ESTRATÉGIAS DE SOFTWARE E HARDWARE PARA OTIMIZAÇÃO DE SISTEMAS DE MEMÓRIA COMPARTILHADA DISTRIBUÍDA

Raquel Coelho Gomes Pinto

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

Aprovada por:

Prof. Ricardo Bianchini, Ph.D.

Prof. Cláudio Luis de Amorim, Ph.D.

Prof. Edil Severiano Tavares Fernandes, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Sérgio Takeo Kofuji, D.Sc.

Prof. Wagner Meira Junior, Ph.D.

RIO DE JANEIRO - RJ, BRASIL

DEZEMBRO DE 2001

PINTO, RAQUEL COELHO GOMES

Estratégias de Software e Hardware para Otimização de Sistemas de Memória Compartilhada Distribuída [Rio de Janeiro] 2001

X, 72p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2001)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Sistemas Operacionais

2 - Memória Compartilhada Distribuída

I. COPPE/UFRJ II. Título (série)

Agradecimentos

Primeiramente gostaria de agradecer ao meu orientador Ricardo Bianchini por sua compreensão, paciência e amizade. Apesar dele ter estado longe no final da tese, nunca deixou de me incentivar, e sempre respondeu prontamente às minhas dúvidas com ótimas sugertões.

Gostaria de agradecer ao meu outro orientador Claudio Luis de Amorim pelo apoio e estímulo.

Agradeço ao Lauro por estar todos os dias ao meu lado no laboratório me ajudando com infindáveis *bugs* e por sua amizade inquestionável. A Cristiana por estar sempre pronta a me ajudar. E ao resto do pessoal das reuniões de quarta-feira (que passaram para sexta-feira e que agora infelizmente não existem mais).

Gostaria de agradecer especialmente a minha família pelo apoio e amor sempre presentes. Ao meu marido pela compreensão e por aturar minhas rabugices nos momentos de pressão devido a "*deadlines*". Aos meus pais por estarem sempre prontos a me ajudar, principalmente com a minha preciosidade que é a minha filha.

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

ESTRATÉGIAS DE SOFTWARE E HARDWARE PARA OTIMIZAÇÃO DE SISTEMAS DE MEMÓRIA COMPARTILHADA DISTRIBUÍDA

Raquel Coelho Gomes Pinto

Dezembro/2001

Orientadores: Ricardo Bianchini
Cláudio Luis de Amorim
Programa: Engenharia de Sistemas e Computação

Um dos maiores custos remanescentes em sistemas de memória compartilhada distribuída com coerência de dados mantida por *software* (*software DSMs*) é a alta latência de acesso a dados remotos. Sendo assim, esta tese apresenta o estudo e o desenvolvimento de estratégias de *software* e *hardware* que toleram e/ou reduzem a latência de acesso a dados remotos em *software DSMs*. Em termos de estratégias de *software* nós propomos uma nova técnica de pré-busca, chamada *Adaptive++*, que visa otimizar aplicações paralelas, utilizando o histórico das falhas de acesso à memória para guiar as pré-buscas. Nós avaliamos *Adaptive++* isoladamente e quando combinada a outras duas técnicas de tolerância a latência: adaptação entre protocolos único-escritor e múltiplos-escritores, e coerência baseada no envio seletivo de atualizações. Nossos resultados mostram que *Adaptive++* reduz o tempo de acesso a dados remotos das aplicações regulares em até 62% em um *cluster* de 8 PCs. Aplicações irregulares apresentam reduções mais modestas. Os resultados de desempenho mostram que *Adaptive++* consegue oferecer um aumento de *speedup* de até 35%. Em adição, os resultados mostram que a combinação das três estratégias alcança ganhos de *speedup* de até 118%. Em termos de estratégias de *hardware*, nós propomos a técnica de diff dinâmico que utiliza o controlador de protocolos projetado com o nosso auxílio para o sistema NCP_2. O controlador monitora o barramento de memória, e marca todas as escritas a dados compartilhados em vetores de *bits*. Posteriormente, estes vetores de *bits* são inspecionados para determinar as modificações feitas a páginas (diffs). Mostramos que esta estratégia reduz os custos de coerência em até 92% em um *cluster* de 4 PCs. Esta redução é traduzida em ganhos de desempenho de até 12%.

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

SOFTWARE AND HARDWARE STRATEGIES FOR OPTIMIZING DISTRIBUTED SHARED-MEMORY SYSTEMS

Raquel Coelho Gomes Pinto

December/2001

Advisors: Ricardo Bianchini

Cláudio Luis de Amorim

Department: Computing and Systems Engineering

One of the most serious overheads that remains in software-based distributed shared-memory systems (software DSMs) is the high latency of remote data accesses. Thus, this thesis presents the study and development of software and hardware strategies that can tolerate and/or reduce remote access latencies in software DSMs. In terms of software strategies, we propose a novel prefetching technique, called Adaptive++, that optimizes parallel applications by using the past history of memory access faults to guide prefetching. We evaluate Adaptive++ in isolation and when combined with two other latency-tolerance techniques: adaptation between single-writer and multiple-writer protocols, and selective update-based coherence. Our results show that Adaptive++ reduces the remote data access overheads of regular applications by as much as 62% on a cluster of 8 PCs. Irregular applications exhibit more modest reductions. The performance results show that Adaptive++ can provide speedup improvements as significant as 35%. Furthermore, the results show that the combination of the three techniques can achieve speedup improvements of up to 118%. In terms of hardware strategies, we propose the dynamic diff technique that uses the protocol controller we helped design for the NCP_2 system. The controller snoops the memory bus and records all shared data writes in bit vectors. These bit vectors can later be inspected to determine the modifications made to pages (diffs). We show that this strategy can reduce coherence overheads by up to 92% on a cluster of 4 PCs. These gains translate into overall performance improvements of up to 12%.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Os dois modelos de programação paralela mais difundidos hoje em dia são memória compartilhada e passagem de mensagens. Apesar de ambos os modelos apresentarem um alto grau de dificuldade de programação, o primeiro é amplamente reconhecido como sendo mais "amigável" do que o segundo. Entretanto, desenvolver uma implementação eficiente da memória compartilhada é muito mais difícil. Basicamente, o modelo de programação de memória compartilhada pode ser implementado diretamente em *hardware*, em *software*, ou com alguma combinação dos dois. Sistemas de memória compartilhada distribuída com coerência de dados mantida por *software*, denominados *software DSMs*, oferecem a facilidade de programação do modelo de memória compartilhada com o baixo custo das arquiteturas distribuídas. No entanto, são poucas as classes de aplicações que alcançam um bom desempenho nesses sistemas, devido à alta taxa de comunicação e ao custo gerado pela manutenção da coerência dos dados.

Software DSMs baseados em modelos de consistência relaxada podem reduzir esses custos atrasando e/ou restringindo a comunicação e as transações de coerência tanto quanto possível. *Software DSMs* baseados em consistência relaxada e que permitem múltiplos escritores tentam reduzir a comunicação e os custos de coerência ainda mais permitindo que dois ou mais processadores modifiquem concorrentemente suas cópias locais de dados compartilhados, e combinem essas modificações em operações de sincronização. Esse mecanismo diminui o efeito negativo do falso compartilhamento em sistemas cuja unidade de coerência é grande, como os *software DSMs* que utilizam a página de memória virtual como unidade de coerência.

O problema é que os custos de desempenho desses sistemas otimizados ainda são muito altos. Em particular, um dos maiores custos remanescentes em *software*

DSMs é a alta latência de acesso a dados remotos, que ocorre no caminho crítico da computação. Logo, nesta tese avaliamos o desempenho de estratégias para a otimização de sistemas DSM nos atendo ao problema da alta latência de acesso a dados remotos. Nossos estudos estão focalizados em sistemas DSM que possuem as seguintes características: a unidade de coerência é a página, oferecem suporte para múltiplos escritores, e utilizam um modelo de consistência de memória relaxado. As técnicas consideradas são as seguintes:

- *Prefetching* (pré-busca): onde o dado remoto é buscado com antecedência, isto é, antes de ser efetivamente necessário.
- Envio de atualizações: onde o dado é enviado com antecedência pelo último escritor.
- Adaptação entre protocolos único-escritor e múltiplos-escritores: no primeiro, quando um nó necessita atualizar um dado compartilhado ele o requisita ao último escritor. Enquanto que no caso de múltiplos escritores, é necessário buscar as modificações realizadas pelos vários escritores e combiná-las.
- Diff Dinâmico: onde a codificação das modificações feitas aos dados compartilhados (chamada de "diff") é gerada com o auxílio de *hardware*.

Todas essas técnicas foram desenvolvidas com o intuito de diminuir a latência de acesso a dados remotos de *software DSMs* sendo todas elas estratégias executadas em tempo de execução. A última técnica, no entanto, depende do uso de um **controlador de protocolos** programável, que se baseia em uma placa PCI comercial que acoplamos a uma lógica customizada. Logo, classificamos essas técnicas em estratégias de *software*, que incluem as três primeiras técnicas, e estratégias de *hardware*. Essas estratégias foram implementadas no *software DSM* TreadMarks, mas são suficientemente genéricas para serem aplicadas em qualquer *software DSM* que apresente as características descritas acima. No entanto, no caso de *software DSMs* baseados em *homes* é necessário fazer alguma adaptação.

1.1 Estratégias de *Software*

Nesta seção discutimos cada uma das técnicas de *software* que estudamos no decorrer da dissertação.

1.1.1 Estratégia de Pré-busca

Em geral, não é simples implementar técnicas de pré-busca que sejam efetivas para *software DSMs*. As técnicas relativamente simples propostas para *hardware DSMs* [44], por exemplo, não são úteis para *software DSMs*. Sumariamente, o uso eficiente de estratégias de pré-busca em *software DSMs* pode ser difícil de atingir devido principalmente a duas razões: (1) Pode não ser fácil prever os acessos futuros a dados; e (2) Pré-buscas geram grandes atrasos quando solicitadas desnecessariamente.

Neste trabalho apresentamos uma nova técnica de pré-busca, *Adaptive++* [9] que apenas realiza pré-buscas quando as falhas de acesso podem ser previstas com razoável segurança. Mais especificamente, *Adaptive++* é uma estratégia de pré-busca que melhora o desempenho de aplicações regulares, sem a necessidade da intervenção de usuários ou compiladores sofisticados. A técnica utiliza o histórico das falhas de acesso à memória para se adaptar a um dos seguintes modos de operação: `repetição-de-fase` ou `repetição-de-stride`. *Adaptive++* não realiza pré-buscas quando a aplicação não apresenta um desses dois comportamentos, apresentando portanto um comportamento irregular.

Utilizamos como sistema base para a implementação desta estratégia o *software DSM* TreadMarks [6]. Nossos resultados experimentais mostram que a nossa técnica reduz significativamente a latência de acesso a dados remotos das aplicações regulares. Essas aplicações alcançam uma redução no tempo de acesso a dados de até 62%. *Adaptive++* também melhora o desempenho de aplicações que não são estritamente regulares, mas que apresentam períodos de regularidade. Essas aplicações exibem reduções mais modestas da latência de acesso a dados. Em termos de desempenho geral, nossos resultados demonstram que *Adaptive++* consegue oferecer um aumento de *speedup* de até 35% em 8 processadores.

Adaptive++ não é a primeira técnica de pré-busca em tempo de execução proposta na literatura para *software DSMs*. A nossa técnica (B+) [7], a de Karlsson and Stenström (KS) [21], e a de Amza *et al.* (*Dynamic Aggregation*) [4] são as propostas anteriores mais conhecidas. A técnica B+ foi a primeira proposta de pré-busca em tempo de execução para *software DSMs*; a partir dos estudos realizados com B+ iniciamos o desenvolvimento de *Adaptive++*. A técnica KS é similar à nossa em certos aspectos, mas não pode ser comparada diretamente à *Adaptive++*. Em [9] apresentamos uma comparação simulada de *Adaptive++* com as estratégias B+ e *Dynamic Aggregation*, mostrando que a nossa técnica oferece desempenho igual ou

melhor ao apresentado por B+. Além disso, *Adaptive++* tem um melhor desempenho do que *Dynamic Aggregation* para as aplicações regulares, enquanto que para as aplicações não-regulares os desempenhos dessas duas técnicas são comparáveis. Essas comparações demonstram a competitividade em desempenho de *Adaptive++*, assim como a sua habilidade de otimizar um conjunto maior de aplicações.

1.1.2 Estratégia de Envio de Atualizações

A estratégia de envio de atualizações está relacionada ao tipo de protocolo usado na manutenção da coerência dos dados compartilhados em *software DSMs*. Os dois tipos de protocolos mais comumente utilizados são aqueles baseados em invalidações, e aqueles baseados em atualizações. Os protocolos baseados em atualizações geralmente transferem mais dados do que o necessário. No entanto, eles conseguem alcançar uma melhora de desempenho com relação aos protocolos baseados em invalidações quando o padrão de compartilhamento é tal que a maioria das atualizações são realmente utilizadas. Os principais padrões de compartilhamento que se beneficiam de um protocolo baseado em atualizações são os dados migratórios acessados dentro de seções críticas, e dados acessados a partir de um compartilhamento do tipo produtor/consumidor(es).

Utilizamos como implementação desta estratégia o sistema ADSM [30]. Tendo em vista que o envio de atualizações em ADSM depende da estratégia de adaptação único-escritor/múltiplo-escritores, nós implementamos uma versão de ADSM que realiza o envio de atualizações sem utilizar a adaptação entre único-escritor/múltiplo-escritores. Desta forma conseguimos identificar os ganhos oferecidos apenas pelo envio de atualizações. Nossos resultados mostraram que esta estratégia alcança ganhos de desempenho de até 29%, reduzindo o número de falhas de acesso em até 56%.

1.1.3 Estratégia de Adaptação Entre Único-Escritor e Múltiplos-Escritores

Os protocolos que permitem a existência de múltiplos escritores concorrentes diminuem o efeito do falso compartilhamento (custo de comunicação e manutenção de coerência) permitindo que dois ou mais processadores alterem concorrentemente suas cópias locais de dados compartilhados e combinem essas modificações nos pontos de sincronização. Entretanto, este tipo de protocolo gera atrasos adicionais no

acesso aos dados que não estão sujeitos a falso compartilhamento, pois é necessário detectar, guardar e combinar as modificações realizadas a dados compartilhados. Esses atrasos adicionais podem ser eliminados, permitindo apenas um escritor para os dados que não sofrem falso compartilhamento. Logo, a terceira estratégia é uma adaptação entre um protocolo que permite múltiplos escritores (usado quando existem vários escritores para um dado) e um que só permite um escritor de cada vez.

Nós utilizamos como implementação desta estratégia a adaptação proposta para o sistema ADSM denominada *Single/Multiple Writer Adaptation* (SMA). A estratégia SMA demonstrou ser a que consegue atingir a maioria das aplicações apresentando ganhos de desempenho de até 56%.

1.1.4 Combinação das Estratégias Anteriores

As estratégias apresentadas atacam individualmente diferentes classes de aplicações. Desta forma é necessário identificar a classe da aplicação em uso para em seguida escolher a estratégia que irá oferecer um melhor desempenho. Sendo que ainda existem aplicações que podem se beneficiar de mais do que uma estratégia. Logo, nesta tese apresentamos o estudo da combinação de *Adaptive++* com a estratégia de envio de atualizações e a de adaptação entre protocolos múltiplos-escritores/único-escritor.

Decidir como as estratégias devem ser combinadas não é uma tarefa trivial tendo em vista os problemas que cada uma pode gerar. Por exemplo, utilizar a estratégia de envio de atualizações para dados alterados por vários escritores pode além de não melhorar o desempenho da aplicação, aumentar seu tempo de execução. Os escritores teriam que gerar diffs antes deles serem requisitados, podendo assim aumentar o número de diffs gerados, e o processador ao receber as atualizações, tem que ter certeza de que todos os diffs necessários foram recebidos. Logo, todos os escritores de um determinado dado têm que enviar atualizações para o mesmo conjunto de processadores de forma que os receptores das atualizações consigam aplicar todas as modificações necessárias antes de acessar o dado.

A nossa proposta de combinação escolhe em tempo de execução a melhor estratégia para o padrão de acessos corrente da aplicação em execução. Combinamos essas estratégias da seguinte forma:

- Os dados alterados por um único escritor por vez são tratados por um protocolo

que não utiliza a técnica de *twinning/diffing* (a ser descrita no capítulo 2), de forma que se um processador quiser acessar esses dados, eles são transferidos integralmente para o processador solicitante.

- Os dados alterados por um único escritor por vez, e que são acessados de acordo com o padrão de compartilhamento migratório dentro de seção crítica ou produtor/consumidor(es), são enviados através de atualizações nos pontos de sincronização correspondentes.

- Os dados alterados por vários escritores concorrentemente são buscados antecipadamente através da pré-busca de diffs.

No momento, a literatura contém trabalhos onde pré-busca e o envio de atualizações são estudados separadamente (e.g. [9, 30, 40]) e onde essas técnicas são comparadas diretamente (e.g. [5]). Entretanto, a combinação eficiente dessas técnicas foi preliminarmente simulada apenas em [14] até onde sabemos. Nosso trabalho pretende ser bem mais profundo e detalhado no estudo dessa combinação que em [14].

Avaliamos a combinação dessas estratégias mostrando inicialmente os ganhos alcançados por cada uma e em seguida o ganho apresentado pela combinação das estratégias. Algumas aplicações se beneficiam de apenas uma das estratégias de forma que a combinação depende apenas do ganho alcançado pela estratégia correspondente. No entanto, existem algumas aplicações que exploram as características de mais do que uma estratégia e a combinação consegue somar os ganhos oferecidos individualmente pelas estratégias. A combinação das estratégias alcançou ganhos de desempenho de até 53%.

1.2 Estratégias de *Hardware*

A última estratégia denominada Diff Dinâmico utiliza um controlador de protocolos que monitora o barramento de memória guardando em vetores de *bits* quais dados compartilhados foram escritos. Esses vetores de *bits* posteriormente são inspecionados para determinar os diffs. Diferentemente, os *software DSMs* geralmente utilizam o mecanismo de *twinning/diffing* para coletar as modificações feitas aos dados compartilhados. Neste mecanismo as páginas compartilhadas são inicialmente protegidas contra escrita de forma que, a primeira escrita a uma página gera uma

violação de acesso. No tratamento desta violação, é criado uma cópia da página, denominada *twin*, e a escrita à página original é liberada. Quando é necessário coletar as modificações feitas à página, o *twin* é comparado palavra por palavra à página modificada, criando-se assim o diff correspondente.

A combinação de um protocolo de coerência baseado em páginas (com consistência relaxada e protocolos múltiplos escritores) e o controlador de protocolos produz um sistema híbrido, onde combinamos as características de DSMs implementados em *hardware* com aqueles implementados em *software*. Este sistema híbrido foi inicialmente proposto em [7], onde avaliamos em simulação várias técnicas de tolerância a latência que utilizam o nosso controlador de protocolos incluindo a técnica de diff dinâmico. Mostramos que o controlador de protocolos pode ser utilizado para executar tarefas básicas de comunicação e coerência, além de auxiliar técnicas de pré-busca aumentando seus ganhos. Posteriormente apresentamos em [34] uma avaliação preliminar do desempenho do diff dinâmico no contexto de um protótipo do controlador implementado como parte do projeto NCP_2.

Nossa avaliação da técnica de diff dinâmico é baseada na aplicação desta técnica ao sistema TreadMarks. Nossos resultados mostram que diff dinâmico reduz consistentemente os custos de criação de diffs de TreadMarks, alcançando ganhos de até 92%. No entanto, os custos de criação de diffs não são dominantes para o pequeno conjunto de aplicações analisado. Sendo assim, os ganhos obtidos pela técnica de diff dinâmico são traduzidos em ganhos de desempenho de até 12%.

Vários sistemas já foram propostos com o intuito de diminuir os custos relacionados com diffs. O sistema baseado em *homes* chamado GeNIMA [11] propôs o método de "diff direto", onde, durante a comparação do *twin* com a página requisitada, as palavras modificadas são enviadas diretamente para o *home* da página. Dessa forma os custos de empacotamento/desempacotamento, interrupção e aplicação de diffs são eliminados. No entanto, o diff direto pode aumentar substancialmente o número de mensagens necessárias para atualizar uma página no seu nó *home*. Outros *software DSMs* anteriores tentaram eliminar a computação de diffs e *twins*. O sistema baseado em *homes* AURC [20] se baseia no uso de uma cache *write-through* e da *interface* de rede SHRIMP [12], que monitora as escritas, para propagá-las automaticamente para os nós *home*. De forma similar, o sistema Cashmere [26] utiliza a característica de escrita remota em granularidade fina oferecida pela *interface* de rede DEC Memory Channel. No entanto, é utilizada uma instrumentação do

código para detectar as escritas que devem ser propagadas ao nó *home*, ao invés de ter essas escritas monitoradas por *hardware*. Todos os sistemas mencionados usam características específicas da *interface* de rede. Em contraste, o nosso sistema não depende de nenhuma característica especial.

Vários outros *software DSMs* baseados em páginas e que permitem múltiplos escritores já foram propostos, e devido à generalidade da nossa estratégia, diff dinâmico pode beneficiar todos esses sistemas.

1.3 Estratégias de *Software* vs. *Hardware*

Os nossos resultados para as técnicas de *software* (apresentados no capítulo 3) e *hardware* (capítulo 4) infelizmente não podem ser diretamente comparados. A razão disso é que o *hardware* de diff dinâmico e o sistema em que os experimentos com esta técnica foram executados não estão mais disponíveis, fazendo com que as nossas avaliações tenham sido realizadas sobre plataformas distintas. Sendo assim, a seguir fazemos apenas uma comparação qualitativa entre as estratégias.

A técnica SMA é aquela cujos objetivos mais se assemelham à técnica de diff dinâmico. SMA elimina os custos de criação e aplicação de diffs assim como os custos de geração de *twins* para as páginas modificadas por um escritor por vez. Enquanto que a técnica de diff dinâmico não atinge o custo de aplicação de diffs, mas por outro lado, reduz os custos de criação de diffs e de geração de *twins* independente do compartilhamento de dados. Além disso, as duas técnicas possuem o mesmo efeito colateral de diminuir o problema de poluição de cache que pode aparecer devido a criação de diffs e de *twins*. Tendo em vista as melhoras de desempenho alcançadas por SMA, acreditamos que diff dinâmico também extrairia bons resultados dessa plataforma. Além disso, devemos levar em consideração a combinação dessas técnicas. Uma vez que SMA não atinge as páginas múltiplo-escritores e consegue eliminar os custos relacionados a diffs, podemos utilizar adicionalmente a técnica de diff dinâmico para as páginas múltiplo-escritores.

As demais estratégias de *software* não têm uma relação direta com a técnica de diff dinâmico, mas podem explorar o uso do controlador de protocolos. Em [7] apresentamos um estudo simulado da utilização do controlador de protocolos combinado com técnicas de tolerância a latência. Mais particulamente, mostramos como o controlador de protocolos pode ser utilizado para acelerar uma técnica de

pré-busca, reduzindo principalmente a interferência que pré-buscas podem gerar em nós remotos. Dessa forma, existe uma indicação de que o controlador de protocolos possa melhorar os ganhos obtidos pela técnica *Adaptive++*.

1.4 Contribuições da Pesquisa

Esta tese contribuirá para o estado da arte em técnicas de tolerância à latência de acesso a dados remotos em *software DSMs* de quatro formas principais:

- Propondo e avaliando novas técnicas de *software*, tais como *Adaptive++*, que demonstra ser a melhor técnica de pré-busca em tempo de execução para *software DSMs* até onde sabemos.

- Propondo e avaliando técnicas baseadas em um suporte de *hardware* de baixo custo, o controlador de protocolos, cujo principal objetivo é reduzir os custos de criação de diffs.

- Demonstrando como estratégias de pré-busca, de envio de atualizações e de adaptação entre protocolos único-escritor e múltiplos-escritores podem ser combinadas para diminuir ainda mais a latência de acesso a dados remotos.

- Avaliando o desempenho do sistema ADSM em uma plataforma diferente daquela utilizada (SP2) quando o sistema foi proposto. O SP2 possui uma razão computação/comunicação menor do que a plataforma que utilizamos para os nossos experimentos. Isto quer dizer que o custo da comunicação no SP2 é comparativamente menor, e, portanto, as latências de *software DSMs* podem se tornar mais significativos na nossa plataforma (a ser descrita no capítulo 3). Isto sugere que o sistema ADSM deve alcançar maiores ganhos. No entanto, a utilização de um processador bem mais rápido diminui os custos relacionados a diffs e *twins*, indicando uma redução dos possíveis ganhos obtidos por ADSM.

1.5 Organização da Tese

O restante da tese está organizado da seguinte forma: no capítulo 2 apresentamos os conhecimentos básicos necessários para o entendimento do resto do trabalho: o comportamento de *software DSMs*, a descrição de técnicas utilizadas para esconder a latência de acesso a dados e uma descrição do sistema ADSM. No capítulo 3

apresentamos uma descrição das estratégias desenvolvidas em *software*, assim como os resultados obtidos. A descrição da técnica de diff dinâmico e seus resultados são mostrados no capítulo 4. Os trabalhos relacionados são descritos no capítulo 5. E as conclusões aparecem no capítulo 6.

Capítulo 2

Conhecimentos Básicos

Neste capítulo introduzimos os conhecimentos básicos necessários para um melhor entendimento da tese. Tendo em vista que *software DSMs* representam a base do nosso trabalho, inicialmente apresentamos as características básicas desses sistemas e seus principais problemas. Em seguida, apresentamos as principais técnicas usualmente empregadas para solucionar os problemas presentes em *software DSMs*. Por último descrevemos o sistema ADSM para o qual foram desenvolvidas algumas estratégias utilizadas nesta tese.

2.1 Software DSMs e Seus Custos

A figura 2.1 apresenta a estrutura geral de um sistema de memória compartilhada distribuída (*Distributed Shared Memory* - DSM), onde o *software* ou o *hardware* fornece a ilusão de um espaço de endereçamento único, alterando o comportamento das memórias que são fisicamente distribuídas. O foco desse trabalho é o estudo de sistemas DSM implementados em *software*. A implementação de DSM em *software* não é uma tarefa trivial, pois o gerenciamento dos dados compartilhados por *software*

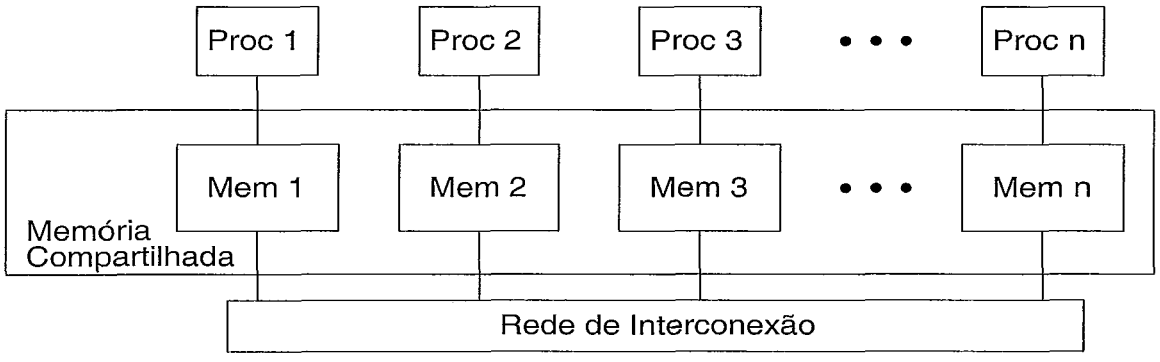

Figura 2.1: Estrutura geral de um sistema DSM.

deve ser realizado de forma a evitar a geração de custos excessivos. Muitos *software DSMs* atacam este problema tirando vantagem do *hardware* existente na maioria dos microprocessadores: os *bits* de proteção da memória virtual. Esse *hardware* é empregado na detecção de potenciais violações de coerência e na garantia da coerência a nível de página. Desta forma, a unidade de coerência passa a ser a página, isto é, utiliza-se uma granularidade alta de compartilhamento, podendo assim gerar o problema de falso compartilhamento de dados entre os processadores. Objetivando a minimização do impacto do falso compartilhamento, esses sistemas buscam garantir a consistência de memória apenas nos pontos de sincronização, e, além disso, permitem que múltiplos processadores escrevam concorrentemente na mesma página [13].

TreadMarks é um exemplo de um sistema que garante a consistência de forma relaxada, utilizando o modelo *Lazy Release Consistency* (LRC) [24], e que permite a existência de múltiplos escritores concorrentes. O algoritmo de LRC divide a execução do programa em intervalos e calcula um vetor de *timestamp* para cada intervalo. Este vetor descreve uma ordem parcial entre os intervalos de processadores distintos [1]. Em uma operação de aquisição de *lock*, o último processador que liberou o *lock* determina o conjunto de notificações de escrita (identificação das modificações feitas a dados compartilhados também chamadas de *write notices*) que o processador que está recebendo o *lock* precisa receber, isto é, o conjunto de modificações a dados compartilhados que precedem, na ordem parcial, a operação corrente de aquisição de *lock*. Ao receber as notificações, o processador altera então o estado da sua memória de acordo com as mesmas. Nenhuma ação é tomada na operação de liberação de *lock*. Um episódio de barreira pode ser visto como uma liberação de *lock* seguido por uma aquisição, executados por cada processador.

Em TreadMarks, o recebimento de uma notificação de escrita representa uma invalidação da página correspondente. A propagação das modificações feitas à página compartilhada é postergada até que o processador de posse do *lock* sofra uma falha de acesso à página. Essas modificações são determinadas através da utilização dos mecanismos de *twinning* e *diffing* descritos a seguir. Uma página é inicialmente protegida contra escrita, de forma que na primeira escrita a ela é gerada uma violação. Durante o tratamento desta violação, é feita uma cópia exata da página (um *twin*), e a escrita à cópia original da página é liberada. Quando a coleta das modificações é necessária, o *twin* e a versão corrente da página são comparados para

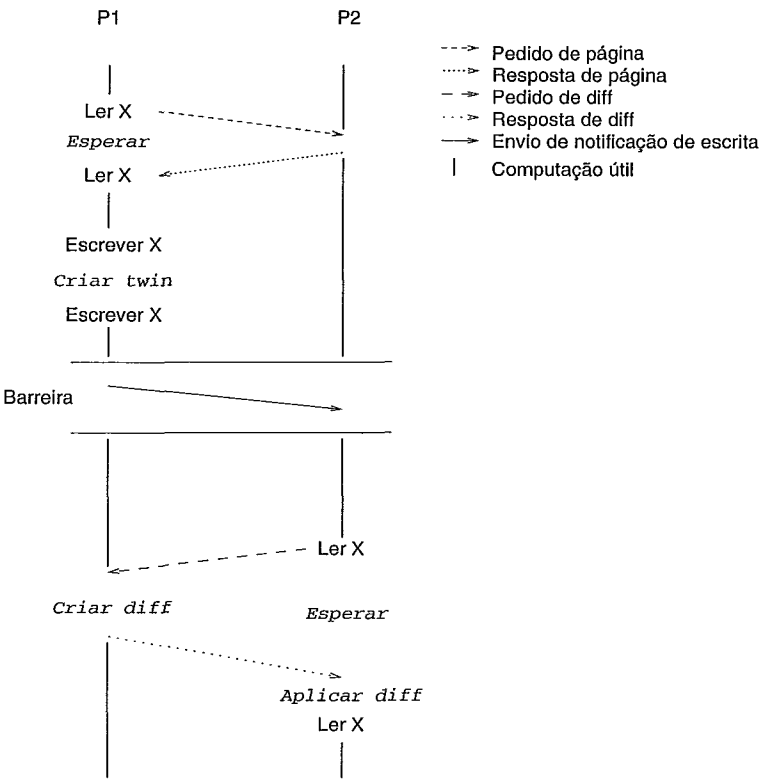

Figura 2.2: Coerência de dados em TreadMarks.

criar uma codificação das modificações (um *diff*). A utilização de *twins* e diffs em TreadMarks permite aos processadores modificar simultaneamente suas cópias locais de uma página compartilhada.

Desta forma, quando ocorre uma falha de acesso, o processador consulta a sua lista de notificações de escrita para descobrir quais os diffs necessários para atualizar a sua página. O processador então solicita os diffs correspondentes e espera que eles sejam criados, enviados e recebidos. Depois de receber todos os diffs requisitados, o processador que sofreu a falha os aplica à sua cópia desatualizada da página. Uma descrição mais detalhada de TreadMarks pode ser encontrada em [25].

Um visão simplificada das ações de coerência de TreadMarks é mostrada na figura 2.2. A figura mostra as ações tomadas por dois processadores (`P1` e `P2`) que compartilham a página onde está localizada a variável `X`. Inicialmente apenas `P2` possui uma cópia válida da página. De acordo com a figura, o primeiro acesso à página é uma leitura feita pelo processador `P1`, que resulta em uma violação de acesso e aciona o *trap handler* que busca a página inteira do nó 2. Depois que a página é recebida, ela é tornada válida e a instrução que causou a violação é re-executada. A seguir, o processador `P1` escreve na página, o que gera outra violação já que a página

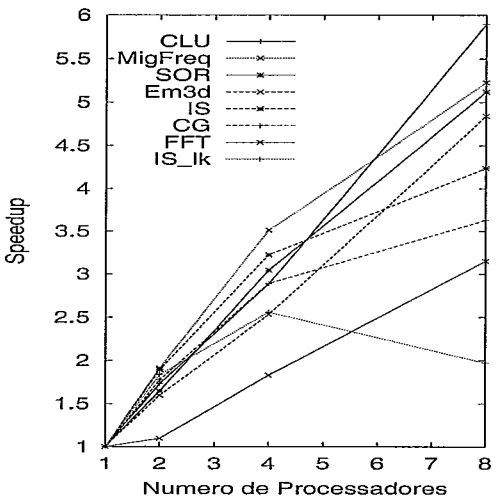

Figura 2.3: Speedup das aplicações sobre TreadMarks.

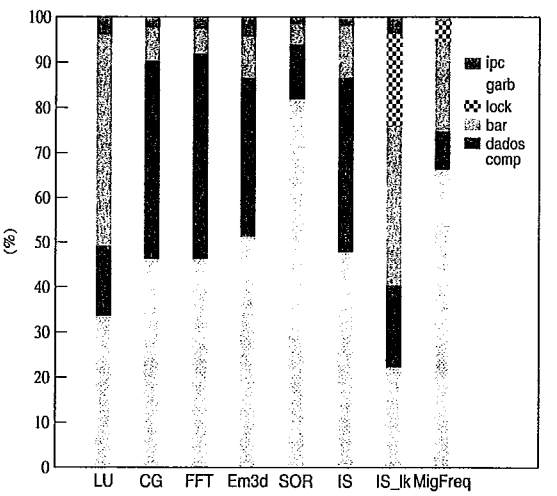

Figura 2.4: Tempo de execução das aplicações sobre TreadMarks em 8 processadores.

estava protegida contra escrita. Esta última violação resulta na criação de um *twin* da página, na atualização dos *bits* de proteção da página (permitindo a sua escrita), e na re-execução da instrução de escrita. Quando os dois processadores alcançam a barreira, o processador P2 recebe uma notificação de escrita como efeito da escrita realizada por P1 na página que contém a variável X. (Note que a seta de notificação de escrita na figura é uma simplificação do que realmente acontece em TreadMarks; a notificação de escrita não é enviada diretamente de P1 para P2.) Isto ocasiona a invalidação da página no nó 2. Depois da sincronização, P2 tenta ler X, o que novamente gera uma falha de acesso. Agora, o nó 2 já tem uma cópia da página, mas recebeu uma notificação de escrita do nó 1. Como resultado, P2 só tem que solicitar um diff a P1. O recebimento deste pedido no nó 1, força P1 a gerar o diff requisitado e enviá-lo a P2. O diff em seguida deve ser aplicado à página, de forma que os *bits* de proteção de página possam ser atualizados, e a instrução de leitura possa ser re-executada no nó 2.

Como pode ser visto nesta figura, as principais fontes de atrasos em *software DSMs* estão relacionadas com as latências de comunicação e com as ações de coerência. Latências de comunicação causam atrasos no processador degradando o desempenho do sistema. As ações de coerência (geração e aplicação de diffs e geração de *twin*) também podem afetar negativamente o desempenho do sistema, pois não realizam nenhum trabalho útil e estão no caminho crítico da computação. O

impacto dos custos de comunicação e coerência são ampliados pelo fato de que processadores remotos estão envolvidos em todas as transações correspondentes.

Para demonstrar a extensão do problema de latência no caso de TreadMarks, considere as figuras 2.3 e 2.4[1]. A figura 2.3 apresenta os *speedups* alcançados por nossas aplicações rodando sobre TreadMarks em 8 processadores. A figura mostra que essas aplicações exibem uma grande variação de *speedups*. Alguns desses *speedups* são baixos mas esse comportamento é muito comum em *software DSMs*.

A figura 2.4 apresenta uma visão detalhada do desempenho de aplicações sobre TreadMarks em 8 processadores. As barras representam tempos de execução normalizados divididos em: tempo de computação, latência de busca de dados remotos, tempo de sincronização (subdividido em barreiras e *locks*), custo da coleta de lixo e custo de *inter-processor communication* (IPC). O tempo de computação inclui as latências de falha na cache e na TLB. A latência de busca de dados é uma combinação do tempo de processamento da coerência e da latência de rede, envolvidos na geração de *twins* e na busca de páginas e diffs como resultado de falhas de páginas. O tempo de sincronização representa os atrasos envolvidos na espera em barreiras e em aquisições/liberações de *lock*, incluindo os custos de processamento de intervalos e notificações de escrita. O custo de IPC contabiliza o tempo de computação que o processador despende servindo pedidos vindos de processadores remotos.

Esta figura mostra que TreadMarks apresenta custos significativos de busca de dados remotos e sincronização. Os custos de IPC não são tão significativos pois são freqüentemente escondidos pelas latências de busca de dados e de sincronização. No entanto, esses tempos se tornam importantes ao usarmos técnicas de pré-busca, o que será mostrado no capítulo 3. Na seção seguinte apresentamos as principais técnicas utilizadas para reduzir essas latências.

2.2 Técnicas para Redução da Latência de Acesso a Dados

Nesta seção apresentamos uma descrição genérica das principais técnicas de redução da latência de acesso a dados em *software DSMs*. A maior parte dos trabalhos anteriores concentra-se em estratégias baseadas em envio de atualizações, como por exemplo [18, 22, 40, 30]. Por outro lado, as técnicas de pré-busca e *multithreading*

[1]Os detalhes do ambiente de execução e das características das aplicações são apresentados no capítulo 3.

para *software DSMs* têm recebido menos atenção [17, 21, 4, 32, 9].

2.2.1 Técnicas de Pré-busca

Pré-busca é uma técnica utilizada para reduzir a latência de acesso a dados remotos movendo-os para o mais próximo possível do processador antes que os mesmos sejam necessários. A técnica de pré-busca implementada por *hardware* já foi exaustivamente estudada para sistemas uniprocessadores e multiprocessadores, e.g. [15, 16, 43, 44]. No entanto, utilizar técnicas similares às desses trabalhos não é eficiente no contexto de *software DSMs*, uma vez que a falta de suporte de *hardware* para técnicas de pré-busca em sistemas distribuídos e o uso de uma unidade de coerência maior em *software DSMs* tornam os erros de previsão mais comuns e custosos.

A técnica de pré-busca implementada por *software* através do compilador ou mesmo do programador também já foi estudada para sistemas uniprocessadores e multiprocessadores, e.g. [33, 8]. Uma técnica similar pode ser utilizada em *software DSMs*, como demonstrado em [32]. No entanto, estratégias que se baseiam em suporte de compiladores ou programadores são necessariamente limitadas, uma vez que compiladores sofisticados e programadores capazes raramente estão disponíveis.

Devido à falta de suporte de *hardware*, ao tamanho da unidade de coerência de *software DSMs* e às limitações de compiladores e programadores, a maior parte das técnicas de pré-busca já propostas para *software DSMs* é implementada no sistema de tempo de execução. Denominamos esse tipo de pré-busca de pré-busca dinâmica.

Todas as estratégias de pré-busca dinâmica desenvolvidas para *software DSMs* se baseiam no histórico passado das falhas de acesso. Dentre as principais técnicas de pré-busca dinâmica estão B+ e *Dynamic Aggregation*. B+ é uma técnica que nós propusemos em [7] e que se baseia nas invalidações de páginas para guiar a pré-busca. A técnica supõe que uma página acessada recentemente por um processador e que é invalidada por outro processador tem grande chance de ser acessada novamente em um futuro próximo. Sendo assim, a técnica B+ inicia pré-busca de diffs para cada uma das páginas invalidadas em pontos de sincronização, desde que a página estivesse válida no nó local antes do recebimento da invalidação. As pré-buscas são iniciadas logo depois de uma aquisição de *lock* ou de uma saída de barreira. B+ combina várias pré-buscas a serem enviadas a um mesmo processador em uma única mensagem.

Dynamic Aggregation agrupa páginas de acordo com as falhas de acesso ocorridas nos processadores, de forma que pode-se aumentar a unidade de busca sem gerar os efeitos negativos do falso compartilhamento. Quando um processador sofre uma falha de acesso para qualquer uma das páginas de um grupo, uma operação de pré-busca para o grupo inteiro é iniciada. No entanto, apenas a página correspondente à falha de acesso corrente se torna válida na chegada dos dados requisitados. Dessa forma, pode-se detectar alterações no comportamento da aplicação. Depois de buscar um grupo de páginas, o grupo é destruído. Além disso, todas as respostas a requisições de pré-busca devem ser recebidas antes de se ultrapassar um subseqüente ponto de sincronização.

Os grupos de páginas são computados em cada ponto de sincronização de acordo com as falhas de acesso ocorridas no processador antes da sincronização. Uma falha ocorre a cada primeiro acesso a uma página inválida. Todas as falhas de acesso geram uma seqüência que é dividida em grupos de forma que um grupo é totalmente preenchido antes que o próximo seja criado. O tamanho máximo dos grupos é definido pelo usuário.

Assim como B+, *Dynamic Aggregation* melhora o desempenho através da combinação de vários pedidos direcionados a um mesmo processador em uma única mensagem, reduzindo assim o número de mensagens envolvidas na busca de diffs para validar as páginas de um grupo.

2.2.2 Envio de Atualizações

O protocolo de coerência determina como as modificações aos dados compartilhados são propagadas aos demais processadores. Existem duas formas básicas de propagação: através do envio de notificações de escrita (protocolo de invalidação), ou através do envio das atualizações propriamente ditas (protocolo de atualização).

No primeiro caso, um processador ao receber uma notificação de escrita, simplesmente invalida a unidade de coerência que contém o dado em questão. As modificações ao dado compartilhado só serão requisitadas quando o processador que recebeu a notificação for efetivamente acessá-lo, sofrendo assim um falha de acesso.

Em um protocolo de atualização, a notificação de modificação de um dado compartilhado é a própria atualização realizada. Desta forma, o processador não sofre uma falha de acesso aos dados compartilhados posteriormente. No entanto, o protocolo de atualização gera um tráfego maior na rede, visto que várias atualizações

podem ser enviadas inutilmente antes que o dado seja realmente acessado.

Tendo em vista o problema do aumento de tráfego na rede gerado pelo uso de um protocolo de atualização, a maior parte dos protocolos de coerência utilizam a estratégia de invalidação. Entretanto, vários protocolos de coerência procuram utilizar uma forma mais seletiva da estratégia de atualização, criando portanto protocolos híbridos. Protocolos híbridos já foram estudados no contexto de multiprocessadores por vários pesquisadores, e.g. [45, 19]. Nesses estudos um controlador de cache invalida um de seus blocos se tal bloco receber um certo número de atualizações sem que aconteça uma referência ao bloco pelo próprio processador local. Tal estratégia não se aplica a *software DSMs*, os quais relaxam a consistência de memória e permitem múltiplos escritores concorrentes para reduzir o tráfego de dados de coerência.

Em *software DSMs*, as estratégias híbridas se concentram em transferir atualizações apenas quando há uma grande chance delas serem necessárias. No sistema *Lazy Hybrid* [18], o processador que libera um *lock* sabe quais *diffs* o processador que está requisitando o *lock* precisa receber, e os envia juntamente com o *lock*. No sistema AEC [40], o envio de atualizações se baseia na técnica de previsão da ordem de aquisição de *locks* (*Lock Acquirer Predictor* - LAP). LAP prevê o padrão de acesso aos *locks*, de forma que o processador que irá adquirir um *lock* pode receber os dados compartilhados que irá precisar antes mesmo de requisitar o *lock*. No sistema ADSM [30], modificações realizadas em páginas classificadas como migratórias dentro de seções críticas são enviadas junto com o *lock* correspondente. Além disso, modificações realizadas em páginas classificadas como produtor/consumidor são enviadas aos respectivos consumidores durante operações de barreira. Na seção 2.3 apresentamos uma descrição mais detalhada de ADSM.

2.2.3 Adaptação Entre Protocolos Único e Múltiplos Escritores

A maioria dos protocolos que permitem múltiplos escritores utilizam o mecanismo de *twinning* e *diffing* descrito anteriormente. Esses protocolos conseguem minimizar o efeito do falso compartilhamento permitindo que vários processadores alterem concorrentemente uma página. No entanto, protocolos múltiplos-escritores criam custos adicionais no caso de páginas que não estão sujeitas a falso compartilhamento, pois é necessário detectar, armazenar e consolidar modificações a essas páginas. Esses custos adicionais podem ser eliminados no caso de páginas que não sofrem

falso compartilhamento, permitindo que apenas um processador as altere por vez. Tendo em vista a existência de aplicações que podem se beneficiar dos dois tipos de protocolos, o ideal seria a utilização de uma estratégia que se adapte entre um protocolo múltiplos-escritores e um que só permite um escritor de cada vez. Basicamente é necessário, portanto, identificar o padrão de compartilhamento das páginas com o intuito de decidir quando usar um dos dois protocolos.

Em [3] foi proposta uma versão de TreadMarks conhecida como *Adaptive TreadMarks* (ATmk) que se adapta dinamicamente entre protocolos único-escritor e múltiplos-escritores. O protocolo múltiplos-escritores utiliza o mecanismo de *twinning* e *diffing* de TreadMarks, e o protocolo único-escritor é uma extensão daquele proposto para o sistema CVM [23]. Este último usa o conceito de posse e versões de páginas, de forma que só o dono da página pode alterá-la. A posse de páginas é transferida em falhas de escrita podendo então gerar mensagens adicionais no caso da falha de escrita ser referente a uma página localmente válida. A adaptação é realizada dinamicamente a nível de página, e o sistema monitora o padrão de acesso às páginas decidindo o modo a ser usado (modo único-escritor ou modo múltiplo-escritor).

O sistema ADSM também propõe uma adaptação entre protocolos múltiplos-escritores e único-escritor. Assim como em ATmk, ADSM utiliza o mecanismo de *twinning* e *diffing* para páginas que possuem múltiplos escritores, enquanto que no caso de páginas que possuem somente um escritor, a página inteira é transferida para manter a coerência. Mas, por outro lado, ADSM não utiliza nenhuma mensagem adicional para implementar a adaptação entre esses dois modos, empregando portanto uma categorização mais detalhada a ser descrita na seção 2.3.

2.3 Sistema ADSM

Considerando que utilizamos as estratégias de tolerância a latência desenvolvidas para o sistema ADSM, apresentamos nesta seção uma breve descrição deste sistema.

O desenvolvimento do sistema ADSM é baseado no sistema TreadMarks, sendo que ADSM faz uma adaptação de acordo com o padrão de acesso da aplicação com o intuito de melhorar o seu desempenho. O protocolo implementa dois tipos de adaptação:

- Adaptação entre estratégias de manutenção de coerência baseadas em invali-

dação e atualização.

- Adaptação entre modos de operação único-escritor e múltiplos-escritores.

Estes dois tipos de adaptação utilizam uma categorização dinâmica do tipo de compartilhamento que cada página está sujeita devido à execução da aplicação. A categorização é denominada *Sharing Pattern Categorization* (SPC) e se baseia na associação entre variáveis de *lock* e as páginas que sofrem falhas dentro das seções críticas correspondentes. SPC classifica as páginas como: falsamente-compartilhadas, migratórias, ou produtor/consumidor(es). As páginas classificadas como migratória e produtor/consumidor(es) são tratadas no modo único-escritor, enquanto que as páginas classificadas como falsamente-compartilhadas são tratadas no modo múltiplo-escritor. As páginas que são alteradas por múltiplos escritores concorrentes são tratadas pelo mecanismo de *twinning* e *diffing*. Por outro lado, a coerência das páginas que são escritas por um processador por vez é mantida através da transferência da página inteira. Uma página identificada por possuir um único escritor só pode ser modificada pelo seu proprietário corrente. Páginas com múltiplos escritores não têm nenhum proprietário associado. As páginas identificadas como produtor/consumidor(es) têm proprietários fixos (o próprio produtor). No entanto, o proprietário de uma página migratória muda ao longo da execução da aplicação. Um processador ao adquirir um *lock*, se torna proprietário de todas as páginas migratórias associadas a este *lock*. Enquanto que a propriedade de uma página migratória que não está associada a nenhum *lock* é transferida juntamente com a página. Portanto, ADSM realiza a transferência de propriedade de páginas sem utilizar nenhuma mensagem extra.

A manutenção de coerência baseada no envio de atualizações é usada para os dados guardados por *locks* e classificados como migratórios, e para os dados protegidos por barreira e classificados como produtor/consumidor(es). No caso dos dados protegidos por *locks*, o processador que está liberando o *lock* envia as atualizações logo depois de enviar o *lock* para o processador que está requisitando-o. Enquanto que no caso dos dados protegidos por barreiras, os produtores enviam atualizações para os consumidores durante a barreira, sendo que esta transferência de dados é realizada durante a espera na barreira, sobrepondo assim os dois tipos de atrasos. Além disso, não existe a necessidade do processador esperar pelo recebimento das atualizações nos pontos de sincronização em nenhuma das duas situações. Uma

descrição mais detalhada de ADSM pode ser encontrada em [30] ou em [31].

Capítulo 3

Estratégias de *Software*

Neste capítulo iremos apresentar uma nova estratégia de pré-busca denominada *Adaptive++*. Além disso, analisamos o desempenho dessa técnica isoladamente e quando combinada com os dois tipos de adaptação implementados no sistema ADSM.

3.1 A Técnica de Pré-Busca *Adaptive++*

3.1.1 Evolução da Técnica

A primeira técnica de pré-busca que nós desenvolvemos foi a técnica B+, que utiliza as invalidações como indicação das páginas a serem acessadas em um futuro próximo e que, portanto, são buscadas via pré-busca. Verificamos que esta técnica é muito agressiva tendo em vista que cobre a maioria das falhas de páginas mas muitos dos dados trazidos através de pré-buscas não são utilizados, gerando degradação de desempenho para algumas aplicações. Devido a esses resultados realizamos um estudo do comportamento de aplicações paralelas executando sob o *software DSM* TreadMarks, considerando os aspectos relacionados com técnicas de pré-busca. Este estudo se baseia em resultados de simulações e foi apresentado em [10].

Neste estudo nós analisamos a seqüencia das falhas de acesso que requerem busca de dados remotos de acordo com sua concentração espacial e temporal, e com relação ao compartilhamento. Nossos resultados mostraram que, em média, a quantidade de computação útil entre o ponto onde uma pré-busca pode ser iniciada e o momento em que a página é realmente usada é suficiente para esconder a maior parte da latência de busca de dados remotos. Nossos resultados também mostraram que o padrão de falhas de acesso muda significativamente durante a execução das aplicações, sendo necessária portanto uma técnica de pré-busca que se adapte

em tempo de execução. Verificamos ainda que freqüentemente as falhas de páginas são espacialmente concentradas, mas não apresentam concentração temporal, demonstrando que a pré-busca seqüencial utilizada em *hardware DSMs* também pode trazer benefícios para *software DSMs*. Finalmente, nossos resultados mostraram que o conjunto de invalidações de páginas recebido em pontos de sincronização não é geralmente uma boa indicação dos próximos acessos a páginas, o que significa que as invalidações não devem ser usadas para guiar pré-buscas.

Esses resultados nos guiaram no desenvolvimento de uma nova técnica de pré-busca: *Adaptive++*.

3.1.2 Descrição

Adaptive++ prevê os acessos a dados remotos que devem acontecer em um futuro próximo e inicia pré-busca para esses dados antes que eles sejam efetivamente acessados. A técnica considera as informações geradas em tempo de execução sobre os dados acessados por cada nó do sistema e visa a otimização do desempenho de aplicações regulares. De acordo com a nossa experiência [10], estas aplicações apresentam dois principais tipos de comportamentos:

- O conjunto dos acessos a dados remotos realizados durante uma fase de execução (delimitada por eventos consecutivos de barreira) é repetido durante uma fase subseqüente.
- O intervalo espacial (*stride*) entre os diferentes acessos remotos durante uma fase é repetido em outras fases.

Adaptive++ não solicita pré-buscas quando a aplicação não apresenta um desses comportamentos, isto é, quando são gerados acessos remotos "não-esperados". Sendo assim, objetivando melhorar o desempenho das aplicações regulares, nossa técnica se adapta entre dois modos de operação: **repetição-de-fase** ou **repetição-de-stride**.

A implementação de cada um desses modos depende obviamente do *software DSM* empregado. Com a finalidade de tornar a descrição da técnica mais concreta, nós detalhamos a seguir a implementação de *Adaptive++* em *TreadMarks*, enquanto descrevemos os diferentes modos de execução e o conceito de acessos remotos "esperados" no contexto desta implementação.

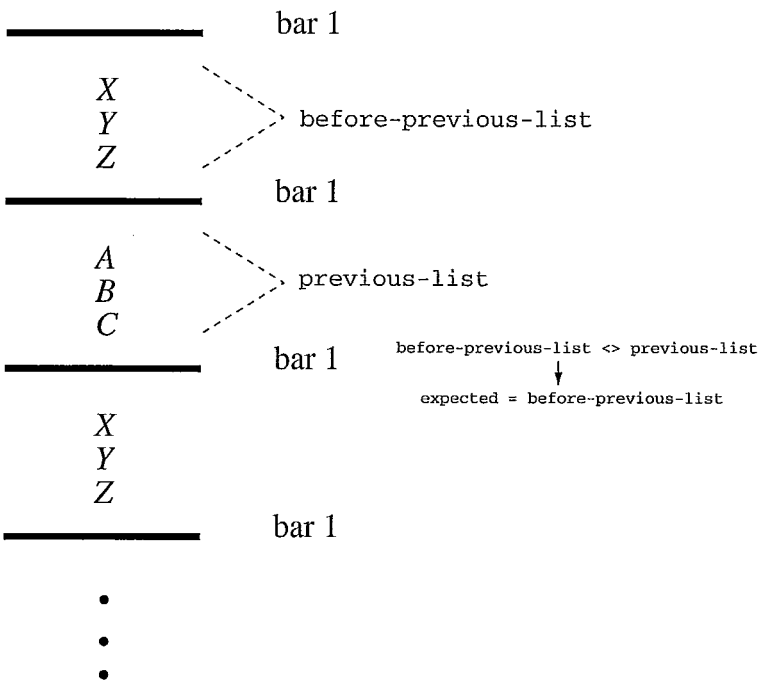

Figura 3.1: Exemplo do modo repetição-de-fase.

Modo repetição-de-fase. Para determinar quais páginas serão requisitadas via pré-buscas neste modo de operação, nossa técnica mantém duas listas: `previous-list` e `before-previous-list`. `previous-list` contém as identificações das páginas que sofreram falhas fora de seção crítica na fase de execução anterior, ao passo que `before-previous-list` registra as falhas que ocorreram fora de seção crítica durante a fase antes da anterior. Uma dessas listas deverá ser escolhida como o conjunto de falhas esperadas (lista `expected`), isto é, como uma previsão das falhas de páginas que provavelmente o processador sofrerá na próxima fase.

No terceiro episódio de uma sincronização de barreira, a similaridade entre as listas `previous-list` e `before-previous-list` determina qual delas será escolhida para representar a lista `expected` nos eventos de barreira. Listas similares determinam que a lista correspondente à fase anterior será escolhida em todos os episódios subseqüentes de barreira. Quando as listas não são similares, a lista correspondente à fase antes da última será sempre escolhida. Duas listas são consideradas similares se mais de 50% de suas páginas pertencem às duas listas. Observe que esta estratégia somente busca determinar a similaridade entre duas fases consecutivas, e assume que, se as fases não são similares, existe uma alternância entre fases similares. A razão desta suposição vem da raridade com que as aplicações apresentam fases similares que não sejam consecutivas ou alternadas.

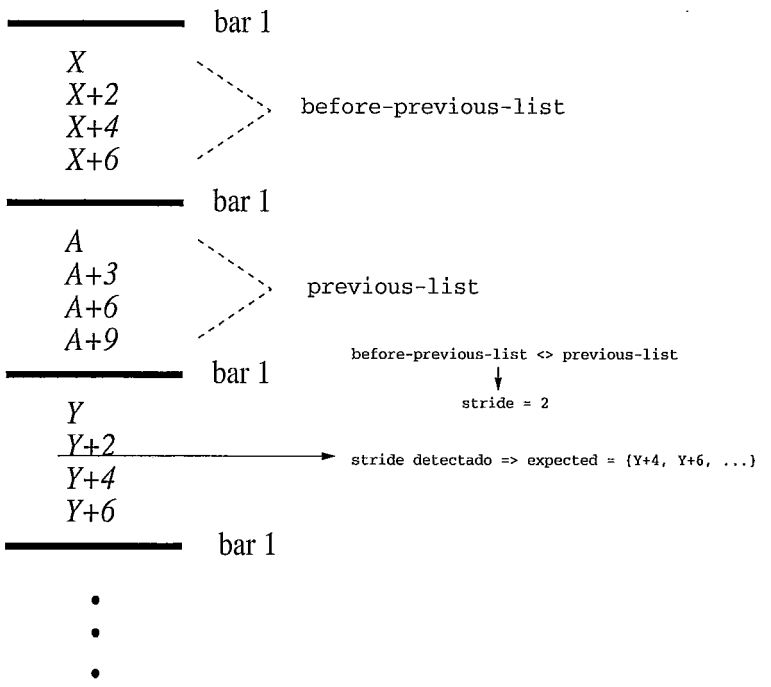

Figura 3.2: Exemplo do modo repetição-de-stride.

Na figura 3.1 ilustramos o modo repetição-de-fase. Os dois primeiros episódios da barreira 1 limitam uma fase de execução. As falhas de acesso às páginas `X`, `Y` e `Z` formam a lista `before-previous-list`. No terceiro episódio da barreira 1, temos a lista `previous-list` que é composta pelas páginas `A`, `B` e `C`. Neste momento, as duas listas são comparadas, e tendo em vista que elas são diferentes, assumimos que a repetição da fase é alternada. Dessa forma, a lista `before-previous-list` é escolhida como sendo a lista de acessos esperados (lista `expected`).

Modo repetição-de-stride. A escolha das páginas requisitadas por pré-buscas é diferente no modo de operação repetição-de-stride, embora este modo também envolva as duas listas de falhas de páginas e a escolha da que será usada baseia-se também na similaridade entre as duas listas. O modo repetição-de-stride usa o *stride* mais freqüente entre as falhas de páginas dentro da lista escolhida, para determinar as páginas a serem solicitadas por pré-buscas na fase seguinte. A determinação da lista ordenada das falhas esperadas (`expected`) depende inicialmente da detecção da primeira ocorrência do *stride* esperado no início da fase e da subseqüente identificação da falha de página correspondente. A lista `expected` será composta então por todas as páginas cuja diferença entre sua identificação e a da página identificada anteriormente é múltipla deste *stride*. A figura 3.2 ilustra o modo repetição-de-stride. No terceiro episódio da barreira 1, verifica-se que as duas

listas (`before-previous-list` e `previous-list`) são diferentes, portanto utiliza-se o stride da lista `before-previous-list`. No entanto, a lista `expected` só será formada depois de se detectar a ocorrência do stride esperado, isto é, depois de acessar a página `Y+2`.

Observe que no modo repetição-de-stride, *Adaptive++* pode iniciar pré-buscas para páginas que nunca foram acessadas, logo pré-buscas podem ser iniciadas tanto para páginas inteiras quanto para os seus diffs. No modo repetição-de-fase, *Adaptive++* somente inicia pré-buscas para páginas que já tenham sido tocadas antes, iniciando assim somente pré-buscas de diffs.

Escolhendo um modo. A decisão de qual modo aplicar em uma fase é tomada durante o episódio de barreira que inicia a fase. A decisão é baseada na técnica mais adequada à fase. Se nenhuma técnica parece apropriada, decide-se parar de fazer pré-buscas pelo menos até o próximo episódio de barreira.

A métrica que determina o potencial de sucesso da técnica é diferente para cada modo de operação. No caso do modo de operação repetição-de-fase, a métrica é a percentagem de pré-buscas úteis (aquelas que evitaram operações remotas em falhas) que este modo iniciou (se foi o modo escolhido na fase anterior) ou deveria ter iniciado (se não foi o modo escolhido na fase anterior). No caso do modo repetição-de-stride, a métrica é a freqüência do *stride* mais comum de acesso a páginas observado na lista de falhas escolhida. Em todos os eventos de barreira, o modo a ser usado na próxima fase é aquele que apresenta o maior valor de sua métrica. No caso de um empate, o modo repetição-de-fase é o escolhido. A escolha de um modo obviamente envolve algum custo adicional, que nós mostraremos ser quase sempre completamente escondido pelo tempo de espera na barreira.

Iniciando pré-buscas. No modo repetição-de-fase, um número de páginas definido pelo usuário (24 páginas em nossos experimentos), cujas identificações pertencem a lista `expected`, é requisitado por pré-buscas logo depois que cada processador sai de um evento de barreira. Além disso, o modo repetição-de-fase tenta iniciar pré-buscas para um outro número de páginas (definido pelo usuário e igual a 2 em nossos experimentos) seguintes a falha de página corrente na lista `expected`, tendo em vista que a falha ocorreu fora de seção crítica. Note que pré-buscas não são iniciadas para: (a) páginas que estão válidas na memória local; (b) páginas para as quais pré-buscas já completaram (isto é, os dados requeridos ainda estão na memória local); e (c) páginas para as quais pré-buscas já foram iniciadas, mas estão

pendentes. Nenhuma ação é tomada nas falhas que ocorrem dentro de seção crítica ou nas páginas que não estão na lista `expected`.

Em contraste com o modo repetição-de-fase, nenhuma pré-busca é iniciada no ponto de barreira no modo repetição-de-stride, tendo em vista que neste ponto ainda não é possível determinar o conjunto de falhas esperadas. Nas falhas fora de seção crítica, o modo repetição-de-stride tenta iniciar pré-buscas para um outro número de páginas definido pelo usuário (2 páginas no nosso experimento) que seguem a falha de página corrente na lista `expected`. Novamente, pré-buscas só são iniciadas nas falhas de páginas encontradas em `expected`, considerando que as condições a, b e c estejam satisfeitas.

Recebendo respostas de pré-buscas. As respostas de pré-buscas são guardadas até que um acesso real à página seja realizado pelo processador, neste ponto quaisquer diffs que não tenham sido buscados por alguma pré-busca (ou que ainda estão para ser recebidos) são coletados, todos os diffs (trazidos por pré-buscas ou não) são aplicados à versão desatualizada da página, e a página é tornada válida. Observe que uma falha de acesso é sofrida pelo nó, mesmo que todos os *diffs* necessários já tenham sido recebidos. Pré-buscas são iniciadas tanto neste tipo de falhas quanto nas falhas para as quais nenhuma pré-busca de diff foi pedida, se a falha de página pertencer à lista `expected`.

3.1.3 Discussão

A seguir apresentamos as justificativas para as principais decisões de projeto relacionadas com a definição da técnica *Adaptive++*. Nossa técnica ataca as aplicações regulares, tendo em vista que seu desempenho pode ser melhorado sem a intervenção de usuários ou compiladores sofisticados inserindo chamadas explícitas de pré-buscas no código fonte da aplicação. É possível prever o comportamento das falhas dessas aplicações baseando-se em seus históricos passados de falhas.

Adaptive++ focaliza somente aplicações regulares que apresentam os comportamentos de repetição-de-fase ou repetição-de-stride. A razão para esta decisão é que nossa experiência anterior [10] mostrou que essas são as duas formas de regularidades mais importantes encontradas em um grande conjunto de aplicações paralelas. O pequeno número de propostas anteriores de técnicas de pré-busca dinâmica não se mostraram efetivas para ambos os tipos de aplicações regulares.

Adaptive++ pode iniciar pré-buscas para um número relativamente grande de páginas (até 24 páginas), logo depois de um evento de barreira. Por outro lado, a técnica é muito menos agressiva ao iniciar (no máximo 2) pré-buscas em falhas de acesso. Esta estratégia usa o fato de que processadores freqüentemente sofrem falhas logo no início de cada fase de execução, permitindo uma sobreposição entre os custos de pedidos entre processadores e de busca de dados. Ao longo da execução da fase, a ocorrência deste tipo de sobreposição se torna menos provável e as pré-buscas acabam interferindo mais freqüentemente na computação dos processadores remotos.

Adaptive++ sempre verifica se a falha de página corrente é uma das esperadas antes de iniciar qualquer pré-busca. Com isso, *Adaptive++* consegue garantir uma maior segurança de que a aplicação está se comportando regularmente, e que sendo assim terá a capacidade de melhorar seu desempenho. Em aplicações irregulares, as falhas sofridas por cada processador são, na maior parte das vezes, "não-esperadas".

Adaptive++ evita a iniciação de pré-buscas depois de pontos de aquisição de *locks*. A razão para isso é que a nossa experiência anterior [7] mostrou que várias aplicações paralelas apresentam seções críticas curtas que podem ser aumentadas pela iniciação de pré-buscas. Seções críticas mais longas aumentam significativamente os custos de sincronização quando há contenção pelo *lock*. Uma desvantagem de restringir pré-buscas a eventos de barreira é que *Adaptive++* não consegue otimizar aplicações que não possuam sincronização de barreira. Nós não consideramos isto um problema grave tendo em vista que a vasta maioria das aplicações paralelas envolvem barreiras.

Adaptive++ também evita pré-buscas de páginas que sofrem falhas de acesso dentro de seções críticas. A razão dessa restrição é que essas páginas geralmente são invalidadas no ponto de aquisição do *lock*, forçando o sistema a coletar os diffs nos primeiros acessos às páginas depois do *lock*.

3.2 Combinação de Estratégias de *Software*

Nesta seção propomos a combinação de três estratégias que otimizam *software DSMs* através da redução do tempo de acesso a dados remotos. As técnicas utilizadas são as seguintes:

- Estratégia de pré-busca: busca antecipada de dados remotos. Mais especificamente, a técnica *Adaptive++*.

- Envio de atualizações: adaptação entre a manutenção de coerência via invalidação e via atualização. Mais especificamente, a adaptação sugerida para o sistema ADSM.

- Adaptação entre o modo único e múltiplo escritor (*Single/Multiple writer Adaptation* - SMA): dados modificados por vários escritores concorrentemente têm sua coerência mantida através do uso do mecanismo de *twinning* e *diffing*. Em contraste, dados alterados por um único escritor de cada vez têm sua coerência mantida através da transmissão integral do dado. Vamos utilizar também a adaptação proposta para o sistema ADSM.

As três estratégias têm suas restrições, de forma que uma pode complementar a outra através de uma combinação. A técnica *Adaptive++* não cobre falhas de acesso que ocorrem dentro de seções críticas, que por outro lado são cobertas pelas duas outras técnicas. A estratégia de envio de atualizações só atinge páginas escritas por um único escritor e que sejam acessadas através dos padrões de acesso migratório em *locks* ou produtor/consumidor(es), tentando assim não aumentar o volume de tráfego na rede. Logo as páginas acessadas por múltiplos escritores concorrentemente não são cobertas por esta estratégia, mas podem ser cobertas pela estratégia de pré-busca.

A combinação das três estratégias se baseia na categorização realizada por SPC proposta em [30] para o sistema ADSM. O acesso às página classificadas como falsamente-compartilhadas é otimizado através da utilização da técnica *Adaptive++*. A estratégia SMA atinge as páginas com um único escritor, sendo que aquelas que são acessadas segundo os padrões de acesso migratório em *lock* ou produtor/consumidor(es) também têm seu acesso otimizado pelo envio de atualizações.

3.3 Metodologia

Nossos experimentos foram realizados em um *cluster* de 8 nós, cada um com 512Mbytes de memória, interligados por uma *switch* Myrinet de 1.8Gb/s. Os processadores são Pentium III de 650MHz, cada um com três caches: duas caches de nível 1 (uma de dados e outra de instruções) com 16Kbytes cada e uma de nível 2 com 256Kbytes.

Aplicação	Entrada
LU	2800 × 2800, bloco de 128 bytes
CG	14000 × 14000 elementos com 2030000 não-zeros
FFT	128 × 128 × 128, 10 iterações
Em3d	80000 nós, 0.1% remotos, 200 iterações
SOR	1000 × 3072 elementos, 100 iterações
IS	$N = 2^{23}, Bmax = 2^{15}$, 100 iterações
IS_lk	$N = 2^{23}, Bmax = 2^{15}$, 100 iterações
MigFreq	2048 × 128

Tabela 3.1: Aplicações e suas entradas.

A implementação de *Adaptive++* foi desenvolvida para TreadMarks, assim como o sistema ADSM teve sua implementação baseada no TreadMarks, logo analisaremos o desempenho alcançado pelas estratégias apresentadas através da comparação com o sistema TreadMarks original. A nossa avaliação desses sistemas foi baseada no comportamento de 8 aplicações: LU, CG, FFT, Em3d, SOR, duas versões de IS, uma baseada em *locks* (IS_lk) e a outra baseada em barreiras (IS), e MigFreq. LU é do conjunto SPLASH-2 de *benchmarks* [47], Em3d foi desenvolvida na *University of California at Berkeley*, MigFreq é um *benchmark* desenvolvido pela Petrobrás e as outras aplicações fazem parte do pacote de distribuição de TreadMarks. Essas aplicações são exemplos de aplicações científicas que apresentam comportamentos regulares, isto é, é possível prever as falhas de acesso futuras. Na tabela 3.1 apresentamos as entradas utilizadas em cada aplicação. As entradas que utilizamos são relativamente pequenas devido à limitação do sistema TreadMarks, que replica a área de memória usada para armazenar os dados compartilhados em todos os nós do sistema. A seguir faremos uma breve descrição das aplicações.

LU fatora uma matriz densa em um produto de duas matrizes triangulares, sendo uma superior e a outra inferior. No *benchmark* **CG** utiliza-se um método de gradiente conjugado para calcular uma aproximação do menor *eigenvalue* de uma matriz simétrica, positiva, grande e esparsa. **FFT** soluciona uma equação diferencial parcial 3D usando FFTs. **Em3d** simula a propagação de ondas eletro-magnéticas em objetos tridimensionais. **SOR** executa relaxações sucessivas de uma matriz densa. **IS** (*Integer Sort*) classifica um vetor de `N` inteiros usando chaves no intervalo [0, Bmax] através da técnica *bucket sort*. **MigFreq** é um *benchmark* que realiza migração sísmica 2D pós-estaqueamento usando o método $w - x$. IS_lk e MigFreq têm os acessos a dados compartilhados guardados por *locks*, enquanto que as demais

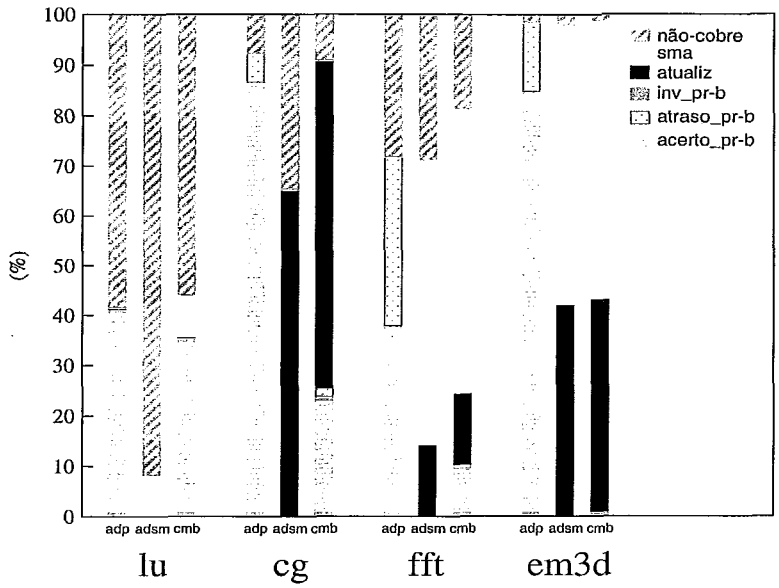

Figura 3.3: Cobertura oferecida pelas estratégias de *software*.

aplicações utilizam apenas barreiras.

3.4 Resultados Experimentais

Nesta seção apresentaremos resultados individuais de cada estratégia de tolerância a latência de acesso a dados remotos estudada, assim como os resultados da combinação dessas estratégias. Dividiremos os nossos resultados experimentais em dois grandes grupos: aqueles que se relacionam a qualidade das estratégias e aqueles que se relacionam com o desempenho alcançado pelas técnicas.

3.4.1 Cobertura e Utilidade

Cobertura. Inicialmente avaliamos as técnicas estudando a sua cobertura. A cobertura é definida como o percentual de falhas de acesso para as quais as estratégias realizam alguma operação. Nas figuras 3.3 e 3.4 apresentamos o número normalizado de falhas de acesso sofridas por cada aplicação. Nas figuras temos três barras por aplicação representando a cobertura oferecida pela técnica *Adaptive++* (`adp`), pelo sistema ADSM (`adsm`) e pela combinação das técnicas (`cmb`). Cada barra da figura é dividida em:

`acerto_pr-b`: falhas de acesso para as quais no momento da falha os dados necessários (diffs ou página) já haviam sido coletados através de pré-buscas;

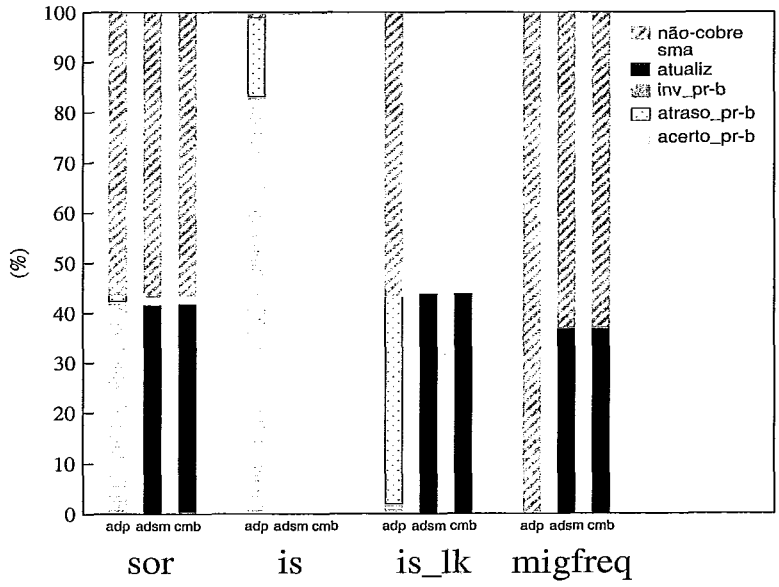

Figura 3.4: Cobertura oferecida pelas estratégias de *software*.

`atraso_pr-b`: falhas de acesso para as quais pré-buscas para a página (ou diffs) foram iniciadas, mas nem todas as respostas já haviam sido recebidas no momento da falha;

`inv_pr-b`: falhas de acesso para as quais a página a ser atualizada via pré-busca foi invalidada antes dos dados necessários (diffs ou página) serem recebidos;

`atualiz`: falhas de acesso que foram evitadas pelo recebimento de atualizações;

`sma`: falhas de acesso sofridas por páginas categorizadas como único-escritor, e que por conseguinte, são atualizadas pela busca da página inteira;

`não-cobre`: falhas de acesso para as quais não foi realizada nenhuma operação.

O fator de cobertura de *Adaptive++* é definido como sendo a soma das três primeiras categorias. No sistema ADSM, a cobertura é definida pela soma de `atualiz` e `sma`. Esta soma também compõe a cobertura da estratégia SMA, sendo que a categoria `atualiz` seria considerada como parte da categoria `sma`. E o fator cobertura da combinação das estratégias é a soma de todas as categorias, com excessão, é lógico, da categoria `não-cobre`.

Analisando inicialmente a técnica *Adaptive++*, as figuras 3.3 e 3.4 mostram que CG, FFT, Em3d e IS são as aplicações que apresentam maior cobertura, justamente por serem regulares. Essas quatro aplicações tiram enorme vantagem do modo

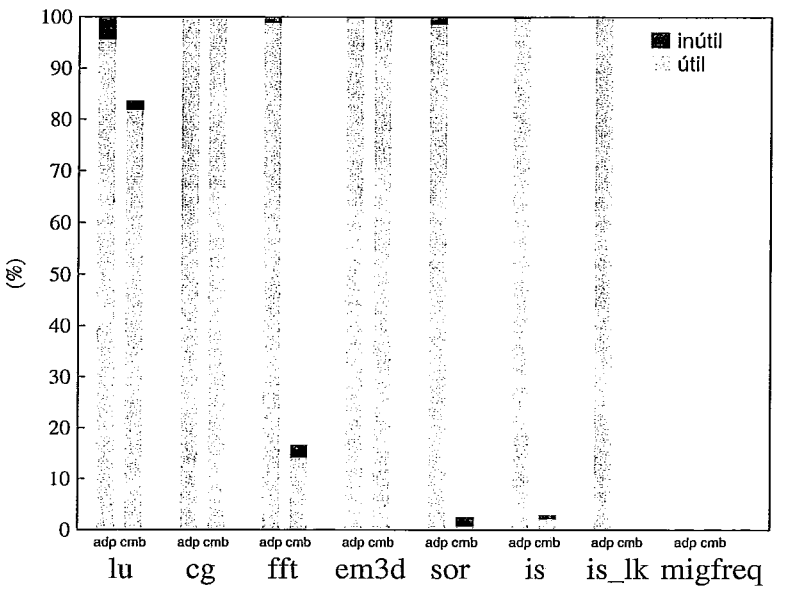

Figura 3.5: Utilização de *Adaptive++*.

repetição-de-fase de *Adaptive++*. Os comportamentos de CG e IS são semelhantes: suas fases similares são consecutivas e elas tiram vantagem do modo repetição-de-stride durante a fase inicial de execução. No modo repetição-de-stride as pré-buscas são iniciadas durante uma falha de acesso, de forma que algumas vezes os dados coletados por essas pré-buscas não chegam a tempo, o que explica o percentual de `atraso_pr-b` obtidos por CG e IS. O comportamento de FFT é muito similar aos de CG e IS, onde a única diferença é que as suas fases similares são alternadas. Já Em3d tem um comportamento diferente. Seu código apresenta duas barreiras principais, que também utilizam o modo repetição-de-fase, sendo que na primeira barreira temos fases similares alternadas, enquanto na segunda barreira as fases similares são consecutivas. Além disso, temos pré-buscas iniciadas tanto no final do evento de barreira, quanto durante algumas falhas de acesso, o que explica o percentual de falhas de acesso cujas pré-buscas chegaram atrasadas. Nessas quatro aplicações o percentual de falhas de acesso onde não é realizado nenhuma pré-busca se deve à fase inicial de execução, quando a técnica ainda não "descobriu" qual é o comportamento da aplicação.

O comportamento de SOR é diferente dessas aplicações. A matriz de SOR é particionada em faixas que são distribuídas entre os P processadores. As linhas da matriz que são compartilhadas são as duas que estão nas bordas das faixas, sendo cada linha compartilhada por dois processadores. Em SOR, as fases similares são alternadas e o percentual de `não-cobre` se deve ao fato que os processadores 0 e

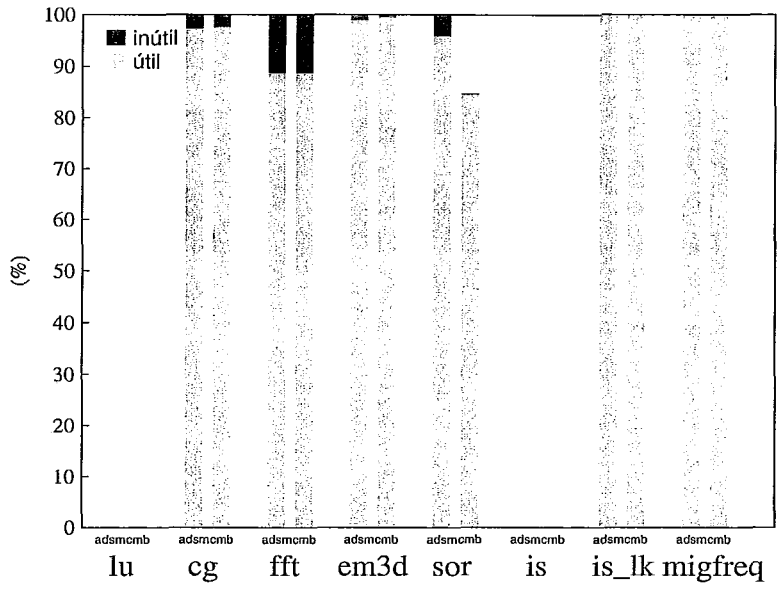

Figura 3.6: Utilização de atualizações.

$P-1$ só experimentam falhas de acesso referentes a uma única linha da matriz por fase e, por isso, não iniciam pré-buscas em *Adaptive++*. Em adição existe também o período de aprendizado onde a técnica ainda não identificou o comportamento da aplicação. O percentual de pré-buscas atrasadas (`atraso_pr-b`) apresentado na figura se justifica pelo fato de que os processadores iniciam pré-buscas logo depois do evento de barreira, mas precisam acessar uma das páginas compartilhadas no início de cada fase.

LU não tem cobertura tão boa quanto essas aplicações altamente regulares. LU é uma aplicação onde muitas falhas de acesso apresentam comportamento seqüencial, logo LU utiliza o modo de repetição-de-stride com *stride* 1. Entretanto, essa aplicação não apresenta esse comportamento durante toda a sua execução. Inicialmente são feitos acessos irregulares e, quando a aplicação passa a se comportar regularmente, existe uma predominância do *stride* 1 dentro de uma fase, mas existem *strides* diferentes dentro da mesma fase. Assim, quando a aplicação passa de um *stride* para outro, *Adaptive++* não consegue prever a nova seqüência de acessos. Isso explica a baixa cobertura (41%) apresentada por LU.

Apesar da aplicação IS_lk usar *locks* para acessar os dados compartilhados, existe uma parte do seu código que não está protegida por *locks* e que acessa uma estrutura compartilhada. Nesta parte do código uma estrutura de dados local é atualizada de acordo com uma estrutura compartilhada, que é percorrida em ordem inversa. Logo, durante a execução desta parte do código, IS_lk utiliza o modo repetição-de-

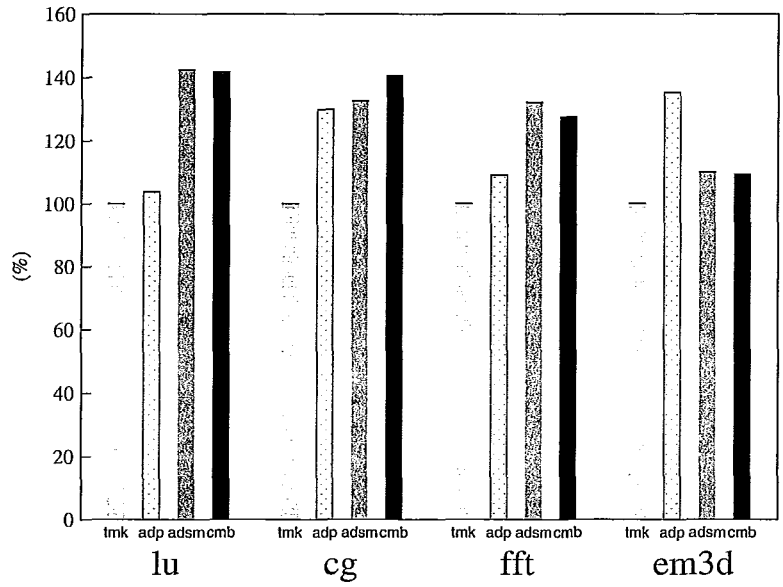

Figura 3.7: *Speedup* com 8 processadores.

stride com *stride* -1. Novamente observa-se a geração do recebimento de pré-buscas atrasadas neste modo.

MigFreq não se beneficia da estratégia de pré-busca tendo em vista ser totalmente baseada em *locks*.

Vamos nos concentrar agora na cobertura oferecida pelo sistema ADSM. Olhando para a barra `adsm` (barra do meio) de cada aplicação das figuras 3.3 e 3.4, verificamos que as aplicações CG, FFT, Em3d, IS e IS_lk apresentam uma alta cobertura. No caso de CG, a cobertura é dominada pelo percentual de atualizações (`atualiz`), pois a maioria das falhas de acesso se referem a páginas categorizadas como produtor/consumidor(es). FFT possui um baixo fator de atualizações (17%), devido ao mesmo tipo de falhas de acesso, mas sua maior cobertura se deve ao fator `sma`, demonstrando que basicamente todas as páginas são alteradas por um processador por vez. As falhas de acesso não cobertas se referem às falhas de acesso "frias" (*cold starts*), isto é, falhas de acesso geradas devido a primeira referência à página feita pelo processador. Em3d e IS_lk têm suas coberturas basicamente divididas igualmente entre as categorias `atualiz` e `sma`, sendo que as atualizações geradas em Em3d se devem a existência de páginas produtor/consumidor(es), enquanto que as atualizações de IS_lk são geradas para páginas categorizadas como migratórias em *lock*.

A baixa cobertura de LU, SOR e MigFreq se deve ao grande número de falhas de acesso "frias" experimentado por essas aplicações. No caso de LU, *Adaptive++*

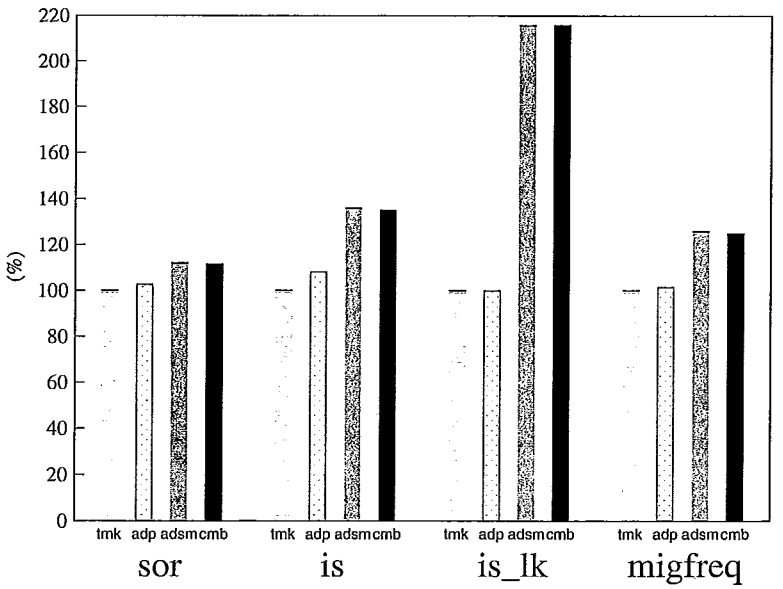

Figura 3.8: *Speedup* com 8 processadores.

consegue um fator de cobertura maior pois esta técnica também cobre falhas de acesso "frias" no modo repetição-de-stride. A cobertura de SOR é dominada pelo envio de atualizações para páginas produtor/consumidor(es). Por outro lado, a cobertura de MigFreq se deve apenas às atualizações feitas a páginas categorizadas como migratórias em *lock*.

Analisando agora a terceira barra de cada aplicação nas figuras 3.3 e 3.4, que corresponde a cobertura da combinação das estratégias, verificamos que LU, CG, FFT e Em3d são as aplicações que se beneficiam desta combinação. Apesar do padrão de compartilhamento de LU, FFT e Em3d ser dominado por páginas com um único escritor, *Adaptive++* dispara pré-buscas para falhas de acesso "frias". Por outro lado, CG apresenta tanto páginas único-escritor (mais especificamente páginas produtor/consumidor(es)) quanto páginas múltiplos-escritores, se beneficiando assim das estratégias de pré-busca e de envio de atualizações. As demais aplicações têm suas páginas categorizadas como único-escritor e seus comportamentos não utilizam o modo repetição-de-stride, de forma que *Adaptive++* não inicia nenhuma pré-busca. Portanto, suas coberturas são iguais as geradas pelo sistema ADSM.

Utilização. A avaliação de técnicas de pré-busca e de envio de atualizações não pode se basear apenas no fator de cobertura. Técnicas que iniciam pré-buscas ou enviam atualizações de forma agressiva alcançam, naturalmente, um fator de cobertura alto, mas às vezes essa alta cobertura está associada a um grande número de pré-buscas ou atualizações inúteis, isto é, pré-buscas para dados ou envio de

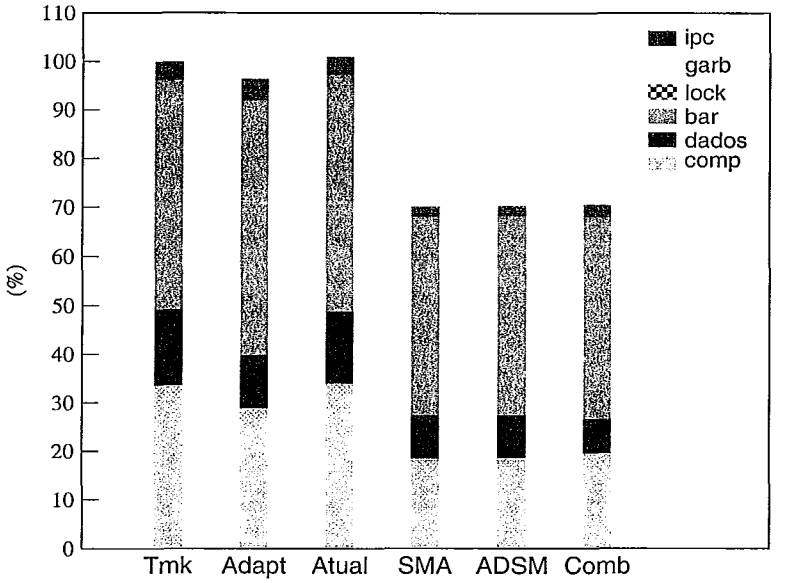

Figura 3.9: Tempo de execução de LU.

atualizações que não serão subseqüentemente utilizados.

A figura 3.5 apresenta o percentual de utilização das pré-buscas. Na figura temos duas barras por aplicação representando a utilização da estratégia *Adaptive++* (adp) e da combinação das estratégias (cmb). Cada barra é dividida em pré-buscas úteis e inúteis. Note que as pré-buscas úteis correspondem às pré-buscas que cobrem as categorias acerto_pr-b e atraso_pr-b que compõem o fator cobertura.

Adaptive++ inicia pré-buscas apenas quando tem uma razoável segurança de que a pré-busca será útil. Isso é demonstrado pelos baixos percentuais de pré-buscas inúteis apresentados na figura. Observamos novamente que apenas CG inicia um considerável número de pré-buscas na combinação das estratégias.

Na figura 3.6 temos o percentual de utilização das atualizações. Apresentamos duas barras por aplicação na figura: a primeira barra representa a utilização das atualizações enviadas pelo sistema ADSM; e a segunda barra mostra a utilização obtida pela combinação das estratégias.

Esta figura demonstra claramente que a estratégia de restringir o envio de atualizações apenas para as páginas categorizadas como produtor/consumidor(es) ou migratória em *lock*, gera baixos percentuais de atualizações inúteis.

3.4.2 Resultados Gerais

Uma técnica de tolerância a latência de acesso a dados remotos é efetiva quando cobre a maior parte das falhas de acesso utilizando um número relativamente pequeno

Aplicação	*Adaptive++*	Atualizações	SMA	ADSM	Combinação
LU	30%	5%	43%	43%	54%
CG	62%	62%	8%	62%	76%
FFT	19%	14%	29%	39%	38%
Em3d	51%	25%	-72%	-7%	-6%
SOR	25%	27%	12%	37%	37%
IS	57%	-2%	62%	62%	61%
IS_lk	0%	-16%	62%	73%	73%
MigFreq	5%	43%	66%	77%	77%

Tabela 3.2: Redução do tempo de acesso a dados.

de pré-buscas e/ou atualizações (i.e. tendo um percentual relativamente pequeno de operações inúteis). Isso se reflete diretamente na redução do custo de acesso a dados remotos.

Na tabela 3.2, apresentamos o percentual de redução do tempo de acesso a dados obtido por cada estratégia em relação aos resultados de TreadMarks original. A tabela mostra que as aplicações regulares baseadas em barreiras (LU, CG, FFT, Em3d, SOR e IS) apresentam significativas reduções no tempo de acesso a dados com a utilização de *Adaptive++*, variando de 19% a 62%. Esses resultados demonstram que esta técnica tem bastante sucesso em reduzir o custo que ela ataca. Além disso, podemos observar na tabela que *Adaptive++* é a única técnica que nunca gera um aumento do tempo de acesso a dados.

LU e CG são as únicas aplicações da nossa seleção onde a combinação das estratégias consegue um maior percentual de redução do tempo de acesso a dados. LU se beneficia da combinação de *Adaptive++*, através das pré-buscas a falhas de acesso "frias", com a técnica SMA. Enquanto que em CG, a redução do tempo de acesso a dados remotos se deve basicamente à utilização de *Adaptive++* e da técnica de envio de atualizações. A estratégia SMA não beneficiou CG tendo em vista que o custo de acesso a dados tem como fator dominante nesta aplicação a latência da rede.

A aplicação IS obteve uma maior redução do tempo de acesso a dados com a utilização da estratégia SMA, pois essa aplicação modifica suas páginas quase que inteiramente, de forma que o custo de criação e aplicação de diffs é muito alto.

O sistema ADSM consegue obter uma maior redução do tempo de acesso a dados para as aplicações: FFT, SOR, IS_lk e MigFreq. Note que ADSM utiliza uma

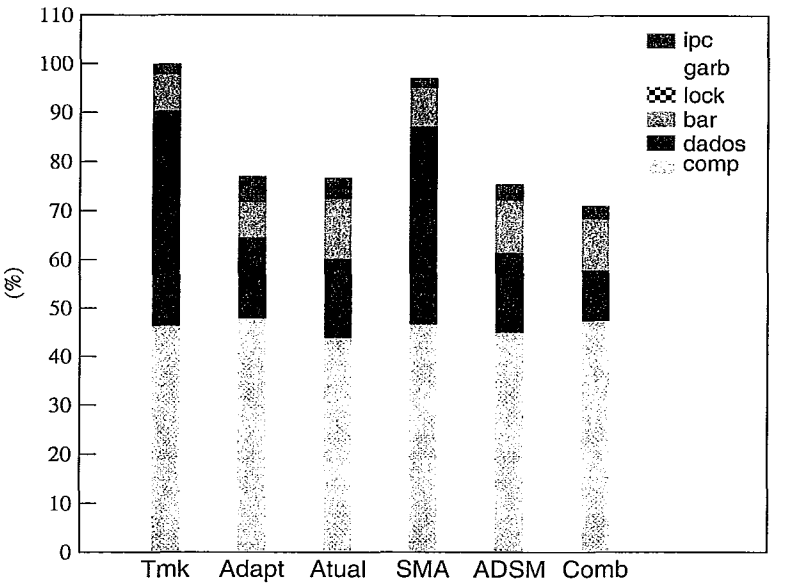

Figura 3.10: Tempo de execução de CG.

combinação de SMA com o envio de atualizações. SOR se beneficia mais do envio de atualizações enquanto que FFT, IS_lk e MigFreq se beneficiam mais da estratégia SMA.

De qualquer forma, mais importantes são os resultados finais de *speedup* e tempo de execução. As figuras 3.7 e 3.8 apresentam os ganhos de *speedup* promovidos por *Adaptive++* (adp), SMA, ADSM e combinação das estratégias (cmb) em relação aos resultados de TreadMarks original, todos os sistemas executando em 8 processadores. Estas figuras demonstram que *Adaptive++* aumenta o *speedup* das aplicações regulares baseadas em barreiras, variando de 5% a 36% de aumento. Dentre essas aplicações, CG e Em3d são aquelas que alcançam um melhor desempenho, enquanto que LU, SOR e IS apresentam ganhos insignificantes, apesar da alta redução do tempo de acesso a dados remotos. Esta anomalia será discutida logo a seguir na análise detalhada dos tempos de execução.

O sistema ADSM consegue extrair ganhos substanciais de *speedup* das aplicações LU, CG, FFT, IS, IS_lk e MigFreq. CG é a única aplicação cujo ganho de *speedup* obtido pela combinação das estratégias é maior do que os ganhos obtidos individualmente pelas técnicas apresentadas. As demais aplicações, com excessão de Em3d, têm seus ganhos com ADSM repetidos na combinação das estratégias. O desempenho de Em3d será detalhado mais adiante.

Para completar a análise de desempenho das estratégias, apresentamos nas figuras 3.9 a 3.16 a quebra do tempo de execução das aplicações sob TreadMarks original,

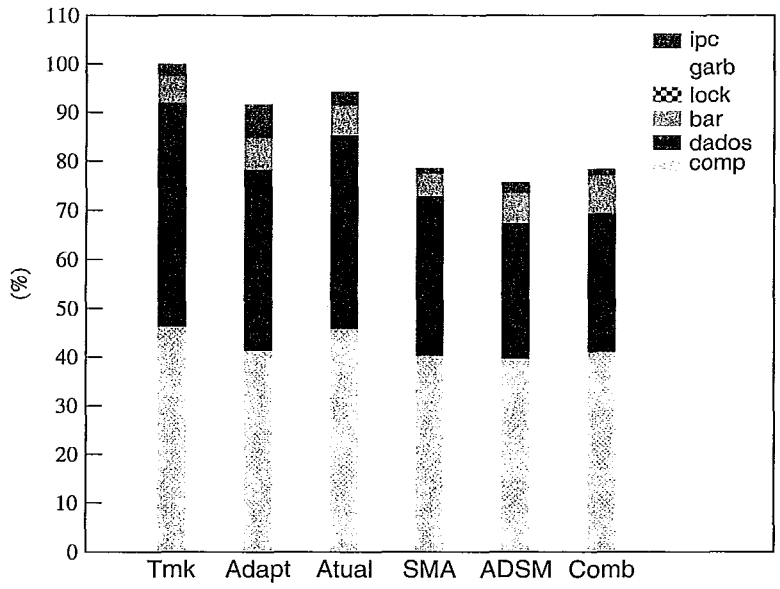

Figura 3.11: Tempo de execução de FFT.

sob TreadMarks utilizando *Adaptive++*, sob TreadMarks utilizando a categorização SPC para guiar o envio de atualizações, sob ADSM sem o envio de atualizações (SMA), sob ADSM, e sob a combinação de ADSM com *Adaptive++*. As execuções foram realizadas em 8 processadores e as barras estão subdivididas da mesma forma que na figura 2.4: tempo de computação útil, tempo de acesso a dados, tempo de sincronização (lock e barreira), tempo despendido em *garbage collection* e tempo de IPC.

Analisando inicialmente o desempenho oferecido pela técnica *Adaptive++*, estas figuras mostram que as aplicações regulares baseadas em barreiras alcançam uma melhora de desempenho de até 25% em Em3d. Note, no entanto, que as reduções significativas do tempo de acesso a dados não são diretamente refletidas no tempo total de execução devido ao aumento do tempo de IPC. O aumento desse custo está relacionado ao recebimento das respostas de pré-buscas. Em uma falha de acesso, os diffs são recebidos sem a ocorrência de interrupções, tendo em vista que o processador está esperando por eles. No caso de uma pré-busca, o processador não sabe quando a resposta será recebida e, portanto, é necessário gerar uma interrupção para indicar o recebimento da resposta de cada pré-busca. Além disso, quando pré-busca é utilizada, os diffs têm que ser copiados para um *pool*. Tal operação não é necessária em uma falha de acesso sem pré-busca, pois os diffs já são recebidos neste *pool*.

Observando o tempo de computação útil das aplicações, verificamos que *Adap-*

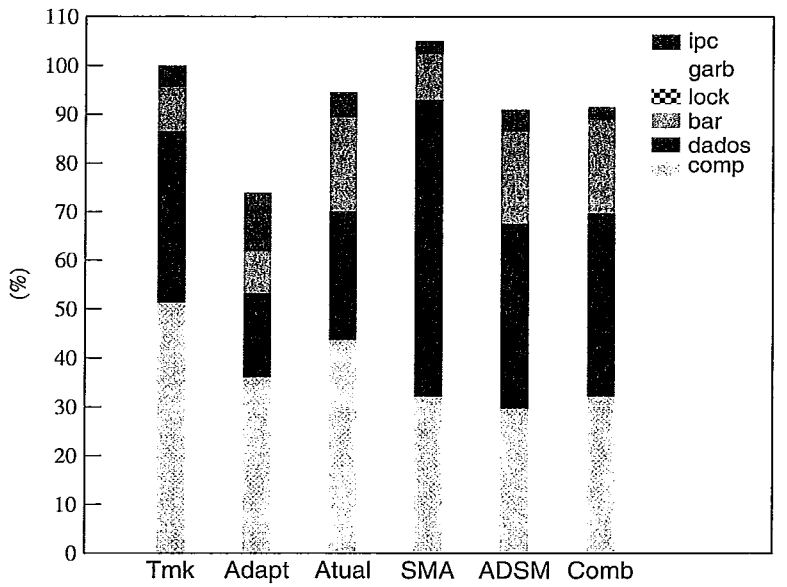

Figura 3.12: Tempo de execução de Em3d.

tive++ diminui este tempo em LU, FFT e Em3d. O comportamento de FFT e Em3d induziu o uso do modo repetição-de-fase iniciando, portanto, pré-buscas durante a barreira. Logo, os diffs que eram criados ao longo de uma fase, passam a ser requisitados (e criados) no início da fase, concentrando o problema de poluição de cache a esse período inicial e diminuindo esse efeito no tempo de computação. LU, por outro lado, utiliza o modo repetição-de-stride que requisita pré-buscas em falhas de acesso, concentrando a criação de diffs em pequenos intervalos de tempo (pois iniciamos pré-buscas de 2 páginas durante uma falha de acesso), diminuindo o efeito da poluição de cache também.

A técnica SMA também reduz consideravelmente o tempo de computação das aplicações LU, FFT, Em3d e SOR. Esta redução também está relacionada a diminuição do efeito da poluição de cache, pois esta técnica elimina a criação de diffs para as páginas com um único escritor.

Outro efeito colateral observado é a redução do custo da barreira. As duas aplicações que utilizam *locks* (IS_lk e MigFreq) têm seus tempos de barreira reduzidos como efeito da redução do tempo de acesso a dados. Ao reduzirmos o tempo dispendido dentro de uma seção crítica (devido a redução do tempo de acesso a dados guardados por *locks*), diminuímos a contenção pelo *lock* e, por conseguinte, diminuímos o desbalancemanto de carga causando uma redução do tempo de espera na barreira.

Observando qual técnica mais contribui para a melhora de desempenho de cada

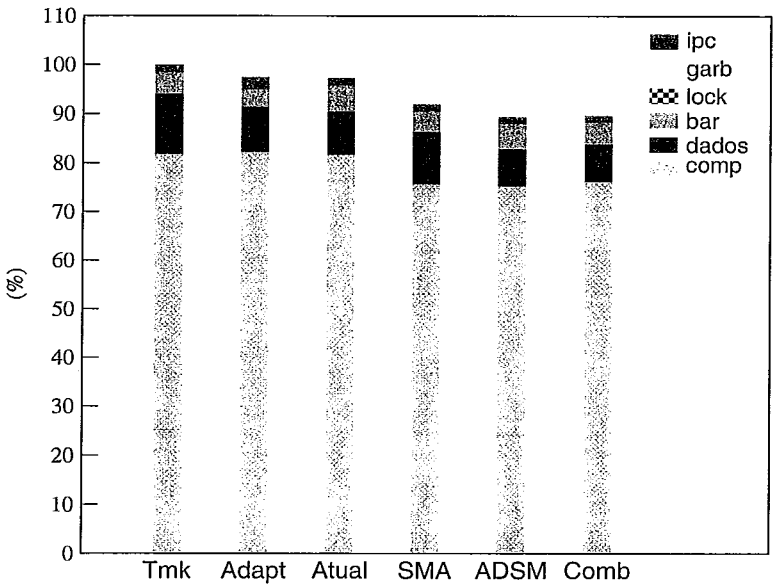

Figura 3.13: Tempo de execução de SOR.

aplicação, verificamos que:

- Em3d se beneficia basicamente da técnica *Adaptive++*.
- LU, FFT e IS têm seu desempenho melhorado principalmente pela técnica SMA.
- SOR, IS_lk e MigFreq obtêm melhor desempenho com a utilização do sistema ADSM, isto é, com a combinação da técnica SMA e do envio de atualizações.
- CG consegue alcançar um melhor desempenho com a combinação de *Adaptive++* com o envio de atualizações.

Além disso, verificamos que CG e Em3d são as únicas aplicações que não se beneficiam do uso da técnica SMA. Na aplicação CG, o custo de tratamento dos pedidos de diff diminui com o uso da técnica (pois 40% dos diffs são eliminados), mas o custo de tratamento dos pedidos de página aumenta de forma que a latência de acesso a dados se mantém igual. O custo de um pedido de página em SMA é maior pois parte da categorização do SPC é implementada no tratamento deste pedido. No caso de Em3d, SMA não melhora seu desempenho porque esta técnica substitui a criação, envio e aplicação de diffs pequenos, pelo envio de páginas, aumentando muito o tráfego na rede. A quantidade de bytes transferidos aumentou 2.4 vezes, enquanto que o tempo de espera por dados devido a rede aumentou 71%. Logo, o

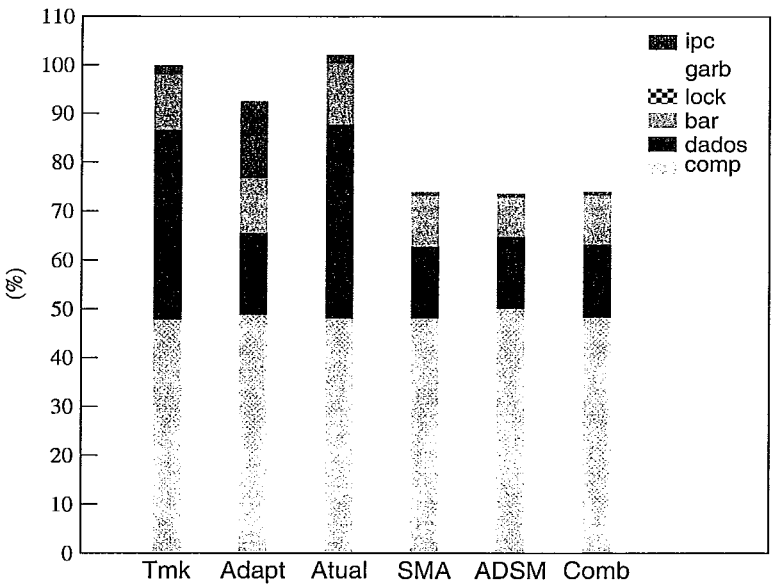

Figura 3.14: Tempo de execução de IS.

aumento destes custos compensou a diminuição do custo de criação e aplicação de diffs.

Observando o desempenho da combinação das estratégias, verificamos que esta consegue extrair ganhos maiores ou iguais aos ganhos produzidos pelas estratégias individualmente, com exceção de Em3d. Os resultados de Em3d demonstram que devemos levar em consideração não apenas o padrão de compartilhamento dos dados, mas também o tamanho dos diffs no momento da escolha da estratégia a ser utilizada. Deste modo evitaríamos o congestionamento gerado na rede devido a troca de páginas que são em média 7.7 vezes maior do que os diffs gerados por esta aplicação. O aumento da quantidade de bytes transferidos devido à utilização da técnica SMA já foi verificado na proposta inicial do sistema ADSM. No entanto, este problema não tinha se traduzido em diminuição de desempenho, como ocorreu no nosso caso, pois utilizamos uma plataforma cuja razão das velocidades do processador e da rede é maior do que do SP2 (plataforma utilizada originalmente pelo sistema ADSM).

3.5 Resultados Simulados de *Adaptive++*

Em [9] apresentamos uma avaliação completa de *Adaptive++* em um ambiente simulado. Os resultados dessa avaliação são semelhantes aos resultados que acabamos de apresentar, confirmando a qualidade das nossas simulações iniciais como indicações do comportamento da nossa técnica.

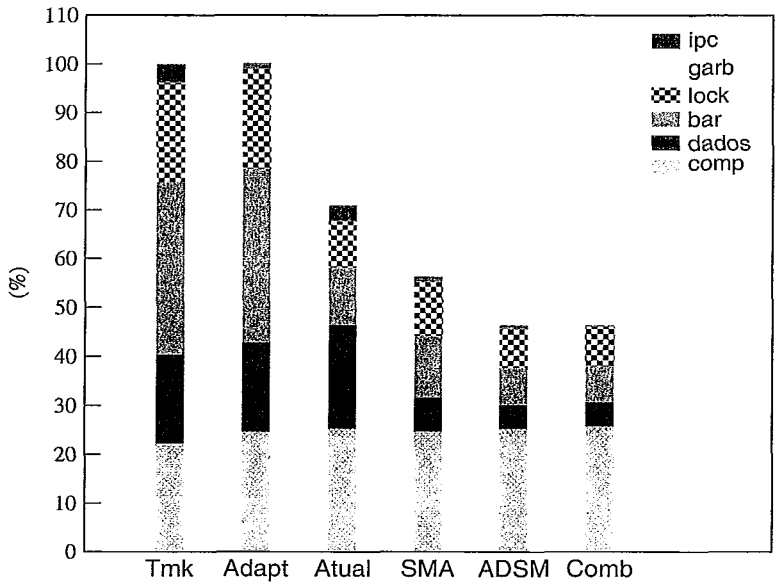

Figura 3.15: Tempo de execução de IS_lk.

Em vista dessa qualidade, não tentamos reproduzir em execuções reais vários resultados interessantes sobre a comparação entre *Adaptive++*, B+ e *Dynamic Aggregation* que são apresentados em [9]. Esses resultados mostram que todas as três técnicas cobrem a maior parte das falhas para aplicações regulares dominadas por falhas de coerência. *Adaptive++* é a única técnica que alcança uma cobertura razoável para aplicações com falhas de acesso "frias" separadas por *stride* fixo. B+ atinge as coberturas mais altas para as aplicações não-regulares. Além disso, *Dynamic Aggregation* e *Adaptive++* iniciam uma fração pequena de pré-buscas inúteis, enquanto B+ em alguns casos desperdiça uma grande quantidade de recursos em pré-buscas inúteis. *Adaptive++* geralmente atinge a melhor eficiência (relação entre cobertura e utilização), gerando grandes reduções no tempo de acessos a dados. A alta eficiência de *Adaptive++* é refletida no seu menor tráfego de dados.

Nossos resultados simulados mostram também que *Adaptive++* não apenas obtém melhor desempenho que as outras técnicas, mas também pode otimizar uma classe maior de aplicações. B+ só lida eficientemente com aplicações que apresentam fases que são repetidas o tempo todo, enquanto que *Adaptive++* pode lidar com essas aplicações e também com aquelas que têm *strides* repetidos. Além disso, *Adaptive++* não degrada o desempenho de aplicações não-regulares. *Dynamic Aggregation* também não degrada o desempenho dessas aplicações, mas obtém desempenho pior do que *Adaptive++* na maioria dos casos.

Nenhuma das técnicas otimiza aplicações não-regulares significativamente; es-

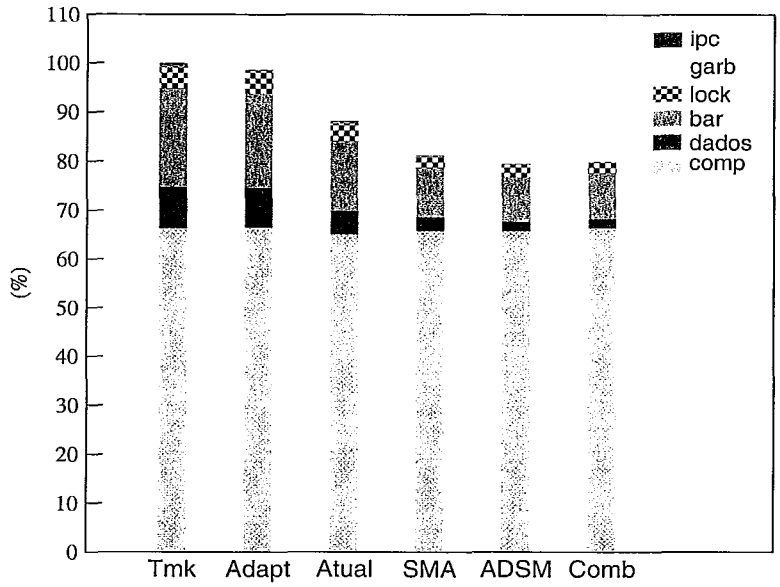

Figura 3.16: Tempo de execução de MigFreq.

sas aplicações precisam de outras melhorias de desempenho. Entretanto, aplicações não-regulares são muito mais difíceis de tratar que aplicações regulares em termos de pré-busca, uma vez que nas primeiras a história de falhas de acesso não serve como uma previsão de falhas futuras. Nesses casos, é necessária a ajuda de compiladores ou programadores que possam inserir chamadas de pré-buscas nas aplicações. Infelizmente, essa situação não é ideal, já que depende de compiladores ou programadores sofisticados. Na verdade, mesmo os compiladores que fazem pré-buscas para *software DSMs* ainda não produzem código que possa competir com código otimizado manualmente [32].

Capítulo 4

Estratégia de *Hardware*

Os principais custos do sistema TreadMarks (e da maioria dos outros *softwares DSMs*) estão relacionados à latência de comunicação e às ações de coerência. A figura 4.1 apresenta uma visão detalhada do desempenho de outras aplicações executando sobre TreadMarks no nosso cluster de 4 nós (uma descrição desse cluster será apresentada na seção 4.2). As barras na figura mostram os tempos de execução normalizados e divididos em: tempo de computação, latência de acesso a dados remotos, e tempo de sincronização. O tempo de computação representa a quantidade de trabalho útil desenvolvido pelo processador na computação. Latência de acesso a dados remotos é uma combinação do tempo de processamento da coerência e da latência da rede envolvida na busca de páginas e diffs como resultado de uma violação de acesso a páginas. O tempo de sincronização representa o retardo envolvido na espera na barreira e na aquisição/liberação de *locks*, incluindo os tempos de processamento de intervalos e de notificações de escrita.

A figura 4.1 mostra que quatro de nossas aplicações tiveram um desempenho muito bom em nosso pequeno cluster. As outras duas aplicações sofreram severos custos de busca de dados remotos e de sincronização quando rodaram sobre TreadMarks. Estes custos devem, certamente, ser muito mais substanciais em configurações de clusters maiores. O tempo de busca de dados remotos inclui os tempos dispendidos na criação e aplicação de diffs e na geração de *twins*. A técnica de diff dinâmico foi desenvolvida para reduzir esses custos através da criação de diffs *on-the-fly* e eliminando a necessidade de *twins*. Infelizmente, dado ao nosso pequeno conjunto de aplicações e ao tamanho pequeno do cluster, não existe muito espaço nesta avaliação para que o diff dinâmico possa atuar e melhorar o desempenho como um todo.

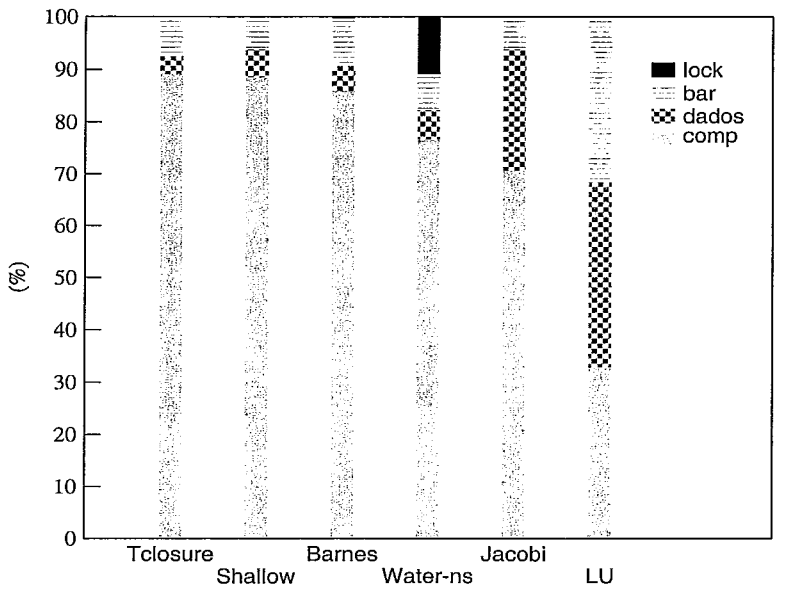

Figura 4.1: Tempos de execução normalizados sob TreadMarks.

4.1 Descrição da Técnica de Diff Dinâmico

Nesta seção apresentaremos a técnica de diff dinâmico depois de apresentar o *hardware* do controlador de protocolos.

4.1.1 *Hardware* para Esconder Latências

O suporte de *hardware* proposto por nós em [7] como parte do projeto NCP_2 é um controlador de protocolos associado a cada processador de uma estação de trabalho do cluster. No sistema NCP_2, o controlador de protocolos e a *interface* de rede estão ambos ligados no barramento PCI. A figura 4.2 oferece uma visão geral do controlador de protocolos, que é composto por uma placa comercial PCI e duas peças de *hardware* customizado. A placa PCI inclui uma *interface* lógica PCI, um microprocessador i960, com até 32 Mbytes de memória, e um *slot* para cartão mezanino. Nossa lógica customizada (rotulada como **SNOOPY_604** e **SNOOPY_CP**) é utilizada para monitorar o barramento de memória detectando todos as escritas feitas na memória distribuída e na manutenção de um registro dessas escritas. O registro de todas as palavras modificadas de uma página é mantido em um vetor de *bits* (chamado de Memória de Vetor), onde cada *bit* representa uma palavra. Sempre que o controlador de protocolos detecta que o processador escreveu em uma pala-

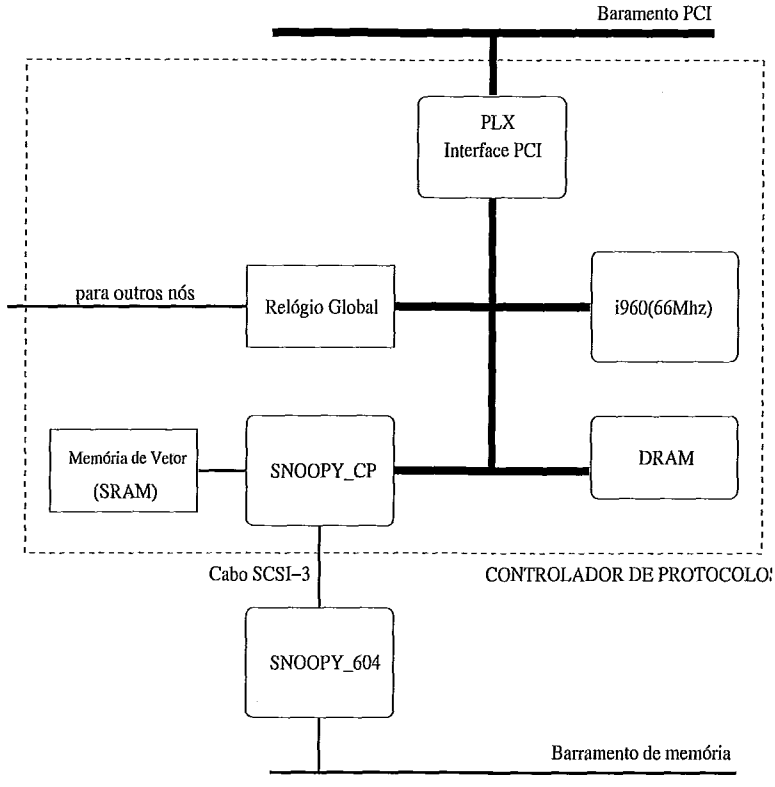

Figura 4.2: Controlador de protocolos.

vra da memória, ele simplesmente liga[1] o *bit* correspondente em sua memória para registrar o evento. O nosso protótipo do controlador é capaz de registrar escritas a no máximo 32 Mbytes de dados distribuídos. A principal parte da lógica dedicada é implementada em uma única *Erasable Programmable Logic Device* (EPLD) acoplada a um cartão mezanino, e é responsável pela codificação das escritas no vetor de *bits*. A lógica do monitoramento do barramento de memória é implementada em outra EPLD localizada no ponto de conexão do barramento (no conector da cache L2).

Embora o controlador de protocolos possa implementar e/ou simplesmente beneficiar outras técnicas de tolerância a atrasos em *software DSMs*, restringiremos nossa avaliação a técnica de diff dinâmico.

4.1.2 Gerando Diffs Dinamicamente

Nosso objetivo é reduzir os custos de operações relacionadas com diffs de *software DSMs* cuja unidade de coerência é a página e permite a existência de múltiplos escritores concorrentes, usando TreadMarks como um exemplo de tais sistemas.

[1]Usaremos os termo ligar/desligar um *bit* para indicar que o *bit* receberá o valor 1 ou 0 correspondentemente

Nesses sistemas a criação de diffs é antecipada pela criação de uma cópia *twin* de cada página trazida à memória local no instante da primeira escrita na página. O *twin* é posteriormente comparado com a versão corrente da página para definir as modificações que foram feitas desde que o *twin* foi criado. Esta técnica, a qual nós denominamos **diff estático**, apresenta duas principais desvantagens:

- A criação de *twin* gera um custo na coerência de dados e ainda pode poluir as caches;
- A geração de diffs envolve uma comparação palavra-por-palavra de duas páginas, que não somente representa mais custos à coerência e pode poluir as caches, como também, aumentar o tráfego no barramento de memória.

Na técnica de diff dinâmico, não existe a necessidade de páginas *twins*, uma vez que todas as modificações feitas a uma página compartilhada são monitoradas durante a execução da aplicação. Isto só é possível pois obrigamos a cache a escrever os dados compartilhados através do barramento de memória, de forma que é mantido um registro de todas as palavras modificadas, conforme explicado na seção anterior.

Quando um processador recebe um pedido de diff, ele simplesmente lê (através do barramento PCI) o vetor de *bits* correspondente à página requisitada, verificando todos os *bits* que estão no conjunto, e lê as palavras correspondentes da memória para gerar o diff. (Mesmo que nós pudéssemos ter usado o próprio vetor de *bits* para descrever o conteúdo de um diff, para simplificar nossa implementação, nós preferimos não modificar o formato atual do diff usado pelo TreadMarks). Assim, uma operação de *twinning* sob a técnica de diff dinâmico envolve, simplesmente, o desligamento do vetor de *bits* correspondente à página solicitada.

Em resumo, a técnica de diff dinâmico pode melhorar o desempenho, eliminando alguns dos atrasos causados pelo diff estático, bem como melhorando os tempos de sincronização (espera pelo *lock* e desbalanceamento de carga) como um efeito colateral de uma manutenção de coerência mais rápida.

4.2 Metodologia

Nossos experimentos foram realizados no protótipo NCP_2, que tem um cluster com 4 nós interconectados por uma *switch* Myrinet. Cada nó do cluster é formado por um PowerPC-604 de 100MHz, com 64Mbytes de memória, 16Kbytes divididos em duas

Aplicação	Entrada
Barnes	16 K corpos, 10 iterações
Jacobi	2048 × 2048 reais, 100 iterações
LU	800 × 800 doubles
Shallow	500 × 500 doubles
Tclosure	1024 × 1024 longs
Water	4096 moléculas

Tabela 4.1: Aplicações e suas entradas.

caches (uma de dados e uma de instruções) de primeiro nível, e uma controladora de protocolos.

O sistema TreadMarks tem sido usado como nossa linha básica em sistemas *software DSMs* com algumas modificações na parte de comunicação, devido a utilização da biblioteca da Myricom e do seu programa de controle de redes (MCP). Nós comparamos esta versão de TreadMarks, que usa diff estático, com uma que utiliza diff dinâmico. Nossos resultados são baseados na execução das seis aplicações apresentadas na tabela 4.1 juntamente com suas entradas. A seguir temos uma breve descrição dessas aplicações.

Barnes simula a interação de um sistema de `N` corpos, sob a influência de forças gravitacionais usando o método *Barnes-Hut hierarchical N-body.* **Jacobi** é um método iterativo para resolver equações diferenciais parciais, tendo como computação principal a média dos vizinhos mais próximos. **LU** fatora uma matrix densa em um produto de duas matrizes triangulares, sendo uma superior e a outra inferior. **Shallow** é um *benchmark* que soluciona equações diferenciais em um *grid* de duas dimensões para a predição do tempo. A computação do *transitive closure* (**Tclosure**) de um grafo G ($G = (V, E)$, onde V é conjunto de vértices e E é o conjunto de arestas), equivale a descobrir o grafo $G+ = (V, E+)$ de forma que para todo v, w pertencentes a V existe uma aresta (v, w) em $E+$ se e somente se existir um caminho não nulo de v para w em E. **Water** é uma simulação dinâmica de moléculas, que calcula forças entre- e intra-moleculares sobre um conjunto de `N` moléculas de água.

4.3 Resultados Experimentais

Nesta seção apresentamos a nossa avaliação do desempenho da técnica de diff dinâmico. Todas as figuras nesta seção mostram duas barras para cada aplicação,

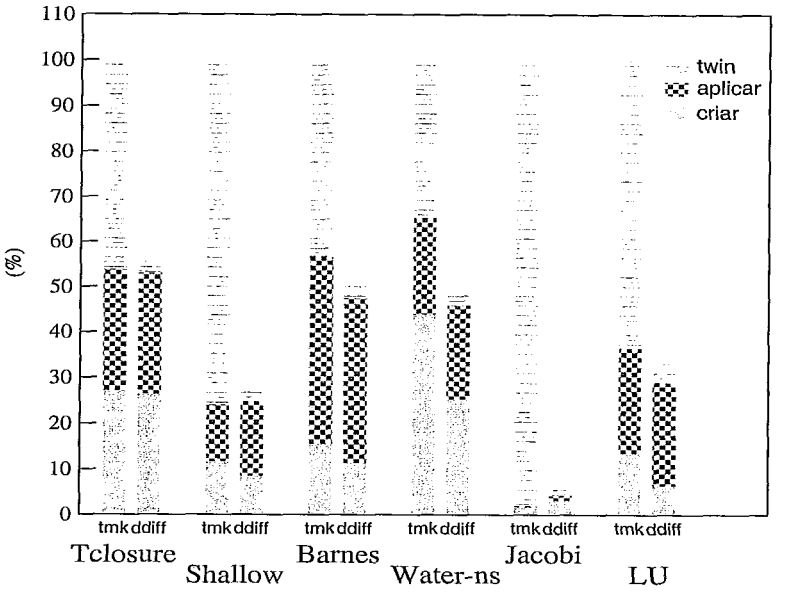

Figura 4.3: Custos relacionados a diffs.

representando: diff estático (à esquerda) e diff dinâmico (à direita). Tendo em vista que a nossa técnica focaliza os custos relacionados a diffs, vamos inicialmente apresentar esses custos na figura 4.3. A figura separa os custos relacionados a diffs em: criação de diff, aplicação de diff, e criação de *twin*. Todas as barras estão normalizadas com respeito aos resultados do diff estático. A figura mostra que a técnica de diff dinâmico reduz significativamente os custos relacionados a diffs; reduções na faixa de 42% a 92%. Considerando cada custo separadamente, verificamos que o custo de criação de *twins*, que é o custo dominante em diff estático, é quase totalmente eliminado ao utilizarmos diff dinâmico. O custo de aplicação de diffs permanece o mesmo, uma vez que não modificamos o formato do diff e, deste modo, utilizamos o código original de TreadMarks para aplicar os diffs.

Com respeito ao custo de criação de diffs, verificamos que Tclosure não se beneficia com a utilização do diff dinâmico, uma vez que suas páginas são quase inteiramente modificadas todas as vezes que são escritas. Por outro lado, Shallow, Barnes, Water e LU obtiveram reduções em seus custos numa faixa de 33% a 43%. Jacobi é a única aplicação que apresenta um aumento de seus custos sob diff dinâmico. O problema com esta aplicação é que a maioria de suas escritas não altera o conteúdo das palavras escritas. Como resultado, a maioria dos diffs estão vazios usando diff estático. Em contra partida, com a utilização do diff dinâmico, o *hardware* de monitoramento detecta que uma palavra foi modificada sem nenhum conhecimento do seu conteúdo. Sendo assim, para esta aplicação, os diffs dinâmicos são muito

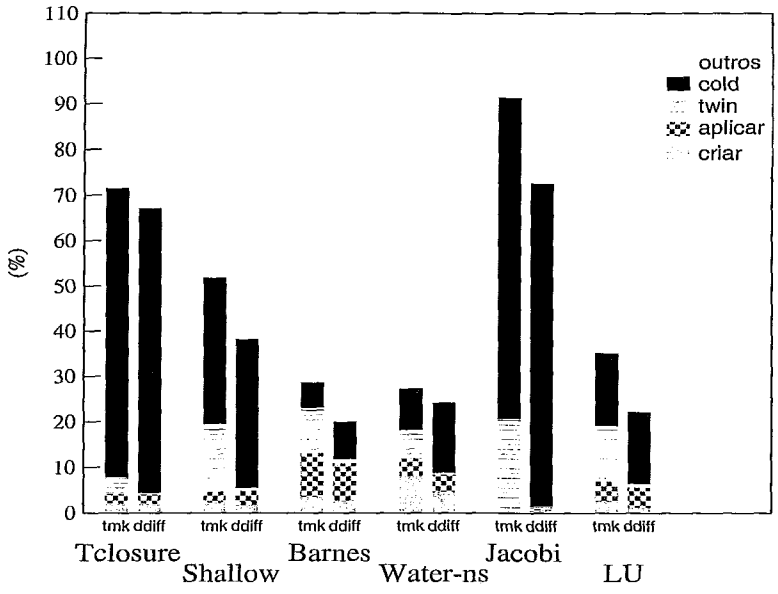

Figura 4.4: Tempo de acesso a dados.

maiores do que os diffs estáticos, provocando uma dilatação do tempo de criação de diffs dinâmicos com relação ao tempo de criação de diffs estáticos.

A figura 4.4 ilustra o impacto do diff dinâmico no tempo de acesso a dados (experimentado por cada processador) como um todo. A figura separa os tempos de acesso a dados nas três categorias estudadas na figura 4.3, assim como o custo de trazer as páginas quando são acessadas pela primeira vez (*cold start*) e outros custos que incluem latência da rede e atrasos devido a contenções remotas. Novamente todas as barras são normalizadas em relação aos resultados de diff estático. A figura mostra que a busca de páginas "frias" e outros custos dominam o tempo de acesso a dados em todos os casos. Os custos relacionado a diffs representam no máximo 20% usando diff estático. No entanto, o diff dinâmico pode reduzir o tempo de acesso a dados em até 30% (Barnes), sendo a maior parte desta redução resultado da diminuição dos custos relacionados a diffs e pelo fato de que nós remotos passam menos tempo tratando de pedidos.

Finalmente, a figura 4.5 apresenta a melhora no tempo de execução alcançado por cada aplicação ao utilizar diff dinâmico. As barras nesta figura estão divididas nas mesmas categorias apresentadas na figura 4.1. Todas as barras estão normalizadas com respeito aos resultados de diff estático. Podemos ver nesta figura que o diff dinâmico pode reduzir o tempo de execução em até 12% para as aplicações selecionadas.

As melhoras de desempenho alcançadas por Tclosure e Shallow sob diff dinâmico

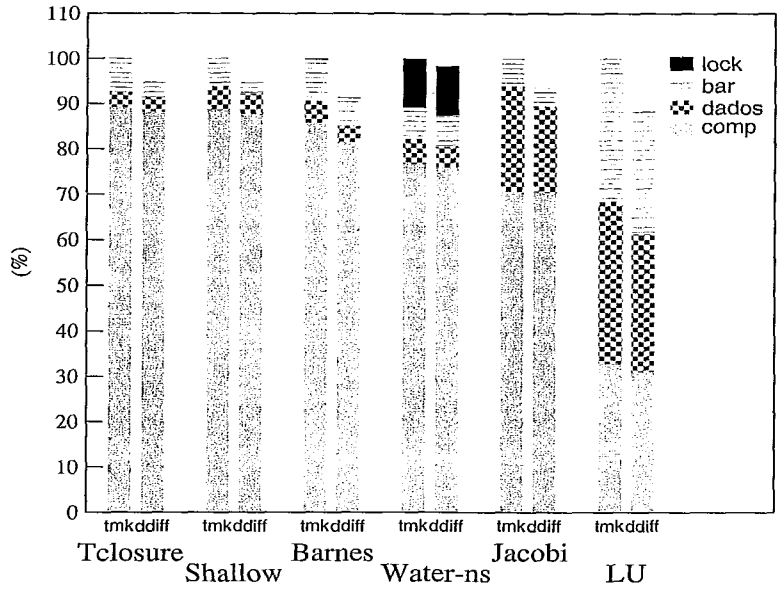

Figura 4.5: Tempo de execução.

se devem principalmente a melhoras no tempo dispendido nas barreiras. Nestas aplicações as barreiras são aceleradas por causa da forma com que os dados são compartilhados. Mais especificamente, as estruturas de dados compartilhadas são particionadas entre os nós de forma que cada nó escreve em sua própria partição. Assim que os dados compartilhados são inicialmente alocados no nó 0, este nó tem permissão de ler e escrever em todas as páginas compartilhadas, inclusive as páginas da sua partição. Sob diff estático, os outros nós têm que criar um *twin* para cada página recebida do nó 0 e posteriormente escrita. Entretanto, mesmo sob diff estático, o nó 0 não tem que criar *twins* e assim alcança mais rapidamente as barreiras do que os outros nós. Utilizando diff dinâmico, nenhum nó precisa criar *twins*, dessa forma o desbalanceamento de carga é reduzido. As barreiras em Jacobi têm seu desempenho melhorado devido a uma razão similar.

O desempenho das barreiras em Barnes e LU é também melhorado com o uso de diff dinâmico. A redução do tempo de barreira verificada em Barnes é consequência do seu algoritmo. Durante cada iteração de Barnes, uma árvore é gerada por somente um dos nós enquanto que os outros esperam num ponto de barreira. Como a geração dessa árvore é executada mais eficientemente com o auxílio do diff dinâmico, os outros nós levam menos tempo esperando na barreira, de forma que o desbalanceamento de carga é reduzido. Barnes também apresenta ganhos no tempo de acesso a dados e no tempo de computação útil com a utilização de diff dinâmico. A razão para a diminuição do tempo de acesso a dados foi apresentada na seção anterior,

Aplicação	Criação de diffs	Geração de *twins*	Total
LU	3.5%	1.9%	5.4%
CG	1.7%	0.9%	2.6%
FFT	10.5%	4.6%	15.1%
Em3d	8.8%	3.9%	12.6%
SOR	5.4%	6.5%	11.9%
IS	3.2%	0.5%	3.7%
IS_lk	3.6%	0.8%	4.4%
MigFreq	5.4%	1.8%	7.2%

Tabela 4.2: Percentual do tempo de acesso a dados remotos.

enquanto que a razão para esta última melhoria está relacionada a ocorrência de um melhor comportamento da cache (menos poluição da cache).

LU é a aplicação que apresenta melhor ganho de desempenho com o uso da nossa técnica. Ela apresenta ganhos relacionados ao comportamento na barreira e de acesso a dados. A diminuição do tempo de acesso a dados em LU é resultado da eliminação dos *twins*. As barreiras de LU se tornaram mais rápidas basicamente pelas mesmas razões que as de Barnes.

Em resumo, os nossos experimentos demonstram que a técnica de diff dinâmico é efetiva na redução dos custos relacionados a diffs. Nossa avaliação também mostra que essa redução freqüentemente se reflete em outros aspectos de desempenho das aplicações, incluindo o custo de barreira. No entanto, o ganho de desempenho alcançado por diff dinâmico não é significativo para as nossas aplicações (que já apresentam um bom desempenho) no nosso pequeno cluster.

4.4 Resultados Estimados

Nesta seção apresentamos uma estimativa dos ganhos que a técnica de diff dinâmico poderia obter na plataforma usada na avaliação das estratégias de *software* usando as aplicações empregadas naquela avaliação.

Na tabela 4.2 mostramos o percentual do tempo de acesso a dados remotos que é despendido na criação de diffs e na geração de *twins* das aplicações rodando sobre TreadMarks original no cluster de 8 processadores. Comparando com os resultados experimentais, mais especificamente com a figura 4.4, verificamos que a principal diferença é que o percentual do tempo usado na geração de *twins* é menor na plata-

Aplicação	Operação com diffs	Computação útil	Total
LU	0.5%	14.6%	15.1%
CG	0.7%	0.0%	0.7%
FFT	4.3%	6.2%	10.5%
Em3d	3.8%	20.2%	24.0%
SOR	0.8%	6.8%	7.6%
IS	0.9%	0.0%	0.9%
IS_lk	0.5%	0.0%	0.5%
MigFreq	0.4%	0.0%	0.4%

Tabela 4.3: Percentual de redução do tempo total de execução.

forma de 8 nós.

Vamos assumir que a técnica de diff dinâmico reduz os custos das operações com diffs em 62% que corresponde à média das reduções obtidas pelos resultados experimentais. A estratégia obteria então as reduções no tempo total de execução apresentadas na segunda coluna da tabela 4.3. Além disso, tendo em vista que esta técnica reduz o efeito de poluição de cache da mesma forma que a estratégia SMA, usamos o percentual de redução do tempo de computação útil alcançado por SMA para estimar este efeito na utilização do diff dinâmico. A terceira coluna da tabela 4.3 apresenta o percentual de redução do tempo total de execução assumindo que a estratégia de diff dinâmico obtém a mesma redução do tempo de computação útil gerada por SMA. E finalmente, na última coluna da mesma tabela apresentamos o percentual total de redução estimado para o tempo de execução das aplicações. Esses resultados sugerem que os ganhos que obteríamos no novo *cluster* de 8 nós seriam bem mais significativos do que os ganhos apresentados na seção anterior.

É importante notar também que os maiores ganhos estimados se devem principalmente a diminuição do tempo de computação útil, que só foi observada em uma aplicação no NCP_2. O efeito da poluição de cache é mais acentuado no novo *cluster* pois os processadores têm uma velocidade 6.5 vezes maior, de forma que o custo de acessar a memória principal é muito maior. Além disso, não utilizamos cache de nível 2 no NCP_2 devido a problemas operacionais do protótipo.

Capítulo 5

Trabalhos Relacionados

Os trabalhos relacionados a esta tese serão divididos em duas partes: trabalhos relacionados às estratégias de *software* e aqueles relacionados à estratégia de *hardware*.

5.1 Estratégias de *Software*

Os principais trabalhos relacionados a este tipo de estratégia são aqueles que tratam de estratégias de pré-busca dinâmica para *software DSMs*, e os que tratam de combinações entre estratégias de tolerância à latência de acessos a dados remotos no contexto de *software DSMs*.

Técnicas de pré-busca dinâmica. Em termos de técnicas de pré-busca dinâmica para *software DSMs*, dois trabalhos se destacam: B+ e *Dynamic Aggregation*. B+ foi proposto por nós em [7]. Essa técnica foi a primeira proposta de pré-busca para sistemas *software DSM*. Em [9] apresentamos uma comparação detalhada entre os desempenhos de *Adaptive++*, de B+ e do *Dynamic Aggregation*. Os resultados dessa comparação nos foram amplamente favoráveis devido às diferenças entre a nossa estratégia e as suas competidoras.

B+ difere de *Adaptive++* em vários pontos importantes:

- B+ solicita pré-buscas depois da aquisição de *locks* e barreiras, enquanto que *Adaptive++* não faz pré-buscas depois da aquisição de *locks* e requisita pré-buscas em falhas de acesso também;
- B+ se baseia em invalidações de páginas, enquanto que ambos os modos de operação de Adaptive++ se baseiam no histórico das falhas de acesso;
- B+ não pára de executar pré-buscas em aplicações irregulares, enquanto que

Adaptive++ pára de solicitar pré-buscas ao identificar a irregularidade da aplicação; e

- B+ não trata de falhas de acesso que são "frias" (*cold start*), enquanto que Adaptive++ ataca este tipo de falha de acesso.

Dynamic Aggregation difere de *Adaptive++* em pelo menos dois pontos importantes:

- *Dynamic Aggregation* só realiza pré-buscas em falhas de acesso que necessitam de comunicação remota, enquanto que *Adaptive++* também solicita pré-buscas em falhas que não requerem comunicação remota adicional, e depois de eventos de barreira no modo repetição-de-fase; e

- *Dynamic Aggregation* não lida com falhas de acesso "frias", enquanto que *Adaptive++* lida.

Além dessas técnicas, o trabalho que mais se aproxima do nosso é o de Karlsson e Stenström [21]. Nele, os autores propõem uma nova técnica de pré-busca (chamada por nós de KS) que é, na essência, similar a nossa técnica *Adaptive++* no sentido de que também é dinâmica, ajustável, e está implementada sob TreadMarks. No entanto, os detalhes específicos das duas técnicas diferem bastante. Assim como na nossa técnica de pré-busca, KS guarda o conjunto de falhas de páginas geradas durante cada fase da aplicação. Além dessa informação, KS também coleciona as invalidações recebidas dos nós remotos. Essas informações são usadas para implementar uma estratégia de pré-busca baseada em histórico, onde é buscado um compartilhamento de dados do tipo produtor/consumidor. Se este padrão é encontrado, a técnica prevê que ele será repetido no futuro e as pré-buscas correspondentes são requisitadas. Entretanto, se este padrão não é encontrado, a técnica recorre para pré-busca estritamente seqüencial (tal como em *hardware DSMs*, mas com uma unidade de pré-busca substancialmente maior). A nossa técnica difere de KS de cinco formas significativas:

- O impacto dos custos de coerência e de pré-busca é menos severo em KS do que em *Adaptive++*. Isto resulta do fato de KS ter sido proposto para uma rede de multiprocessadores, em vez de uma rede de uniprocessadores, como *Adaptive++*. Mais especificamente, o uso de nós multiprocessadores em KS

permite que algumas operações relacionadas a pré-busca sejam executadas em qualquer processador que esteja disponível (não esteja executando nenhuma computação útil) no momento no nó multiprocessado. Além disso, o tratamento de interrupções pode ser distribuído entre os diferentes processadores de cada multiprocessador;

- KS inicia pré-buscas logo depois de pontos de aquisição de *locks* ou saída de barreiras no modo de pré-busca por histórico, enquanto que *Adaptive++* não inicia pré-buscas depois da aquisição de *locks* para evitar o aumento das seções críticas. KS é capaz de solicitar pré-buscas na aquisição de *locks* sem ter seu desempenho seriamente penalizado, porque o custo gerado por pré-buscas pode ser escondido como mencionado acima;

- KS não espera pela ocorrência de falhas de acesso para requisitar pré-buscas no modo de pré-busca por histórico. Por outro lado, *Adaptive++* tenta espalhar as requisições de pré-buscas ao longo do tempo nos dois modos de operação. KS é capaz de requisitar um número potencialmente grande de pré-buscas de uma única vez sem sofrer uma séria queda de desempenho, porque o custo de interromper nós remotos pode ser distribuído da forma mencionada acima;

- KS restringe as pré-buscas a dados do tipo produtor/consumidor, enquanto que *Adaptive++* é mais genérico, não fazendo este tipo de restrição; e

- KS aplica pré-busca estritamente seqüencial, enquanto que *Adaptive++* é, novamente, mais genérico e pode lidar com *strides* maiores de falhas de páginas como aqueles encontrados em aplicações tais como FFT.

Uma ferramenta interessante de suporte a técnicas de pré-busca dinâmica foi proposta por Keleher [22]. A sua biblioteca *Sparks DSM* provê uma clara interface para registrar o histórico de falhas de páginas, da mesma forma que a nossa técnica de pré-busca registra. A interface é suficientemente genérica, de forma que técnicas sofisticadas de atualização ou pré-busca podem ser facilmente implementadas.

Finalmente, em um dos nossos trabalhos anteriores [7], propusemos o uso de um suporte simples de *hardware* para tolerar agressivamente latências em *software DSMs*. Nós avaliamos a técnica de pré-busca B+ implementada para TreadMarks padrão e para uma versão modificada do sistema que explora o *hardware* extra. Nossos experimentos detectaram os problemas de desempenho do B+, mas mostraram

que ele pode se beneficiar substancialmente do uso do nosso suporte de *hardware*. Acreditamos que a técnica *Adaptive++* pode se beneficiar deste suporte ainda mais significativamente do que B+.

Combinação de técnicas de tolerância à latência. A literatura não contém muitos trabalhos onde diferentes técnicas de tolerância à latência de acessos a dados remotos são combinadas no contexto de *software DSMs*. Em [32], a técnica *software prefetching* (pré-busca dirigida pelo compilador) é combinada com *multithreading*, ambos implementados sobre TreadMarks. A combinação dessas técnicas não levou a resultados conclusivos, uma vez que, em alguns casos, uma das técnicas aplicada isoladamente obtém desempenho superior a sua combinação. Esse resultado é uma conseqüência do fato que *software prefetching* e *multithreading* procuravam tolerar a latência das mesmas falhas de acesso.

As técnicas de pré-busca e envio de atualizações foram combinadas em [14], também implementado sobre TreadMarks. Nesse trabalho, as técnicas são combinadas de forma semelhante a que propomos no capítulo 3. Mais especificamente, a técnica de pré-busca é aplicada àquelas páginas categorizadas como tendo múltiplos escritores. As pré-buscas são iniciadas em operações de barreira. Já a técnica de envio de atualizações é utilizada para páginas categorizadas como tendo um único escritor, com atualizações enviadas tanto em transferências de *lock* quanto em barreiras. A combinação dessas técnicas foi estudada em simulação para apenas uma aplicação científica, demonstrando bons resultados.

Existem algumas diferenças importantes entre o nosso trabalho e o apresentado em [14]:

- As técnicas que são combinadas no nosso trabalho são, em geral, mais eficientes, isto é, cobrem um grande número de falhas de acesso gerando um baixo percentual de operações inúteis e, por conseguinte, diminuem o custo de acesso a dados remotos. [14] utiliza uma técnica de pré-busca muito semelhante a B+, se baseando portanto nas invalidações recebidas. A principal diferença é que B+ não verifica o padrão de compartilhamento da página a ser atualizada via pré-buscas.

- Outra diferença está na categorização do padrão de compartilhamento. A categorização proposta no sistema ADSM é muito mais simples, gerando o mínimo de intervenção possível no protocolo, mas podendo não ser tão exata;

e

- Nossos resultados foram obtidos através da execução real dos sistemas, não através de simulações. É muito difícil reproduzir em simulações o instante e a ordem exata do acontecimento de eventos. Logo, a categorização do padrão de compartilhamento, por exemplo, pode sofrer algum desvio na simulação, mudando totalmente as ações que deveriam ser tomadas pelo sistema real.

Em [5], a técnica *Dynamic Aggregation* foi combinada com uma estratégia de atualizações que atualiza páginas tanto em transferências de *lock* quanto em barreiras, como em ADSM, mas que não leva em conta o padrão de compartilhamento das páginas compartilhadas. Os resultados dessa combinação não foram favoráveis, uma vez que as duas técnicas combinadas atacavam as mesmas falhas de acesso.

O sistema ADSM proposto em [30] também utiliza uma combinação de estratégias. ADSM se adapta entre protocolos único-escritor e múltiplos-escritores, e utiliza a estratégia de envio de atualizações para a manutenção da coerência das páginas categorizadas como migratórias em *locks* ou como produtor/consumidor(es). Nosso trabalho faz uma extensão do sistema ADSM adicionando uma estratégia de pré-busca ao conjunto de estratégias combinadas. Além disso, fazemos uma análise do sistema ADSM em uma plataforma mais atual do que o sistema SP2 que foi inicialmente utilizado na proposta de ADSM.

Outro exemplo de um sistema que combina várias estratégias de tolerância a latência de acesso a dados remotos, é o sistema HAP proposto em [46]. Este sistema faz uma adaptação das estratégias propostas em ADSM para um sistema baseado em *homes*, mais especificamente para o sistema HLRC [48]. Além disso, HAP propõe uma migração dinâmica do *home* das páginas categorizadas como migratórias e produtor/consumidor(es). No entanto, HAP não estuda uma combinação com uma estratégia de pré-busca.

5.2 Estratégia de *Hardware*

Devido a sua simplicidade e a sua estruturação sobre componentes comerciais, nosso suporte de *hardware* leva a custos baixos e a um ciclo curto de projeto, colocando-o em uma classe diferente de sistemas de memória compartilhada como os multiprocessadores com coerência a nível de cache (e.g. [29, 2, 28]). Tais multiprocessadores alcançam um alto desempenho as custas de um projeto complexo de *hardware*.

Devido à complexidade do *hardware*, sistemas desta classe podem terminar com microprocessadores obsoletos, isto é, o microprocessador escolhido no início do projeto pode no fim ser mais lento do que outro microprocessador comercial devido ao tempo gasto na operacionalização da máquina. Basear o projeto da máquina em um microprocessador da próxima geração pode minimizar esse problema, mas pode não resolvê-lo dependendo do tempo despendido na construção da máquina.

Nosso trabalho utiliza idéias relacionadas com a pesquisa em processadores de protocolos programáveis onde os projetos FLASH de Stanford [27] e Typhoon de Wisconsin [35] foram pioneiros. Estes sistemas procuram fornecer um compartilhamento de dados com granularidade fina, flexível e eficiente, a nível de blocos de cache. A utilização de unidades de coerência e de transferência de dados tão pequenas requer processadores de protocolos extremamente otimizados para evitar que o seu desempenho se torne um gargalo. Mais importante, entretanto, é a baixa latência de rede requerida pela transferência de mensagens curtas para garantir uma execução eficiente; estas redes devem estar altamente acopladas ao restante do nó e ao processador de protocolo, tornando o projeto mais complexo e caro. Em contraste, nossa coerência baseada em páginas permite o uso de um processador de protocolos muito mais simples e de baixo custo, e de redes comerciais ligadas através de um barramento PCI.

Os sistemas que tentam fornecer compartilhamento de dados com granularidade fina em *software*, mas sem a utilização de um *hardware* tão customizado [36, 37, 38] também requerem uma forte integração entre a rede de interconexão e o resto do nó e exige muito da rede, especialmete com relação a latência.

O sistema GeNIMA [11] baseado em *home* propôs o método do "diff direto" que diminui os custos de empacotar/desempacotar, de interromper e de aplicar diffs no lado da recepção. Quando o processador local tem que computar um diff, ele envia as palavras que foram modificadas de forma contínua e direta ao *home* enquanto compara a página com o seu *twin*, em vez de empacotar todo o grupo de modificações, enviando-os em uma única transferência. Infelizmente, os diffs diretos podem aumentar substancialmente o número de mensagens requeridas para atualizar uma página no nó *home* correspondente. Outros *software DSMs* anteriores tentaram evitar a computação e a aplicação de diffs. O sistema baseado em *homes* denominado *Automatic Update Release Consistency* (AURC) [20] tira vantagem da propagação automática das escritas para os nós *home*, baseado em uma cache *write-through* e

no monitoramento de escritas usando a *interface* de rede SHRIMP [12]. De forma similar, o sistema Cashemere [26] usa a capacidade de escritas remotas com granularidade fina da *interface* de rede DEC Memory Channel. Entretanto, o barramento de memória não é monitorada pela *interface* de rede. Sendo assim, utiliza-se uma instrumentação do código para propagar as escritas relevantes ao nó *home*. Todos os sistemas mencionados usam características específicas da *interface* de rede para melhorar o desempenho. Em contrapartida, nossa abordagem não se baseia em nenhuma característica especial da *interface* de rede.

Muitos *softwares DSMs* baseados em páginas e que permitem múltiplos escritores concorrentes têm sido propostos desde o projeto e a avaliação de Munin [13], e.g. TreadMarks [25], HLRC [48], Cashemere [42], Brazos [41], e AEC [39]. Várias otimizações para esses sistemas têm sido propostas, e.g. [9, 30, 5]. Nossa estratégica de diff dinâmico pode trazer benefícios a todos esses sistemas.

Capítulo 6

Conclusões

6.1 Resumo dos Resultados

Nesta tese apresentamos novas estratégias de *software* e *hardware* que toleram e/ou reduzem a latência de acesso a dados remotos em *software DSMs*. Considerando inicialmente as estratégias de *software*, nós propusemos e avaliamos a técnica *Adaptive++* de pré-busca dinâmica e adaptativa para o sistema TreadMarks, cujo objetivo é otimizar o desempenho de aplicações regulares. Nossos resultados experimentais mostraram que a nossa técnica de pré-busca alcança uma melhora de *speedup* em relação a TreadMarks de até 35% para aplicações regulares rodando em 8 processadores. Uma comparação simulada de *Adaptive++* com outras técnicas de pré-busca dinâmica mostrou que nossa estratégia alcança as maiores reduções da latência de acesso a dados remotos.

Estudamos também a combinação de *Adaptive++* com duas estratégias de tolerância a latência de acesso a dados remotos: adaptação entre protocolos único-escritor e múltiplos-escritores (SMA), e adaptação entre a manutenção da coerência de memória via invalidação e via atualização. Esta combinação foi implementada através da combinação do sistema ADSM com a nossa técnica de pré-busca. Nossos experimentos demonstram que a combinação das estratégias consegue extrair ganhos maiores ou iguais aos ganhos produzidos pelas estratégias individualmente, com exceção de uma aplicação. As aplicações baseadas em barreiras tiveram seus *speedups* melhorados em até 40%, enquanto as aplicações baseadas em *locks* obtiveram ganhos de *speedup* de até 118%.

Em relação a estratégias de *hardware*, nós propusemos a técnica de diff dinâmico, que procura reduzir os custos de coerência induzidos pelos *softwares DSMs* baseados em página, através do auxílio de um controlador de protocolos. Nós mostramos que

o diff dinâmico reduz os custos de coerência de forma significativa e pode melhorar o desempenho em até 12% em um *cluster* de 4 processadores, principalmente pela eliminação do custo de criação de cópias de páginas. Estimativas do desempenho dessa técnica no nosso novo *cluster* de 8 nós sugerem uma redução do tempo total de execução de até 24%. Em contraste com os nossos resultados experimentais, os maiores ganhos estimados se devem a diminuição do efeito de poluição de cache.

6.2 Trabalhos Futuros

Um estudo da combinação das estratégias de *software* e *hardware* não foi apresentado porque o *hardware* de diff dinâmico e o sistema em que os experimentos com esta técnica foram executados não estão mais disponíveis. No entanto, os estudos simulados da utilização do controlador de protocolos combinado com técnicas de tolerância a latência que apresentamos em [7], demonstram que o controlador de protocolos consegue reduzir a interferência que pré-buscas podem gerar em nós remotos. Além disso, as técnicas de adaptação e de diff dinâmico podem ser consideradas complementares, tendo em vista que elas reduzem os custos relacionados a operações de diffs para páginas único-escritor, enquanto que diff dinâmico pode ser usado para atacar os mesmos custos para as páginas múltiplos-escritores. *Adaptive++* também pode se beneficiar da estratégia de diff dinâmico, já que o menor custo de diffs reduziria a quantidade de tempo útil de computação necessária para a pré-busca. Sendo assim, acreditamos que a combinação de todas as estratégias é possível, e que ainda geraria ganhos maiores.

Nós também não realizamos um estudo de todas as possíveis formas de combinação das diferentes estratégias de *software*. A nossa escolha se baseou em um estudo qualitativo das técnicas de forma que, cada falha de acesso é coberta por apenas um tipo de estratégia evitando assim, a ocorrência de interferências negativas entre as estratégias. No entanto, não exaurimos todas as possibilidades.

A alta latência de acesso a dados remotos se deve principalmente ao comportamento da aplicação. Nossos estudos se basearam em aplicações que apresentam comportamentos regulares dos seguintes tipos: repetição de fase, repetição de *stride*, e repetição de um padrão de compartilhamento. Entretanto, não desenvolvemos nenhuma estratégia que resolva a questão de contenção por acesso a um determinado conjunto de dados compartilhados, que representa um outro tipo de comportamento

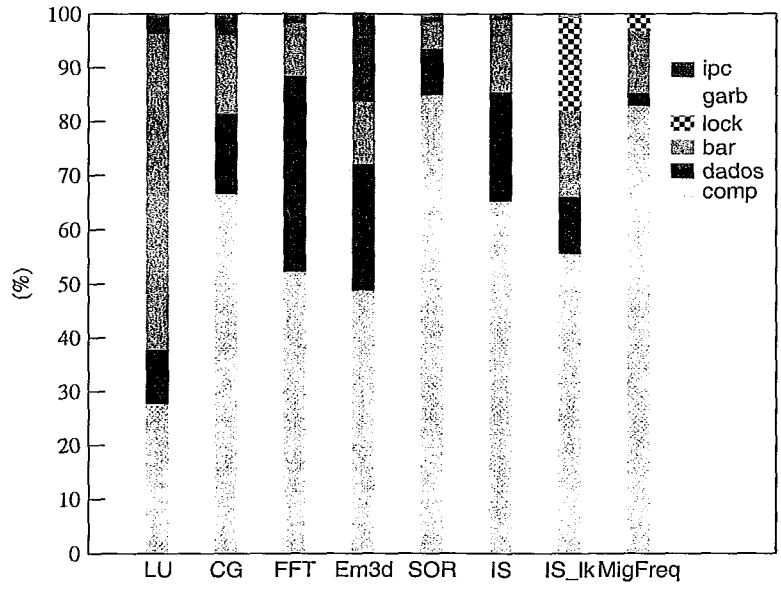

Figura 6.1: Tempo de acesso a dados.

de aplicações.

Considerando ainda o comportamento das aplicações, nós também não desenvolvemos nenhuma estratégia para reduzir a latência de acesso a dados remotos das aplicações que possuem um comportamento irregular. Uma idéia seria utilizar técnicas de *multithreading* para otimizar este tipo de aplicação.

6.3 Considerações Finais

As estratégias de *software*, especialmente quando combinadas, alcançam ganhos significativos de desempenho. A nossa técnica de *hardware* também gera altos ganhos de desempenho, mas quando comparada às estratégias de *software* verificamos que a técnica de *hardware* oferece uma razão custo/benefício muito pior. Observamos que as estratégias de *software* apresentam ganhos maiores do que a técnica de *hardware*, além de serem muito mais flexíveis pois não dependem de nenhuma característica da plataforma utilizada. Além disso, a técnica de *hardware* ainda apresenta o custo de implementação do projeto como fator negativo.

Finalizando, observamos que a aplicação das nossas estratégias para otimização de *software DSMs*, reduz significativamente o tempo de acesso a dados remotos de tal forma que para algumas aplicações o principal custo deixou de ser a latência de acesso a dados. Isto pode ser verificado na figura 6.1 onde apresentamos o tempo de execução das aplicações utilizando a combinação das estratégias de *software*,

sendo que para Em3d apresentamos o tempo de execução produzido pela utilização de *Adaptive++*. Os tempos de execução estão normalizados e divididos da mesma forma que na figura 2.4. Observando a figura verificamos que o peso do custo de sincronização (custos de barreira e *locks*) aumentou significativamente, demonstrando a necessidade de uma maior investigação em estratégias para reduzir este custo em *software DSMs*.

Bibliografia

[1] S. Adve and M. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–614, Junho 1993.

[2] 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. Em *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp 2–13, Junho 1995.

[3] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt Between Single Writer and Multiple Writer. Em *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pp 261–271, Fevereiro 1997.

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

[5] C. Amza, A. L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W.Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. *Proceedings of IEEE, Special Issue on Distributed Shared Memory Systems*, 87(3):397–532, Março 1999.

[6] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, Fevereiro 1996.

[7] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software

DSMs. Em *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pp 198–209, Outubro 1996.

[8] R. Bianchini and B.-H. Lim. Evaluating the Performance of Multithreading and Prefetching in Multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):83–97, Agosto 1996.

[9] R. Bianchini, R. Pinto, and C. L. Amorim. Data Prefetching for Software DSMs. Em *Proceedings of the International Conference on Supercomputing'98*, pp 385–392, Julho 1998.

[10] R. Bianchini, R. Pinto, and C. L. Amorim. Page Fault Behavior and Prefetching in Software DSMs. Relatório Técnico ES-401/96, COPPE Engenharia de Sistemas, Universidade Federal do Rio de Janeiro, Julho 1996, Revisado Fev 1998.

[11] A. Bilas, C. Liao, and J. P. Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. Em *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp 282–293, Maio 1999.

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

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

[14] M. Castro and C. de Amorim. Efficient Categorization of Memory Sharing Patterns in Software DSM Systems. Em *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, Abril 2001.

[15] T. Chen and J. L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. Em *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp 223–232, Abril 1994.

[16] T. Chen and J. L. Baer. Effective Hardware-Based Data Prefetching for High Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, Maio 1995.

[17] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. Em *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pp 186–197, Outubro 1996.

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

[19] F. Dahlgren and P. Stenström. Reducing the Write Traffic for a Hybrid Cache Protocol. Em *Proceedings of the 1994 International Conference on Parallel Processing*, pp 166–173, Agosto 1994.

[20] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. Em *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pp 14–25, Fevereiro 1996.

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

[22] P. Keleher. Sparks: Coherence as an Abstract Type. Em *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, Outubro 1996.

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

[24] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. Em *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp 13–21, Maio 1992.

[25] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Em *Proceedings of the Winter 1994 USENIX Conference*, pp 115–131, Janeiro 1994.

[26] L. Kontothanassis and *et al.* VM-Based Shared Memory on Low-Latency Remote-Memory-Access Networks. Em *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp 157–169, Maio 1997.

[27] 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. L. Hennessy. The Stanford FLASH Multiprocessor. Em *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp 302–313, Abril 1994.

[28] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. Em *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp 241–251, Junho 1997.

[29] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Janeiro 1993.

[30] L. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. Em *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pp 289–299, Fevereiro 1998.

[31] L.R.R. Monnerat. Adaptação Eficiente a Padrões de Compartilhamento em Software DSMs. Tese de Mestrado, COPPE/UFRJ, Dezembro 1997.

[32] T. Mowry, C. Chan, and A. Lo. Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory. Em *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pp 300–309, Fevereiro 1998.

[33] 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, Junho 1991.

[34] R. Pinto, L. Whately, M. deMaria, R. Santos, R. Bianchini, and C. L. Amorim. Evaluating the Dynamic Diffing Technique for Distributed Shared-Memory Systems. Em *Proceedings of the 12th Symposium on Computer Architecture and High Performance Computing*, pp 209–216, Outubro 2000.

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

[36] S. K. Reinhardt, R. A. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. Em *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp 34–43, Maio 1996.

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

[38] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. Em *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pp 297–307, Outubro 1994.

[39] C. Seidel, R. Bianchini, and C. L. Amorim. Evaluating the Impact of the Programming Model on the Performance and Complexity of Software DSM Systems. Em *Proceedings of the 1999 International Conference on Parallel Processing*, Setembro 1999.

[40] C. B. Seidel, R. Bianchini, and C. L. Amorim. The Affinity Entry Consistency Protocol. Em *Proceedings of the 1997 International Conference on Parallel Processing*, pp 208–217, Agosto 1997.

[41] W. E. Speight and J. K. Bennett. Using Multicast and Multithreading to Reduce Communication in Software DSM Systems. Em *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, pp 312–322, Fevereiro 1998.

[42] R. Stets, S. Dwarkadas, N. Hardavellas, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write

Network. Em *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp 170–183, Outubro 1997.

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

[44] S. VanderWiel and D. Lilja. When Caches Aren't Enough: Data Prefetching Techniques. *Computer*, 30(7):13–30, Julho 1997.

[45] J. E. Veenstra and R. J. Fowler. The Prospects for On-Line Hybrid Coherency Protocols on Bus-Based Multiprocessors. Relatório Técnico TR-490, Dept. of Computer Science, The University of Rochester, 1994.

[46] L. Whately, R. Pinto, M. Rangarajan, L. Iftode, R. Bianchini, and C. L. Amorim. Adaptive Techniques for Home-Based Software DSMs. Em *Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing*, pp 164–171, Setembro 2001.

[47] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH2 Programs: Characterization and Methodological Considerations. Em *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp 24–36, Maio 1995.

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