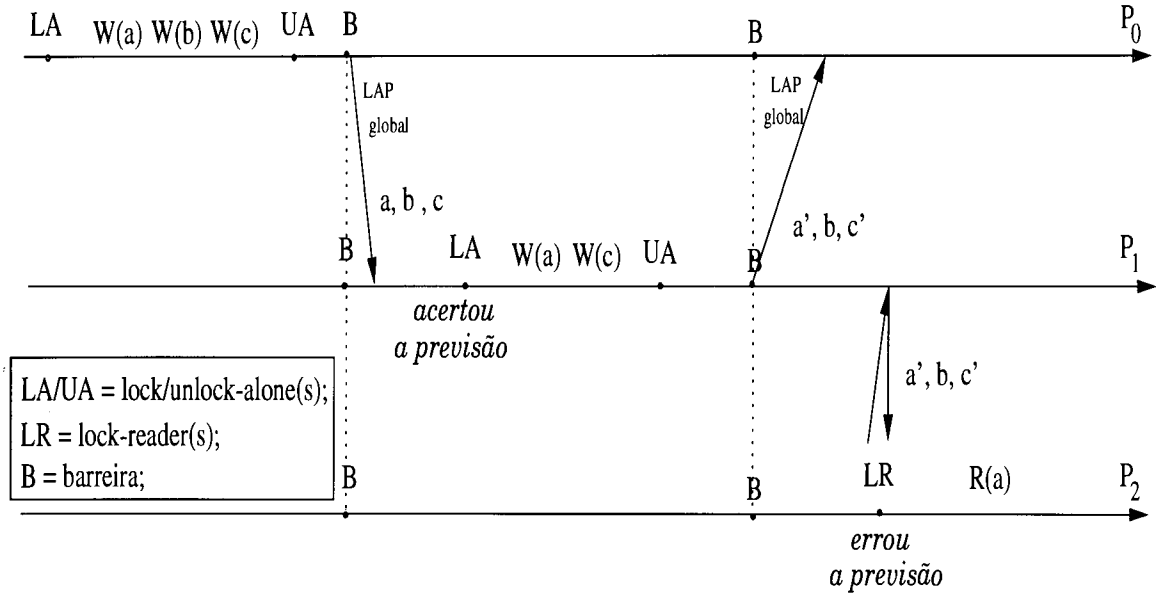

Figura 6.1: O uso de LAP global no protocolo de coerência de AEC-light.

6.3 Sincronização Local e a Utilização de LAP

Conforme mostramos no modelo de programação de AEC-light, o programador pode utilizar três tipos diferentes de operações de *lock*: *lock*; *lock-reader*; e *lock-alone*.

As operações *lock/unlock* padrão e o respectivo uso de LAP são implementados em AEC-light exatamente da mesma forma que em AEC (seção 5.3). Portanto, vamos nos ater aqui à explicação da implementação das outras duas operações de *lock*.

6.3.1 O Algoritmo de Lock-Reader e Unlock-Reader

A operação *lock-reader* é implementada de forma bem simples. A cada operação de *lock-reader* numa variável de sincronização s, os seguintes passos (ilustrados na figura 6.2) são realizados pelo processador p:

- se, de acordo com as informações recebidas na barreira (conforme explicado a seguir), o processador não está no conjunto $U_l(s)$ do último dono l, i.e., se ele não foi previsto como provável próximo *acquirer* de s, então ele deve solicitar a l todos os *diffs* associados a s;
- ao receber os *diffs*, o processador os aplica em suas respectivas páginas.

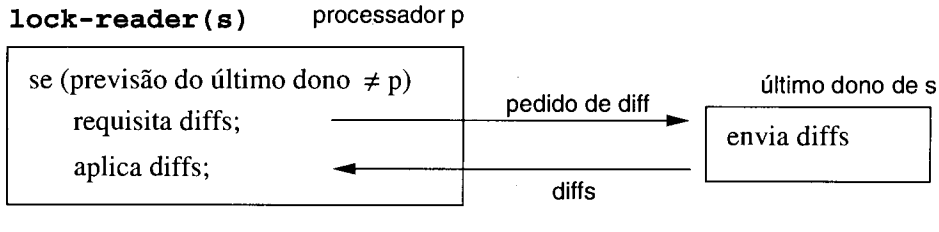

Figura 6.2: Operação *lock-reader* no protocolo de coerência de AEC-light.

A operação *unlock-reader* não realiza nenhuma função; ela pode ser utilizada opcionalmente pelo programador para efeito de delimitação da seção crítica.

6.3.2 O Algoritmo de Lock-Alone e Unlock-Alone

As operações *lock-alone/unlock-alone* também não requerem nenhum tipo de sincronização com outros processadores. Os passos realizados pelo protocolo de coerência de AEC-light no processador p, durante a execução de uma operação *lock-alone* em uma variável de sincronização s, estão ilustrados na figura 6.3 e descritos abaixo:

- se o processador não está no conjunto $U_l(s)$ do último dono l, ele deve requisitar a l todos os *diffs* de s;
- como haverá escrita no dado, antes de aplicar os *diffs* recebidos, são criados *twins* para suas páginas, para se obter o efeito, já explicado anteriormente, de combinação de *diffs*.

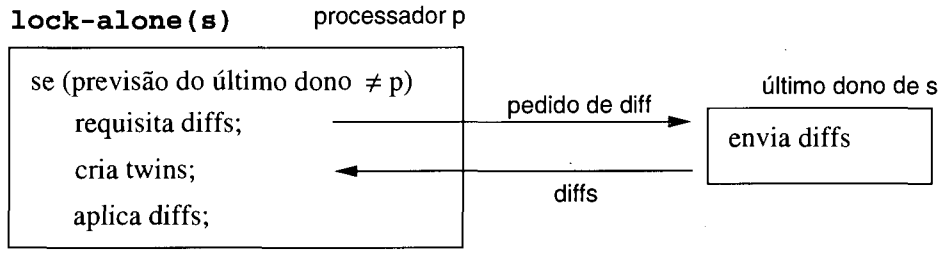

Figura 6.3: Operação *lock-alone* no protocolo de coerência de AEC-light.

Na operação *unlock-alone*, o processador cria os *diffs* relativos às modificações no dado e os associa a s.

6.4 Sincronização Global e a Utilização de LAP

Em AEC-light, a barreira funciona basicamente como sincronização global; não há necessidade de se garantir coerência de dados. Portanto, não há necessidade de se utilizar avisos de escrita (`write-notices-out` e `write-notices-in`) e nem de se invalidar páginas.

Na verdade, além da sincronização, a barreira em AEC-light é utilizada também para implementar a técnica de afinidade global de LAP e para trocar algumas in-

formações úteis às operações *lock-reader/unlock-reader* e *lock-alone/unlock-alone* de fases posteriores. Essas informações são: a identificação do último dono de cada seção crítica s; o conteúdo do contador c_s; e o conjunto de páginas alteradas sob cada *lock-alone*. A troca dessas informações na operação de barreira evita que, num pedido de *lock-reader* ou *lock-alone* em s, o processador p tenha que requisitar ao gerente de s a identificação do último dono de s, l; evita que, quando $p \in U_l(s)$, ele tenha que requisitar a l o conjunto de *diffs* associados a s; e também evita que a cada *unlock-alone* em s, o processador tenha que informar ao gerente sobre o conjunto m_s de páginas alteradas sob s.

6.4.1 O Algoritmo de Barreira

Os passos realizados pelo processador p numa operação de barreira estão ilustrados na figura 6.4 e descritos abaixo:

- ao chegar à barreira, o processador envia uma mensagem ao gerente contendo as seguintes informações: a identificação de cada variável de sincronização s utilizada numa operação *lock* ou numa operação *lock-alone* na fase anterior; o contador c_s; e o conjunto m_s;

- logo em seguida, enquanto espera pela sincronização global, o processador envia para os processadores do conjunto $UG_p(k)$ os *diffs* relativos às modificações realizadas sob um *lock-alone* na variável de sincronização k;

- após enviar as mensagens para $UG_p(k)$, se a sincronização ainda não terminou, o processador pode aplicar *diffs* recebidos por estar em $U_l(s)$ de alguma seção crítica s.

- depois que o gerente recebeu mensagens de todos os processadores, ele determina, pelos contadores c_s recebidos, os últimos donos de cada *lock* s;

- o gerente envia mensagem de volta para todos os processadores liberando a barreira e informando os últimos donos de cada *lock* s, juntamente com m_s e c_s.

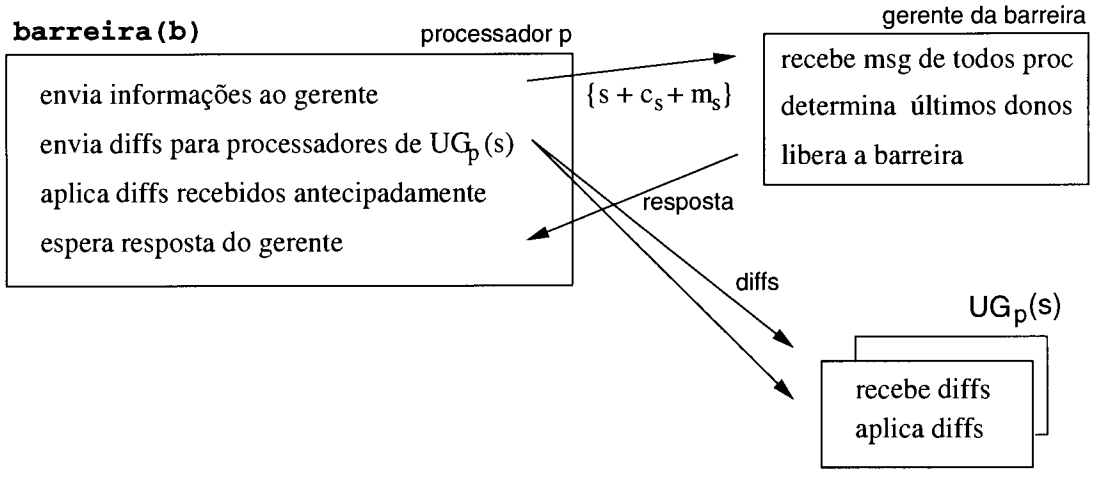

Figura 6.4: Operação de barreira no protocolo de coerência de AEC-light.

Ao receber o conteúdo do contador c_s do último dono l, um processador pode determinar, através dos contadores dos conjuntos de *diffs* recebidos com antecedência, se ele estava em $U_l(s)$ e assim, numa operação *lock-reader* ou *lock-alone* na fase seguinte, ele não precisa requisitar os *diffs* associados a s.

6.5 Interface de Programação

O sistema AEC-light utiliza a mesma interface de programação do sistema AEC, exceto pela inclusão dos pares de primitivas `Aec_lock_reader/Aec_unlock_reader` e `Aec_lock_alone/Aec_unlock_alone`, para implementar as operações *lock/unlock-reader* e *lock/unlock-alone*.

Na figura 6.5 apresentamos o mesmo programa da figura 5.7, mas sob a interface de programação de AEC-light. Como todos os acessos devem ser realizados dentro de seções críticas, na fase de inicialização do vetor compartilhado foram inseridas as primitivas `Aec_lock_alone` e `Aec_unlock_alone`. Na última fase da computação foi inserida uma primitiva `Aec_lock_reader`.

```
#include "Aec.h"
int  *shared_array, arrayDim = 10 ;
main(int argc, char **argv)
{
    int start, end, i, p, c, b, s;

    Aec_startup(argc, argv);
    a = 1; s = 2;
    /* Aloca o vetor compartilhado e distribui enderecos */
    if (Aec_proc_id == 0) {
        shared_array = (int *) Aec_malloc(arrayDim * sizeof(int));
        Aec_distribute(&array, sizeof(array)); }

    Aec_barrier(b);

    /* Primeira fase - inicializa vetor compartilhado */
    if (Aec_proc_id == 0) {
        Aec_lock_alone(s);
        for (i = 0; i < arrayDim; i++)
             shared_array[i] = 0;
        Aec_unlock_alone(s);}
    Aec_barrier(b);

    /* Segunda fase - escreve pid no vetor */
    Aec_lock(s);
    for (i = 0; i < arrayDim; i++)
        shared_array[i] += Aec_proc_id;
    Aec_unlock(s);
    Aec_barrier(b);

    /* Terceira fase - imprime conteudo do vetor */
    Aec_lock_reader(s);
    for (i = 0; i < arrayDim; i++)
        printf(" %d", shared_array[i]);
    Aec_exit(0);
}
```

Figura 6.5: Exemplo simples da utilização da interface de programação de AEC-light.

Capítulo 7

O Sistema AEC-bind

Nesse Capítulo apresentamos o sistema *Affinity Entry Consistency-Bind* (AEC-bind). O sistema AEC-bind foi desenvolvido para o mesmo modelo de programação de AEC-light com um requerimento extra: associação explícita de dado compartilhado com variável de sincronização. Utilizando associação explícita, foi possível desenvolver um protocolo de coerência ainda mais simples.

Todos os aspectos de implementação de AEC-bind são idênticos aos de AEC-light. A seguir, apresentamos as diferenças principais de AEC-bind em relação a AEC-light no que diz respeito ao modelo de programação, ao protocolo de coerência empregado e à interface de programação.

7.1 Modelo de Programação

O modelo de programação utilizado pelo protocolo AEC-bind é semelhante ao modelo de programação de AEC-light, já que requer que todos os acessos a dados compartilhados sejam realizados dentro da seção crítica correspondente e provê operações *lock-reader* e *lock-alone* para implementar respectivamente acesso não exclusivo e acesso único a determinada seção crítica. Para uma aplicação executar corretamente em AEC-bind, ela deve atender às mesmas restrições do modelo de programação de AEC-light. Essas restrições estão descritas na seção 6.1.

A única diferença entre os modelos de programação de AEC-light e AEC-bind

está na associação das variáveis de sincronização com dados compartilhados. O sistema AEC-light realiza essa associação dinamicamente, de forma transparente à aplicação. Em contraste, o sistema AEC-bind requer que essa associação seja feita explicitamente, pelo programador, através da operação `bind`.

A operação bind faz a ligação de cada estrutura compartilhada com a variável de sincronização que a protege. Uma mesma variável de sincronização pode ser utilizada para proteger várias estruturas compartilhadas. Todas as estruturas protegidas por uma mesma variável de sincronização s são representadas por O_s e chamadas de `objeto` de s.

7.2 Protocolo de Coerência

O protocolo empregado em AEC-bind utiliza o objeto como unidade de coerência e emprega as técnicas de LAP para enviar as atualizações de forma antecipada e seletiva. A idéia é bastante simples, cada processador p ao sair de uma seção crítica s envia o objeto O_s para os processadores do conjunto $U_p(s)$. Numa operação de barreira, O_s é enviado para os processadores de $UG_p(s)$.

O protocolo de AEC-bind é ainda mais simples do que o protocolo de AEC-light. O principal motivo dessa simplicidade é que AEC-bind não precisa tratar falhas de acesso. Em conseqüência disso, as operações de coerência ocorrem somente num pedido de *lock* (quando LAP não acerta sua previsão). Além disso, como a unidade de acesso é idêntica à unidade de coerência, não há falso compartilhamento e, portanto, não há necessidade de se prover nenhum tipo de suporte a múltiplos escritores como criação de *twins*, criação de *diffs* e aplicação de *diffs*. Podemos ilustrar essa simplificação a grosso modo através do número de linhas de código gasto na implementação de falhas de acesso, *locks* e barreiras. Em comparação a AEC-light, AEC-bind elimina todo o código de falhas de acesso, além de reduzir em 48 e 35% os códigos de *locks* e barreiras, respectivamente.

Em AEC-bind os objetos inteiros são transferidos nos envios de atualizações. Este tipo de transferência pode causar um aumento do tráfego de dados. No entanto, nossa motivação para transferir objetos inteiros está em dois fatos:

- Quando aplicações utilizam seções críticas para proteger objetos muito grandes, normalmente esses objetos são completamente reescritos (pois, caso contrário, o código estaria excessivamente serializado). Quando um objeto é totalmente reescrito, ele deve ser enviado de qualquer forma.

- Quando aplicações utilizam seções críticas para proteger objetos pequenos, a inclusão de alguns bytes extras na mensagem, mesmo que desnecessários, não gera *overhead* significativo no sistema.

Dessa forma, diferente de alguns sistemas *software DSM* baseados em objetos (como por exemplo Midway[11] e Shasta[63]), AEC-bind não realiza detecção das escritas realizadas no objeto, distinguindo que um dado é modificado apenas pelo uso das operações *lock* ou *lock-alone*.

7.3 Sincronização Local e a Utilização de LAP

O protocolo de atualização empregado em AEC-bind para implementar uma operação *lock* padrão segue os mesmos princípios básicos do protocolo utilizado em AEC. Quando LAP acerta sua previsão, o próximo processador a utilizar a seção crítica já está com o objeto coerente e tem que esperar somente pela sincronização. No caso de LAP errar a previsão, o funcionamento do protocolo é diferente. O processador requisita a versão coerente do objeto ao último dono da seção crítica, ao entrar na seção crítica e não na primeira falha de acesso ocorrida (em AEC-bind não há falhas de acesso). A figura 7.1 mostra o funcionamento do protocolo de AEC-bind num exemplo idêntico ao da figura 5.1. As setas contínuas representam

envios de atualizações e as setas pontilhadas representam o encadeamento das operações de sincronização. Considerando que a operação bind foi realizada no início da aplicação e realizou a associação de s com os dados a, b e c, podemos observar que em AEC-bind as modificações não vão sendo acumuladas a cada *lock*, mas em todos as passagens de *locks*, o objeto inteiro é enviado pelo protocolo. Além disso, podemos observar que P_2 não espera o acesso a b para buscar o objeto de P_1, ele o faz na entrada da seção crítica.

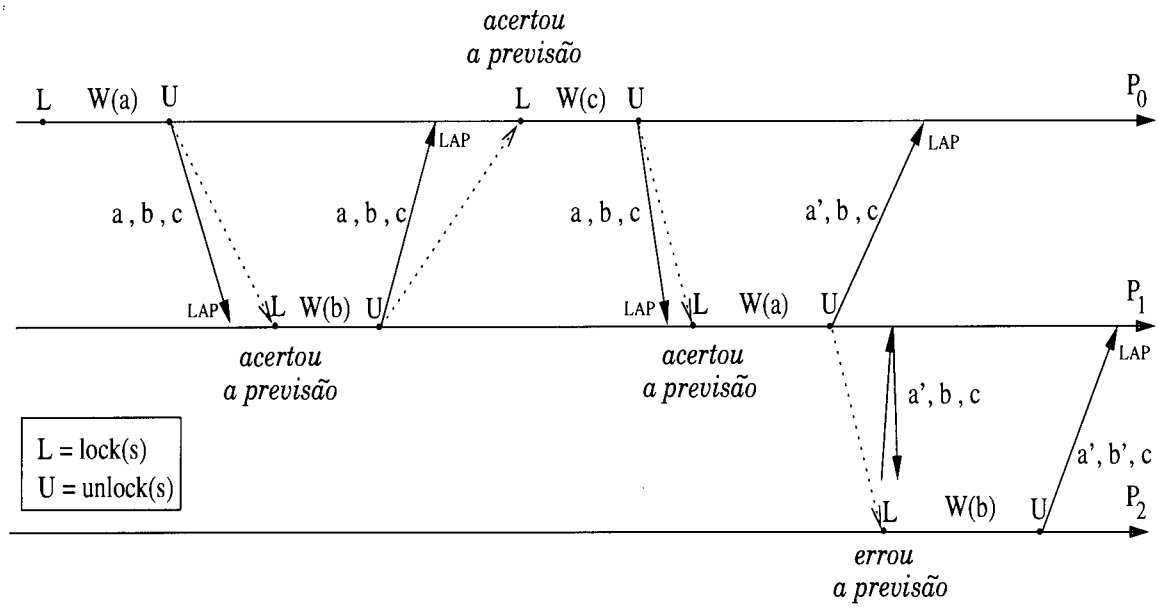

Figura 7.1: O uso da técnica de previsão de afinidade local no protocolo de coerência de AEC-bind.

Como o protocolo de AEC-bind não gera ou aplica *diffs*, durante o tempo de espera pela sincronização nenhuma operação de coerência é escondida.

7.3.1 Os Algoritmos de Lock/Unlock

Há um processador gerente para cada variável de sincronização s, responsável por implementar LAP e formar os conjuntos $U_p(s)$ para cada processador p. A determinação do gerente de cada variável de sincronização é realizada da mesma forma que em AEC. A cada operação de *lock* numa variável de sincronização s, os seguintes passos (ilustrados na figura 7.2) são realizados pelo processador p:

- envia mensagem a G_s requisitando o acesso à seção crítica protegida por s;
- ao receber a resposta do gerente, o processador recebe também a identificação do último dono de s, l; o seu conjunto de atualização, $U_p(s)$; e a indicação se $p \in U_l(s)$;
- se $p \notin U_l(s)$ (de acordo com as informações recebidas na barreira), então o processador requisita o objeto O_s a l.

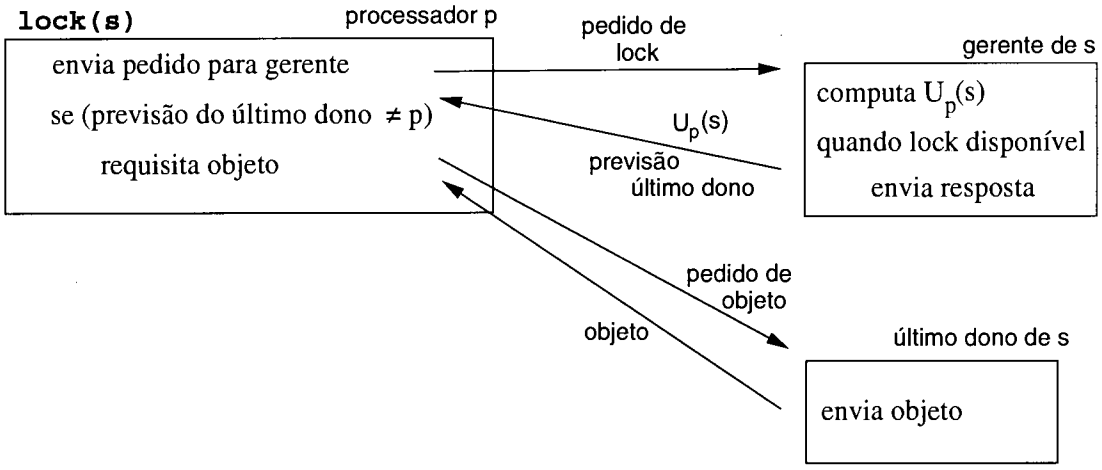

Figura 7.2: Operação lock no protocolo de coerência de AEC-bind.

As operações realizadas no processador p durante a execução de uma operação *unlock* em uma variável de sincronização s, estão ilustrados na figura 7.3 e descritos abaixo:

- envia mensagem para G_s, informando que a seção crítica está livre;
- envia para os processadores de $U_p(s)$ o objeto O_s.

7.3.2 Os Algoritmos de Lock/Unlock-Reader e Lock/Unlock-Alone

Já nas operações de *lock-reader* e *lock-alone* numa variável de sincronização s, o único passo realizado pelo processador p está ilustrado na figura 7.4 e descrito abaixo:

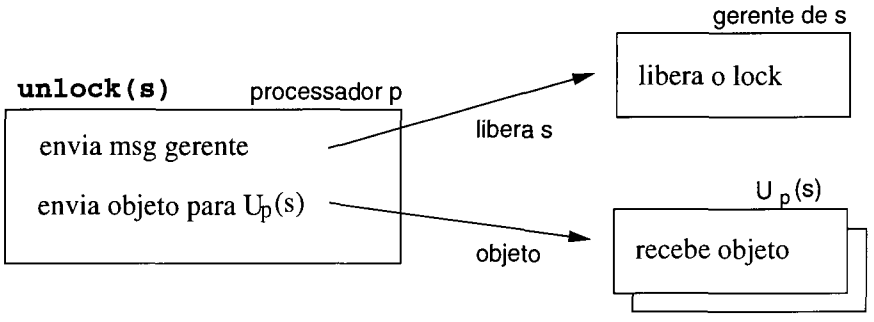

Figura 7.3: Operação *unlock* no protocolo de coerência de AEC-bind.

- se $p \notin U_l(s)$ (de acordo com as informações recebidas na barreira), então o processador requisita o objeto O_s a l.

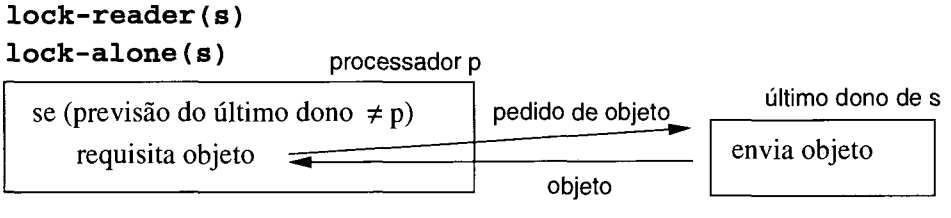

Figura 7.4: Operação *lock-reader* ou *lock-alone* no protocolo de coerência de AEC-bind.

As operações *unlock-reader* ou *unlock-alone* não realizam nenhuma função; elas podem ser utilizadas opcionalmente pelo programador para efeito de delimitação da seção crítica.

Em AEC-bind as operações de *lock*, *lock-reader* e *lock-alone* em uma mesma seção crítica s também são totalmente ordenadas por um contador, c_s. Porém, o uso desse contador é muito mais restrito em AEC-bind. Ele é utilizado apenas na barreira, conforme veremos adiante, para determinar qual processador foi o último dono de cada seção crítica s.

7.4 Sincronização Global e a Utilização de LAP

A barreira de AEC-bind funciona de modo idêntico a de AEC-light, visto que em ambos os sistemas a barreira não requer operação de coerência de dados. Tal qual em AEC-light, o protocolo de AEC-bind aproveita a troca de mensagens necessária

para a sincronização, para que cada processador conheça o último dono de cada seção crítica e o valor de c_s. Através dessas informações, numa operação de *lock-reader* ou *lock-alone* em s, o processador sabe se já recebeu o objeto do último dono de s ou não. Se ele já recebeu, então não tem que requisitá-lo. No caso de não ter recebido, ele tem que requisitá-lo de qualquer forma, mas não precisa requisitar ao gerente de s a identificação do último dono de s.

7.5 Interface de Programação

O sistema AEC-bind utiliza interface de programação idêntica à utilizada pelo sistema AEC-light, com a inclusão da primitiva `Aec_lock_bind`, para realizar a ligação da variável de sincronização com o objeto compartilhado. A primitiva `Aec_lock_bind` recebe como parâmetros o número da variável de sincronização, o endereço inicial e o tamanho (em bytes) do objeto. Quando um objeto consiste de n áreas descontíguas da memória, então a primitiva `Aec_lock_bind` deve ser chamada n vezes, uma para cada parte contígua do objeto.

Na figura 7.5 mostramos o mesmo programa exemplo utilizado na apresentação de AEC e AEC-light, sob a interface de programação de AEC-bind. Como AEC-bind possui interface muito parecida com a de AEC-light, a única diferença do programa apresentado nessa figura para o programa apresentado na figura 6.5 está na inclusão de uma chamada a `Aec_bind` para ligar o vetor compartilhado à variável de sincronização s.

```
#include "Aec.h"
int  *shared_array, arrayDim = 10 ;
main(int argc, char **argv)
{
    int start, end, i, p, c, b, s;

    Aec_startup(argc, argv);
    a = 1; s = 2;
    /* Aloca o vetor compartilhado e distribui enderecos */
    if (Aec_proc_id == 0) {
        shared_array = (int *) Aec_malloc(arrayDim * sizeof(int));
        Aec_distribute(&array, sizeof(array)); }

    /* Estabelece associacao dado-variavel de sincronizacao */
    Aec_bind(s, shared_array, arrayDim * sizeof(int));

    Aec_barrier(b);

    /* Primeira fase - inicializa vetor compartilhado */
    if (Aec_proc_id == 0) {
        Aec_lock_alone(s);
        for (i = 0; i < arrayDim; i++)
             shared_array[i] = 0;
        Aec_unlock_alone(s); }
    Aec_barrier(b);

    /* Segunda fase - escreve pid no vetor */
    Aec_lock(s);
    for (i = 0; i < arrayDim; i++)
        shared_array[i] += Aec_proc_id;
    Aec_unlock(s);
    Aec_barrier(b);

    /* Terceira fase - imprime conteudo do vetor */
    Aec_lock_reader(s);
    for (i = 0; i < arrayDim; i++)
        printf(" %d", shared_array[i]);
    Aec_exit(0);
}
```

Figura 7.5: Exemplo simples da utilização da interface de programação de AEC-bind.

Capítulo 8

Metodologia

8.1 Ambiente

Para avaliar LAP e o desempenho da implementação de AEC, AEC-light e AEC-bind utilizamos um multicomputador IBM SP2 do Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro, com oito nós de processamento. Cada nó é composto de um processador Power2 de 66MHz com 128M de memória. Os nós são conectados por um *switch* de alto desempenho com banda passante nominal de 40MB/s. Todos nossos experimentos foram realizados com a máquina totalmente dedicada. A instrumentação necessária para coletar os tempos medidos afeta o desempenho dos sistemas de forma negligível.

8.2 Aplicações

Utilizamos seis aplicações em nossos experimentos: IS, Water, FFT, SOR, MigDepth e MigFreq. IS, Water, FFT e SOR fazem parte do pacote de distribuição de TreadMarks. MigFreq e MigDepth são duas versões de migração sísmica 2D da Petrobrás [29]. A tabela 8.1 mostra as entradas utilizadas em cada aplicação.

A seguir faremos um breve resumo de cada aplicação e mostraremos as alterações necessárias para que elas possam atender às restrições dos modelos de programação de AEC-light e AEC-bind. Todas as seis aplicações executam corretamente, sem nenhuma alteração adicional, no modelo de programação de AEC. Não estamos

Aplicação	Tamanho do Problema
IS	$N = 2^{23}$, $B_{max} = 2^{15}$, 10 iterações
MigDepth	128×128
MigFreq	128×128
FFT	$64 \times 64 \times 16$, 100 iterações
SOR	2000×1000, 100 iterações
Water	512 moléculas, 10 passos

Tabela 8.1: Tamanho das entradas utilizadas em cada aplicação.

considerando nessa tese nenhum tipo de reestruturação das aplicações, as alterações realizadas incluem somente inclusão de operações *lock-reader*, *lock/unlock-alone* e *bind*.

8.2.1 IS

IS classifica um vetor de N inteiros, utilizando chaves no intervalo $[0, B_{max}]$, através da técnica *bucket sort*. As chaves são igualmente divididas pelos processadores e cada iteração consiste de três fases. Na primeira fase, o vetor global é inicializado pelo processador 0. Na segunda fase processadores armazenam os resultado locais no vetor global. Este armazenamento é feito dentro de uma única seção crítica. Na última fase todos os processadores lêem o vetor global para classificar suas chaves locais. A entrada utilizada foi $N = 2^{23}$, $B_{max} = 2^{15}$ e 10 iterações.

Para atender às restrições impostas pelo modelo de AEC-light, IS precisou de algumas alterações. Na primeira fase em que o vetor global é inicializado por um único processador, foi necessário inserir primitivas *lock-alone/unlock-alone*. A segunda fase permanece idêntica e no início da terceira fase foi necessária a inserção de um *lock-reader* na variável que protege todo o vetor global. Para atender ao modelo de programação de AEC-bind, a aplicação foi alterada para incluir a associação explícita do vetor global com a única variável de sincronização da aplicação.

8.2.2 MigDepth

MigDepth faz migração 2D pós-estaqueamento usando o método $\omega - x$. A paralelização é obtida através de particionamento por profundidade. Cada processador extrapola a seção sísmica para um determinado conjunto de profundidades. Como a migração de cada profundidade depende da anterior, a computação é feita em modo *pipeline*. Este *pipeline* é controlado com uso de *locks*. Toda a computação paralela é realizada em uma única fase. A entrada utilizada foi de 128×128.

Para atender às restrições impostas pelo modelo de AEC-light, MigDepth precisou de uma única alteração, a inclusão de um par de primitivas *lock-alone/unlock-alone* na fase de inicialização. Isto porque em MigDepth, durante a execução paralela, como há *locks* para proteger o *pipeline*, todos os acessos aos dados compartilhados são executados dentro de seções críticas. Para o modelo de programação de AEC-bind, a cada variável de sincronização é associado explicitamente um bloco de freqüências da matriz que representa a seção sísmica. O fator de bloco utilizado em MigDepth para a entrada de 128×128 é de 8.

8.2.3 MigFreq

MigFreq resolve o mesmo problema que MigDepth usando particionamento por frequências. Cada processador migra um bloco de frequências sem a necessidade de comunicação. Em MigFreq toda a computação paralela é realizada em uma única fase. A comunicação entre os processadores é realizada somente no final da fase, quando todos os processadores devem acumular as frequências migradas localmente à seção sísmica final. A acumulação das frequências migradas é realizada dentro de uma seção crítica, sendo que em cada seção, toda a seção sísmica é reescrita.

Tal qual MigDepth, MigFreq requer alterações mínimas para executar corretamente no modelo de AEC-light. Essas alterações são a inclusão de primitivas *lock-alone/unlock-alone* na fase de inicialização e a inclusão de primitivas lock/unlock

para proteger o cálculo da seção seção sísmica (foi incluído um *lock* diferente para cada processador). A associação de variável de sincronização com dado compartilhado, necessária ao modelo de AEC-bind é mais simples, como há um *lock* diferente para cada processador, esse *lock* é explicitamente associado à seção sísmica pela qual o processador é responsável.

8.2.4 FFT

FFT resolve um conjunto de equações diferenciais parciais usando FFTs e FFTs inversas tridimensionais. A entrada é de uma matriz A de complexos com dimensão $n_1 \times n_2 \times n_3$. Os dados são distribuídos de modo que para qualquer i, todos os elementos de $A[i,*,*]$ $(0 \leq i < n_1)$ são atribuídos ao mesmo processador. Cada iteração é composta de duas fases. Na primeira fase, uma série de FFTs-1D são realizadas na matriz de entrada. Primeiro, uma FFT de n_3 pontos é realizada em cada matriz $n_1 \times n_2$. Em seguida uma FFT de n_2 pontos é realizada em cada matriz $n_1 \times n_3$. Cada processador computa a sua porção da matriz A, sem necessidade de comunicação. Na segunda e última fase de cada iteração, a matriz resultante é transposta em uma outra matriz $n_2 \times n_3 \times n_1$, para em seguida ser realizada uma FFT de n_1 pontos em cada matriz $n_2 \times n_3$.

Para atender às restrições impostas pelo modelo de AEC-light, FFT precisou de alterações. Todos os acessos às matrizes compartilhadas são realizados sob *locks*. Para isso, foi criada uma matriz de *locks* L:$nprocs \times nprocs$. A matriz A é dividida na primeira dimensão em ($nprocs*nprocs$) subcubos distintos e cada *lock* L_{ij} protege um subcubo S_k, onde $k = i * nprocs + j$.

Na primeira fase, as FFTs são realizadas sob operações de *lock-alone*. Cada processador adquire uma linha da matriz de *locks* L. Na fase seguinte, a transposição também é realizada sob operações *lock-alone*, mas os *locks* são adquiridos de forma diferente. Cada processador adquire uma coluna da matriz de *locks* L, visto que

vai operar com porções distintas da matriz. Para atender ao modelo de AEC-bind, cada variável de *lock* é associada explicitamente ao subcubo correspondente. Esses subcubos, entretanto, não estão alocados contiguamente na memória. Assim são necessárias várias operações de bind para cada variável de *lock*.

8.2.5 SOR

SOR resolve equações diferenciais parciais usando uma estratégia do tipo "vermelho-preto" para realizar relaxações sucessivas. A entrada é de uma malha bidimensional, divida em duas matrizes diferentes: a "matriz preta" e a "matriz vermelha". Cada iteração é composta de duas fases. A primeira fase computa cada valor da matriz preta baseado nos quatro valores da matriz vermelha que estão em torno dele. A fase seguinte computa os valores da matriz preta a partir de valores da matriz vermelha. Cada processador recebe um bloco contínuo de linhas das matrizes preta e vermelha. Portanto, os únicos valores que são realmente compartilhados pelos processadores são os valores das bordas de cada bloco.

Para atender às restrições impostas pelo modelo de AEC-light, foram inseridas primitivas de *lock-alone* e *lock-reader* nas duas fases de SOR. Para cada borda de cada matriz há uma variável de sincronização diferente. Na fase de computação da matriz preta, foram inseridas uma primitiva *lock-reader* para leitura de cada borda da matriz vermelha e um par de primitivas *lock-alone/unlock-alone* para a computação de cada a borda da matriz preta. Na fase de computação da matriz vermelha, foram inseridas uma primitiva *lock-reader* para leitura de cada borda da matriz preta e um par de primitivas *lock-alone/unlock-alone* para a computação de cada a borda da matriz vermelha. A associação explícita necessária ao modelo de AEC-bind é trivial; a cada variável de sincronização é associada uma borda de uma das matrizes.

8.2.6 Water

Water calcula as forças e potenciais de um sistema de moléculas de água no estado líquido. As moléculas são distribuídas igualmente entre os processadores. Em cada iteração várias fases são computadas. Numa fase inicial, o processador 0 inicializa algumas somas globais. Na fase seguinte, chamada fase intramolécula, cada processador computa valores de deslocamento para o seu conjunto de moléculas. A terceira fase, chamada de intermolécula, utiliza os valores de deslocamento gerados na fase anterior para computar as forças entre as moléculas. Para um total de m moléculas, cada processador computa a interação entre todas as moléculas em sua partição e outras $m/2$ moléculas. Nessa fase, a atualização das forças das moléculas é realizada dentro de seção crítica, sendo que há uma seção crítica por molécula. A quarta fase utiliza valores das forças calculados na fase anterior e calcula os valores corretos para as moléculas, condições de fronteira e a energia cinética do sistema. Nessa fase, cada processador realiza as computações no seu conjunto de moléculas. Utilizamos como entrada um conjunto de 512 moléculas com 10 iterações.

Para executar no modelo de programação de AEC-light, algumas alterações foram necessárias em Water. Na primeira fase, foram inseridas primitivas *lock-alone/unlock-alone* para inicialização das somas globais. Na fase intramolécula, foram inseridas um par de primitivas *lock-alone/unlock-alone* para cada molécula na atualização dos seus deslocamentos. Na fase intermolecula, foram inseridas primitivas *lock-reader* para leitura dos valores de deslocamento relativos a moleculas calculadas por outros processadores. Na quarta fase, foram inseridas primitivas *lock-alone/unlock-alone* para calcular os valores corretos para as moléculas e as condições de fronteira. Para atender ao modelo de AEC-bind, foi inserida a associação explícita entre cada molécula de água e a variável de *lock* que a protege.

8.2.7 Diversidade das Aplicações

As aplicações do nosso *workload* diferem bastante no tipo e na frequência das operações de sincronização e na granularidade de acesso aos dados compartilhados. A tabela 8.2 sumariza algumas dessas características. Na coluna `Granularidade` mostramos a granularidade de acesso aos dados compartilhados. Estamos considerando que a granularidade de acesso é fina, média ou grossa, de acordo com a porção da estrutura de dados central que é acessada pelo processador entre duas operações de sincronização. Apresentamos nessa coluna uma definição apenas qualitativa da granularidade de acesso, por exemplo, se a estrutura de dados central da aplicação é uma matriz, consideramos que o acesso entre duas operações de sincronização é realizado com granularidade grossa se toda a matriz é acessada, com granularidade média se um bloco de linhas (ou colunas) da matriz é acessado, ou com granularidade fina se apenas um elemento da matriz é acessado. Na coluna `Sincronização` mostramos o tipo dominante de sincronização utilizada no modelos de programação de AEC. Na coluna `Sinc/seg` mostramos a taxa de operações de sincronização executadas por segundo em cada uma das aplicações.

Aplicação	Granularidade	Sincronização	Sinc/seg
IS	grossa	*locks* e barreiras	228
MigDepth	média	*locks* e barreiras	52
MigFreq	grossa	*locks* e barreiras	9
FFT	grossa	barreiras	13
SOR	média	barreiras	51
Water	fina	*locks* e barreiras	661

Tabela 8.2: Características de sincronização e granularidade de acesso cada aplicação.

SOR é uma aplicação simples e regular com sincronização baseada em barreiras. Nessa aplicação, apesar das matrizes preta e vermelha serem declaradas como compartilhadas, somente as bordas das matrizes são realmente compartilhadas, por isso

consideramos que, em SOR, a granularidade de acesso aos dados "realmente compartilhados" é fina. FFT também é uma aplicação regular baseada em barreiras, mas os acessos aos dados compartilhados ocorrem com granularidade grossa. Em FFT, depois da fase de transposição da matriz, os processadores operam em grandes seções completamente distintas da matriz. Como FFT e SOR têm sincronização baseada somente em barreiras, em cada fase da computação cada processador escreve sozinho no dado compartilhado, para que na fase seguinte outro(s) processdor(es) possa(m) ler os dados. Essa característica de FFT e SOR impõe um padrão de compartilhamento produtor-consumidor explorado pela técnica de afinidade global de LAP.

IS e Water usam sincronização por *locks* e barreiras. Entretanto, a sincronização em Water é muito mais frequente do que em IS e a principal diferença das duas aplicações está na granularidade de acesso aos dados compartilhados. Em IS a granularidade é grossa; as seções críticas são grandes (o vetor compartilhado inteiro é acessado dentro de uma seção crítica). Em Water, a granularidade de acesso é fina; cada seção crítica compreende o acesso a uma única molécula do sistema.

MigDepth e MigFreq apresentam sincronização apenas por *locks*, barreiras são utilizadas somente no início e no fim da computação. Em MigFreq um processador opera sobre o seu bloco de frequência, sem a necessidade de sincronização. A sincronização ocorre apenas no final, onde, em cada seção crítica, toda a seção sísmica é reescrita. MigDepth por sua vez, utiliza *locks* durante a migração e a cada seção crítica uma porção bem menor da matriz é alterada, essa porção compreende um bloco de freqüências da matriz.

De uma forma geral, as aplicações que utilizamos para avaliar nossos sistemas representam uma grande variedade de algoritmos, estilos de sincronização e granularidades de acessos. Essa diversidade garante que os resultados de nosso estudo são representativos para uma grande classe de programas.

Capítulo 9

Resultados

Nesse Capítulo apresentamos os resultados obtidos com a execução das seis aplicações do nosso *workload*. Avaliamos inicialmente a precisão das previsões realizadas por LAP e em seguida avaliamos o desempenho dos três sistemas propostos nessa tese, comparando com o desempenho do sistema TreadMarks.

9.1 Avaliação das Previsões de LAP

Nesta seção apresentamos uma avaliação das previsões realizadas por LAP para as seis aplicações do nosso *workload* (uma avaliação das previsões LAP para um conjunto diferente de aplicações pode ser encontrada em [67]). Inicialmente avaliamos as técnicas fila de espera e afinidade local, utilizadas por LAP para o modelo de programação de AEC e em seguida avaliamos a técnica de afinidade global para aplicações no modelo de programação empregado em AEC-light e em AEC-bind.

9.1.1 Previsões no Modelo de Programação Convencional

Para avaliar a precisão das previsões realizadas pelas técnicas de fila de espera e afinidade local, utilizamos quatro aplicações do nosso *workload* executando no sistema AEC: Water, IS, MigDepth e MigFreq. As aplicações selecionadas são as que utilizam *locks* intensamente para a sincronização entre os processadores. Na tabela 9.1 temos quantificados, para cada uma das aplicações, o número de variáveis

de sincronização, o número total de eventos *lock/unlock* realizados e o número médio de eventos *lock/unlock* por variável de sincronização.

	Water	IS	MigDepth	MigFreq
# variáveis	515	1	8	1
# eventos	25760	80	1028	8
Eventos/var	50	80	128	8

Tabela 9.1: Características dos eventos de sincronização das aplicações.

Para cada variável de sincronização s, o processador gerente de s computa a taxa de acerto $H(s)$ da seguinte forma:

$$H(s) = \frac{\#\ vezes\ acquirer\ q\ estava\ em\ U_p(s)\ do\ releaser\ p}{\#\ locks\ executados\ na\ variavel\ s}$$

Antes de apresentarmos resultados das taxas de acerto obtidas por LAP, vamos determinar os valores ideais dos parâmetros z (número de elementos do conjunto de atualização) e T (valor mínimo de afinidade em relação ao total de passagens de *lock*). Tais parâmetros têm impacto tanto nas taxas de acerto das previsões, como na "seletividade" das atualizações realizadas por LAP. Estamos interessados, portanto, em dimensionar z e T de modo a realizar as atualizações da forma mais seletiva possível (gerando menor *overhead*), mas mantendo uma taxa de acerto alta.

O gráfico da figura 9.1 mostra a variação da taxa de acerto quando o número de elementos do conjunto de atualização é aumentado de 0 até 3 processadores. Como algumas aplicações possuem uma grande quantidade de variáveis de sincronização, apresentamos somente os resultados para as variáveis de sincronização com uma quantidade razoável de eventos de *lock*. Além disso, agrupamos algumas variáveis de sincronização que estão logicamente relacionadas e, assim, apresentamos no gráfico uma única taxa de acerto por aplicação. Em Water, são agrupadas as variáveis que protegem as moléculas de água. Em MigDepth são agrupadas as variáveis de controlam o acesso ao *pipeline*. Conforme podemos observar no gráfico,

para essas aplicações, o aumento no tamanho do conjunto de atualização tem efeito muito pequeno nas taxas de acerto obtidas. Para as aplicações MigFreq e IS, onde a técnica fila de espera é a mais utilizada, esse é um resultado previsível, dado que quando há um processador na fila, o conjunto de atualização é formado por um único processador. A explicação para a pequena variação obtida em MigDepth e Water, porém, é dependente de características da aplicação, dado que nessas aplicações a técnica de previsão mais utilizada é a de afinidade local. Em MigDepth, o processamento é feito como um pipeline, onde a migração de cada profundidade depende da profundidade anterior. A ordem com que os processadores realizam a migração é mantida para todas as profundidades e, assim, o valor $z = 1$ é suficiente para obtenção de altas taxas de acerto. Em Water, cada processador começa computando as suas moléculas e só depois é que computa a dos vizinhos. Como cada molécula é acessada apenas por quatro processadores, ocorrem pequenas variações na ordem que esses quatro processadores acessam as moléculas. A variação da taxa de acerto de Water em relação ao valor de z seria maior para Water se considerássemos um número maior de processadores. Concluímos que para nossos experimentos com esse conjunto de aplicações, o valor de z ideal para uma alta taxa de acerto com atualização bastante seletiva é de 1 processador.

O limite T é utilizado para que a técnica de afinidade local não considere o envio de atualizações para processadores cuja afinidade com o processador que está liberando o *lock* seja muito pequena em relação ao total das passagens de *lock*. O gráfico da figura 9.2 mostra a variação da taxa de acerto em relação ao valor de T, para as aplicações MigDepth e Water (as únicas onde a técnica afinidade local é efetiva) executando em 8 processadores. Variamos T de 0 a 100% e obtivemos taxas de acerto muito baixas para $T > 20\%$, o que significa que as afinidades dos processadores representam em média de 10 a 20% do total de passagens de *lock*. Esses resultados mostram que para um processador p, as afinidades de p em relação

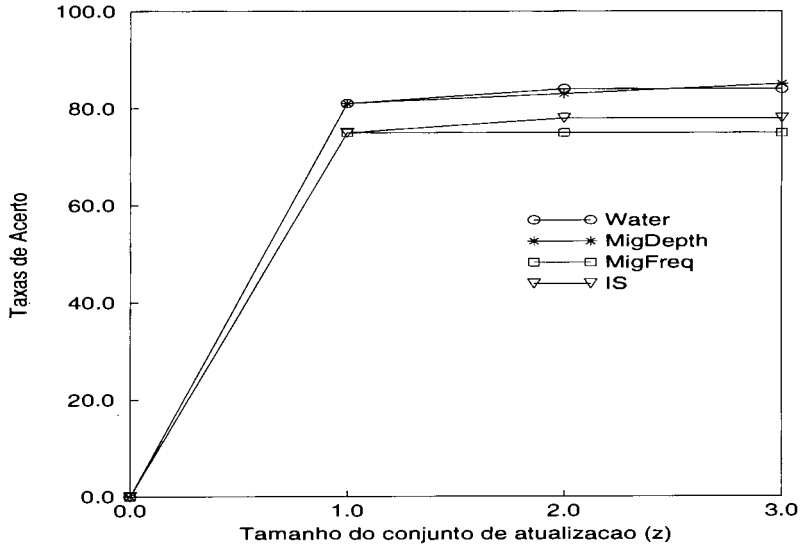

Figura 9.1: Taxa de acerto obtida com a variação de z.

aos outros processadores que compartilham determinado *lock* com p são bem distribuídas em relação ao total de eventos desse *lock*. Isso significa que, para alcançar altas taxas de acerto, T deveria receber um valor baixo. Note que T baixo não causa maiores problemas, uma vez que z já é o mínimo possível para as nossas aplicações. Segundo a figura, podemos usar T igual a 10%, por exemplo.

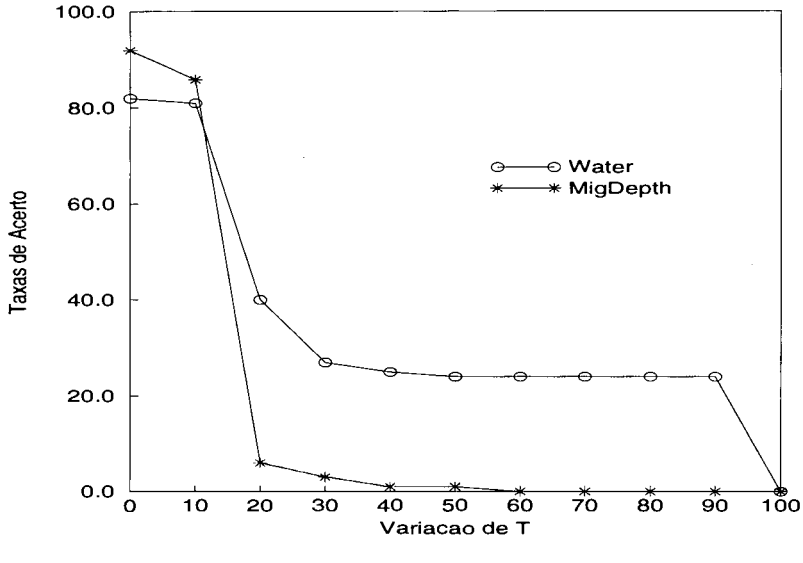

Figura 9.2: Taxa de acerto obtida com a variação de T.

Dado que já dimensionamos $z = 1$ e $T = 10\%$, podemos avaliar as taxas de

acerto de LAP para as aplicações. Na tabela 9.2 avaliamos a contribuição das duas técnicas (fila de espera e afinidade local) na taxa de acerto de LAP para cada uma das aplicações testadas. A tabela mostra, para cada variável de sincronização da aplicação, o número de eventos de *lock* na variável, a porcentagem desses eventos no número total de *locks* da aplicação e a taxa de acerto de LAP para cada uma das técnicas. Na coluna `LAP` apresentamos $H(s)$; na coluna `Fila` apresentamos a contribuição da técnica fila de espera em $H(s)$; e na coluna `Afinidade` apresentamos a contribuição da técnica de afinidade local em $H(s)$.

Aplicação	# var	# de eventos *lock*	% do total de eventos	taxas de acerto		
				LAP	Fila	Afinidade
Water	4-515	25280	98.1%	81.3%	0.0%	81.3%
IS	1	80	100.0%	82.0%	76.0%	6.0%
MigDepth	1-8	1028	100.0%	90.0%	7.3%	82.7%
MigFreq	1	8	100.0%	75.0%	75.0%	0.0%

Tabela 9.2: Taxas de acerto para $z = 1$ e $T = 10\%$.

Para as variáveis que aparecem agrupadas na tabela 9.2, a taxa de acerto do grupo corresponde à média, ponderada pela quantidade de eventos, das taxas de acerto de cada variável.

Conforme podemos observar na tabela, a taxa de acerto de LAP é alta variando de 75% a 90% para as variáveis de sincronização mais importantes do conjunto de aplicações. Para IS e MigFreq, a técnica fila de espera é a mais efetiva. Enquanto que, para Water e MigDepth, onde não há contenção pela seção crítica, a previsão é quase toda baseada na técnica de afinidade local.

Para avaliar a robustez de nossas previsões, implementamos LAP em outro sistema *software DSM*, TreadMarks. Obtivemos, para o mesmo conjunto de aplicações taxas de acerto bem próximas das obtidas para AEC. A variação obtida foi de no máximo 10% [67].

9.1.2 Previsões em Outros Modelos de Programação

Para avaliar a precisão das previsões realizadas pela técnica de afinidade global, utilizamos quatro aplicações do nosso *workload*, Water, IS, SOR e FFT, executando no sistema AEC-light[1]. As aplicações selecionadas são as que apresentam características produtor-consumidor entre diferentes fases da computação e portanto podem se beneficiar da técnica de afinidade global.

Na tabela 9.3 temos quantificados, para cada uma das aplicações, o número de variáveis de sincronização e o número total de pedidos de *lock* (de qualquer tipo) realizados logo após uma barreira. Esses pedidos de *lock* são os que podem se beneficiar das atualizações enviadas pela técnica de afinidade global.

	Water	IS	SOR	FFT
# variáveis	515	1	32	128
# eventos	46894	70	2772	11972

Tabela 9.3: Características dos eventos de sincronização tipo produtor-consumidor entre fases distintas.

O cálculo da taxa de acerto $HG(s)$ é um pouco diferente do cálculo de $H(s)$. Em $HG(s)$ não estamos interessados em todas as passagens de *locks*, mas somente nas passagens tipo produtor-consumidor entre fases distintas. São as passagens de *lock* em que um processador requisita o *lock* (de qualquer tipo) após a última liberação na fase anterior de um *lock* ou um *lock-alone*. Dessas passagens, $HG(s)$ mostra a porcentagem das vezes que a afinidade global acertou sua previsão. Para cada variável de sincronização s, o processador computa a taxa de acerto $HG(s)$ da seguinte forma:

$$HG(s) = \frac{\#\ vezes\ acquirer\ q\ estava\ em\ UG_p(s)\ do\ releaser\ p}{\#\ pedidos\ de\ lock\ na\ variável\ s\ depois\ de\ barreira}$$

[1] A avaliação das previsões foi realizada também para o sistema AEC-bind com resultados praticamente idênticos.

Avaliamos a técnica de afinidade global de LAP, considerando que o número máximo de processadores para o qual um processador pode enviar atualizações (w) é igual ao número de processadores do sistema. Estamos utilizando o maior valor possível para w, dado que essas atualizações são enviadas na barreira e, nesse caso, a maioria dos processadores envolvidos nas atualizações já está esperando pela sincronização. Na verdade, as atualizações não são efetivamente enviadas para todos os processadores. O limite mínimo de afinidade (L) impede que um processador p envie atualizações para processadores com afinidade muito baixa em relação ao total de passagens de *lock* do tipo produtor-consumidor entre fases distintas. O número médio de processadores para os quais LAP envia atualizações segundo a técnica de afinidade global, quando $L = 50\%$, por exemplo, é de 4 processadores para Water, 7 processadores para IS, 1 processador para SOR e 1 processador para FFT.

A figura 9.3 mostra a relação entre a variação de L com as taxas de acerto obtidas. Segundo as curvas descritas no gráfico, observamos que até $L = 50\%$ as taxas de acerto se mantêm altas, o que significa que as afinidades globais são superiores a 50% do total de passagens de *lock* tipo produtor-consumidor. Concluímos que 50% é um valor razoável para L, porque permite limitar o número de processadores para os quais são enviadas atualizações, mas mantém taxas de acerto altas.

Na tabela 9.4 mostramos as taxas de acerto $HG(s)$ para os grupos de variáveis de sincronização de cada aplicação, considerando $w = 8$ e $L = 50\%$.

Aplicação	# var	HG(s)
Water	4-515	80.0%
IS	1	35.0%
SOR	1-32	99.4%
FFT	1-128	77.1%

Tabela 9.4: Taxas de acerto da técnica afinidade global.

Conforme podemos observar na tabela, a taxa de acerto de LAP é alta para três

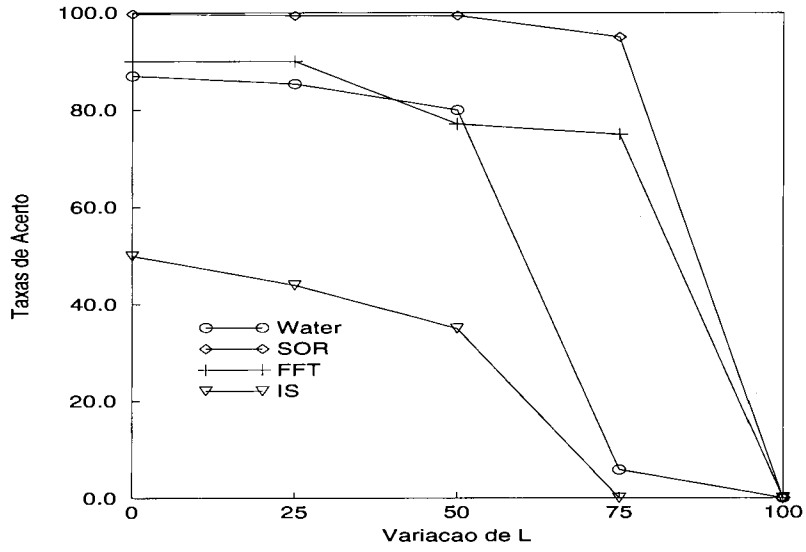

Figura 9.3: Taxa de acerto obtida com a variação de L.

aplicações, Water, SOR e FFT, e baixa para IS. As aplicações FFT e SOR apresentam altas taxas de acerto, pois possuem comportamento regular e padrão de acesso produtor-consumidor entre todas as suas fases. Water apresenta padrão produtor-consumidor nas transições das fases inter e intra-molécula. Como cada molécula é acessada por no máximo 4 processadores, o conjunto de consumidores de cada molécula é facilmente definido depois de algumas iterações. O produtor também é facilmente definido, cada processador é produtor do seu conjunto de moléculas. IS apresenta padrão de acesso produtor-consumidor entre duas fases, mas as previsões de LAP não são precisas. Da fase de inicialização para a fase de atualização do vetor global, o consumidor varia a cada iteração de IS, visto que representa o primeiro processador a adquirir o *lock* na fase de atualização. Da fase de atualização para a fase de leitura do vetor global, os consumidores são facilmente detectados (são todos os processadores), mas o produtor varia, visto que representa o último processador a adquirir o *lock* na fase anterior. Por esse motivo, as taxas de acerto de IS apresentadas no gráfico da figura 9.3 são bem inferiores às taxas apresentadas para as outras aplicações.

Para as previsões de LAP utilizando afinidade global, não temos resultados de taxas de acerto para outros sistemas *software DSM*, visto que essa técnica se aplica somente ao modelo de programação empregado por AEC-light e AEC-bind.

9.2 Avaliação dos Sistemas *Software DSM*

Tendo estudado o comportamento de LAP, apresentamos nesta seção uma avaliação do desempenho dos três sistemas DSM propostos. Estamos interessados em comparar seus desempenhos com relação ao sistema TreadMarks e em investigar os benefícos obtidos pelas técnicas propostas nessa tese. Inicialmente, apresentamos uma visão geral dos resultados obtidos. Em seguida, para isolar os benefícios particulares de desempenho obtidos por cada uma da aplicações, apresentamos uma análise aplicação por aplicação.

9.2.1 Visão Geral

Nesta seção apresentamos os resultados gerais de desempenho para as seis aplicações no nosso *workload*. Inicialmente, vamos apresentar o *speedup* obtido em cada um dos sistemas. Em seguida, vamos mostrar a comunicação gerada, apresentando o número de mensagens e a quantidade de bytes trocados. Por fim, vamos avaliar a contribuição da técnica LAP nos *speedups* obtidos.

Speedups

O gráfico da figura 9.4 mostra os *speedups* das seis aplicações executando nos sistemas TreadMarks, AEC, AEC-light e AEC-bind. Todos os *speedups* são calculados em relação à versão paralela da aplicação no modelo de programação tradicional (TreadMarks) executando em um processador. Além disso nossas medidas não consideram a fase de inicialização das aplicações. Tanto nesses resultados quanto em todos os outros apresentados à frente, os valores que assumimos para z, T e L foram

1, 10% e 50%, respectivamente.

A figura mostra comportamentos diferentes para as seis aplicações. Para as aplicações IS, MigDepth e MigFreq, os *speedups* dos três sistemas propostos apresentam ganhos em relação a TreadMarks. Esses ganhos são de: 38% para AEC, 60% para AEC-light e 82% para AEC-bind em IS; 143% para AEC, 143% para AEC-light e de 200% para AEC-bind em MigDepth; e 18% para AEC, 32% para AEC-light e de 62% para AEC-bind em MigFreq. Em FFT, TreadMarks e AEC apresentam o mesmo *speedup*, AEC-bind apresenta ganhos de 158% em relação ao *speedup* de TreadMarks, mas AEC-light apresenta *speedup* três vezes inferior a TreadMarks. Em SOR, AEC apresenta *speedup* 7% inferior ao speedup de TreadMarks, e AEC-light e AEC-bind apresentam ganhos praticamente idênticos de 20% em relação ao *speedup* de TreadMarks. A aplicação Water ilustra um exemplo em que os sistemas AEC, AEC-light e AEC-bind não têm bom desempenho. Em relação a TreadMarks, as perdas são de 12% para AEC; 31% para AEC-light e 15% para AEC-bind.

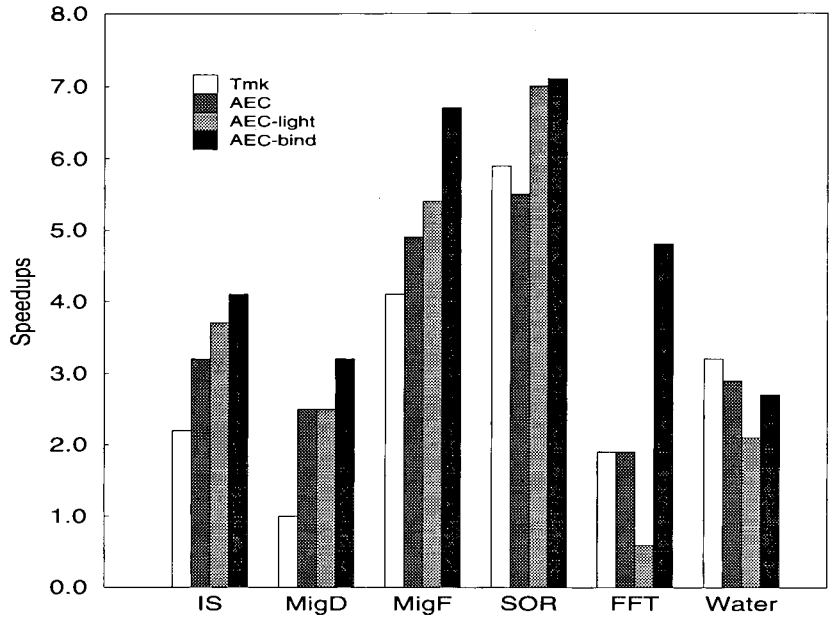

Figura 9.4: Speedups obtidos para TreadMarks, AEC, AEC-light e AEC-bind.

De uma forma geral, o gráfico da figura 9.4 apresenta resultados bastante inte-

ressantes. Uma comparação entre TreadMarks e AEC mostra que o nosso sistema tem desempenho bem melhor para 3 aplicações, enquanto que, para as outras 3 aplicações, AEC tem desempenho igual ou um pouco inferior a TreadMarks. Essas diferenças são resultado apenas das características dos dois protocolos, já que, para as nossas aplicações, TreadMarks e AEC levam ao mesmo modelo de programação.

Em termos do impacto de diferentes modelos de programação no desempenho das aplicações, observamos que o aumento da complexidade do modelo não necessariamente leva a ganhos de desempenho, o que é contrário à expectativa dominante na comunidade científica. Uma comparação entre os resultados de AEC e AEC-light confirma essa observação. Para uma única aplicação, o ganho de AEC-light é considerável, enquanto que para duas outras aplicações ocorrem perdas significativas de desempenho. Somente comparando AEC-light com AEC-bind é que observamos ganhos reais de desempenho provenientes da diferença dos modelos de programação. AEC-bind obtém melhor desempenho que AEC-light para todas as nossas aplicações, sendo que, na sua maioria, as diferenças de desempenho são substanciais. Comparando os nossos extremos em complexidade do modelo de programação, AEC e AEC-bind, vemos que, como esperado, o modelo mais complexo permite melhor desempenho.

Comunicação

A comunicação entre processos é um dos aspectos da execução de sistemas *software DSM* em redes de *workstations* que tem maior influência nos *speedups* alcançados por estes sistemas. Nos gráficos das figura 9.5 e 9.6 apresentamos respectivamente o número de mensagens e a quantidade de bytes transmitidos para as seis aplicações executando nos sistemas TreadMarks, AEC, AEC-light e AEC-bind. As barras estão normalizadas para os resultados de TreadMarks.

Segundo o gráfico da figura 9.5, os três sistemas propostos apresentam redução na

quantidade de mensagens transferidas em relação a TreadMarks, exceto para Water. Para AEC, a redução é obtida pela eliminação de falhas de acesso. Essa eliminação ocorre devido a dois fatores principais: o uso de LAP (evita falhas dentro de seções críticas) e o recebimento numa única mensagem de todos os *diffs* associados ao mesmo *lock* (*diffs* para diferentes páginas são agrupados numa mesma mensagem).

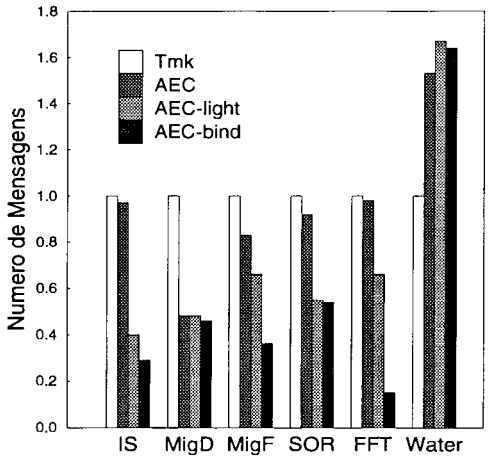

Figura 9.5: Número de mensagens para TreadMarks, AEC, AEC-light e AEC-bind.

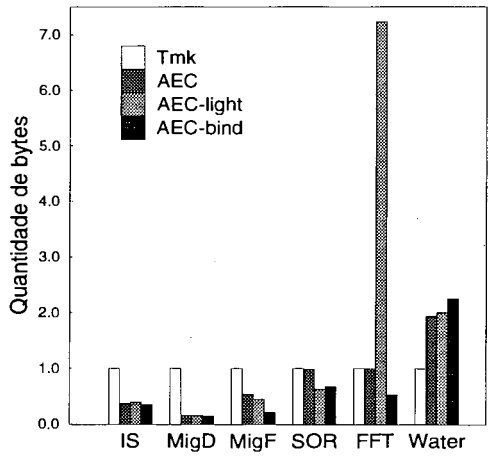

Figura 9.6: Quantidade de bytes transferidos para TreadMarks, AEC, AEC-light e AEC-bind.

Para AEC-light e AEC-bind, o uso de *locks* para proteger os acessos aos dados permite aos sistemas evitar um número ainda maior de falhas de acesso. As falhas de acesso geradas fora de seções críticas em AEC e TreadMarks são combinadas em uma única requisição de *diffs* (ou objeto) na operação de *lock* em AEC-light e AEC-bind.

O aumento no número de mensagens transmitidas para Water se deve a uma característica especial da aplicação, a qual analisaremos detalhadamente mais tarde, em que a requisição de dados feita a cada *lock* por AEC, AEC-light e AEC-bind leva a um número maior de falhas de acesso do que no sistema TreadMarks.

Conforme podemos observar pelo gráfico da figura 9.6, exceto para FFT e Water, AEC, AEC-light e AEC-bind conseguem reduzir também a quantidade de bytes trocados pelos processadores. Essa redução se deve principalmente à redução na

quantidade de falhas de acesso e ao armazenamento combinado de *diffs*. FFT tem desempenho desastroso em AEC-light porque a associação dinâmica de dado compartilhado com variável de sincronização é tal que toda a matriz de entrada é associada a todos os *locks*, conforme explicaremos mais tarde. Portanto, as requisições de dados nos pedidos de *lock* e nas falhas de acesso envolvem enorme quantidade de bytes trocados. Em Water, o aumento na quantidade de bytes trocados se deve também ao maior número de falhas de acesso geradas por AEC, AEC-light e AEC-bind.

Contribuição de LAP

Além da comunicação entre processadores, a técnica LAP também tem influência nos *speedups* alcançados pelos sistemas que propomos nesta tese. Nos gráficos das figuras 9.7, 9.8 e 9.9 mostramos, para AEC, AEC-light e AEC-bind respectivamente, a contribuição de LAP nos *speedups* apresentados anteriormente.

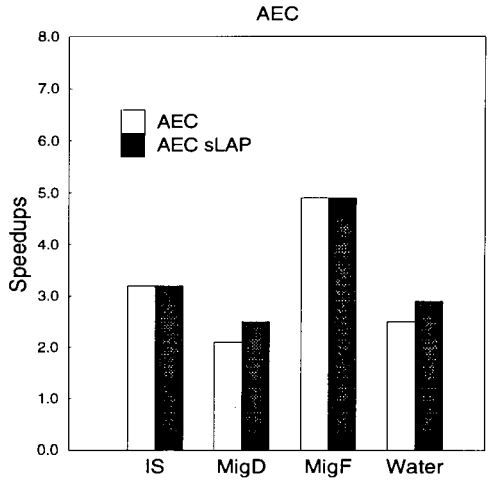

Figura 9.7: Speedups obtidos para AEC e AEC_{sLAP}.

O sistema AEC é comparado ao sistema AEC_{sLAP}, que é idêntico a AEC, mas toda vez que o processador entra numa seção crítica, ele deve gerar uma falha de acesso para buscar os *diffs* associados ao *lock* do último dono da seção crítica. O sistema AEC-light é comparado a três sistemas distintos AEC-$light_{sLAPl}$, AEC-

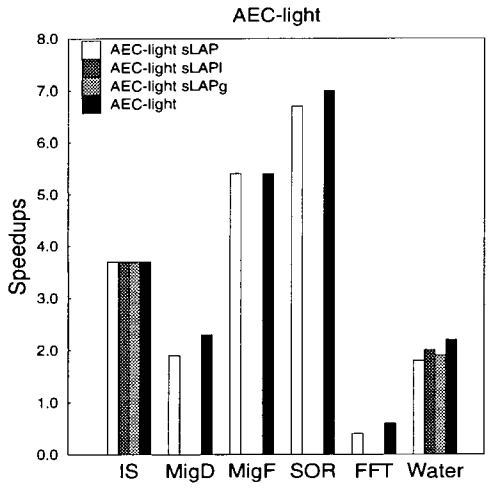

Figura 9.8: Speedups obtidos para AEC-light, AEC-light$_{sLAPl}$, AEC-light$_{sLAPg}$ e AEC-light$_{sLAP}$.

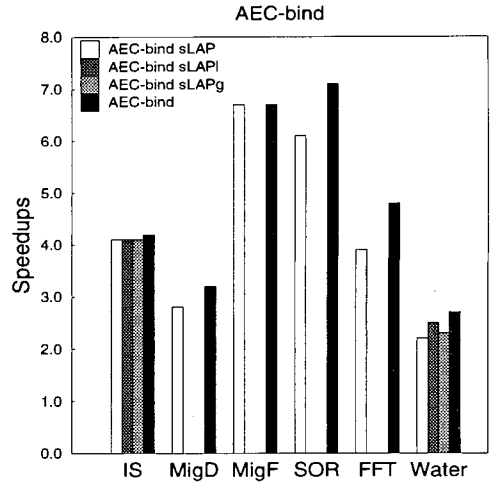

Figura 9.9: Speedups obtidos para AEC-bind, AEC-bind$_{sLAPl}$, AEC-bind$_{sLAPg}$ e AEC-bind$_{sLAP}$.

light$_{sLAPg}$ e AEC-light$_{sLAP}$. No sistema AEC-light$_{sLAPl}$ somente as técnicas de previsão local (fila de espera e afinidade local) são desligadas; um processador ao entrar numa seção crítica gera uma falha de acesso para buscar os dados do último dono da seção. No sistema AEC-light$_{sLAPg}$ são mantidas as técnicas de previsão local, mas é desligada a técnica de previsão global. Assim, num pedido de *lock* posterior a uma barreira, o processador sempre deve buscar os *diffs* do último dono do *lock* na fase anterior. No sistema AEC-light$_{sLAP}$ tanto as previsões locais como as previsões globais são desligadas. O sistema AEC-bind é comparado a três sistemas distintos AEC-bind$_{sLAPl}$, AEC-bind$_{sLAPg}$ e AEC-bind$_{sLAP}$. Como em AEC-bind não há falhas de acesso, a ausência de LAP requer que na operação de *lock* o objeto seja buscado do último dono da seção crítica.

No gráfico da figura 9.7 apresentamos somente os resultados do impacto de LAP em AEC para IS, MigDepth, MigFreq e Water. FFT e SOR no modelo de programação de AEC não empregam LAP, porque não possuem *locks*. Conforme podemos observar no gráfico, LAP apresenta aumento no *speedup* apenas para MigDepth e Water. Para essas aplicações, os ganhos no *speedup* são respectivamente de 19% e 16%.

Os gráficos das figuras 9.8 e 9.9, onde estudamos o impacto de LAP em AEC-light e AEC-bind, não incluem algumas barras, já que as aplicações MigDepth e MigFreq não empregam previsão global e as aplicações FFT e SOR não empregam previsão local. AEC-light e AEC-bind apresentam resultados bastante parecidos no que diz respeito ao benefício de LAP nos *speedups* das aplicações. IS e MigFreq não se beneficiam da previsão local (conforme ocorreu em AEC) e, em IS, dadas as baixas taxas de acerto, a previsão global também não tem efeito no *speedup* obtido. MigDepth apresenta 21% de aumento no *speedup* de AEC-light e 14% de aumento no *speedup* de AEC-bind com o uso de LAP. FFT e SOR apresentam aumentos de *speedup* de 5% e 50%, respectivamente, para AEC-light e de 16% e 23%, respectivamente, para AEC-bind. Water é a única aplicação que se beneficia tanto da previsão local quanto da previsão global. Em Water, o uso de previsão global permite aumento de 15% no *speedup* de AEC-light e 17% no *speedup* de AEC-bind. Já o uso de previsão local permite aumento de 10% no *speedup* de AEC-light e 8% no *speedup* de AEC-bind. O emprego das duas técnicas juntas aumenta o *speedup* de Water em 22% para AEC-light e em 22% para AEC-bind.

Note que esses resultados isolam apenas uma parte da contribuição de LAP nos nossos sistemas. Não é possível isolar LAP completamente em AEC, AEC-light e AEC-bind porque quase todas as características dos três sistemas são direcionadas a facilitar o emprego de LAP. Assim, mesmo os sistemas sem LAP continuam apresentando o armazenamento combinado de *diffs* e a requisição de todos os *diffs* associados ao *lock* na primeira falha, que por si só permitem ganhos de desempenho. O armazenamento combinado de *diffs* permite ganhos quando o padrão de acesso ao dado é migratório. Nesse caso, mesmo não tendo recebido atualizações de LAP, numa falha de acesso, ao invés de receber um conjunto de n *diffs* gerados pelas n requisições de *lock* anteriores, o processador recebe um único *diff* referente à combinação dos n *diffs* anteriormente gerados. A requisição de todos os *diffs* associados

ao *lock* permite que, na primeira falha dentro de uma seção crítica, um processador receba *diffs* de todas as páginas associadas ao *lock* e mesmo sem ter recebido atualizações de LAP, o sistema está evitando falhas seguintes nas outras páginas. Quando implementamos os sistemas sem LAP, consideramos que se o sistema já tem a infra-estrutura para permitir ganhos de desempenho, ela deve ser mantida para fornecer uma comparação mais justa. Dessa forma, os ganhos de LAP apresentados não são tão proeminentes, quanto o seriam caso LAP fosse completamente eliminada dos sistemas.

As figuras 9.7, 9.8 e 9.9 confirmam que os ganhos significativos de *speedup* alcançados pelos sistemas em relação a TreadMarks não são devidos unicamente a LAP, mas a características intrínsecas às aplicações e a técnicas empregadas nos sistemas para suportar LAP e para tolerar *overheads* de criação de *diffs*. A seguir faremos uma análise detalhada de cada aplicação para distinguir quais são essas características e como as outras técnicas utilizadas em AEC, AEC-light e AEC-bind permitem ganhos de desempenho.

9.2.2 MigDepth

A figura 9.10 mostra o tempo de execução paralela de MigDepth executando sob os sistemas TreadMarks, AEC, AEC-light e AEC-bind. Os tempos estão normalizados para o tempo de execução de TreadMarks. Na figura, o tempo de execução é dividido em tempo de computação (`comp`) que inclui também tempo de falha na cache e na TLB; tempo de sincronização dividido em `lock` (tempo gasto com todas as primitivas de *lock*), `unlock` (o tempo gasto com todas as primitivas de *unlock*) e `barreira` (o tempo gasto com sincronização global); *overhead* de falha de acesso (`falhas`); e *overhead* de *ipc* (`ipc`) que representa o tempo que um processador gasta servindo pedidos vindos de outros processadores.

Conforme podemos observar na figura 9.10, AEC apresenta ganhos consideráveis

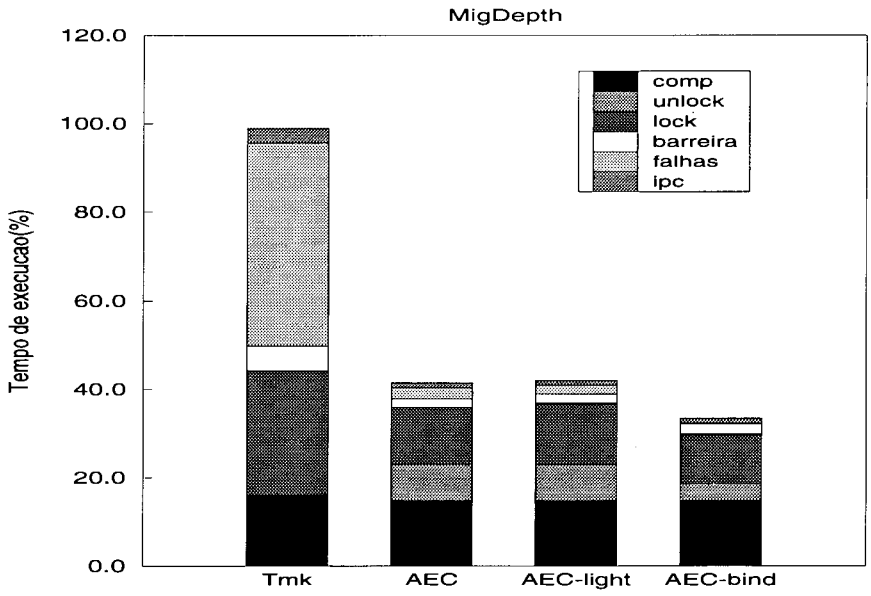

Figura 9.10: Tempo de execução de MigDepth em TreadMarks, AEC, AEC-light e AEC-bind.

em relação a TreadMarks; AEC-light obtém o mesmo tempo de execução que AEC e AEC-bind reduziu um pouco mais o tempo de execução em relação a AEC.

AEC × TreadMarks

AEC obtém redução de 58% no tempo de execução de MigDepth em relação a TreadMarks, como resultado de reduções nos *overheads* de *lock* e falhas de acesso. Mais especificamente, AEC obtém 95% de redução no *overhead* de falha de acesso e 54% de redução no *overhead* de *lock*. A seguir vamos analisar as reduções obtidas nos *overheads* de falhas de acesso e sincronização em mais detalhes.

Overhead de Falha de Acesso. Em MigDepth, todas as falhas de acesso ocorrem dentro de seções críticas. A tabela 9.5 mostra o número e o tempo gasto com falhas de acesso ocorridas dentro de seções críticas em MigDepth executando em TreadMarks, AEC e AEC_{sLAP}. A redução no *overhead* de falha de acesso em MigDepth é obtida unicamente pela eliminação de 95% das falhas de acesso ocorridas dentro de seções críticas, devido ao uso de LAP. Entretanto, retirando-se LAP de AEC, continuamos obtendo uma quantidade de falhas de acesso 4 vezes inferior a TreadMarks,

conforme podemos constatar na linha de AEC_{sLAP} na tabela 9.5. Como mencionado anteriormente, isto se explica pelo fato de que em AEC_{sLAP} quando um processador gera uma falha dentro de uma seção crítica, ele recebe de uma vez todos os *diffs* associados ao *lock*. Como em MigDepth, dentro de uma seção crítica, são acessadas em média 4 páginas diferentes, na primeira falha ocorrida, AEC_{sLAP} evita outras 3 falhas seguintes dentro da seção crítica.

Sistema	falhas dentro de sc	
	# de falhas	tempo(ms)
TreadMarks	5497	9240
AEC	281	402
AEC_{sLAP}	1105	2135

Tabela 9.5: Falhas de acesso dentro de seções em MigDepth, para TreadMarks, AEC e AEC_{sLAP}.

Overhead de Sincronização. Com relação ao *overhead* de *lock*, a redução de 57% é obtida porque o custo do *lock* é menor em AEC do que em TreadMarks, quando não há espera pela seção crítica. Em AEC, numa operação de *lock* em uma seção crítica disponível, um processador p troca um par de mensagens com o gerente do *lock*. Em TreadMarks, p deve enviar mensagem para o gerente, que envia para o último dono, que então envia a liberação de volta para p. Em MigDepth, como não há contenção pelos *locks*, o *overhead* de *lock* compreende somente o tempo dessas trocas de mensagens.

AEC-light × AEC

AEC e AEC-light apresentam praticamente o mesmo tempo de execução para MigDepth. Isso se deve a que a única alteração realizada em MigDepth para executar corretamente no modelo de programação de AEC-light ocorre na fase de inicialização, a qual não é computada no tempo de execução paralela.

AEC-bind × AEC-light

O uso da associação explícita de variável de sincronização com dado compartilhado e do objeto como granularidade de coerência permite a AEC-bind uma redução de 25% em relação ao tempo de execução de AEC-light. Mais especificamente, essa redução é obtida pela ausência em AEC-bind de *overheads* relativos ao suporte de múltiplos escritores: criação de *twins*, criação de *diffs* e aplicação de *diffs*. A ausência desses *overheads* tem efeito direto nos tempos de *lock* e *unlock* de AEC-bind. O tempo de *lock* de AEC-bind é 24% menor do que o tempo de *lock* somado ao tempo de falha de acesso de AEC-light, dado que o custo de criação de *twins* e aplicação de *diffs* corresponde a 26% na soma desses dois tempos em AEC-light. O tempo de *unlock* é reduzido em 52% em AEC-bind em relação a AEC-light, porque em AEC-light 64% do tempo de *unlock* é gasto com criação de *diffs*.

Além disso, em MigDepth, cada vez que a seção sísmica é atualizada, ela é completamente reescrita. Portanto, essa característica se adapta perfeitamente ao esquema proposto em AEC-bind, onde todo o objeto é enviado, sem necessidade de detecção de escritas.

9.2.3 MigFreq

A figura 9.11 mostra o tempo de execução paralela de MigFreq executando sob os sistemas TreadMarks, AEC, AEC-light e AEC-bind. Conforme podemos observar na figura, os tempos de execução de MigFreq diminuem em relação a TreadMarks de acordo com o aumento da complexidade do modelo de programação, i.e., diminuem de AEC para AEC-light e depois para AEC-bind.

AEC × TreadMarks

MigFreq consegue realizar boa parte da computação em paralelo sem a necessidade de comunicação entre os processadores. O uso de *lock* ocorre somente uma vez no fim

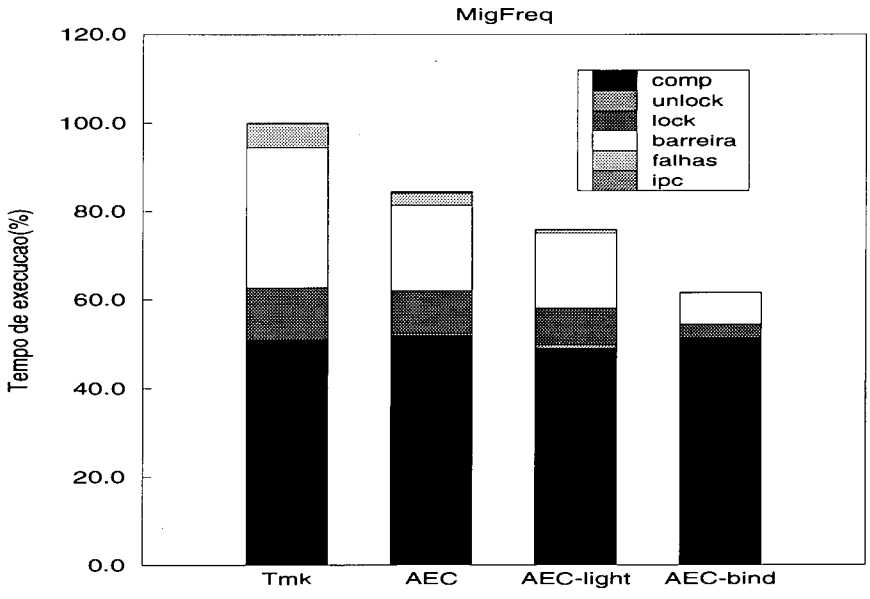

Figura 9.11: Tempo de execução de MigFreq em TreadMarks, AEC, AEC-light e AEC-bind.

da aplicação, quando as freqüências são acumuladas. Portanto, MigFreq apresenta *speedups* altos e mesmo reduções significativas nos *overheads* têm efeito pequeno no tempo total de execução.

A serialização imposta pelas operações de *lock* é responsável por praticamente todo o *overhead* gerado na aplicação. Como LAP atua diretamente nesse tipo de serialização, AEC obtém redução de 16% no tempo de execução de MigFreq em relação a TreadMarks. Mais especificamente, AEC consegue reduzir 48% do *overhead* de falha de acesso dentro de seção crítica, 19% do *overhead* de *lock* em MigFreq e 39% do *overhead* de barreira. A seguir vamos justificar as reduções obtidas nesses dois *overheads*.

Overhead de Falha de Acesso. A tabela 9.6 mostra o número e o tempo gasto com falhas de acesso dentro de seções críticas em TreadMarks, AEC e AEC_{sLAP}. Comparando os dados de AEC e AEC_{sLAP}, vemos que o uso de LAP em AEC não tem efeito no número de falhas. Isso ocorre porque a matriz que armazena os valores acumulados de frequência é acessada pela primeira vez dentro da seção crítica. Portanto, independente de já ter recebido os *diffs* associados ao *lock*, o processador

gera falhas de acesso para buscar as primeiras cópias das páginas correspondentes. Em relação a TreadMarks, entretanto, AEC gera 36% menos falhas de acesso. Essa redução é obtida pela da criação antecipada de *twins* de AEC. Em MigFreq, antes de escrever suas frequências na matriz, o processador lê o valor acumulado por outros processadores. Em TreadMarks, então, ele gera primeiro uma falha de leitura e em seguida uma falha de escrita (para criação do *twin*). Em AEC, na primeira falha de leitura ao receber os *diffs* associados ao *lock*, o processador cria antecipadamente *twins* para cada página correspondente, evitando assim a falha de escrita subseqüente. Com relação ao tempo gasto com falhas de acesso, LAP reduz apenas 24% desse tempo em AEC, por causa dos 25% dos casos em que LAP não acerta a sua previsão.

Sistema	falhas dentro de sc	
	# de falhas	tempo(ms)
TreadMarks	126	262
AEC	80	111
AEC_{sLAP}	80	146

Tabela 9.6: Falhas de acesso dentro de seções em MigFreq, para TreadMarks, AEC e AEC_{sLAP}.

Overhead de Sincronização. AEC reduz em 19% o tempo de *lock* e em 39% o tempo de espera pela barreira. O tempo de *lock* é reduzido devido a redução no tempo de espera pela seção crítica. Como os processadores terminam praticamente ao mesmo tempo a migração da sua seção sísmica, há contenção pelo único *lock* de MigFreq. Assim, qualquer redução no tempo gasto dentro da seção crítica (em falhas de acesso) ocasiona redução no tempo de espera pelo *lock*. Note que a redução obtida no tempo de *lock* não é diretamente proporcional à redução no *overhead* de falha de acesso ocorrida dentro de seção crítica. Isso porque a redução no tempo da seção crítica afeta somente o tempo de espera pelo *lock* e o tempo total da operação *lock* depende também do tempo de troca de mensagens com o gerente e com o último

dono do *lock*. Em AEC, na requisição de um *lock* não diponível, um processador p deve aguardar por duas trocas de mensagens, do *releaser* para o gerente e do gerente para p. Em TreadMarks, p aguarda somente a mensagem de liberação do *releaser*.

O efeito da redução no tempo de seção crítica tem impacto também no *overhead* de barreira, porque a última barreira de MigFreq ocorre logo em seguida à seção crítica de acumulação das frequências. Conforme podemos observar na figura 9.12, quando uma seção crítica com contenção ocorre imediatamente antes da execução de uma barreira, mesmo o primeiro processador a adquirir o *lock* espera na barreira que todos os outros processadores terminem suas seções críticas. Assim, quanto menos tempo o processador gasta na seção crítica, menor o tempo de espera na barreira.

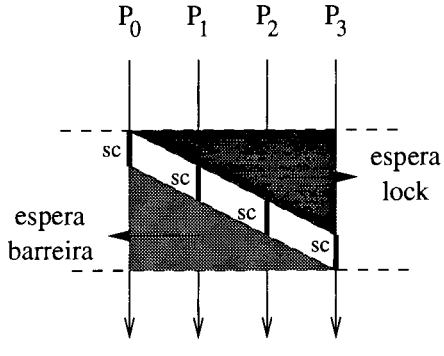

Figura 9.12: Efeito na barreira da serialização da seção crítica.

AEC-light × AEC

No modelo de programação de AEC-light, MigFreq apresenta tempo de execução 11% menor do que o tempo de execução em AEC. Esse resultado é obtido unicamente pela redução de 75% no *overhead* de falha de acesso. Essa redução é explicada pela seguinte característica de MigFreq: o processador 0 realiza a inicialização da matriz de entrada e em seguida cada processador migra a sua seção sísmica de forma independente. Como há uma barreira entre a inicialização e a computação da seção sísmica, no modelo de programação de AEC, a inicialização e a computação das seções sísmicas são realizadas fora de seções críticas. Portanto, em AEC, quando o

processador começa a calcular a seção sísmica, ele gera falha em todas as páginas relativas à matriz de entrada, inicializadas pelo processador 0. Já no modelo de programação de AEC-light, a inicialização da matriz é realizada com operações *lock/unlock-alone* em s e após a barreira os processadores executam um *lock-reader* em s. No pedido de *lock-reader*, o processador recebe de uma vez os *diffs* de todas as páginas inicializadas pelo processador 0, evitando assim várias falhas de leitura seguintes na matriz de entrada.

AEC-bind × AEC-light

MigFreq apresenta, tal qual MigDepth, a característica de reescrever completamente a estrutura compartilhada dentro da seção crítica e, portanto, se adequa bem ao esquema de AEC-bind de não detectar escritas.

AEC-bind obtém em MigFreq redução de 18% no tempo de execução em relação a AEC-light. Essa redução é obtida pela redução de 70% no tempo de *lock* de AEC-bind, comparado à soma dos tempos de *lock* e falhas de acesso em AEC-light. Essa redução é obtida pela eliminação da criação de *twins* e aplicação de *diffs* na operação *lock* e seu efeito é aumentado porque é propagado de um processador para outro, dado que há contenção pelo *lock*. O tempo de *unlock* é insignificante para os dois sistemas.

9.2.4 IS

A figura 9.13 mostra o tempo de execução paralela de IS executando sob os sistemas TreadMarks, AEC, AEC-light e AEC-bind. Conforme podemos observar na figura, tal qual ocorreu em MigFreq, os tempos de execução de IS diminuem em relação a TreadMarks de acordo com o aumento da complexidade do modelo de programação.

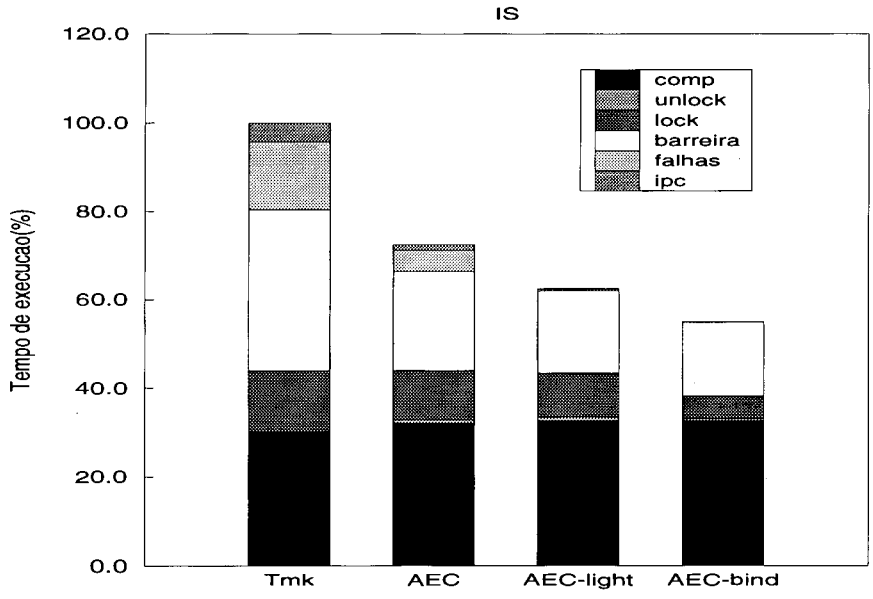

Figura 9.13: Tempo de execução de IS em TreadMarks, AEC, AEC-light e AEC-bind.

AEC × TreadMarks

AEC obtém em relação a TreadMarks uma redução de 27% no tempo total de execução de IS, em grande parte através de grandes reduções nos *overheads* de sincronização e falha de acesso. Mais especificamente, AEC obtém 69% de redução no *overhead* de falha de acesso, 38% de redução no tempo de espera por barreiras e 18% de redução no tempo de *lock*. Nos parágrafos seguintes apresentamos as justificativas para essas reduções.

Overhead de Falha de Acesso. AEC obtém redução significativa no *overhead* de falha de acesso, mas essa redução é causada por motivos diferentes para falhas ocorridas dentro e fora de seções críticas. A tabela 9.7 mostra o número e o tempo gasto com falhas de acesso ocorridas dentro de seções críticas em TreadMarks, AEC e AEC_{sLAP}. Em relação a TreadMarks, AEC obtém redução de 51% no número de falhas e 63% no tempo de falhas. Entretanto, conforme podemos observar pela comparação do número de falhas geradas por AEC e por AEC_{sLAP}, essa redução não é gerada devido unicamente à LAP.

Sistema	falhas dentro de sc	
	# de falhas	tempo(ms)
TreadMarks	2482	4556
AEC	1213	1692
AEC_{sLAP}	1217	2394

Tabela 9.7: Falhas de acesso dentro de seções em IS, para TreadMarks, AEC e AEC_{sLAP}.

LAP tem efeito limitado em IS devido a uma característica específica da aplicação. As falhas de acesso ocorridas dentro de seções críticas acontecem em IS somente na segunda fase da computação, quando os processadores atualizam o vetor global. Como na fase anterior o processador 0 inicializa esse vetor fora de seção crítica, na barreira seguinte os processadores recebem avisos de escrita do processador 0. Assim, embora LAP evite que um processador gere uma falha de acesso para requisitar os dados do último dono do *lock*, não há como evitar a falha ocorrida por causa do aviso de escrita recebido. Portanto, AEC e AEC_{sLAP} geram quase o mesmo número de falhas de acesso, mas em AEC o custo da falha é 30% menor, visto que o processador deve buscar *diffs* apenas do processador 0. Portanto, os maiores ganhos obtidos por AEC no *overhead* de falha de acesso ocorrida dentro de seção crítica, em IS, não são devidos a LAP por si só, mas a técnicas empregadas no protocolo de coerência para suportar LAP.

A redução no número de falhas de acesso é conseqüência da técnica de criação antecipada de *twins*. Em IS, tal qual em MigFreq, dentro da seção crítica o processador realiza uma leitura seguida de uma escrita no vetor global. Em TreadMarks, das 2482 falhas ocorridas dentro de seção crítica, 1232 são falhas de leitura e 1250 são falhas de escrita. AEC só gera as falhas de leitura.

A redução no tempo de uma falha de acesso é em grande parte consequência da técnica de combinação de *diffs*. Em TreadMarks, ao buscar os *diffs* do último dono do *lock*, o processador recebe um conjunto de *diffs* gerados por todos os processadores

que já acessaram o *lock*. Em AEC esses *diffs* são combinados em um único *diff*.

Com relação às falhas de acesso ocorridas fora de seções críticas, AEC obtém uma redução de 71% no tempo gasto por falha, embora gere o mesmo número de falhas que TreadMarks, conforme mostra a tabela 9.8. Isto se explica também pelo efeito de combinação de *diffs* de AEC. As falhas fora de seções críticas ocorrem na terceira fase da computação, onde os processadores lêem (sem *locks*) o vetor global para classificar suas chaves locais. Em TreadMarks, um processador p, ao gerar uma falha fora de seção crítica numa página k, recebe em média 5 *diffs* diferentes, relativos aos intervalos que ele ainda não recebeu do processador que foi o último dono da seção crítica na fase anterior. Em AEC, p recebe somente 1 *diff* do processador que foi o último dono da seção crítica na fase anterior.

Sistema	falhas fora de sc	
	# de falhas	tempo(ms)
TreadMarks	1343	8994
AEC	1292	2573

Tabela 9.8: Falhas de acesso fora de seções críticas em IS, para TreadMarks e AEC.

Overhead de Sincronização. Com relação ao tempo gasto com operações *lock*, a redução de 18% obtida por AEC pode ser explicada pela redução no número de falhas ocorridas dentro de seção crítica. Em IS, como há apenas um *lock* para proteger todo o vetor global, a contenção por esse *lock* é alta. A redução no tempo de *lock* não é proporcional à redução nas falhas porque, quando há contenção, uma parte da operação *lock* é mais custosa em AEC do que em TreadMarks, conforme explicado anteriormente. O tempo de *unlock*, embora pouco significante no tempo total, aumenta bastante em AEC devido à criação de *diffs*, necessária à implementação de LAP.

O efeito da redução no tempo de seção crítica tem impacto também no *overhead* de barreira, porque a segunda fase da computação de IS apresenta espera serial

pelo *lock* seguida de uma operação de barreira. Nesse caso, conforme ilustramos na figura 9.12, a redução obtida no tempo de espera pelo *lock* é refletida diretamente no tempo de espera na barreira.

AEC-light × AEC

O uso de um modelo de programação diferenciado em AEC-light levou a uma redução de 14% no tempo de execução de IS em relação a AEC. Comparando os *overheads* gerados pelos dois protocolos, vemos que em AEC-light praticamente não há *overhead* de falha de acesso, e os tempos de barreira e *lock* são reduzidos em respectivamente 17% e 10%. Entretanto, nessa comparação de *overheads*, vale ressaltar que o *overhead* de falha de acesso ocorrida fora de seção crítica de AEC não existe em AEC-light. Este *overhead* é transferido para *overhead* de *lock* em AEC-light, visto que os acessos realizados fora de seção crítica em AEC são realizados sob operações *lock-alone* ou *lock-reader* em AEC-light. Nessas operações, o processador requisita os *diffs* na própria operação de *lock*. A seguir vamos analisar as reduções obtidas nos *overheads* de falhas de acesso e sincronização em mais detalhes.

Overhead de Falha de Acesso. A tabela 9.9 mostra o número de falhas e tempo gasto com falhas de acesso ocorridas dentro de seções críticas para AEC e AEC-light. Conforme podemos observar, AEC-light reduz drasticamente o número de falhas de acesso e conseqüentemente o tempo total gasto com essas falhas. Essa redução é obtida na segunda fase da computação. Em AEC todos os processadores recebem avisos de escrita e geram falhas para requisitar os *diffs* do processador 0. Em AEC-light, como a inicialização do vetor é realizada sob operações de *lock*, na segunda fase somente o primeiro processador a requisitar o *lock* gera falha para buscar os *diffs* do processador 0.

Overhead de Sincronização. Com relação ao tempo gasto com operações de *lock*, vamos analisar o tempo de cada uma das três primitivas de *lock* diferentes, presentes

Sistema	falhas dentro de sc	
	# de falhas	tempo(ms)
AEC	1213	1692
AEC-light	46	393

Tabela 9.9: Falhas de acesso dentro de seções críticas em IS, para AEC e AEC-light.

no modelo de programação de AEC-light: *lock*, *lock-reader* e *lock-alone*. O tempo da primitiva *lock* padrão é 34% menor em AEC-light do que o tempo de *lock* em AEC. A redução nesse tempo é consequência direta da redução no *overhead* de falha de acesso dentro de seção crítica. As primitivas *lock-reader* e *lock-alone* não existem em AEC, mas o tempo gasto com elas em AEC-light pode ser comparado ao tempo gasto com falha de acesso fora de seção crítica em AEC, nesse caso o custo de falha de acesso ocorrida fora de seção crítica é equivalente ao custo de se buscar dados numa operação *lock-alone* ou *lock-reader*.

AEC-bind × AEC-light

IS apresenta, tal qual MigDepth e MigFreq, a característica de reescrever completamente a estrutura compartilhada dentro da seção crítica e, portanto, também se adequa bem ao esquema de AEC-bind de não detectar escritas.

AEC-bind obtém redução de 12% no tempo de execução em relação a AEC-light. Essa redução se deve à redução de 38% no tempo de *lock* de AEC-bind, comparado à soma dos tempos de *lock* e falhas de acesso de AEC-light. Isto se explica pela eliminação dos *overheads* de criação e aplicação de *diffs* na operação de *lock* e, tal qual para MigFreq, seu efeito é aumentado devido à contenção pelo *lock*. O tempo de *unlock* é reduzido consideravelmente em AEC-bind, visto que em AEC-light 50% desse tempo é gasto com criação de *diffs*.

9.2.5 SOR

A figura 9.14 mostra o tempo de execução paralela de SOR executando sob os sistemas TreadMarks, AEC, AEC-light e AEC-bind. Conforme podemos observar pela figura, nessa aplicação, AEC obtém tempo de execução próximo a TreadMarks; e AEC-light obtém praticamente o mesmo tempo de execução que AEC-bind, ambos apresentam redução de 15% em relação a Treadmarks.

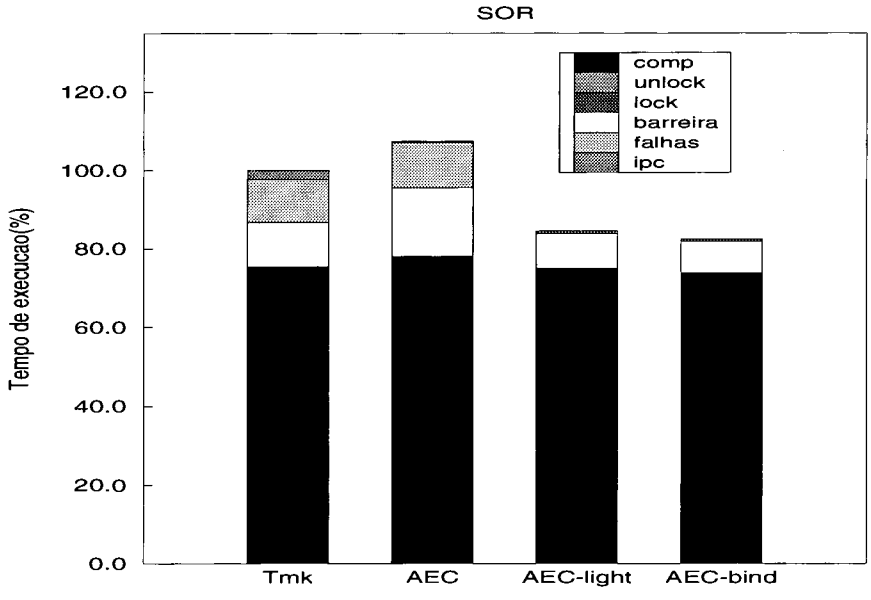

Figura 9.14: Tempo de execução de SOR em TreadMarks, AEC, AEC-light e AEC-bind.

AEC × TreadMarks

Conforme podemos observar na figura, AEC apresenta apenas um pequeno aumento de 7% no tempo de execução de SOR em relação a TreadMarks. No modelo de programação de AEC e TreadMarks, SOR não contém *locks*. Por esse motivo, AEC apresenta comportamento bastante parecido com TreadMarks, visto que utiliza somente protocolo de invalidações.

A pequena diferença no tempo de execução dos dois sistemas é causada pelo aumento no *overhead* de barreira de AEC. Esse aumento é ocasionado pelas primeiras fases da computação. Como AEC aproveita o tempo de barreira para computar

diffs antecipadamente, os primeiros processadores a chegarem na barreira, criam alguns *diffs* e protegem as respectivas páginas para escrita. Na fase seguinte, os processadores que criaram *diffs* na barreira, vão gerar falhas de escrita ao acessarem as páginas. Em compensação, o processador que não teve tempo de criar *diffs* não gera essas falhas e chega mais cedo à próxima barreira.

Esse efeito ocasiona um desbalanceamento de carga e um conseqüente aumento do tempo de espera na barreira, mas ele ocorre somente nas primeiras fases da computação. AEC possui um mecanismo que impede a criação (mesmo que antecipada) de *diffs* para páginas que não estão sendo efetivamente compartilhadas. Depois de duas fases sem receber requisição de *diffs* para uma determinada página, o processador a marca como não-compartilhada até que uma próxima requisição seja recebida. Em SOR apenas as páginas das bordas das matrizes são compartilhadas entre os processadores. Portanto, depois da terceira fase da computação, AEC consegue detectar as páginas não efetivamente compartilhadas e o efeito citado acima torna-se insignificante.

AEC-light × AEC

O sistema AEC-light apresenta redução de 15% no tempo total de execução de SOR em relação a AEC. Em SOR, no modelo de programação de AEC-light, há uma operação *lock* para cada acesso a uma borda da matriz compartilhada. Na própria operação de *lock*, o processador requisita a borda da matriz ao último dono do *lock*. No entanto, o *overhead* de falha de acesso de AEC não é completamente transferido para o *overhead* de *lock* de AEC-light, devido ao uso de LAP global em AEC-light. Por ser uma aplicação regular e com padrão de compartilhamento bastante previsível, a técnica de previsão global de LAP em SOR obtém alta taxa de acerto e envio de atualizações seletivas para um número mínimo de processadores. Em 98% das operações de *lock* de SOR em AEC-light, o processador já recebeu a

borda da matriz antecipadamente.

AEC-bind × AEC-light

Os sistemas AEC-light e AEC-bind têm praticamente o mesmo tempo de execução em SOR. Como em AEC-light é possível estabelecer dinamicamente a relação entre cada *lock* e cada borda da matriz, ambos os sistemas apresentam comunicação mínima, representada pela transferência das bordas entre os processadores envolvidos.

Embora AEC-light apresente *overhead* de criação e aplicação de *diffs* e de criação de *twins*, esse *overhead* é muito pequeno em comparação com o tempo total de execução. Além disso, boa parte da criação dos *diffs* é escondida no tempo de espera pela barreira.

9.2.6 FFT

A figura 9.15 mostra o tempo de execução paralela de FFT executando sob os sistemas TreadMarks, AEC, AEC-light e AEC-bind. Conforme podemos observar na figura, nessa aplicação, AEC e TreadMarks apresentaram o mesmo tempo de execução; AEC-light apresenta desempenho desastroso; e AEC-bind obtém grande redução no tempo de execução em relação a TreadMarks e AEC.

AEC × TreadMarks

Conforme podemos observar, AEC e TreadMarks apresentam comportamento similar para FFT. Tal qual em SOR, em FFT não há *locks* e, portanto, ambos os sistemas utilizam somente protocolo de invalidação.

A pequena diferença nos *overheads* gerados pelos dois sistemas está nos *overheads* de falha de acesso e de barreira. Tal qual explicado para SOR, em AEC alguns *diffs* são criados antecipadamente na barreira, reduzindo o custo das falhas na fase seguinte. Em compensação, o processador que não teve tempo de criar seus *diffs* na

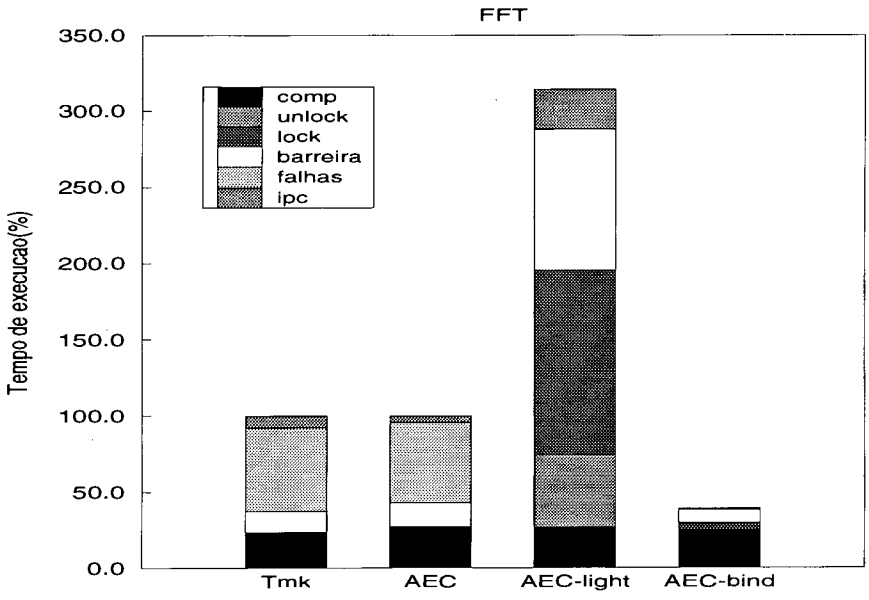

Figura 9.15: Tempo de execução de FFT em TreadMarks, AEC, AEC-light e AEC-bind.

barreira, atrasa a computação da fase seguinte. Portanto, a redução obtida por AEC no *overhead* de falha de acesso tem como conseqüência um aumento no *overhead* de barreira.

AEC-light × AEC

O sistema AEC-light triplica o tempo de execução de FFT em relação a AEC. Esse resultado ocorre devido à associação dinâmica de variável de sincronização com dado compartilhado realizada por AEC-light em FFT. Para a inicialização das duas matrizes de entrada, o processador 0 adquire de modo aninhado todos os *locks* que protegem os subcubos das matrizes. Assim o protocolo de coerência de AEC-light associa as matrizes inteiras a todas as variáveis de *lock* da aplicação. A cada pedido de *lock*, então, as duas matrizes são requisitadas. O excesso de dados trocados entre os processadores, ilustrado na figura 9.6 da seção 9.2.1, é responsável pelo aumento considerável gerado em todos os *overheads*.

AEC-bind × AEC-light

O sistema AEC-bind resolve o grave problema de desempenho do sistema AEC-light em FFT, através da associação explícita de variável de sincronização com dado compartilhado. Como o programador especifica os trechos da matriz que são associados a cada *lock*, a quantidade de dados trocados entre os processadores é extremamente menor.

AEC-bind apresenta tempo de execução 61% inferior aos tempos obtidos para AEC e TreadMarks. Esse resultado é obtido pela redução de 91% no tempo de *lock* de AEC-bind comparado ao tempo de falhas de AEC e TreadMarks. Dois fatores contribuíram para essa redução: uso da técnica de previsão global (figura 9.9) [2]; e o uso de objetos para representar blocos não contíguos de memória. A busca de um objeto inteiro evita que várias tranferências de dados sejam realizadas para um mesmo subcubo da matriz, visto que em AEC e TreadMarks, esses blocos podem pertencer a páginas distintas. Note ainda que o fato de que os subcubos são completamente reescritos dentro das seções críticas permite que AEC-bind tire vantagem do envio do objeto inteiro a cada pedido de *lock* em FFT.

9.2.7 Water

A figura 9.16 mostra o tempo de execução paralela de Water executando sob os sistemas TreadMarks, AEC, AEC-light e AEC-bind. Conforme podemos observar pela figura, Water apresenta comportamento diferente das outras aplicações. O tempo de execução aumenta para os três sistemas propostos em relação a TreadMarks, sendo que AEC e AEC-bind apresentaram praticamente o mesmo tempo de execução e AEC-light apresenta aumento considerável em relação aos outros dois sistemas.

[2]Note que, AEC-light também emprega LAP global, mas o excesso de dados associados a cada *lock* não permite que as atualizações seletivas de LAP tenham qualquer efeito positivo no tempo de execução.

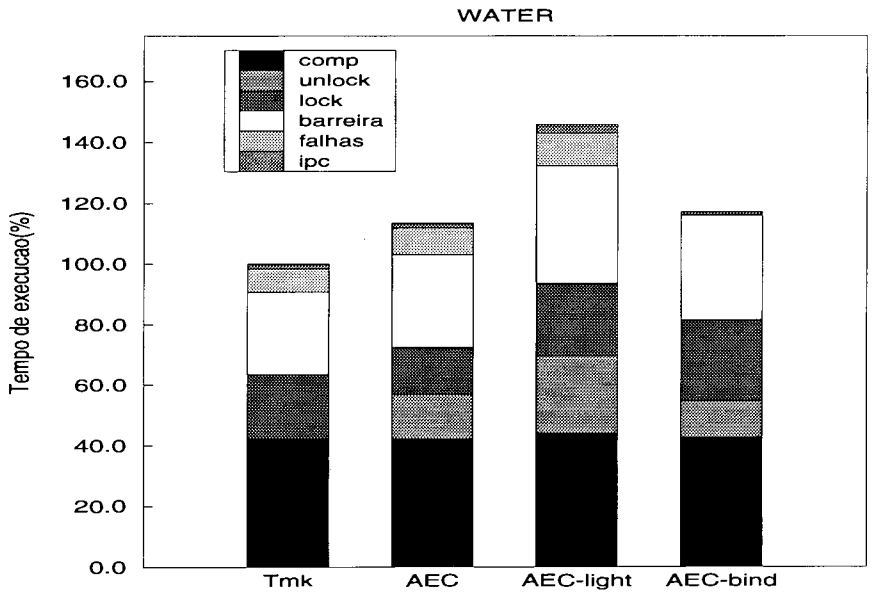

Figura 9.16: Tempo de execução de Water em TreadMarks, AEC, AEC-light e AEC-bind.

AEC × TreadMarks

AEC apresenta aumento de 13% no tempo de execução de Water em relação a TreadMarks. Analisando os *overheads* mais importantes, vemos que AEC aumenta o *overhead* de falha de acesso em 13%, aumenta também em 13% o tempo de barreira, reduz em 26% o tempo de *lock* e apresenta tempo de *unlock* 250 vezes superior. Em seguida, apresentamos uma análise mais detalhada dos *overheads* de falha de acesso e sincronização.

Overhead de Falha de Acesso. O aumento considerável no *overhead* de falha de acesso se deve basicamente ao *overhead* relativo às falhas geradas dentro de seções críticas. A tabela 9.10 mostra o número e o tempo gasto com essas falhas para TreadMarks, AEC e AEC_{sLAP}. Conforme podemos observar, LAP permite redução de 65% no número de falhas dentro de seção crítica. Mesmo assim, AEC obtém 34% mais falhas do que TreadMarks. Isso se explica por uma característica especial de Water. Cada processador é responsável por um conjunto de moléculas que estão alocadas contiguamente na memória; cada página contém aproximadamente

8 moléculas. Na fase de computação intermolécula, onde se concentra todo o processamento com *locks*, um processador começa acessando as suas moléculas e em seguida acessa as moléculas de outros processadores. Em TreadMarks, quando um processador p começa a acessar um conjunto de moléculas da página k, ele gera somente uma falha de acesso ao acessar a primeira molécula de k (dado que p acessa todas as outras moléculas de k antes dos outros processadores). Em AEC, entretanto, as modificações realizadas em seções críticas diferentes são tratadas de forma independente. Assim, para acessar as moléculas de k, p gera 8 falhas quando LAP não acerta sua previsão. Na verdade, mesmo quando LAP acerta a previsão, p deve criar novo *twin* para k a cada operação *lock*.

Sistema	falhas dentro de sc	
	# de falhas	tempo(ms)
TreadMarks	6673	4447
AEC	8956	5740
AEC_{sLAP}	25760	25595

Tabela 9.10: Falhas de acesso dentro de seções em Water, para TreadMarks, AEC e AEC_{sLAP}.

Com relação ao *overhead* de falhas de acesso ocorridas fora de seções críticas, observamos que ele se mantém equivalente em AEC e TreadMarks, visto que o tratamento dessas falhas é similar em AEC e TreadMarks.

Overhead de Sincronização. A redução obtida no tempo de *lock* se deve ao fato de não haver contenção pelos *locks*, e, conforme explicado anteriormente, o custo do *lock* é menor em AEC do que em TreadMarks quando não há espera pelo *lock*. O aumento considerável ocorrido no tempo de *unlock* se explica pela necessidade de criação de um *diff* a cada *lock*, e é ainda exacerbado pela grande quantidade de operações *unlock* dessa aplicação.

AEC-light × AEC

Pela figura 9.16 observamos que o uso de modelo de programação diferenciado e do protocolo AEC-light compromete o desempenho de Water. Para esta aplicação, AEC-light aumenta o tempo de execução em 29% em relação a AEC.

O tempo gasto com a primitiva *lock* padrão é o mesmo para os dois protocolos. Entretanto, comparando *overheads* de falha de acesso observamos que: o custo da falha de acesso dentro de seção crítica é equivalente para os dois protocolos, enquanto que o tempo gasto com busca de dados pelas primitivas *lock-reader* e *lock-alone* de AEC-light é o dobro do *overhead* de falha de acesso ocorrida fora de seção crítica em AEC. Esta diferença se deve a que *lock-readers* buscam dados com granularidade muito menor que uma página. Em AEC, numa falha de acesso na página k fora de seção crítica, um processador p requisita as modificações realizadas nas moléculas de k na fase anterior. Se um mesmo processador foi o último dono do *lock* que protege todas as moléculas de k, em uma única mensagem ele envia todas as modificações para p e a página está coerente para os acessos seguintes. Em AEC-light, essa mesma operação é realizada com uma chamada a *lock-reader* para cada molécula. Assim, tendo 8 moléculas em uma página, p vai requisitar as 8 moléculas, uma de cada vez, mesmo que todas as requisições sejam para o mesmo processador.

Em AEC-light, o tempo de *unlock* aumenta consideravelmente por causa do custo da operação *unlock-alone*, a qual envolve criação de *diffs*. Em AEC, a criação desses mesmos *diffs* é realizada durante a espera por uma barreira.

AEC-bind × AEC-light

AEC-bind permite redução de 19% no tempo de execução de Water em relação a AEC-light. Em Water, devido à grande quantidade de operações *lock*/*unlock* realizadas, o custo da criação de *twin* e da aplicação de *diffs* numa operação *lock* e da criação de *diffs* numa operação *unlock* tem peso considerável no tempo de

execução. Em AEC-light esses custos representam 10% do *overhead* gerado. Assim, a ausência desses *overheads* em AEC-bind levou a reduções de 20% no tempo de *lock* (comparado com a soma dos tempos de *lock* e falha em AEC-light) e 20% no tempo de *unlock* em relação a AEC-light.

9.3 Sumário dos Resultados

Os resultados apresentados mostram que LAP consegue realizar suas previsões com alta precisão; as taxas de acerto obtidas variam de 75% a 90% para LAP local e de 35% a 99% para LAP global.

Com relação à execução das aplicações nos sistemas *software DSM* propostos nessa tese, mostramos que nossos sistemas, na grande maioria das vezes, apresentam ganhos significativos de *speedup* comparados ao sistema Treadmarks.

Com relação à comunicação gerada pelos protocolos de coerência, mostramos que AEC, AEC-light e AEC-bind em geral produzem menor número de mensagens que TreadMarks. Essa redução é obtida pela eliminação de falhas de acesso, pelo uso de LAP e pelo recebimento, numa única mensagem, de todos os *diffs* ou objeto associados a um mesmo *lock*. Quanto à quantidade de bytes trocados, exceto em raras exceções, os três sistemas propostos produzem ganhos, principalmente porque reduzem a quantidade de falhas de acesso, armazenam os *diffs* de forma combinada, e/ou transferem objetos inteiros.

Com relação aos *overheads* de tempo de execução gerados pelos protocolos, mostramos que tanto para AEC como para AEC-light e AEC-bind, o uso das atualizações seletivas de LAP permite redução razoável no *overhead* de busca de dados compartilhados acessados dentro de uma seção crítica e tem efeito direto no tempo de espera pela seção crítica e, em alguns casos, no *overhead* de barreiras. Em AEC, mesmo para aplicações que não se beneficiam do protocolo de atualizações, como FFT e SOR, mostramos que o protocolo de invalidações de AEC obtém desempenho

semelhante a TreadMarks, embora os dois sistemas sejam bastante diferentes.

Em AEC e AEC-light, observamos que não só as atualizações seletivas de LAP permitem ganhos de desempenho, mas algumas características dos protocolos de coerência, próprias para o suporte de LAP (criação antecipada de *twins* e armazenamento combinado de *diffs*) permitem sensíveis reduções no número de falhas de acesso geradas.

Em AEC-bind, observamos que o uso de objeto como granularidade de coerência extingue *overheads* envolvidos no suporte a múltiplos escritores. Na maior parte das aplicações estudadas, esses *overheads* têm peso significativo no *overhead* total gerado pelo protocolo. Observamos ainda que, conforme esperávamos, os dados acessados dentro de seções críticas são completamente reescritos nas aplicações estudadas e, portanto, essas aplicações se beneficiam do envio de objetos inteiros realizado por AEC-bind.

O modelo de programação de AEC-light e AEC-bind, em que todos os acessos a dados compartilhados são protegidos por *locks*, permite outro tipo de redução no *overhead* de busca de dados. Com uma única requisição ao último dono do *lock* que protege o dado, é possível manter coerente um dado que foi escrito em diferentes fases anteriores. Em contraste, num acesso fora de seção crítica a um dado escrito em várias fases diferentes, AEC e TreadMarks têm que requisitar todas as modificações realizadas nas fases anteriores.

Já para o caso de Water, observamos que, mesmo sendo potencialmente ineficiente manter coerência por página quando a unidade de acesso é muito menor, o uso do modelo LRC, aliado ao padrão de acesso das moléculas, permite que as ações tomadas pelo protocolo para uma molécula adiantem ações de coerência para as outras moléculas alocadas na mesma página.

Essencialmente, nossos experimentos apresentam dois resultados principais. O primeiro é que a técnica LAP e a infra-estrutura que a suporta em geral permitem

ganhos significativos de desempenho para sistemas *software DSM*. O segundo é que o aumento da complexidade do modelo de programação não necessariamente leva a ganhos de desempenho.

Capítulo 10

Trabalhos Relacionados

Neste capítulo apresentamos os trabalhos relacionados a esta tese. Há uma quantidade considerável de trabalhos na área de memória compartilhada distribuída. Aqui sumarizamos esses trabalhos, concentrando nossas discussões na pesquisa mais diretamente relacionada aos temas contidos na tese. Primeiramente, discutimos trabalhos relacionados à programação de sistemas paralelos e ao impacto do modelo de consistência no modelo de programação. Em seguida, comparamos as principais características dos protocolos de coerência implementados pelos três sistemas propostos nesta tese com outros protocolos, no que diz respeito ao seu nível de implementação e ao tipo de protocolo. Além disso, discutimos técnicas que como LAP visam tolerar os *overheads* de execução em sistemas DSM.

10.1 Programação e Consistência de Memória

A programação num sistema paralelo poderia ser uma tarefa trivial se o programador não tivesse que se preocupar com a paralelização do código e nem com a interação entre os processos paralelos. Idealmente, essas tarefas poderiam ser realizadas pelo compilador e, assim, os programadores utilizariam um modelo seqüencial de programação. Alguns compiladores para sistemas paralelos, como por exemplo HPF [46] e SUIF [35], são dirigidos para esse objetivo, mas além de serem limitados em suas análises, em geral requerem intervenção do programador para obtenção de

bom desempenho. Em HPF, por exemplo, é o programador o responsável por definir a distribuição dos dados, o que o afasta de um modelo seqüencial simples.

Como a tecnologia atual em compiladores para sistemas paralelos, em geral, não permite a geração de programas eficientes, a grande maioria dos sistemas paralelos utiliza programação paralela com paralelismo explícito. Há dois paradigmas principais de programação paralela com paralelismo explícito: passagem de mensagens e memória compartilhada. A passagem de mensagens força que não apenas o paralelismo seja explícito, mas também a comunicação entre processos seja explícita, o que dificulta ainda mais a programação desses sistemas. Desta forma, nessa tese nos concentramos no paradigma de memória compartilhada, no qual a comunicação é implícita. Mesmo o modelo de programação de AEC-bind é mais simples que utilizar passagem de mensagens, já que não é necessário identificar qual processador deve prover cada dado e quando ele deve fazê-lo.

Num sistema que implementa o paradigma de programação paralela com memória compartilhada, o modelo de consistência de memória empregado tem impacto tanto no seu desempenho quanto na sua programação. O modelo SC é o mais restritivo de todos os modelos de consistência, mas provê um modelo de programação tão simples quanto o de um uniprocessador multiprogramado.

A maior parte dos modelos relaxados de consistência utiliza modelo de programação quase tão simples quanto o empregado no modelo SC. A única diferença entre eles está em que os modelos relaxados requerem sincronização explícita, i.e., o programador deve sincronizar o programa através de primitivas especiais providas pelo sistema. O modelo de consistência dos sistemas propostos nessa tese, EC, está entre os mais relaxados, junto com o modelo RC e o seu descendente LRC.

O modelo RC foi proposto no sistema DASH [53] e é empregado em uma série de outros sistemas, tais como Munin [22], Quarks [73], CRL [41] e mais recentemente Shasta [63]. A versão "preguiçosa" de RC, LRC, foi introduzida no sistema

TreadMarks [45] e é também utilizada em vários outros sistemas, tais como o de Kontotanassis, Scott e Bianchini [48], HLRC [76], ADSM [57], CVM [43]. Na verdade, CVM possui suporte para implementação de protocolos baseados em RC e em LRC.

O modelo EC se adequa perfeitamente às otimizações propostas pela técnica LAP, pois explora a relação entre variável de sincronização e os dados compartilhados que ela protege. Outros sistemas que também empregam o modelo EC são Midway [11], ScC [39] e Brazos [71]. O sistema AEC utiliza EC com modelo de programação idêntico ao dos sistemas ScC e Brazos, o qual não requer a associação explícita de variável de sincronização com dado compartilhado. Assim, AEC explora LAP com o modelo de programação mais simples possível para EC. Já o modelo de programação do sistema AEC-light também não requer associação explícita, mas força que todos os acessos a dados compartilhados sejam realizados dentro de seções críticas. O sistema CRL, por exemplo, utiliza um modelo de programação semelhante ao de AEC-light. CRL utiliza um modelo baseado em *shared regions* [61]. O programador define regiões de código onde haverá compartilhamento de dados e utiliza anotações especiais para delimitar os acessos às regiões compartilhadas. O sistema AEC-bind utiliza modelo de programação similar a AEC-light, mas com a associação explícita de dado compartilhado com variável de sincronização, como em Midway.

O uso de modelos de programação distintos nos sistemas propostos nessa tese nos permitiu realizar uma avaliação do impacto do modelo de programação na complexidade e no desempenho dos sistemas. Pelo o que sabemos, há um único trabalho [1] que trata da comparação entre sistemas baseados em modelos de programação diferenciados. Este trabalho avalia dois sistemas diferentes (baseados em EC e LRC), mas é centrado apenas em opções para a implementação dos sistemas estudados. Além disso, o trabalho não analisa como o modelo de programação afeta

os *overheads* e a complexidade dos protocolos e não considera técnicas como LAP.

10.2 Protocolo de Coerência e Tolerância a *Overheads*

Nesta seção comparamos os três sistemas propostos nessa tese com outros sistemas no que diz respeito ao nível de implementação e ao tipo de protocolo de coerência utilizado. Além disso, discutimos técnicas que sobrepõem *overheads* com tempo de computação ou outros *overheads*.

10.2.1 Nível de Implementação e Granularidade

O protocolo utilizado para manter as memórias de um sistema DSM coerentes pode ser implementado totalmente em hardware; totalmente em software; ou de forma híbrida.

Implementação em Hardware

Sistemas *hardware DSM* permitem que cada processador tenha acesso direto às memórias dos outros processadores. Os sistemas hardware DSM podem ser divididos em cinco grupos principais: (1) os que implementam coerência de páginas em hardware (e.g., DDM [34] e KSR [20]); (2) os que implementam coerência de linhas de cache em hardware (e.g., DASH, Alewife [5], Origin [52], Exemplar [24] e SCI [40]); (3) os que implementam coerência das caches em software (e.g., T3D [25]); (4) os que utilizam hardware programável altamente otimizado para implementação do protocolo de coerência (e.g. FLASH [50], Typhoon [60] e NUMAQ [55]); e (5) os que simplesmente não permitem o cacheamento de dados remotos (e.g, T3E [65]).

Sistemas *hardware DSM* alcançam ótimo desempenho para uma grande gama de aplicações. No entanto, estes sistemas invariavelmente envolvem grande complexidade. Os sistemas nos grupos 1 e 2 requerem a implementação de todo o protocolo de coerência em hardware, incluindo todas as situações de exceção e problemas de

temporização. Os sistemas nos grupos 2, 3, 4 e 5, por implementarem granularidade muito fina de acessos à memoria, requerem que todo o hardware associado a acessos externos ao processador (máquina de estados do protocolo de coerência, rede de interconexão, interface entre processador e lógica de coerência, barramentos de dados do sistema, etc) seja altamente otimizado. Toda essa complexidade leva a um custo alto e a um longo tempo de projeto.

Implementação em Software

Para atacar diretamente o problema de custo de sistemas *hardware DSM*, sistemas DSM totalmente implementados em software foram propostos. A seguir, discutimos alguns desses sistemas, agrupando-os segundo a granularidade da unidade de coerência empregada e apresentando-os em ordem decrescente de granularidade.

Diversos sistemas se baseiam no hardware de memória virtual e utilizam, tal qual AEC e AEC-light, a página como unidade de coerência, e.g., Ivy [54], Munin, Quarks, TreadMarks, HLRC, CVM, ADSM e Brazos. Devido ao tamanho considerável das suas unidades de coerência, esses sistemas proveêm suporte para permitir a existência de múltiplos escritores numa página e assim aliviar o problema de falso compartilhamento. Exceto por Ivy, que foi o primeiro *software DSM* proposto e não trata do problema de falso compartilhamento, todos os outros sistemas citados suportam múltiplos escritores e modelos relaxados de consistência. O trabalho de Keleher [43], entretanto, compara o uso de modelo relaxado de consistência com suporte a múltiplos escritores e mostra que relaxar o modelo de consistência é mais importante do que suportar múltiplos escritores por página.

Tal qual AEC-bind, alguns sistemas, como Midway, CRL, e Orca [9], utilizam o objeto ou região como unidade de coerência. Na definição das unidades de coerência, os sistemas Midway e CRL se baseiam em suporte de compilação ou programação, enquanto que o sistema Orca se baseia em uma linguagem de programação própria.

Midway, entretanto, emprega mecanismo de detecção de escritas através de um compilador sofisticado que insere código extra para ligar *flags* de escrita (*dirty bits*). Em contraste, AEC-bind não realiza a detecção de escritas, enviando sempre o objeto inteiro para não incorrer no *overhead* associado a cada operação de escrita. Além disso, AEC-bind não se baseia em nenhum suporte de compilação ou linguagem.

A grande maioria dos sistemas *software DSM* utilizam unidade de coerência com granularidade grossa ou média (objetos e regiões em geral são maiores que linhas de cache, mas menores que páginas). No entanto, alguns sistemas *software DSM* utilizam unidade de coerência com granularidade fina. Os sistemas Shasta e Blizzard-S [64] são bons exemplos. Eles utilizam instrumentação especial para cada *load* e *store* que testa se o dado está presente localmente e se pode haver problema de coerência. AEC e AEC-light não têm bom desempenho em aplicações cujo acesso é realizado com granularidade fina, em compensação AEC-bind provê desempenho aceitável para essa classe de aplicações sem necessidade de nenhuma instrumentação no código e os conseqüentes atrasos gerados nas operações de *load* e *store*.

Ainda não há um consenso na comunidade científica, sobre o tamanho ideal da granularidade de coerência para LRC ou EC. O trabalho de Zhou *et al.* [77] compara um protocolo baseado em página com um protocolo com suporte a granularidade fina, os resultados mostram que o protocolo baseado em páginas apresentou desempenho superior. Por outro lado, o trabalho de Bilas e Singh [16] mostra que algumas aplicações podem se favorecer da diminuição do tamanho da unidade de coerência. Já o trabalho de Buck e Keleher [19] compara sistemas baseados em objetos com sistemas baseados em páginas e conclui que os sistemas baseados em páginas apresentam vantagens de desemepenho devido à localidade de referência das aplicações e ao efeito de *prefetching* da unidade de coerência grande. Os resultados de nossas avaliações mostraram que o uso da página como unidade de coerência em aplicações com granularidade fina de acesso ao dado pode obter desempenho razoável no mo-

delo LRC, mas pode ter graves problemas de desempenho em EC.

Implementação Híbrida

Como mencionado anteriormente, sistemas *hardware DSM* alcançam bom desempenho mas a um alto custo, enquanto sistemas *software DSM* têm custo mais baixo mas têm bom desempenho para uma classe menor de aplicações. Para atacar o problema de custo dos sistemas *hardware DSM* e o problema de desempenho dos sistemas *software DSM*, foram propostos sistemas híbridos. Sistemas como SHRIMP [18], Cashmere [47] e NCP_2 [12] implementam protocolo de coerência em software, mas possuem mecanismos de hardware especiais que auxiliam, de forma eficiente, a tarefa do protocolo de coerência. Todos os três sistemas utilizam a página como unidade de coerência, aliviando o problema do falso compartilhamento com suporte a existência de múltiplos escritores numa mesma página e com o uso de modelos relaxados de consistência.

O mulicomputador SHRIMP é formado por uma rede de processadores Pentium. O mecanismo de hardware utilizado para melhorar a eficiência do protocolo de coerência permite o mapeamento de páginas de um nó em páginas de outro nó e assim as escritas realizadas por um nó são automaticamente propagadas para o outro nó que tem a página mapeada. Foram propostos dois protocolos de coerência diferentes para SHRIMP: AURC [36] e ScC. Esses protocolos são baseados em *home-nodes* para tirar proveito do hardware especial para atualizações remotas automáticas. Os protocolos da familia Cashmere têm características semelhantes por também utilizarem hardware para atualizações automáticas.

O sistema NCP_2 em desenvolvimento na COPPE Sistemas/UFRJ utiliza suporte de hardware simples e de baixo custo para implementação do protocolo de coerência. Tal suporte de hardware consiste de um controlador de protocolo programável que pode ser utilizado para dividir as tarefas de coerência com o seu processador asso-

ciado. Além disso, o controlador permite a implementação de técnicas de tolerância à latência de comunicação e a *overheads* de processamento de coerência. O primeiro protocolo de coerência utilizado no NCP_2 será uma versão modificada de TreadMarks.

Todos esses sistemas híbridos foram comparados com TreadMarks e apresentam desempenho superior a ele. AEC, AEC-light e AEC-bind também apresentam desempenho superior a TreadMarks, mas não requerem nenhum suporte de hardware além do que já existe em *workstations* comerciais. No entanto, AEC e AEC-light poderiam se beneficiar bastante do uso do controlador de protocolos do NCP_2, uma vez que este permite sobrepor ou eliminar os *overheads* relacionados a *diffs*. Os sistemas propostos aqui também poderiam utilizar o suporte de hardware de SHRIMP e Cashmere, mas com algumas restrições. Idealmente, nossos sistemas poderiam mapear a(s) página(s) do dado que vai ser modificado em página(s) do próximo processador a utilizar o dado segundo LAP. Assim, o próximo processador a utilizar o dado já receberia automaticamente as atualizações realizadas no mesmo. A eficiência desse tipo de esquema dependeria então do custo de re-mapeamento de páginas e quão frequentemente ele teria que ser realizado.

Há ainda uma outra classe de sistemas DSM híbridos, aquela em que o sistema base é formado por um *cluster* de SMPs (*Symmetric MultiProcessor*) interconectados. Nesse tipo de sistema, é possível utilizar protocolo de coerência em hardware dentro de cada SMP e protocolo em software para permitir memória compartilhada entre SMPs. Os sistemas SoftFLASH [28], Cashmere-2L [72] e SMP-Shasta [62] são exemplos dessa classe de DSMs híbridos. Os sistemas propostos nesta tese consideram *clusters* ou redes de uniprocessadores, mas poderiam ser modificados para tirar proveito das memórias centralizadas dos SMPs. Na verdade, os SMPs melhorariam o desempenho de LAP e suas atualizações, já que todos os processadores de um SMP poderiam ser tratados como um único processador.

10.2.2 Protocolo de Invalidação x Atualização

Como as redes de interconexão normalmente não suportam o excesso de mensagens geradas por um protocolo de atualização, a maior parte dos protocolos de coerência de sistemas DSM são baseados em invalidações (e.g., DASH, Alewife, NCP_2, Shasta, TreadMarks, CRL, e HLRC).

Como exemplo de protocolos baseados unicamente em atualizações temos o protocolo do sistemas Plus [17], RMS [56], Orca e os protocolos base de Munin e Midway. Plus e RMS implementam protocolo de coerência em hardware. Orca se baseia no envio de *broadcasts* das escritas realizadas. O protocolo base de Munin se baseia no conceito de atualizações atrasadas e do modelo RC de consistência. Midway aproveita a associação de variável de sincronização com dado compartilhado do modelo EC para enviar atualizações somente do dado afetado pela sincronização, na aquisição de uma seção crítica.

Alguns sistemas utilizam uma combinação de invalidações e atualizações, para tentar obter os benefícios dos dois protocolos (evitar excesso de mensagens e o grande *overhead* gerado pelas falhas de acesso), e.g., Munin, Midway, Lazy Hybrid [27], Brazos, ScC e ADSM. Os protocolos de AEC, AEC-light e AEC-bind também se incluem na classe dos protocolos híbridos. Eles são baseados na técnica LAP, que permite envio seletivo de atualizações, evitando excesso de mensagens na rede.

Midway e Munin permitem, através de anotações especiais, que o programador especifique o uso de protocolo de invalidações na sua aplicação. Midway utiliza protocolo de invalidação quando o programador especifica o uso de modelo de consistência menos relaxado que EC. Munin permite que o programador defina um tipo diferente de protocolo para cada estrutura compartilhada.

O protocolo Lazy Hybrid é uma versão híbrida do protocolo empregado em TreadMarks. Lazy Hybrid utiliza invalidações da mesma forma que TreadMarks,

mas aproveita para enviar algumas atualizações na mensagem de liberação da seção crítica. As atualizações enviadas são os *diffs* que o *releaser* tem mas que ainda não foram vistos pelo *acquirer*. AEC envia as atualizações de forma ainda mais agressiva, já que todas as atualizações já realizadas nos dados são enviadas numa mensagem. Além disso, essas atualizações são enviadas com antecedência através de LAP.

Os protocolos empregados em ScC e Brazos utilizam suporte de hardware específico para enviar atualizações de forma eficiente. O protocolo ScC avaliado em [39] utiliza o hardware para atualizações automáticas (apesar de uma implementação somente em software ser possível) para manter a página coerente no processador *home*. O protocolo de Brazos se baseia em facilidades de *multicast* da rede de interconexão para enviar *early updates*. *Early Updates* são atualizações de páginas que são acessadas por vários processadores logo após uma barreira. Este mecanismo procura evitar o tráfego que resultaria de várias operações *multicast* concorrentes. Em contraste, AEC não se baseia em nenhum suporte de hardware para envio de atualizações.

O protocolo do sistema ADSM se adapta entre atualização e invalidação segundo o padrão de compartilhamento dinâmico exibido pela aplicação. ADSM utiliza protocolo de atualização em dois casos específicos: para páginas categorizadas como migratórias e associadas a uma variável de *lock*; e para páginas caracterizadas como produtor-consumidor entre duas fases distintas da aplicação. AEC utiliza atualizações para modificações realizadas dentro de seções críticas, independente do padrão de compartilhamento, e não envia a página toda, mas somente os seus *diffs*. AEC não utiliza protocolo de atualizações para dados modificados fora de seções críticas. A estratégia utilizada em ADSM, entretanto, é bastante parecida com a técnica de afinidade global de LAP, empregada em AEC-light e AEC-bind. A técnica de afinidade global, entretanto, é baseada em modelo de programação distinto e utiliza heurística diferente da utilizada por ADSM.

ADSM permite ainda que o protocolo de coerência se adapte em relação ao uso ou não de suporte a múltiplos escritores. Os três sistemas propostos nessa tese não apresentam nenhum tipo de adaptação às características da aplicação. Entretanto, poderiam utilizar estratégias bastante parecidas com as de ADSM, se empregasse o algoritmo de ADSM para categorização das páginas.

10.2.3 Técnicas para Tolerância a Overheads

Nesta seção vamos discutir algumas técnicas que toleram *overheads* através da sua sobreposição com tempo de computação ou com outros *overheads*: *prefetching*, *multithreading*, *pre-diffing*, *forwarding* e *diff* dinâmico.

Prefetching tolera *overhead* de busca de dados, requisitando o dado antes que ele seja realmente necessário. Esses pedidos devem ser feitos com antecedência suficiente para que os dados cheguem antes do efetivo acesso. *Prefetching* já foi exaustivamente estudado no contexto de sistemas *hardware DSM*, como em [21, 59, 26, 14] por exemplo. *Prefetching* para sistemas *software DSM* recebeu muito menos atenção até o momento; somente os trabalhos [12, 7, 42, 15, 58] tratam desse assunto. Bianchini *et al.* [12] propuseram a primeira técnica para sistemas software DSM, a qual usa invalidações para guiar as ações de *prefetching*. No trabalho [15], Bianchini *et al.* estudaram uma técnica adaptativa que leva em consideração as falhas e os *strides* de acesso às páginas. Ambos os trabalhos consideram técnicas baseadas apenas na execução das aplicações, i.e., nenhum suporte de compilação ou programação é assumido. Já o trabalho de Mowry *et al.* [58] considera *prefetching* controlado por software, onde chamadas para rotinas de *prefetching* são inseridas no código da aplicação. A técnica LAP é baseada, tal qual as técnicas de *prefetching*, em antecipar ações de coerência através de previsões. A diferença entre LAP e *prefetching* é que em LAP é o processador que possui a versão atualizada do dado que a envia com antecedência, enquanto que em *prefetching* é o processador que vai

utilizar o dado que requisita a versão mais atualizada com antecedência.

Multithreading tolera *overheads* de comunicação ou coerência através de trocas de contexto em momentos que seriam de espera, mas requer a existência de um conjunto grande o suficiente de *threads* ativadas no nó. Assim como *prefetching*, a técnica *multithreading* já foi implementada tanto em sistemas *hardware DSM* (e.g., [70, 5, 6]) quanto em sistemas *software DSM* (e.g., [30, 74]). O trabalho de Thitikamol e Keleher [74] estuda o impacto de *multithreading* no sistema CVM. Nesse estudo, acontece uma troca de contexto toda vez que uma operação na rede de interconexao é necessária. Como resultados, os autores mostraram que *multithreading* pode melhorar o desempenho das várias aplicações com um pequeno número de *threads*. Os sistemas AEC, AEC-light e AEC-bind se baseiam na execução de uma única *thread* por processador. O uso de *multithreading* nesses sistemas, entretanto, teria menor impacto no desempenho, visto que grande parte do tempo em que um processador está bloqueado é usado para adiantar operações de coerência como criação de *twins* e *diffs*, e aplicação de *diffs*.

Forwarding tolera o *overhead* de falhas de acesso através de envio de atualizações antecipadas. O trabalho de Trancoso e Torrelas [75] propôs a técnica de *forwarding* para evitar falhas dentro de seções críticas. Esta técnica é bastante parecida com a técnica LAP. O trabalho de Trancoso e Torrellas, entretanto, considera *forwarding* apenas para os casos em que há espera pelo *lock*, é baseado em sistemas *hardware DSM* e não faz uma análise do ganho de desempenho dessa técnica isoladamente. O trabalho de Koufaty *et al.* [49] estuda o envio antecipado de atualizações do produtor direto para o consumidor de cada dado, tal como LAP global. No entanto, a análise de qual processador vai consumir o dado é feita unicamente pelo compilador, o que limita o uso desta técnica a aplicações com padrões de compartilhamento estaticamente determináveis.

Pre-diffing tolera o *overhead* de criação de *diffs* através da sobreposição deste com *overheads* de sincronização e comunicação. O trabalho de Swanson [73], por exemplo, implementa *pre-diffing* em Quarks. Numa operação *release* um processador tem que enviar uma série de *diffs* para outros processadores, Quarks permite, então, que a geração de alguns *diffs* seja sobreposta ao envio (não-bloqueante) e espera por *acknowlegments* de outros *diffs*. Outro exemplo de uso de *pre-diffing* é o trabalho de Zhou *et al.* [76], que avalia quatro protocolos LRC no multicomputador Intel Paragon. Dois desses protocolos usam o processador de comunicação da máquina para gerar *diffs* fora do caminho crítico dos processadores de computação. O sistema NCP_2 faz o mesmo, mas utiliza os seus controladores de protocolo para tal. Os protocolos de AEC e AEC-light também fazem uso de *pre-diffing*, mas não utilizam qualquer suporte de hardware para tal, além de explorarem o tempo de espera pela sincronização para criação antecipada de *diffs*.

Diffs Dinâmicos foram propostos no sistema NCP_2. O controlador de protocolos do sistema monitora o barramento de memória para observar todas as escritas a dados compartilhados. Dessa forma, o controlador pode sobrepor a criação de *diffs* com computação útil, além de eliminar a necessidade de criação de *twins* e de comparações de páginas. AEC e AEC-light poderiam se beneficiar bastante de *diffs* dinâmicos para eliminar, reduzir e/ou esconder os *overheads* da criação de *twins* e *diffs*.

Vários trabalhos estudaram a combinação de diferentes técnicas de tolerância a *overheads*, e.g. [23, 33, 14, 12, 58]. *Prefetching* e *multithreading* foram combinados em [33, 14, 58]. O trabalho de Castro e Amorim [23] apresenta um modelo de máquina de estados para identificar de forma bastante precisa o padrão de compartilhamento das páginas da aplicação. Com essas informações, os autores implementaram *forwarding*, *prefetching* e *pre-diffing* de forma combinada no sistema TreadMarks. Os sistemas propostos nesta tese também utilizam mais de uma

técnica de tolerância a overheads, *pre-diffing* e LAP.

Capítulo 11

Conclusões e Trabalhos Futuros

O objetivo dessa tese foi estudar as vantagens oferecidas por LAP em sistemas *software DSM*. Nós atingimos esse objetivo através do desenvolvimento de um conjunto de três sistemas *software DSM* baseados em LAP. Os três sistemas desenvolvidos exploram LAP de forma diferente e, dentro do paradigma de memória compartilhada, utilizam modelos de programação distintos. Tais modelos, embora mais complexos que o modelo de programação tradicional nos permitiram estudar a implicação do modelo de programação na complexidade e no desempenho do protocolo de coerência empregado.

Através de uma análise detalhada da execução de um conjunto representativo de seis aplicações científicas, mostramos que LAP acerta suas previsões com alta precisão. O uso de atualizações seletivas permite uma redução razoável no *overhead* de busca de dados compartilhados acessados dentro de uma seção crítica e tem efeito direto no tempo de espera pela seção crítica e no *overhead* de barreiras. Mostramos também que não só as atualizações seletivas de LAP permitem ganhos de desempenho, mas algumas características dos protocolos de coerência, próprias para o suporte de LAP, como criação antecipada de *twins* e armazenamento combinado de *diffs*, permitem grandes reduções na quantidade de dados transferida e no número de falhas de acesso geradas.

Com relação às implicações do modelo de programação na complexidade do pro-

tocolo de coerência, mostramos que a restrição de que todos os dados devem ser acessados dentro de seções críticas permite o emprego de LAP de forma mais agressiva (através do uso da técnica de afinidade global) e permite que o protocolo de coerência empregado seja mais simples porque não envolve falhas de acesso fora de seções críticas. O uso de associação explícita de variável de sincronização com dado compartilhado permite que o protocolo utilize o objeto como unidade de coerência. O uso do objeto torna o protocolo de coerência ainda mais simples, visto que a única operação de coerência necessária é o envio do objeto nas operações de sincronização.

O modelo de programação também tem efeito direto no desempenho dos sistemas, mas esse efeito é diferente do esperado intuitivamente. O aumento da complexidade do modelo não necessariamente leva a ganhos de desempenho, conforme mostra a comparação de desempenho entre os sistemas AEC e AEC-light. Por outro lado, a comparação entre AEC-light e AEC-bind mostra que a associação explícita entre variáveis de sincronização e dados compartilhados tem efeito bastante positivo no desempenho das aplicações. Comparando os nossos extremos em complexidade do modelo de programação, AEC e AEC-bind, observamos que a complexidade de programação leva a ganhos significativos de desempenho.

Embora os sistemas AEC, AEC-bind e AEC-light tenham se mostrado bastante eficientes se comparados com o sistema TreadMarks, aplicações com granularidade fina de acesso aos dados compartilhados têm desempenho superior em TreadMarks. O uso do modelo LRC, aliado ao padrão de acesso aos dados permite que, em TreadMarks, as ações tomadas pelo protocolo de coerência num acesso a um determinado dado compartilhado adiantem ações de coerência para os outros dados contidos na mesma página.

Baseados nesses resultados e observações, concluímos que a técnica LAP merece a atenção dos projetistas de sistemas *software DSM*, uma vez que pode alcançar bom desempenho sem a necessidade de hardware específico ou suporte de compiladores.

Além disso, concluímos que a melhor escolha de modelo de programação depende do tempo disponível para o desenvolvimento do sistema *software DSM* e da experiência dos programadores das aplicações. Sistemas mais difíceis de programar são mais simples (porque envolvem o tratamento de uma quantidade menor de eventos) e, dependendo do modelo empregado, têm melhor desempenho final. Na verdade, acreditamos que, na maioria dos casos, a complexidade de programação adicional imposta pelo modelo só é justificada pelos correspondentes ganhos de desempenho no caso de AEC-bind.

Futuramente, pretendemos comparar o desempenho dos sistemas propostos nessa tese com outros sistemas DSM, tais como Midway e ADSM. Um outro trabalho que pretendemos realizar envolve o estudo de como a interface de comunicação do sistema computacional tem efeito na estrutura e no desempenho dos nossos sistemas *software DSM*. Finalmente, planejamos re-avaliar a nossa comparação entre modelos de programação, considerando também suporte de compilação que facilite a programação em modelos complexos.

Referências Bibliográficas

[1] S. V. Adve, A. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 26–37, February 1996.

[2] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] S. V. Adve and M. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[4] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[5] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[6] R. Alverson *et al.* The Tera Computer System. In *Proceedings of the International Conference on Supercomputing*, pages 1–16, June 1990.

[7] C. Amza, A. Cox, K. Rajamoni, and W. Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–99, June 1997.

[8] G. R. Andrews. *Concurrent Programming Principles and Practice.* The Benjamin/Cummings Publishing Company, 1991.

[9] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[10] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, 1991.

[11] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the '93 CompCon Conference*, February 1993.

[12] R. Bianchini, L. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 198–209, October 1996.

[13] R. Bianchini, T. J. LeBlanc, and J. E. Veenstra. Categorizing Network Traffic in Update-Based Protocols on Scalable Multiprocessors. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 142–151, April 1996.

[14] R. Bianchini and B.-H. Lim. Evaluating the Performance of Multithreading and Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing, special issue on Multithreading for Multiprocessors*, 37(1):83–97, August 1996.

[15] R. Bianchini, R. Pinto, and C. L. Amorim. Data Prefetching for Software DSMs. In *Proceedings of the International Conference on Supercomputing*, pages 385–392, July 1998. Extended version published as TR ES-463/98. COPPE Systems Engineering, Federal University of Rio de Janeiro, March 1998.

[16] A. Bilas and J. P. Singh. The Effects of Communication Parameters on End Performance of Shared Virtual Memory Clusters. In *Proceedings of Supercomputing'97*, November 1997.

[17] R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 115–124, May 1990.

[18] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[19] B. Buck and P. Keleher. Locality and Performance of Page- and Object-Based DSMs. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 687–693, March 1998.

[20] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, February 1992.

[21] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[22] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[23] M. C. Castro and C. L. Amorim. Avaliação do Potencial de Técnicas Adaptativas Conjugadas para Software DSMs. In *Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, Setembro 1998.

[24] Convex Computer Corp. *Convex Exemplar Architecture*, November 1993.

[25] Cray Research, Inc. *CRAY T3D System Architecture Overview*, September 1993.

[26] F. Dahlgren and P. Stenström. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):385–398, April 1996.

[27] S. Dwarkadas, P. Keleher, A. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 144–155, May 1993.

[28] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analysing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 210–221, October 1996.

[29] P. Figueiredo. Exploitation of Parallelism in Seismic Migration. Master's thesis, University of Illinois at Urbana-Champaign, April 1995.

[30] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 201–213, November 1994.

[31] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[32] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

[33] A. Gupta, J. Henessy, K. Gharachorloo, T. Mowry, and W. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[34] E. Hagersten, A. Landin, and S. Haridi. DDM—A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[35] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler. In *Proceedings of Supercomputing'95*, December 1995.

[36] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. In *Proceedings of the Second*

IEEE Symposium on High-Performance Computer Architecture, pages 14–25, February 1996.

[37] L. Iftode, J. P. Singh, and K. Li. Irregular Applications under Software Shared Memory. Technical Report TR-510-96, Princeton University, February 1996.

[38] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 122–133, May 1996.

[39] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: A Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 122–133, June 1996.

[40] D. V. James. The Scalable Coherent Interface: Scaling to High-Performance Systems. In *Proceedings of CompCon '94*, pages 64–71, November 1994.

[41] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 213–228, December 1995.

[42] M. Karlsson and P. Stenström. Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared Memory Systems. *Journal of Parallel and Distributed Computing*, 43(7):79–93, July 1997.

[43] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.

[44] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[45] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[46] C. Koelbel, D. Loveman, R. Schreider, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1996.

[47] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Ciernak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 157–169, May 1997.

[48] L. Kontothanassis, M. L. Scott, and R. Bianchini. Lazy Release Consistency for Hardware-Coherent Multiprocessors. In *Proceedings of Supercomputing'95*, December 1995.

[49] D. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. *IEEE Transactions on Parallel as Distributed Systems*, 29(2):1250–1264, December 1996.

[50] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[51] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1991.

[52] J. Laudon and D. Lenoski. The SGI Origin: A CC-NUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[53] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[54] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume II, pages 94–101, August 1988.

[55] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[56] S. Lucci, I. Gertner, A. Gupta, and U. Hegde. Reflective-Memory Multiprocessor. In *Proceedings of the 28th Hawaii International Conference on System Science*, pages 85–94, January 1995.

[57] L. R. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, pages 289–299, February 1998.

[58] T. Mowry, C. Chan, and A. Lo. Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory. In *Proceedings of the 4th*

IEEE Symposium on High-Performance Computer Architecture, pages 300–309, February 1998.

[59] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[60] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[61] H. S. Sandhu, B. Gamsa, and S. Zhou. The Shared Region Approach to Software Cache Coherence on Multiprocessors. In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, pages 229–238, July 1993.

[62] D. J. Scales and K. Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 157–169, October 1997.

[63] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1996.

[64] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–307, October 1994.

[65] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, October 1996.

[66] C. B. Seidel, R. Bianchini, and C. L. Amorim. The Affinity Entry Consistency Protocol. In *Proceedings of the 1997 International Conference on Parallel Processing*, pages 65–78, August 1997.

[67] C. B. Seidel, R. Bianchini, and C.L. Amorim. Técnicas para Previsão Dinâmica do Próximo *Acquirer* em *Software DSMs*. In *Anais do Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, pages 203–212, Agosto 1996.

[68] C. B. Seidel, R. Bianchini, and C.L. Amorim. Avaliando a Técnica de Previsão Dinâmica da Passagem de *Locks* em Sistemas DSM. In *Anais do Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, pages 301–316, Outubro 1997.

[69] C. B. Seidel, R. Bianchini, and C.L. Amorim. Projetando e Avaliando Sistemas Baseados em *Entry Consistency* e *Lock Acquirer Prediction*. In *Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, Setembro 1998.

[70] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the 1978 Interbational Conference on Parallel Processing*, 1978.

[71] W. E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1997 USENIX Windows/NT Workshop*, pages 95–106, August 1997.

[72] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 170–184, October 1997.

[73] M. Swanson, L. Stroller, and J. B. Carter. Making Distributed Shared Memory Simple, Yet Efficient. In *Proceedings of the 3rd International Workshop on High-Level Parallel Programming Models and Supportive Envvironments*, pages 207–216, March 1998.

[74] K. Thitikamol and P. Keleher. Per-Node Multithreading and Remote Latency. *IEEE Transactions on Computers*, 47(4):414–426, 1998.

[75] P. Trancoso and J. Torrellas. The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages 79–86, August 1996.

[76] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 75–88, October 1996.

[77] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proceedings of the 6th Symposium on Principles and Practice of Parallel Programming*, pages 193–205, June 1997.