Técnicas para Detecção e Exploração de Padrões de Compartilhamento em Sistemas de Memória Compartilhada Distribuída

Maria Clicia Stelling de Castro

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

Aprovada por:

Prof. Cláudio Luis de Amorim, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Ricardo Bianchini, Ph.D.

Prof. Orlando Gomes Loques Filho, Ph.D.

Prof. Júlio Salek Aude, Ph.D.

Prof. Wagner Meira Júnior, Ph.D.

Rio de Janeiro, RJ - Brasil

DEZEMBRO DE 1998

CASTRO, MARIA CLICIA STELLING DE

Técnicas para Detecção e Exploração de Padrões de Compartilhamento em Sistemas de Memória Compartilhada Distribuída [Rio de Janeiro] 1998

XVI, 123 p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computacão, 1998)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Sistemas Operacionais

2 - Memória Compartilhada Distribuída

I. COPPE/UFRJ II. Título (série)

Aos meus pais Maria José e Alcides

Agradecimentos

Agradeço:

à Cláudio Luis de Amorim, meu orientador, pela confiança, estímulo e apoio incondicional nas horas em que mais precisei;

à Universidade Federal de Juiz de Fora a liberação concedida para que este trabalho pudesse ser desenvolvido. Em particular, agradeço aos amigos que fiz nessa instituição, Carlos Alberto Araújo Zacaron e Aparecida Pinto e Neto, pelo apoio e exemplo de profissionalismo. Agradeço também, a todos os professores do Departamento de Ciência da Computação pelo apoio, compreensão e liberação para que eu pudesse terminar com tranqüilidade meu doutorado;

aos órgãos financiadores Finep, CAPES e CNPq, que deram suporte a realização desse trabalho;

aos professores da minha linha de pesquisa Edil, Valmir, Felipe, Vitor, Eliseu, Ricardo e Inês pelo conhecimento que trocamos e pela convivência proveitosa de tantos anos;

à Leonidas Kontothanassis e Ricardo Bianchini que desenvolveram a versão original do simulador utilizado nesta tese, e a Raquel Pinto que o modificou;

à Cláudia, Solange, Sueli, Mercedes, Ari, Lúcia e Marli, que fazem parte do corpo administrativo do Programa de Engenharia de Sistemas e Computação, que com dedicação e competência ajudam a construí-lo a cada dia e a mantê-lo. Agradeço também aos técnicos, Júlio(s), Adilson, César, Carlos e Frederico, que tiveram paciência para tantas vezes ouvir "a rede caiu de novo", "o disco de SO está cheio", "tem processo perdido na miami", "a porta está batendo novamente";

à minha família pelo apoio e compreensão pelas longas horas de ausência de um convívio tão prazeroso. Em especial, aos meus pais Maria José e Alcides pelo carinho e incentivo;

à Cláudia, Mara e Ana Paula Prata, pela amizade, carinho e inúmeros almoços, cheios de alegria e descontração que tanto me ajudaram a aliviar as tensões do dia a dia;

às minhas amigas Nancy, Inês, Denise, Lúcia, Anna, Laura, Márcia Aparecida, Márcia Cerioli, pelo apoio e incentivo, mesmo que para algumas à distância. A todas meu carinho especial;

aos meus companheiros de sala Cristiana, Carla, Paula, Luiz Favre e Ayru, pela amizade, pelos desabafos, pelas longas conversas para tentar esquecer momentanea-

mente nossos intermináveis *bugs*, e pelo incentivo nas horas onde tudo parecia que não ia ter fim;

à Raquel em especial por me ajudar a encontrar os pontos de *bug* e retirar minhas dúvidas em momentos tão importantes;

à Ricardo e Wagner pela amizade e pelos comentários pertinentes e construtivos;

aos participantes habituais e eventuais dos lanches ao final da tarde, Vanusa, Lauro, Rodrigo, Luis Adaulto e cia, sempre descontraídos e que ajudaram a levantar o ânimo para enfrentar mais horas de trabalho;

à Wagner Arbex por ter me substituído tão bem junto aos meus alunos e

à tantos mais que compartilharam comigo desse caminho.

Às vezes é preciso não só sonhar, mas também é preciso trabalhar, se dedicar e acreditar para alcançar um objetivo.

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

Técnicas para Detecção e Exploração de Padrões de Compartilhamento em Sistemas de Memória Compartilhada Distribuída

Maria Clicia Stelling de Castro

Dezembro de 1998

Orientador: Claudio Luís de Amorim
Programa: Engenharia de Sistemas e Computação

Nesta tese propomos uma nova máquina de estados finitos, denominada FIESTA, que em tempo de execução gera informações sobre os estados de compartilhamento de cada página das aplicações executadas em sistemas de memória compartilhada distribuída implementados em *software*.

Propomos também um novo algoritmo de categorização baseado em FIESTA, denominado RITMO, que identifica eficazmente os padrões de escrita tanto em aplicações regulares quanto em aplicações irregulares ao longo de sua execução.

FIESTA e RITMO são transparentes para o usuário e não requerem suporte de compilação, modificação do modelo de programação ou suporte de *hardware*.

O potencial de FIESTA e de RITMO para reduzir *overheads* de protocolos de coerência em sistemas *software DSM* é demonstrado pela adição de ambos a um simulador do software DSM TreadMarks e aplicando seletivamente técnicas de tolerância à latência, tais como adaptação *single writer/multiple writer*, *forwarding* e *prefetching*, de acordo com os padrões de compartilhamento de memória de cada aplicação.

Nossos resultados mostram que FIESTA e RITMO têm *overhead* insignificantes para ambas as classes de aplicações regulares e irregulares. Além disso, nossos resultados demonstram que, baseados nas informações de FIESTA e RITMO, os

mecanismos de previsão, implementados nas técnicas *forwarding* e *prefetching*, têm alta precisão de acertos.

As técnicas de tolerância à latência baseadas em FIESTA e RITMO reduziram substancialmente os *overheads* devidos a acessos remotos a dados e indiretamente a outros tipos de *overheads*, melhorando significativamente o desempenho da maioria das aplicações testadas.

Esses resultados demonstram que FIESTA e RITMO, conjugados com técnicas de tolerância à latência, oferecem uma ótima opção que deve ser considerada no desenvolvimento de sistemas *software DSM* eficientes para uma ampla classe de aplicações.

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

Techniques for Detection and Exploration of Sharing Patterns in Distributed Shared-Memory Systems

Maria Clicia Stelling de Castro

December, 1998

Thesis Supervisor: Claudio Luís de Amorim

Department: Programa de Engenharia de Sistemas e Computação

In this thesis we propose a new finite-state machine called FIESTA that generates at run-time sharing-state information for each page of applications running on distributed shared-memory systems implemented in software.

Also, we propose a new FIESTA-based cathegorization algorithm called RITMO that identifies efficiently writing patterns for both regular and irregular applications over the course of their execution.

FIESTA and RITMO are transparent to the user and do not require compiler support, alteration to the programming model, or hardware support.

The potential of FIESTA and RITMO to reduce protocol overheads in software DSM systems is demonstrated through adding both to the TreadMarks software DSM system as well as by selectively applying latency-tolerant techniques such as adaptation to single-writer/multiple-writer, forwarding, and prefetching according to the memory sharing patterns of each application.

Our results show that FIESTA and RITMO overheads are negligible for both regular and irregular applications. Furthermore, our results demonstrate that, based on the information provided by FIESTA and RITMO, the inference mechanisms implemented on forwarding and prefetching techniques are highly accurate.

The latency-tolerant techniques based on FIESTA and RITMO reduced substantially the overheads due to remote data accesses and indirectly to other overhead types, improving the performance of most benchmarks applications significantly.

These results demonstrate that FIESTA and RITMO coupled with latency-tolerant techniques offer a very good option that should be considered while developing efficient software DSM systems for a broad class of applications.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nos últimos anos, pudemos observar a crescente expansão de aplicações científicas e comerciais que se beneficiam do grande poder computacional de sistemas de computação paralela. Basicamente, esses sistemas se distinguem pelo modelo arquitetural que implementam: memória distribuída ou memória compartilhada.

Os sistemas de memória distribuída ou multicomputadores possuem um *hardware* facilmente escalável. Com o aumento do número de processadores é possível obter maior desempenho em diversas classes de aplicações. Os sistemas de memória centralizada não apresentam esta característica. A medida que o número de processadores aumenta pode haver contenção no barramento limitando o seu desempenho.

O modelo de programação de passagem de mensagens é mais difícil de ser assimilado que o modelo de memória compartilhada, pois o programador ou o compilador ou ambos, com suporte do sistema operacional, devem gerenciar a distribuição dos dados compartilhados e a troca explícita de mensagens. Já os sistemas que implementam o modelo de programação de memória compartilhada possuem um único espaço de endereçamento global. Seu modelo de programação é mais próximo do modelo de um sistema uniprocessador, o que os torna, portanto, mais fáceis de serem programados.

Uma das tendências atuais enfoca o desenvolvimento de sistemas que têm como características principais o modelo de programação de memória compartilhada, mais amigável, e a escalabilidade proporcionada pelos sistemas distribuídos. Estes sistemas são denominados sistemas de memória compartilhada distribuída (DSM - *Distributed Shared Memory*).

Os sistemas DSM assumem normalmente a existência de um processo por processador e, por este motivo, no restante dessa tese (a menos que explicitamente men-

cionado) utilizamos os termos processo e processador indistintamente. A figura 1.1 ilustra a organização de um sistema DSM.

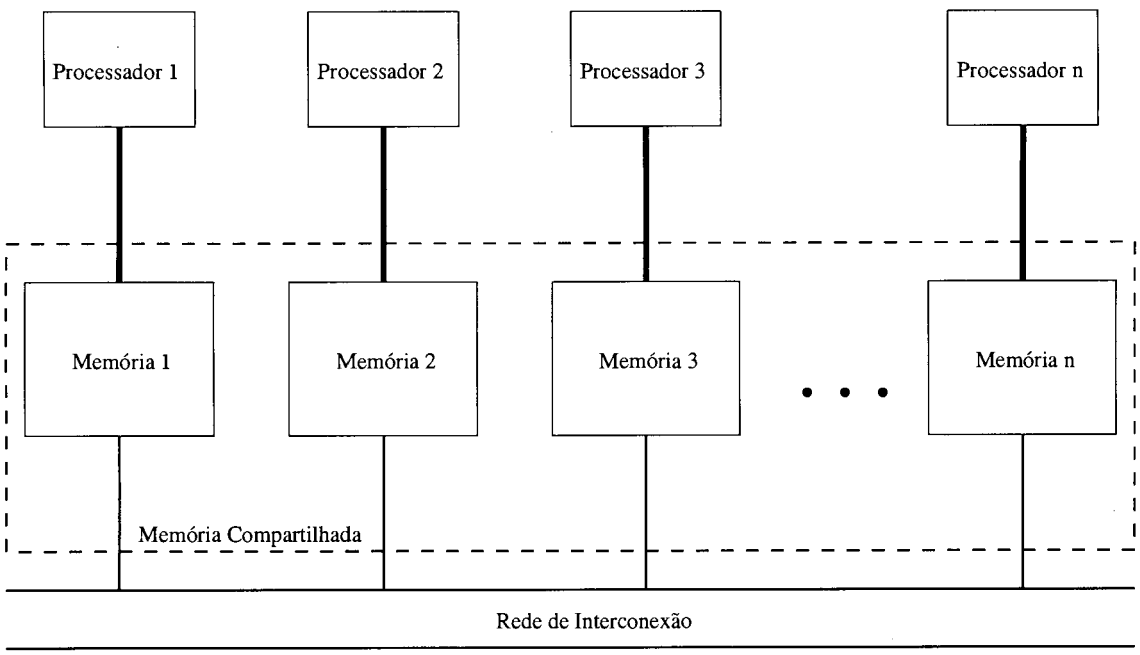

Figura 1.1: Organização de um sistema DSM

Esses sistemas são compostos por múltiplos nós de processamento conectados por uma rede de interconexão. Cada nó pode conter um ou mais processadores, memória privativa, memórias *cache* e dispositivos de entrada e saída.

Cada processo pode realizar operações no espaço de endereços compartilhado. Nos sistemas DSM, a "memória compartilhada" é implementada por mecanismos de *hardware* e/ou *software* que transformam, de modo trasparente ao usuário, os acessos às memórias remotas em mensagens pela rede.

O compartilhamento dos dados nesse modelo é implementado com instruções de acesso às posições de memória (*loads* e *stores*). A cooperação e coordenação entre os processos é realizada através da leitura e da escrita de variáveis compartilhadas e de ponteiros que se referem aos endereços compartilhados. As escritas em dados compartilhados são realizadas de modo exclusivo e ordenadas com operações de sincronização. As barreiras e os *locks* são os dois tipos de implementação de operações de sincronização mais comuns.

As operações de barreira são utilizadas para separar as fases de execução de uma aplicação. Um processo só inicia uma nova fase depois que todos os processos tenham atingido a barreira. Dessa forma, a operação de sincronização de barreira garante que todos os acessos feitos na nova fase estão ordenados em relação aos

acessos realizados na fase anterior.

As operações de *lock* são utilizadas para ordenar os acessos dos processos envolvidos em seções críticas. As seções críticas são trechos de código que só podem ser executados com exclusão mútua, isto é, somente um processo por vez pode executar tal trecho de código.

Os sistemas DSM podem ser implementados utilizando diferentes abordagens: puramente em *hardware*, puramente em *software* ou combinando-se *hardware* e *software*.

Os sistemas *hardware DSM* são escaláveis e alcançam ótimo desempenho em várias classes de aplicações. Porém, um projeto de sistema *hardware DSM* é complexo e exige um longo tempo de desenvolvimento. Este fato pode levá-lo à desatualização tecnológica rápida após ser construído, além de apresentar um alto custo. Os sistemas *software DSM* são uma alternativa para o problema de custo dos sistemas *hardware DSM*. Esses sistemas têm tempo e custo de desenvolvimento menores. O grande atrativo de sistemas *software DSM* é que eles podem ser implementados em multicomputadores ou em redes de *workstations*, que são amplamente difundidas. Entretanto, os sistemas *software DSM* conseguem bom desempenho numa classe limitada de aplicações. Para melhorar o seu desempenho, podem ser empregados, por exemplo, suportes de *hardware* simples e de baixo custo para esconder a latência da rede de interconexão (sistemas híbridos) ou soluções em *software* que são o alvo de nosso estudo.

Em sistemas *software DSM*, os acessos a dados remotos são muito custosos devido à latência da rede de comunicação. Para tentar minimizar este custo, esses sistemas replicam os dados nas diversas memórias privativas dos processadores. Se for permitido aos processadores escreverem livremente em suas cópias, uma visão inconsistente da memória poderá ser formada, levando o programa a resultados imprevisíveis. Por esse motivo, existe a necessidade de se implementar um protocolo para manter a coerência no acesso aos dados compartilhados.

Tipicamente, sistemas *software DSM* utilizam a página como unidade de coerência. O grande tamanho da página pode causar dois problemas: falso compartilhamento e fragmentação. O falso compartilhamento ocorre quando dois ou mais processadores realizam acesso a dados não relacionados que estão armazenados numa mesma página, e pelo menos um deles realiza uma escrita. O falso compartilhamento pode causar o efeito *ping-pong*, que faz com que a página seja transferida

de um processador para outro, aumentando o *overhead* do protocolo de coerência. A fragmentação ocorre quando um processador necessita somente de parte da página mas tem que carregá-la inteira, causando tráfego de dados desnecessário.

Dependendo dos padrões de compartilhamento das aplicações, o grau de fragmentação e falso compartilhamento irão determinar o desempenho de sistemas *software DSM*. Vários estudos foram realizados para minimizar o impacto desses problemas, incluindo reestruturação das aplicações, avaliação de novos modelos de consistência, diferentes técnicas de tolerância à latência, suporte a múltiplos escritores e protocolos adaptativos.

A modificação ou reestruturação das aplicações permitem melhor localidade e maior compatibilidade da aplicação com o tamanho da unidade de coerência (página), reduzindo a freqüência das ações do protocolo[45]. Para minimizar o falso compartilhamento e a fragmentação, essas estratégias apresentam maior eficácia, porém, freqüentemente são necessárias mudanças complexas tanto no algoritmo quanto nas estruturas de dados, exigindo conhecimento especializado das aplicações. Além disso, o alto desempenho não é garantido se a aplicação for executada em outros sistemas *software DSM*.

O emprego de modelos de consistência de memória mais relaxados atrasa a propagação de dados ou ações de coerência até os pontos de sincronização do programa[49, 13, 43], o que alivia os efeitos do falso compartilhamento. A literatura[35, 49, 69, 86] revela que, para uma ampla classe de aplicações, os sistemas baseados em modelos relaxados de consistência possuem melhor desempenho do que os sistemas baseados em modelos fortes de consistência.

O suporte a múltiplos escritores permite escritas concorrentes à mesma unidade de coerência (desde que em diferentes posições de memória) para minimizar o problema de falso compartilhamento. Escritas feitas à mesma posição de memória devem ser ordenadas através de operações de sincronização. O principal problema de se permitir a existência de múltiplos escritores na mesma unidade de coerência são os *overheads* de gerenciar todas as escritas concorrentes realizadas para montar uma versão coerente da unidade de coerência, quando necessário.

As técnicas de tolerância à latência, tais como *prefetching* [17] e *forwarding* [74, 81], visam reduzir o tempo de espera de um processador por dados remotos, buscando ou enviando antecipadamente os dados compartilhados. Estas técnicas implementam mecanismos de previsão de falhas e são baseadas em anotações inseri-

das pelo programador/compilador ou em ações do *run-time system* ou combinação destas.

Protocolos que se adaptam dinamicamente às características das aplicações apresentam certa diversidade. Alguns implementam diferentes modelos de consistência, outros diferentes opções de adaptação tais como atualização ou invalidação das páginas nos pontos de sincronização e adaptação entre um e vários escritores[21, 32, 7, 64].

Os protocolos *software DSM* que admitem algum tipo de adaptação têm se mostrado capazes de explorar melhor o padrão de compartilhamento das aplicações com bons resultados.

Os protocolos adaptativos *software DSM* atuais do tipo único escritor/múltiplos escritores (*Single Writer/Multiple Writers* ou SW-MW) permitem que no modo MW, processadores distintos escrevam concorrentemente em regiões diferentes de uma página, para reduzir o efeito do falso compartilhamento. Nesse modo entretanto, o *overhead* adicional introduzido corresponde ao gerenciamento das modificações feitas pelos processadores em cada página compartilhada e o seu armazenamento para posterior envio aos processadores nas operações de sincronização. Já no modo SW, a página inteira pode ser transferida de seu escritor para o(s) seu(s) leitor(es) sem a necessidade de armazenamento das modificações.

As técnicas de tolerância à latência e os protocolos adaptativos são soluções promissoras para se reduzir os *overheads* de protocolos *software DSM*, porém ainda não é claro quais técnicas ou adaptações podem atingir melhores desempenhos, além dos benefícios serem limitados à uma classe restrita de aplicações. Em geral, as técnicas de tolerância à latência e os protocolos adaptativos se baseiam em métodos que categorizam o padrão de compartilhamento das páginas, de forma aproximada, para realizar as previsões ou ativar as ações de adaptação. Aumentando-se a precisão desses métodos poderemos ampliar as oportunidades de se efetuar mais ações de adaptação e incrementar a eficiência das técnicas, e portanto, reduzir os *overheads* de protocolos *software DSM*. Assim, seria importante que a detecção dos padrões de compartilhamento das aplicações, não só acontecesse de forma não intrusiva e transparente ao usuário, mas também refletisse com maior precisão possível o padrão dinâmico de compartilhamento de cada página ao longo da execução das aplicações.

A precisão da categorização do padrão de compartilhamento é fundamental para que o protocolo execute ações adaptativas que reduzam ao máximo a quantidade

de mensagens trocadas para manter a coerência dos dados compartilhados e, além disso, que as técnicas de tolerância à latência minimizem o tempo de espera pelos dados. A intrusão, se excessiva, pode neutralizar os ganhos decorrentes das ações adaptativas. Captar informações de forma transparente ao usuário evita o aumento da complexidade de programação.

Esta tese mostra que um método de discriminação mais precisa da dinâmica de compartilhamento de uma página em sistemas *software DSM*, permite a categorização dos padrões de compartilhamento tanto em aplicações regulares como em aplicações irregulares. Mais especificamente, mostramos que o método proposto é capaz de detectar padrões em páginas com acessos de escrita irregulares: (1) quando ocorrem dentro e fora de seções críticas, e (2) quando ocorrem em seções críticas guardadas por *locks* distintos.

No nosso método de identificação de padrões é fundamental a distinção entre estado de compartilhamento e padrão de escrita da página. Em aplicações regulares, comumente a repetição de estados de compartilhamento facilmente identifica o padrão de escrita da página. Nas aplicações irregulares, a dinâmica dos acessos às páginas compartilhadas é bem menos óbvia. Tipicamente, nessas aplicações os processadores acessam páginas dentro e fora seção de crítica ou com mais de um *lock*. Essas páginas são consideradas simplificadamente como tendo múltiplos escritores pelas atuais estratégias de categorização do padrão de escrita, limitando as opções de técnicas de tolerância à latência que podem ser empregadas. Mostramos nesta tese que observando a seqüência de estados de compartilhamento dessas páginas, podemos categorizá-las com padrões único escritor, múltiplos escritores ou migratórias, onde as escritas são realizadas por vários processadores, porém de modo exclusivo, isto é, as escritas são realizadas por um processador de cada vez, retirando essa limitação.

O estado de compartilhamento de uma página é definido pelo tipo de acesso realizado à página (leitura ou escrita, dentro ou fora de seção crítica) e por quais processadores realizam os acessos (um ou vários). O padrão de compatilhamento considera a quantidade de processadores que realizam acessos de escrita à página e se tais acessos são realizados dentro ou fora de seção crítica. As classificações dos padrões de compartilhamento são: um único escritor, produtor-consumidor, múltiplos escritores ou ainda migratória.

Consideramos como aplicações regulares, aquelas onde todas as páginas compar-

tilhadas do conjunto da aplicação ou têm acessos somente fora de seção crítica ou somente dentro de uma única seção crítica. Já as aplicações irregulares são aquelas que possuem pelo menos uma página ou com acessos dentro e fora de seção crítica ou dentro de mais de uma seção crítica (página com vários *locks*).

Esta tese é demonstrada através da introdução de uma nova máquina de estados finitos para *software DSM* denominada FIESTA (*FInitE STAte machine*) que efetua transições no estado de compartilhamento das páginas sempre que ocorram operações de coerência a elas associadas.

FIESTA permite determinar com precisão as características dos estados de compartilhamento das páginas ao longo da execução da aplicação para que sejam utilizados por mecanismos de adaptação e técnicas de tolerância à latência pelos protocolos *software DSM*. As transições efetuadas por FIESTA geram a seqüência de estados de compartilhamento que permite identificar com precisão o padrão de compartilhamento de cada página, onde o estado é representado pelos conjuntos de processadores bem como pelos tipos de acesso que realizam nas páginas.

A categorização dos padrões de compartilhamento, para uma página ou coleção de páginas, permite a utilização de técnicas e adaptações específicas para cada aplicação particular. Com as informações fornecidas por FIESTA, diferentes técnicas podem ser inseridas nos protocolos e utilizadas em grupos de páginas distintos, tornando o protocolo adaptável ao comportamento dinâmico das aplicações.

Baseado nas seqüências dos estados de compartilhamento de cada página gerados por FIESTA, um novo algoritmo (RITMO) é desenvolvido para inferir o padrão de compartilhamento dinâmico das aplicações e explorar protocolos adaptativos, mais especificamente categoriza eficazmente páginas com acessos dentro e fora de seção crítica e páginas que são protegidas por mais de uma variável de *lock*. Além disso, propomos e avaliamos estratégias de adaptação e tolerância à latência (único escritor/múltiplos escritores, *forwading* e *prefetching*) , de forma individual e combinada, para minimizar *overheads* em protocolos de sistemas *software DSM*. Essas estratégias são implementadas considerando as informações de FIESTA e RITMO.

A caracterização realizada por FIESTA representa o estado de compartilhamento de cada página ao longo da execução das aplicações. A figura 1.2 ilustra como, a partir da seqüência dos estados de compartilhamento de cada página, podemos categorizar cada uma delas, e fazer com que o protocolo execute ações adaptativas que reduzam os *overheads* relacionados com a manutenção da coerência dos dados

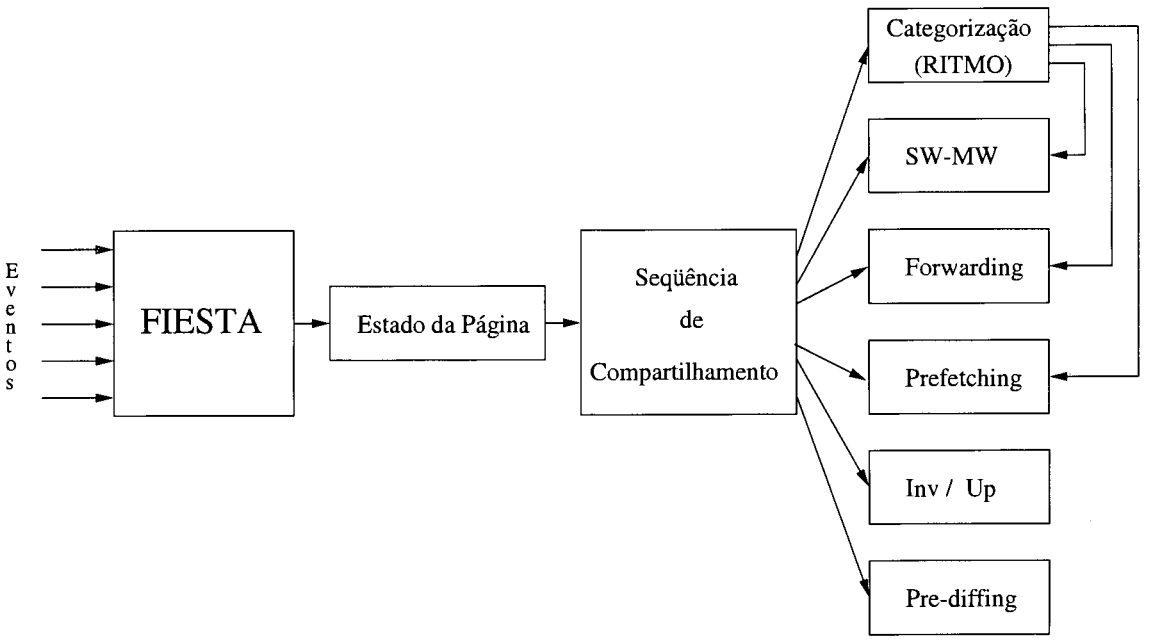

Figura 1.2: Aplicação das informações dos estados de compartilhamento do modelo de FIESTA

compartilhados. Isto é, permite a utilização de mecanismos de adaptação e técnicas de tolerância à latência, como por exemplo, adaptação entre um único escritor e múltiplos escritores (SW-MW), busca (*prefetching*) ou envio antecipado (*forwarding*) dos dados compartilhados, criação antecipada de *diffs*, adaptação entre protocolos de invalidação e atualização (Inv/Up), entre outros. O algoritmo de categorização que utilizamos é o RITMO, porém, outros algoritmos de categorização podem ser desenvolvidos a partir das informações fornecidas por FIESTA. Além disso, utilizamos em nossos experimentos as técnicas SW-MW, *forwarding* e *prefetching*.

Nossos resultados experimentais são baseados na implementação de FIESTA, no algoritmo de categorização do padrão de compartilhamento (RITMO) e de estratégias de adaptação e de tolerância à latência, utilizando um simulador de TreadMarks[50], o *software DSM* mais disseminado nos dias de hoje. Simulamos um multicomputador com oito nós de processamento. A análise do desempenho é realizada comparado-se a versão original de TreadMarks com cada uma das versões modificadas de TreadMarks. Nossos resultados mostram que FIESTA não aumenta o número de mensagens transferidas no protocolo e o aumento na quantidade de *bytes* transmitidos para a maioria das aplicações é insignificante. FIESTA e RITMO permitem que diferentes ações de adaptação possam ser realizadas de forma altamente precisa. Quanto à implementação das técnicas de tolerância à latência e adaptação,

as altas taxas de acertos, no envio e na busca de dados compartilhados, permitem redução significativa nos *overheads* de acesso a dados remotos do protocolo. Essas reduções são significativas para todas as aplicações e variam entre 25% a 89%.

A principal conclusão é que, FIESTA e RITMO têm grande potencial de melhorar o desempenho de sistemas *software DSM* para uma classe de aplicações mais ampla.

Sumariamente, as principais contribuições desta tese são:

- proposta de FIESTA, uma nova máquina de estados finitos para representar estados de compartilhamento de páginas em sistemas *software DSM*;
- proposta de RITMO, um novo algoritmo de categorização de padrão de escrita das páginas nas aplicações em sistemas *software DSM*;
- avaliação do impacto de FIESTA e RITMO na implementação das técnicas de tolerância à latência SW-MW, *forwarding* e *prefetching* de forma isolada e combinada.

Esta tese está organizada da seguinte forma. No capítulo 2 abordamos os conceitos básicos de sistemas de memória compartilhada distribuída, descrevemos as técnicas mais utilizadas atualmente para esconder ou tolerar *overheads* em sistemas *software DSM* e descrevemos também o sistema TreadMarks, no qual baseamos nossos experimentos. No capítulo 3, introduzimos FIESTA. Baseados nas informações de FIESTA, descrevemos como categorizamos o padrão de compartilhamento de cada página através de RITMO, no capítulo 4. Os mecanismos de adaptação e técnicas de tolerância à latência estão explicados no capítulo 5. Nossa metodologia está apresentada no capítulo 6. O comportamento das aplicações segundo FIESTA e RITMO e a análise dos resultados são abordados nos capítulos 7 e 8, respectivamente. No capítulo 9 relacionamos nossos resultados com os principais trabalhos na área. Apresentamos nossas conclusões no capítulo 10.

Capítulo 2

Protocolos *Software DSM*: Conceitos Básicos

A implementação de *software DSMs* eficientes não é uma tarefa simples, e envolve várias decisões que influenciam diretamente no desempenho do sistema, como por exemplo, a escolha entre diferentes modelos de consistência ou mecanismos de coerência. Iniciamos este capítulo revendo alguns conceitos básicos que são utilizados ao longo desta tese. Discutimos os modelos de consistência, a estrutura dos dados e o tamanho da unidade de coerência, os protocolos de coerência e o suporte a múltiplos escritores. Em seguida, descrevemos técnicas utilizadas para tolerar ou minimizar os *overheads* de sistemas *software DSM*. Finalizamos apresentando o protocolo Treadmarks, utilizado como referência de sistema *software DSM* em nossos experimentos.

2.1 Modelos de Consistência

Os modelos de consistência de memória definem a ordem em que os acessos à memória compartilhada devem ocorrer e a ordem em que estes acessos são observados pelos processadores. As restrições à reordenação dos acessos, inerente a cada modelo de consistência, influeciam diretamente o desempenho das aplicações submetidas aos sistemas. Os modelos de consistência podem ser caracterizados como fortes ou relaxados segundo suas restrições às ordenações dos acessos.

Nas seções seguintes apresentamos a descrição de alguns modelos de consistência de memória. O modelos de consistência *Sequencial Consistency* (SC) e *Processor Consistency* (PC), que tratam os acessos de sincronização como operações de leitura e escrita comuns, e, os modelos *Weak Consistency* (WC), *Release Consistency* (RC),

Lazy Release Consistency (LRC) e *Entry Consistency* (EC), que distingüem os acessos à memória em acessos comuns e acessos de sincronização. Nosso objetivo não é apresentar os formalismos da definição de cada modelo, mas sim prover uma noção intuitiva do funcionamento do modelo num sistema DSM.

2.1.1 Modelo de Consistência Seqüencial

O modelo de consistência seqüencial (SC - *Sequencial Consistency*) foi formalizado por Lamport[58] e define que um sistema multiprocessador é seqüencialmente consistente se e somente se o resultado de qualquer execução é o mesmo que seria obtido caso as operações de todos os processadores fossem executadas em alguma ordem seqüencial, e as operações de cada processador na ordem estabelecida pelo programa. Em termos de ordenação dos acessos a dados compartilhados, podemos dizer que uma máquina paralela seqüencialmente consistente funciona como se fosse um único processador multiprogramado.

A forma mais simples de se implementar o modelo SC é garantir que um acesso a um dado compartilhado só possa ser realizado se o acesso anterior tiver sido observado por todos os outros processadores, isto é, cada acesso realizado no dado compartilhado deve estar imediatamente e globalmente coerente.

A implementação do modelo de consistência seqüencial pode levar a um desempenho muito restrito, pois impede a implementação de algumas otimizações consideradas importantes nos processadores atuais. A utilização de *write-buffers*, *pipelines* de escrita, e *overlap* de operações de memória estão entre as otimizações comuns que podem violar o modelo[3].

2.1.2 Modelos de Consistência Relaxados

Vários modelos de consistência de memória relaxados foram propostos para viabilizar o uso de algumas otimizações no acesso à memória compartilhada, e assim, com um modelo menos restritivo, obter melhora de desempenho do sistema.

Modelo *Processor Consistency*

O modelo *Processor Consistency* (PC) foi introduzido por Goodman[37] e elimina algumas das restrições do modelo SC. Este modelo define que as escritas realizadas por cada processador devem ser sempre observadas segundo a ordem em que ele as executou. Entretanto, as escritas realizadas por processadores distintos podem ser

observadas em ordens diferentes. Este modelo garante que as escritas sejam observadas de modo consistente em cada processador e permite que algumas operações de leitura ultrapassem os acessos de escrita (desde que para posições de memória diferentes destas). Com isso, existe a oportunidade de utilização de *write buffers* e *pipelining*, o que permite a obtenção de um desempenho melhor que no modelo SC.

Se considerarmos que a grande maioria das aplicações paralelas têm definidas em alto nível suas necessidades de coerência é possível relaxar as condições de consistência. Os modelos relaxados apresentados a seguir, diferenciam os acessos aos dados em acessos comuns e acessos de sincronização. Esses fazem parte dos chamados modelos de consistência híbridos, nos quais podem ser utilizados modelos de consistência diferentes para os dois tipos de acesso. A coerência da memória compartilhada só é garantida nos pontos de sincronização. Assim, nestes modelos, devem existir pontos de sincronização explícitos onde a coerência é necessária.

A figura 2.1 exemplifica a utilização de sincronização para garantir a coerência. Ela mostra os segmentos de código de dois processos P_0 e P_1, e a sincronização de forma bem genérica através de acessos de leitura e escrita à variável de sincronização s. A sincronização neste exemplo é empregada para garantir que os acessos de leitura de A e B em P_1 necessariamente observem os valores escritos por P_0.

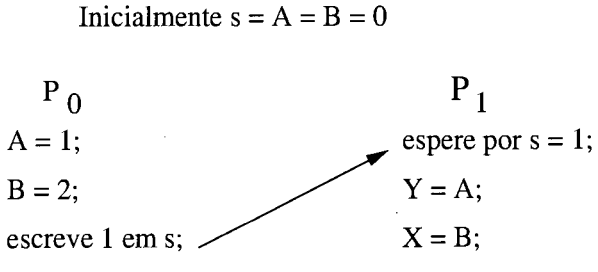

Figura 2.1: Segmentos de código dos processos que utilizam sincronização para garantir a coerência

A utilização de acessos às variáveis de sincronização evita condições de corrida e conseqüentemente resultados não determinísticos.

A execução de uma aplicação propriamente sincronizada num sistema com modelo de consistência relaxado equivale à sua execução num sistema com modelo de consistência seqüencial.

Uma aplicação é propriamente sincronizada se contém sincronização suficiente para evitar condições de corrida. Um programa propriamente sincronizado tem a seguinte definição[36]:

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

Modelo *Weak Consistency*

O primeiro modelo relaxado de consistência considerando a relação entre a ordem dos pedidos de acesso à memória e os pontos de sincronização do programa, foi denominado *Weak Consistency* (WC) e proposto por Dubois, Scheurich e Briggs[30]. Nele as operações de sincronização funcionam como cercas (*fences*) que obedecem ao modelo SC. Esse modelo estabelece que acessos a dados compartilhados realizados entre dois acessos de sincronização podem ser observados por outros processadores em qualquer ordem. Quando um acesso de sincronização é realizado, os acessos anteriores devem ter sido observados por todos os processadores.

Por permitir uma maior possibilidade de reordenação dos acessos à memória, os sistemas que implementam o modelo WC têm um potencial de ganho de desempenho melhor do que naqueles sistemas que implementam os modelos SC e PC.

Modelo *Release Consistency*

O modelo *Release Consistency* (RC) foi introduzido por Gharachorloo[36] e subdivide os acessos de sincronização em *acquires* e *releases*. Para garantir a coerência dos dados compartilhados a aplicação deve, portanto, utilizar sincronizações explícitas através de primitivas do sistema. As primitivas mais comuns fornecidas pelos sistemas DSM para sincronização são: *locks* (*acquires*) e *unlocks* (*releases*), para guardar seções críticas, e barreiras (um *release* seguido de um *acquire*), para sincronização global.

Um *acquire* e um *release* correspondem a uma operação de leitura e de escrita, respectivamente, a uma variável de sincronização. O *acquire* indica que o processador esta iniciando acessos a determinados dados compartilhados que pode depender de valores gerados por outro processador. Já o *release* indica que o processador está terminando os acessos aos dados compartilhados e pode ter gerado valores dos quais outro processador pode depender.

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

- antes que uma operação de *release* tenha sido observada por qualquer processador, todos os acessos anteriores aos dados compartilhados devem ter sido observados por este processador;
- os acessos que seguem uma operação de *acquire* numa variável de sincronização devem esperar que o *acquire* tenha terminado e
- os acessos às variáveis de sincronização devem ser observados segundo os modelos SC ou PC.

A subdivisão de acessos de sincronização *acquire* e *release* permite ao modelo RC um tratamento diferenciado e mais flexível do que o empregado para as cercas do modelo WC.

O modelo RC relaxa as restrições de consistência em relação ao modelo WC, e ainda executa corretamente programas propriamente sincronizados, necessitando porém que os acessos de sincronização sejam mapeados em *acquires* e *releases*. Este fato faz com que a programação tenha que ser um pouco mais cuidadosa devido a inserção das operações de sincronização agora divididas em *acquires* e *releases*. Ele porém, apresenta uma redução sensível na latência de acesso à memória compartilhada, pois os pontos de espera são apenas nas operações de *release*. Como exemplos de sistemas que utilizam o modelo RC podemos citar DASH[60] Munin[21], Quarks[79] e Shasta[71].

Lazy Release Consistency

O modelo *Lazy Release Consistency* (LRC)[36] é uma versão mais relaxada do modelo RC, suportando o mesmo modelo de programação. LRC relaxa as condições de RC porque não necessita que numa operação de *release* os acessos anteriores sejam globalmente visíveis. Ele necessita apenas que os acessos anteriores ao *release* sejam visíveis somente no processador que executar um *acquire*. Isto é, a propagação das modificações feitas por um processador P_i para outro processador P_j são atrasadas até o momento de um *acquire* subseqüente em P_j numa mesma variável de sincronização.

A comunicação é realizada somente entre os dois processadores envolvidos, *acquirer* e *releaser*.

Para determinar quais modificações um processador deve ter ciência no momento de um *acquire*, LRC estabelece uma ordem parcial dos acessos aos dados compartilhados, denominada *happened-before-1* ($hb1$). Esta ordem parcial $hb1$ é baseada na ordem seqüencial de execução de um processador e no encadeamento das operações *acquire* e *release* realizadas em processadores diferentes, mas sob a mesma variável de sincronização. Dois acessos à memória compartilhada a_1 e a_2 são ordenados por $hb1$, denotado por $a_1 \stackrel{hb1}{\rightarrow} a_2$, se:

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

A ordem $hb1$ é dada basicamente pelo fecho transitivo da ordem de execução de um processador com a ordem das operações de sincronização na mesma variável de sincronização. Se $a_1 \stackrel{hb1}{\rightarrow} a_2$ e $a_2 \stackrel{hb1}{\rightarrow} a_3$ então $a_1 \stackrel{hb1}{\rightarrow} a_3$.

Como exemplo entre os sistemas DSM que implementam o modelo de consistência LRC podemos citar TreadMarks[50], HLRC[85], AURC[41] e ADSM[64].

Entry Consistency

O modelo *Entry Consistency* (EC) foi desenvolvido por Bershad *et al.*[13]. Ele relaxa a consistência dos dados em relação a LRC através da associação dos dados compartilhados a variáveis de sincronização. Neste modelo, numa operação de *acquire* é garantido que somente os dados compartilhados que são acessados dentro da seção crítica são coerentes. Assim, a variável de sincronização que controla o acesso a uma seção crítica atua também como um guarda dos dados compartilhados. Informalmente, a condição para que um sistema esteja consistente segundo o modelo EC é:

> Antes que uma operação de *acquire* em uma variável de sincronização tenha terminado em P, todas as atualizações nos dados compartilhados protegidos pela variável de sincronização devem ter sido observados por P.

O relaxamento proposto pelo modelo EC traz uma redução potencial do número de mensagens de coerência e do impacto de falso compartilhamento. Isto porque

não só determina quando mas também quais modificações devem ser observadas. Porém, afeta o modelo de programação, pois alguns programas propriamente sincronizados podem exigir modificações para executar corretamente sob este modelo. Por exemplo, programas que usam uma estratégia de filas de tarefas podem exigir alterações não triviais para executar corretamente sob o modelo EC. Apesar da complexidade adicional do modelo de programação os ganhos de desempenho observados por implementações de EC sobre LRC não foram no entanto significativos [1, 63].

Como exemplo de sistemas DSM que implementam o modelo de consistência EC podemos citar Midway[13] e AEC[74].

2.2 Estrutura dos Dados e Unidade de Coerência

A estrutura e o tamanho da unidade de coerência estão muito relacionados. A estrutura se refere a distribuição dos dados na memória compartilhada. Muitos sistemas não possuem nenhuma estrutura, isto é, a memória é um *array* linear de palavras. Outros sistemas podem possuir algum tipo de estrutura tal como objetos ou tipos de linguagem[68].

A unidade de coerência define o tamanho da unidade de compartilhamento dos dados. As unidades podem ter o tamanho de um byte, de uma palavra, de uma página ou de uma estrutra de dados complexa.

Geralmente, os sistemas *hardware DSM* possuem unidades de coerência menores que os sistemas *software DSM*. Tipicamente, os sistemas implementados em *hardware* utilizam unidades de coerência do tamanho de linhas de *cache* enquanto os sistemas em *software* utilizam unidades de coerência do tamanho de uma página. Este fato faz com que os sistemas *software DSM* possam ficar mais sensíveis aos padrões de compartilhamento das aplicações do que os sistemas *hardware DSM*[42]. Os sistemas *software DSM* são sensíveis a interação entre a estrutura de dados e os padrões de acesso com a unidade de coerência do tamanho de página, que pode provocar a ocorrência de falso compartilhamento e fragmentação.

Vários sistemas *software DSM* fazem uso dos mecanismos de proteção de páginas de memória virtual disponíveis na maioria dos sistemas operacionais modernos. Estes mecanismos permitem inferir se uma página compartilhada possui dados válidos ou inválidos. As páginas que não contém dados válidos localmente são protegidas contra leitura. Dessa forma, o sistema pode interceptar os acessos às posições

de memória inválidas localmente e, assim, iniciar as operações necessárias para buscar os dados em nós remotos de modo a torná-los válidos. Esta abordagem sofre de problemas relacionados a grande unidade de comunicação e coerência (página inteira), como falso compartilhamento e fragmentação.

Os dois problemas que podem ser gerados pela unidade de coerência ser do tamanho da página são o falso compartilhamento e a fragmentação. O falso compartilhamento ocorre quando dois ou mais processadores realizam um acesso a dados não relacionados que estão armazenados numa mesma página, e pelo menos um deles escreve num dado. Este fato faz com que o mecanismo de coerência transfira a página entre os processadores inúmeras vezes (efeito *ping-pong*). A fragmentação ocorre quando um processador necessita somente de parte da página mas tem que buscá-la inteira, causando tráfego de dados desnecessário. Estes problemas, se não solucionados ou reduzidos, muitas vezes anulam as vantagens de uma unidade maior como a página que favorece a localidade de referência.

2.3 Protocolos de Coerência

Para melhorar o desempenho, os sistemas DSM permitem a replicação dos dados compartilhados nas memória privativas dos processadores, na tentativa de reduzir o número de acessos remotos. No entanto, tal replicação exige a utilização de um protocolo para manter a memória coerente. A coerência deve ser mantida segundo um modelo de consistência de memória. Para manipular a replicação dos dados compartilhados existem dois tipos de protocolo de coerência, os baseados em invalidações e os baseados em atualizações. Estes protocolos visam propagar para os demais processadores as modificações das posições de memória compartilhada feitas localmente a um processador.

No mecanismo de invalidação somente é enviada, aos demais processadores, a informação de que os dados foram modificados. Ao receber uma mensagem deste tipo os processadores remotos invalidam seus dados correspondentes. Uma tentativa de acesso a estes dados gera uma falha de acesso. Neste tempo, é buscada uma versão atual do dado modificado remotamente. Neste mecanismo, as falhas de acesso não são evitadas, porém somente são transmitidos dados que realmente serão necessários.

No protocolo de atualização os próprios dados modificados localmente por um processador são enviados aos demais processadores. Dessa forma, se um outro pro-

cessador realizar um acesso ao dado compartilhado, este já estará disponível, não ocasionando falha de acesso. Este mecanismo minimiza as falhas de acesso, porém pode vir a provocar uma carga desnecessária de comunicação se novas modificações forem feitas antes que seja realizado um acesso ao dado[15].

Um protocolo híbrido e auto-adaptável, baseado em ambos protocolos, pode potencialmente melhorar o desempenho. Se é identificado que existe um único escritor no dado compartilhado, por exemplo, é melhor, na maior parte das vezes, utilizar o mecanismo de atualização. Em caso contrário, múltiplos escritores e uma quantidade de dados modificados muito pequena, o mais conveniente é utilizar o mecanismo de invalidação. Suponha, por exemplo, um sistema com unidade de coerência do tamanho de uma página. Suponha ainda que vários processadores compartilham uma página e escrevem em somente uma palavra de modo exclusivo. No protocolo de atualização a página inteira é enviada aos processadores e com isso uma grande quantidade de dados é transmitida desnecessariamente. No protocolo de invalidação, as escritas são apenas notificadas e as modificações são transmitidas somente quando cada processador necessita do dado.

2.4 Suporte a Múltiplos Escritores

Para minimizar o problema de falso compartilhamento quando a unidade de coerência do sistema é grande, como em muitos sistemas *software DSM* que utilizam como unidade a página, são utilizados protocolos de múltiplos escritores. Nestes protocolos são permitidas escritas concorrentes à mesma unidade de coerência, retardando a visão das modificações para um ponto de sincronização posterior entre os processadores. É importante ressaltar que as escritas concorrentes devem ser feitas a diferentes posições de memória. Em caso contrário, estaria caracterizada uma condição de corrida. Esta condição não é permitida em programas propriamente sincronizados. Para este tipo de programa, escritas feitas na mesma posição de memória devem ser ordenadas através de operações de sincronização.

O principal problema de se permitir a existência de múltiplos escritores na mesma unidade de coerência está em combinar todas as escritas realizadas para montar uma versão coerente dela.

Considerando a página como unidade de coerência, na implementação de protocolos com múltiplos escritores, podem ser utilizados os mecanismos de *twinning* e

diffing[22]. Inicialmente, todas as páginas compartilhadas são protegidas contra escrita. Quando um processador tenta modificar um dado compartilhado localmente, é gerada uma falha de acesso à página. Ao detectar a falha, o protocolo *software DSM* faz uma cópia da página (*twin*) e a libera para escrita. Assim, as escritas locais ocorrem livremente na página e o *twin* mantém a versão original. No momento da propagação das modificações locais, o protocolo faz uma comparação entre a cópia atual da página e o *twin*, e cria uma estrutura, denominada *diff*, contendo as modificações realizadas. Dessa forma, para se obter uma página atualizada é necessário aplicar os diversos *diffs* criados pelos diferentes processadores que compartilham a mesma página e a modificaram. A aplicação dos *diffs* deve preservar a ordenação imposta pelas operações de sincronização, para evitar que uma escrita recente seja sobreposta por uma outra escrita mais antiga na mesma posição de memória.

A figura 2.2 mostra em exemplo de uma página sendo escrita por dois processadores (P_0 e P_1) ao mesmo tempo, porém, em dados diferentes. Ao detectar uma falha de proteção de escrita, os processadores geram *twins* para suas páginas locais e a liberam para a escrita. P_0 escreve em a e b e P_1 escreve em c. Quando for necessário uma versão atualizada da página, cada processador cria o *diff* correspondente às suas escritas e a combinação deles forma a nova versão da página.

O suporte a múltiplos escritores consegue aliviar os problemas de falso compartilhamento mas traz alguns *overheads* relativos aos custos de detecção, armazenamento e consolidação das modificações nos dados compartilhados. Estes custos poderiam ser eliminados para as páginas que não são sujeitas a falso compartilhamento, isto é, aquelas onde somente um processador as modifique de cada vez. Assim, sistemas que possam se beneficiar de ambas as estratégias, implementando técnicas que se adaptem a diferentes padrões de compartilhamento, se mostram essenciais.

2.5 Técnicas para Reduzir *Overheads* em Sistemas DSM

Os principais problemas de sistemas *software DSM* são a latência de comunicação e o tempo dispendido com as operações para manter a memória coerente. Técnicas que possam tolerar ou minimizar estes *overheads* são essenciais para melhorar o desempenho dos sistemas *software DSM*. A seguir, apresentamos algumas das técnicas que podem ser utilizadas de forma adaptativa em sistemas DSM, e que lidam com

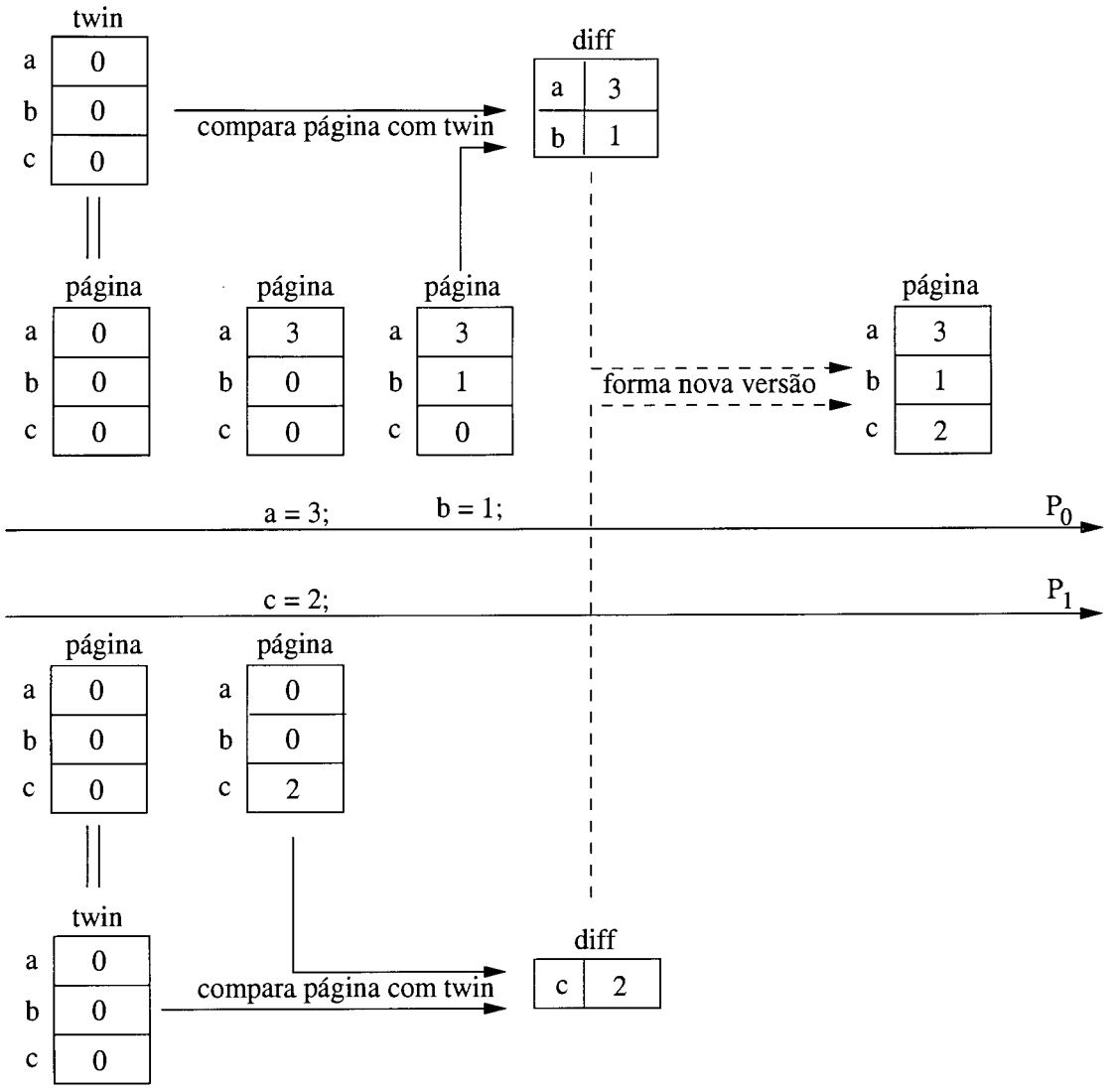

Figura 2.2: Suporte a múltiplos escritores com esquema de criação de *diff*

as falhas de acesso induzidas pelo compartilhamento sobrepondo acessos à memória com computação ou com outros acessos à memória. Para cada uma delas enfocamos os problemas que buscam solucionar, formas de implementação, suas vantagens e desvantagens, e exemplificamos alguns dos sistemas que implementam estas técnicas. A adaptação entre um escritor ou múltiplos escritores, e as técnicas de *forwarding* e *prefetching* são utilizadas nos nossos experimentos. Já a técnica de agregação, *pre-diffing*, adaptações entre protocolos de invalidação e atualização, unidade de coerência, e a técnica *multithreading* não fazem parte do escopo deste trabalho.

Único Escritor/Múltiplos Escritores (SW-MW)

Protocolos com suporte a únicos escritores podem sofrer com o efeito *ping-pong* provocado pelo falso compartilhamento. Protocolos com suporte a múltiplos es-

critores aliviam os efeitos do falso compartilhamento, mas têm *overheads* de geração de *twins*, criação e aplicação de *diffs*.

Se processadores distintos escrevem concorrentemente em regiões diferentes de uma página, é ativado o modo MW. Em caso contrário, somente um processador escreve na página, é ativado o modo SW. Assim, só são introduzidos *overheads*, correspondentes ao gerenciamento das modificações e o seu armazenamento, para as páginas do modo MW. Nas demais páginas no modo SW, a página inteira pode ser transferida de seu escritor para o(s) seu(s) leitor(es) sem a necessidade de armazenamento das modificações. Ao sofrer uma falha de página, o processador deve requisitar a página ou os *diffs* dependendo do modo que esteja ativado.

Os protocolos adaptativos *software DSM* do tipo único escritor/múltiplos escritores (SW-MW) permitem a utilização de ambos os suportes quando identificado o padrão de compartilhamento adeqüado a cada página. Esta adaptação pode ser definida estaticamente, através de anotações realizadas pelo programador[21] ou pelo compilador, ou dinamicamente, através do *runtime system*[7, 64].

A adaptação SW-MW não visa diminuir o número de falhas de página, mas sim diminuir o *overhead* relativo aos mecanismos de *twining* e *diffing* utilizados para manter a coerência das páginas. Numa falha de página ou é transferida a página ou são tranferidos os *diffs*. Pode ocorrer um aumento do tamanho médio das mensagens transferidas, por exemplo, quando numa aplicação existe fragmentação numa página do modo SW, isto é, são modificados apenas poucos dados compartilhados, e a página inteira é transferida quando seria necessário transferir somente parte dela.

Adaptação à Página/*Diff*

Discutimos a adaptação em relação à transferir a página inteira ou *diff*. Esta adaptação considera a quantidade de dados modificados e decide quando seria melhor usar *diffs* ao invés de páginas, usando, por exemplo, um determinado valor limite. Se o tamanho das modificações na página é pequeno, o custo de criação do *twin*, geração e transferência de *diffs* pode ser menor que a transferência de uma página completa. Além da quantidade de modificações realizadas nos dados compartilhados, deve-se considerar a banda passante da rede de interconexão, pois o tempo de transferência é dependente dela. No trabalho de Amza *et al.*[7] são mostrados resultados para um protocolo que implementa essa adaptação.

Esta adaptação se utilizada em conjunto com a adaptação SW-MW pode melho-

rar o desempenho evitando a fragmentação no modo SW, e aliviando a acumulação de *diffs* que ocorre no modo MW, principalmente quando os *diffs* são grandes.

Forwarding

A técnica de *forwarding* evita e/ou reduz o *overhead* das falhas de acesso aos dados remotos enviando antecipadamente as modificações realizadas nos dados compartilhados. Esta técnica admite implementações com diferentes unidades de coerência, por exemplo, envio antecipado de páginas ou *diffs*. Nela, os dados compartilhados modificados são enviados antecipadamente pelo escritor a um conjunto de processadores, de modo seletivo. Os processadores que recebem os dados antecipadamente podem ser selecionados estaticamente[56] ou dinamicamente[60, 64].

Na implementação da técnica de *forwarding*, também devem ser considerados quando enviar os dados compartilhados e quando atualizar destes dados. O envio antecipado dos dados pode ser realizado a cada escrita ou nas sincronizações, por exemplo. Quando os dados compartilhados chegam ao processador destino, eles podem ser atualizados imediatamente ou só quando ocorrer uma falha de acesso.

Se bem sucedida, esta técnica esconde o tempo de espera pelo dado remoto. Dependendo da forma de implementação, pode ainda reduzir o número de falhas de acesso, se os dados forem recebidos, atualizados imediatamente, e acessados antes que novas modificações ocorram.

Existem duas situações onde a técnica de *forwarding* não é apropriada. Primeiro, quando não se consegue selecionar quais processadores necessitam dos dados compartilhados. Segundo, mesmo que se consiga definir tais processadores, eles não acessam os dados antes que eles sejam novamente modificados e enviados. Neste caso, o sistema apresenta maior *overhead* das falhas de acesso aos dados. As reduções nos *overheads* podem também ser menores do que o esperado se as modificações não chegam a tempo no processador destino.

Prefetching

A técnica de *prefetching* visa melhorar o desempenho de sistemas *DSM* em relação ao acesso a dados remotos. Ela tenta esconder a latência do acesso ao dado enviando o pedido pelo dado remoto com antecedência o suficiente do seu uso efetivo. Para tanto, é necessário prever quais dados serão utilizados em avanço no tempo. Esses dados devem ser pedidos com antecedência para que quando forem acessados estejam disponíveis na memória local do processador. Esta abordagem causa *overheads*

de tempo de execução, mesmo que esta decisão seja também feita a tempo de compilação. O despacho de pedido dos dados é feito a tempo de execução o que interfere na execução.

A técnica de *prefetching* já foi estudada tanto no contexto de *hardware DSM*, como em [29, 16, 66], quanto em *software DSM* [14, 47, 17, 65], e também admite implementações considerando páginas ou *diffs*, e, atualizações imediatas dos dados recebidos ou só quando ocorre uma falha de acesso.

Da mesma forma que a técnica de *forwarding*, se bem sucedida, a técnica de *prefetching* esconde o tempo de espera pelo dado remoto, e pode ou não diminuir o número de falhas de acesso se as atualizações dos dados remotos são realizadas imediatamente. Se os dados são pedidos com muita antecedência, eles podem ser novamente modificados sendo necessário então enviar um novo pedido pelo dado compartilhado quando o acesso ocorre. Além disso, na situação inversa, as reduções nos *overheads* podem ser menores do que o esperado se os dados não foram pedidos com antecedência o suficiente.

Agregação

A quantidade de mensagens trocadas para manter a memória coerente pode ser um fator determinante no desempenho de várias aplicações. A técnica de agregação combina o envio de vários dados modificados numa mesma mensagem. A combinação resulta num menor número de mensagens transmitidas e redução dos *overheads* de ativação e recepção das mensagens.

A agregação pode ser especificada de forma estática ou dinâmica. Na forma estática, o programador ou compilador determinam o tamanho do conjunto de páginas contínuas que serão transferidas juntas. Na forma dinâmica, é utilizado um algoritmo baseado em detecções de falhas de acesso para determinar quais páginas serão agrupadas no mesmo conjunto para serem transferidas. A agregação dinâmica permite que páginas não contínuas pertençam a um mesmo conjunto. Em [8] é apresentado um algoritmo que agrega páginas dinamicamente.

Porém, se uma pequena parte das páginas são modificadas muito freqüentemente e as páginas pertencem a um mesmo conjunto, a técnica de agregação pode aumentar o efeito de falso compartilhamento, e como conseqüência, aumentar o número de mensagens e a quantidade de dados trocados. A agregação dinâmica, como qualquer técnica dinâmica, interfere na execução, pois a decisão de quais páginas pertencem

a um determinado conjunto é realizada em tempo de execução.

Pre-diffing

A técnica de *pre-diffing* utiliza o tempo de espera por sincronização para esconder o *overhead* da criação de *diffs*. Sempre que um processador estiver ocioso esperando por uma sincronização, ele cria *diffs* para as páginas que possui e que estão modificadas.

Esta técnica pode aumentar o número de falhas de página e degradar o desempenho do sistema se os *diffs* criados não são utilizados antes que ocorram novas modificações. Neste caso, deve-se abandonar o *diff* criado e recuperar o *twin* anterior. Porém, se os *diffs* são utilizados, o *overhead* da criação de *diffs* terá sido escondido.

Os sistemas Quarks[79] e AEC[74] implementam a criação antecipada de *diffs*.

Invalidação/Atualização

A adaptação entre os protocolos de invalidação e atualização visa melhorar o desempenho dos *overheads* associados à comunicação dos dados remotos.

No mecanismo de invalidação são enviados somente os dados que realmente são modificados, mas não são evitadas falhas de acesso. Já o mecanismo de atualização envia os dados modificados, minimiza as falhas de acesso, porém pode vir a provocar uma carga desnecessária de comunicação se novas atualizações são feitas antes que seja realizado um acesso ao dado[15].

Os sistemas que possuem protocolo híbrido invalidação/atualização podem ser implementados tanto em *hardware* [28] quanto em *software* [21, 32, 64].

Multithreading

Os *overheads* de comunicação ou de coerência podem ser tolerados através da técnica *multithreading*, que explora o paralelismo através de múltiplas *threads* realizando troca de contexto em momentos que seriam de espera. Para tolerar a latência de sincronização e de memória, a técnica *multithreading* não necessita de previsão, e portanto, pode manipular arbitrariamente padrões de acesso imprevisíveis e complexos. Além disto, não necessita de modificações no programa. Porém, envolve *overheads* de tempo de execução para realizar a troca de contexto e pode causar *thrashing*.

A técnica *multithreading* também já foi implementada em sistemas *hardware*[75, 4, 5] e *software DSM*[34, 80, 76]. Ela tem o potencial de melhorar o desempenho de várias aplicações com um pequeno número de *threads* como mostrado em [80].

2.6 TreadMarks

TreadMarks (TM)[50] é um sistema de memória compartilhada distribuída desenvolvido em *software*. Sua implementação é a nível do usuário e utiliza as bibliotecas padrão do Unix para a criação de processos remotos, para a comunicação entre processos e para o gerenciamento de memória. A sua unidade de coerência é a página, sendo a coerência dos dados mantida com uso do protocolo de invalidações. Este protocolo é implementado através da propagação de notificações de escrita (*write notices*) em operações de sincronização. A *interface* de programação oferecida por TM provê dois tipos de primitivas de sincronização: *lock/unlock* e barreira.

Para reduzir os efeitos de falso compartilhamento causado pela unidade de coerência grande (páginas de 4K*bytes*), TreadMarks fornece suporte a múltiplos escritores através dos mecanismos de *twining* e *diffing*. Inicialmente todas as páginas são protegidas contra escrita. Quando há uma tentativa de modificação numa página, é gerada uma falha de acesso. O TM intercepta esta falha, faz uma cópia da página (*twin*) e a libera para escrita. Quando for necessária a propagação das modificações feitas localmente a uma página, TM faz uma comparacação entre o *twin* gerado e a versão modificada, e cria um *diff* contendo todas as modificações locais feitas à página. A cada *diff* existe um *write notice* associado identificando o intervalo e o processador onde o *diff* deve ser criado. Os intervalos são segmentos de tempo na execução de um processador e iniciados cada vez que uma operação de sincronização é executada.

Com o objetivo de reduzir o tráfego de dados pela rede, principalmente o número de mensagens, este sistema adota o modelo de consistência *Lazy Release Consistency* (LRC). O envio e a aplicação dos *diffs* são atrasados até o primeiro acesso à página realizado após uma operação de *acquire*. TreadMarks utiliza a ordem parcial *happens-before-1*[2] para ordenar os intervalos de diferentes processadores. Através de *hb*1 é possível determinar quais intervalos de outros processadores precedem o intervalo corrente de um processador P. As modificações realizadas nos dados compartilhados são associadas aos intervalos em que elas ocorreram e, assim, numa

operação de *acquire* ocorrida num intervalo i, o processador deve ser notificado sobre todas as modificações associadas a intervalos anteriores a i segundo a ordem parcial $hb1$.

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

Quando ocorre uma falha num acesso à página é necessário buscar um conjunto de *diffs* para torná-la válida, havendo então a comunicação com um ou mais processadores. Os *diffs* recebidos são aplicados à página, respeitando a ordenação parcial, para se obter uma versão coerente.

TreadMarks fornece suporte a múltiplos escritores visando diminuir os efeitos do falso compartilhamento e da fragmentação. Entretanto, para aplicações onde não há falso compartilhamento o *overhead* associado ao processamento dos *diffs* pode degradar o sistema. Esse *overhead* inclui o custo da criação de *twins* e de *diffs* e da aplicação dos *diffs*. Além disso, a busca dos *diffs* pode envolver vários processadores e gerar contenção na rede de interconexão. Uma das situações mais críticas seria aquela onde todos os processadores simultaneamente requisitam *diffs*.

A identificação de padrões de compartilhamento do tipo um escritor e um ou mais leitores sugere a utilização de uma adaptação (SW-MW) onde uma vez identificado um único escritor, é melhor enviar a página do que ter o *overhead* de tratamento de validação da página. Esta adaptação só não surte efeito quando a quantidade de dados modificados dentro da página é muito pequena. Neste caso pode ser melhor enviar os *diffs* do que a página inteira.

Por outro lado quando são identificados padrões de múltiplos escritores, a técnica de *prefetching* pode ser mais eficaz na antecipação da busca pelos dados, levando a um menor tempo de espera ou no melhor caso, ter os *diffs* disponíveis no momento do acesso. Esta técnica porém, se não tiver uma previsão correta pode degradar o sistema, pois gera interrupções desnecessárias nos processadores remotos, caso os

dados trazidos com antecedência sejam invalidados antes de serem utilizados.

As seções seguintes detalham as operações executadas nas falhas de acesso e nas sincronizações por barreiras e por *locks*.

2.6.1 Falhas de Acesso

Em TreadMarks, inicialmente todas as páginas do sistema estão válidas no processador 0 e inválidas nos demais processadores. Na primeira falha de acesso, o processador P ($P \neq 0$) deve requisitar uma cópia da página ao processador 0. A menos desta falha inicial, onde os processadores não têm ainda uma cópia da página, as outras falhas de acesso só ocorrem em dois casos: ou a página está válida e o acesso é uma tentativa de escrita numa página que está protegida contra escrita, ou a página está inválida.

Se a falha ocorre devido a um acesso de escrita, TreadMarks simplesmente cria um *twin* e desprotege a página contra escrita.

Quando ocorre uma falha num acesso a uma página inválida é necessário buscar um conjunto de *diffs* para torná-la válida. Os *diffs* devem ser buscados dos processadores que enviaram as notificações de escrita para a página. Uma característica importante de TreadMarks é a criação atrasada de *diffs*. Nele, um processador só cria um *diff* relativo às modificações realizadas em uma determinada página quando chega um requisição para esse *diff*. Além disso, de modo a minimizar o número de mensagens para buscar os *diffs*, TreadMarks emprega um esquema de dominância de intervalos. Este esquema visa a comunicação com um conjunto mínimo de processadores, denominado conjunto de processadores dominantes, que possuam os *diffs* necessários para tornar a página válida. Um intervalo i domina um intervalo j, se j precede i na ordem parcial $hb1$. Assim, as mensagens devem ser enviadas apenas aos processadores cujos intervalos mais recentes não são dominados pelos intervalos mais recentes de outros processadores. Considere, por exemplo, os segmentos de código da figura 2.3. Quando P_2 executa o comando `lock(c)`, ele deve requisitar *diffs* apenas ao processador P_1, pois o intervalo I_2 de P_1 domina os intervalos I_0 e I_1 de P_0 e I_0 de P_1. Note também que neste momento, P_2 também observa as modificações realizadas fora de seção crítica (escrita em z e y).

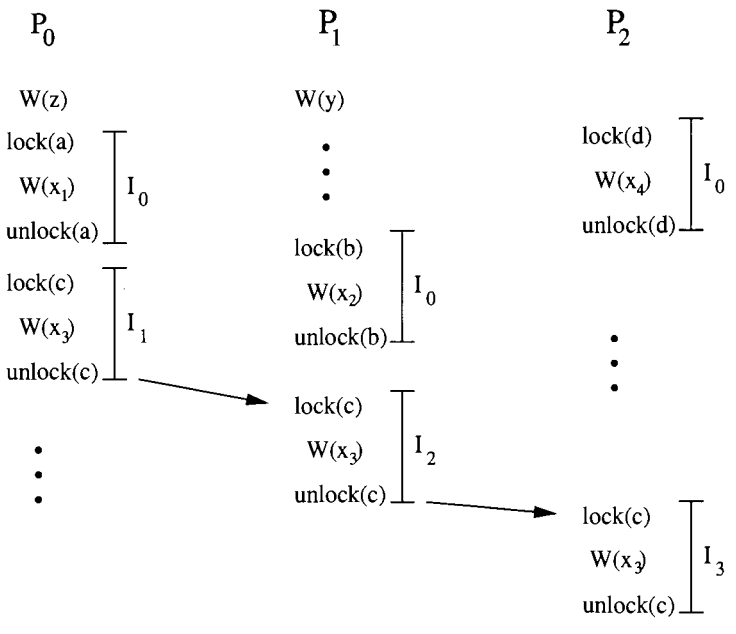

Figura 2.3: Exemplo de dominância de intervalos

2.6.2 Barreiras

Na barreira é realizada uma sincronização global, isto é, cada processador recebe as notificações de todas as modificações feitas pelos demais processadores.

Para cada barreira há um gerente selecionado estaticamente. Ao chegar a uma barreira, cada processador envia ao gerente uma mensagem, contendo todos os seus intervalos. A cada mensagem recebida o gerente da barreira armazena os intervalos associados a mensagem, e invalida as páginas para as quais são recebidas notificações de escrita. Após receber e manipular todas as mensagens enviadas pelos processadores, o gerente envia de volta aos mesmos, mensagens contendo notificações de escrita que cada um deles ainda não recebeu. Ao receber a mensagem enviada pelo gerente, cada processador age da mesma forma que ele, armazena os intervalos e invalida suas respectivas páginas para as quais são recebidas notificações de escrita.

2.6.3 *Locks*

Nas sincronizações realizadas com *locks*, também existe um gerente associado estaticamente a cada variável de sincronização. Numa operação de aquisição de *lock*, o processador *acquirer* envia uma mensagem com seu intervalo para o gerente do *lock*. O gerente verifica qual o último processador que requisitou o *lock* e redireciona para este processador o pedido de posse do *lock*.

Quando o *lock* está disponível, o processador *releaser* (último processador a

adqüirir o *lock*) compara seus intervalos com os intervalos recebidos do processador *acquirer* e envia uma mensagem contendo todos as notificações de escrita que ele ainda não recebeu. Isto é, o processador *releaser* envia todas as notificações de escrita que ele possui e que foram geradas após o intervalo enviado pelo processador *acquirer*, segundo a ordenação parcial $hb1$. Ao receber a mensagem, o processador *acquirer* armazena os intervalos recebidos e invalida as suas páginas correspondentes.

Capítulo 3

Caracterização de Estados e Padrões de Compartilhamento

Neste capítulo introduzimos uma nova máquina de estados, denominada FIESTA (*FInitE STAte machine*), utilizada para caracterizar o estado e o padrão de compartilhamento das páginas das aplicações ao serem executadas em sistemas *software DSM*. A caracterização do estado de compartilhamento de cada página é baseada em transições de estado disparadas por eventos gerados pelo protocolo para manter a memória coerente.

Mostramos como FIESTA foi implementada sobre o modelo de consistência *Lazy Release Consistency* de TreadMarks, sem a necessidade de mensagens adicionais. As informações dos estados são enviadas juntamente com as mensagens trocadas nas sincronizações.

3.1 Estados e Eventos

FIESTA é uma máquina de estados finitos construída sobre o protocolo TreadMarks. Para a captação dos estados de compartilhamento, ela não altera o modelo de consistência LRC e nem o protocolo de coerência. Os eventos utilizados por FIESTA para provocar as transições de estado são os mesmos eventos utilizados pelo protocolo para manter a memória coerente. Além disso, ela não inclui mensagens extras para a captação dos estados de compartilhamento, o que é fundamental para sistemas *software DSM*.

FIESTA capta dinamicamente as informações sobre cada página compartilhada. Ao longo da execução, a página muda o seu estado segundo o tipo de acesso realizado pelos processadores.

Os acessos à uma página podem ser divididos em acessos dentro e fora de seção crítica e acessos de leitura ou de escrita. A partir desta divisão, definimos os estados de uma página, considerando também a quantidade de processadores que realizam esses acessos à página. Dessa forma, uma página pode estar em um dentre os seguintes estados para acessos fora de seção crítica: único escritor (SIWR); único leitor (SIRE); único leitor/único escritor (SRSW); único leitor/múltiplos escritores (SRMW); múltiplos leitores/único escritor (MRSW) e múltiplos leitores/múltiplos escritores (MRMW). Em relação aos acessos dentro de seção crítica temos os seguintes estados: único produtor (SIPR); único consumidor (SICO); único produtor/único consumidor (SPSC); único produtor/múltiplos consumidores (SPMC); múltiplos consumidores (MUCO) e migratória (MIGR). FIESTA possui ainda dois estados definidos como UCLP e INOUT. Estes estados refletem o estado inicial de uma página e o estado onde ocorrem acessos dentro e fora de seção crítica na mesma fase, respectivamente.

A tabela 3.1 mostra as condições que caracterizam cada um dos possíveis estados de uma página segundo FIESTA. A caracterização de cada estado de compartilhamento é representada pelos conjuntos denominados R, W, P e C, e pelo conteúdo de cada um deles. Esses conjuntos representam os tipos de acesso realizados pelos processadores. O conjunto R corresponde aos acessos de leitura e o conjunto W aos acessos de escrita, ambos fora de seção crítica. Os conjuntos C e P correspondem também a acessos de leitura e escrita, respectivamente, porém, dentro de seção crítica. Como exemplo de caracterização, considere o estado SIRE. Para estar neste estado é necessário que exista somente um processador no conjunto dos leitores fora de seção crítica e não existam processadores nos outros conjuntos. O estado INOUT possui uma característica diferente dos demais. Para estar neste estado, basta que exista pelo menos um processador com acesso fora de seção crítica e um outro processador com acesso dentro de seção crítica, não importa se os acessos são de leitura ou escrita.

A transição de um estado para outro é provocada pelos eventos gerados pelo protocolo para manter a memória coerente. Os eventos que disparam as mudanças entre os estados são: falha num acesso de leitura (**F**); violação de proteção de escrita com a criação de um *twin* (**T**); invalidação (**I**); aquisição de uma variável de *lock* (**acq**) e barreira (**B**).

Todos os estados gerados por FIESTA e suas transições estão mostrados na

Estados de Compartilhamento	Condições			
	|R|	|W|	|P|	|C|
UCLP	0	0	0	0
SIRE	1	0	0	0
SIWR	0	1	0	0
SRSW	1	1	0	0
MRSW	>1	1	0	0
SRMW	1	>1	0	0
MUWR	0	>1	0	0
MURE	>1	0	0	0
MRMW	>1	>1	0	0
SICO	0	0	0	1
SIPR	0	0	1	0
SPSC	0	0	1	1
SPMC	0	0	1	>1
MUCO	0	0	0	>1
MIGR	0	0	>1	X
INOUT	>0	X	X	>0
	>0	X	>0	X
	X	>0	X	>0
	X	>0	>0	X

0 : não contém processadores
1 : contém exatamente um processador
>1 : contém mais de um processador
>0 : contém pelo menos um processador
X : não importa a quantidade de processadores

Tabela 3.1: Estados de compartilhamento de uma página segundo FIESTA

figura 3.1. Nessa figura as setas contínuas representam as transições fora de seção crítica, as setas tracejadas as transições dentro de seção crítica e as setas pontilhadas as transições provocadas pelos eventos de barreira. Os eventos que provocam as transições entre os estados estão especificados nos pontos de partida de cada seta.

Para exemplificar como ocorrem as transições entre os estados explicamos a seguir uma pequena parte de FIESTA. Considere uma página no seu estado inicial UCLP. A ocorrência de um evento identificado como falha num acesso de leitura, provoca a transição do seu estado para SIRE. Na ocorrência de um evento de violação de proteção de escrita, fora de seção crítica, pode ocorrer a transição para dois possíveis estados. Transição do estado SIRE para SIWR, se o evento ocorreu no processador leitor ou transição para o estado SRSW se o evento ocorreu num outro processador. Similarmente, para a ocorrência de um evento **F**, dentro de seção crítica numa página

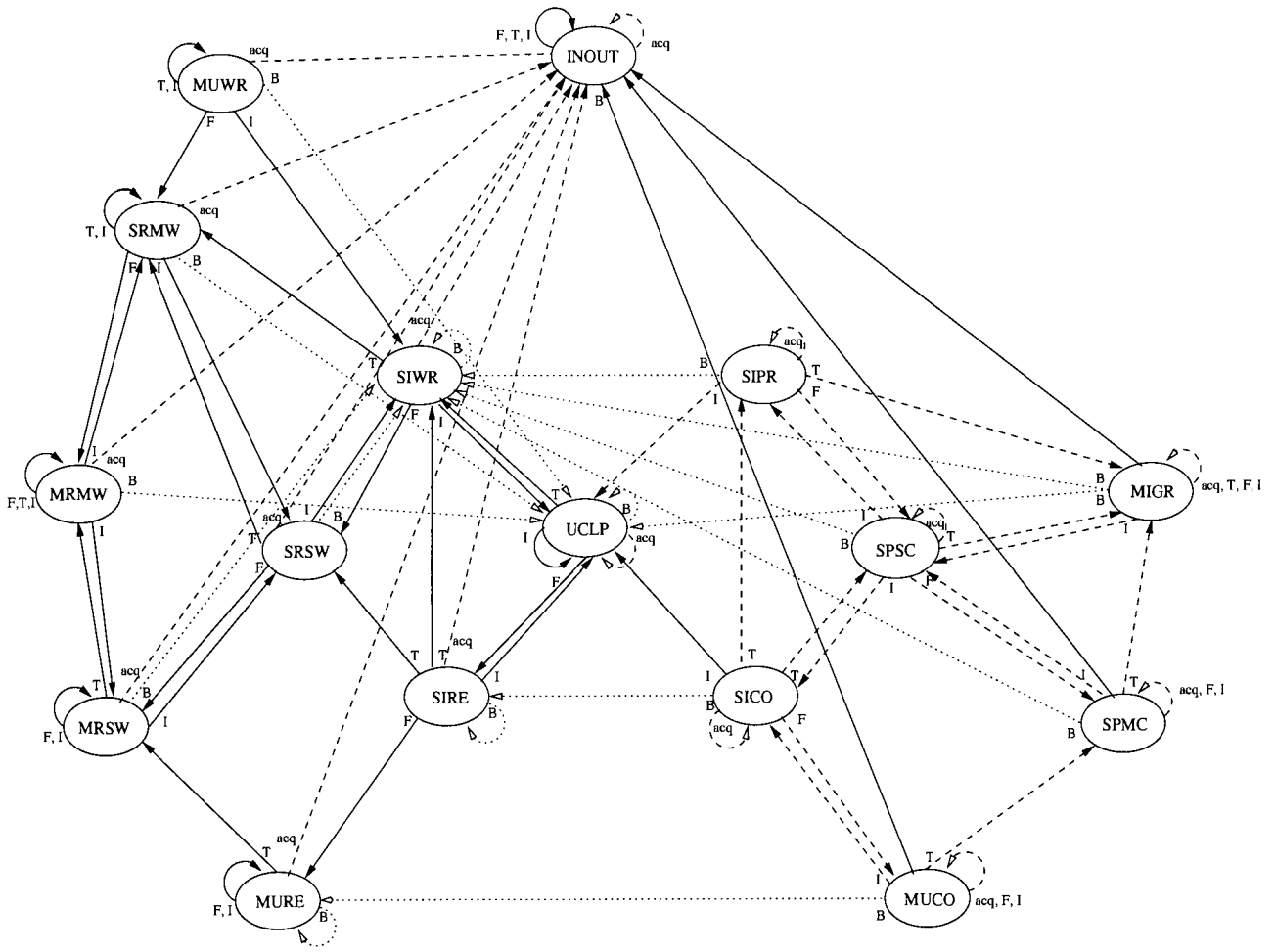

Figura 3.1: Diagrama de transições de estados de FIESTA

UCLP, provoca a transição do seu estado para SICO. Na ocorrência de um evento **T**, pode ocorrer a transição para dois possíveis estados. Transição do estado SICO para SIPR, se o evento ocorreu no processador consumidor ou transição para o estado SPSC se o evento ocorreu num outro processador.

A ocorrência de um evento **acq** quando a página está num estado fora de seção crítica, provoca a transição para o estado INOUT, se o último evento não tiver sido uma barreira. Uma vez neste estado, a única transição possível é para o estado UCLP num evento de barreira. Da mesma forma ,se a página têm acessos somente dentro de seção crítica e ocorre um evento **F** ou **T**, fora de seção crítica, por um processador que não está presente nos conjuntos P e C, a página passa ao estado INOUT.

Diferentemente de FIESTA, máquinas de estados mais simples[64, 6, 7, 8] não conseguem distingüir entre todos os estados de compartilhamento, como por exemplo páginas com acessos dentro e fora de seção crítica, que são então categorizadas de

forma única genérica. Esses estados ocorrem freqüentemente em diversas aplicações com padrões de escrita irregular.

Aplicações irregulares que possuem páginas com acesso dentro e fora de seção crítica ou com páginas que possuem mais de uma variável de *lock*, podem ter uma categorização dos estados de compartilhamento mais precisa utilizando FIESTA.

Para destacar mais facilmente a diferença entre outras abordagens e a nossa proposta de máquina de estados, mostramos, na figura 3.2, uma simplificação de FIESTA. Essa máquina simplificada é composta por cinco grupos diferentes de estados considerando somente os acessos de escrita: UCLP, SW, MW, MIGR e INOUT. Os estados são definidos da seguinte forma. O estado SW é composto pelas páginas onde ocorrem acessos de escrita realizados por um único processador. As páginas onde ocorrem acessos de escrita por mais de um processador pertencem ao estado MW. As páginas onde ocorrem acessos de escrita realizados por mais de um processador mas de modo exclusivo, estão no estado MIGR. Este estado também inclui as páginas que têm acessos de escrita exclusivos com mais de uma variável de *lock*. As páginas pertencentes ao estado INOUT são aquelas onde ocorrem tanto acessos dentro quanto fora de seção crítica. Um outro estado incluído nesta figura é o UCLP que representa o estado inicial das páginas.

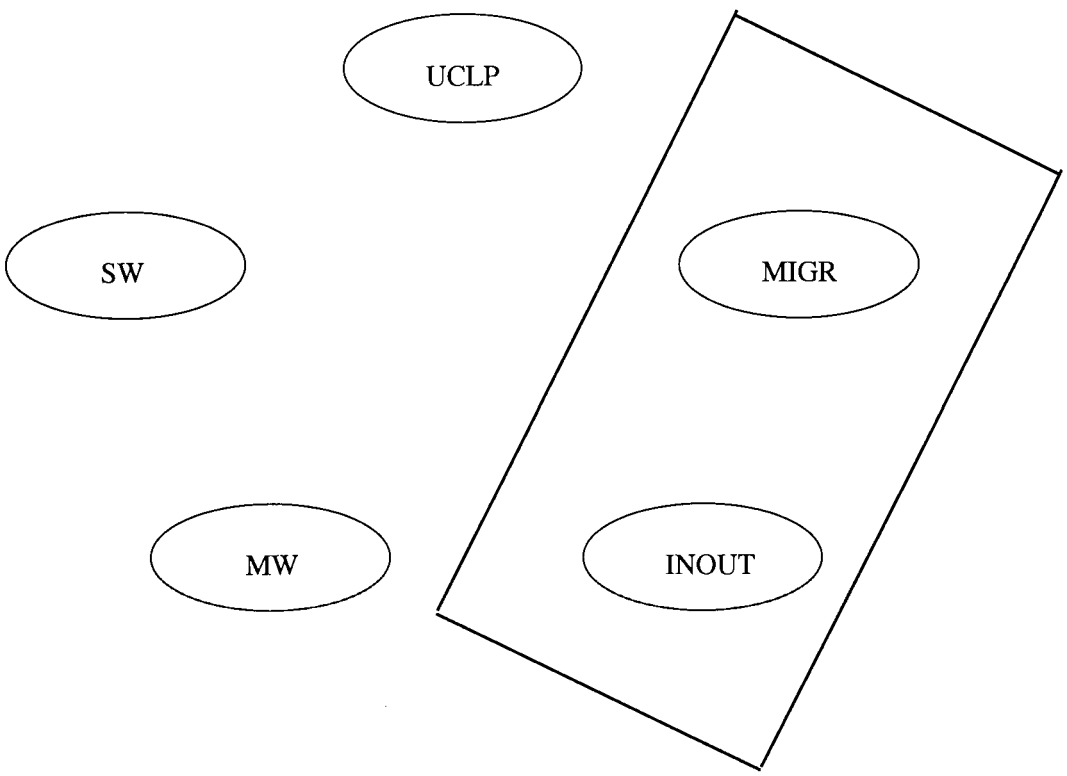

Figura 3.2: Estados de compartilhamento do modelo de FIESTA simplificado

Na figura 3.2 destacamos com um retângulo a contribuição inovadora de FIESTA. As máquinas de estados mais simples não conseguem distingüir e categorizar de forma mais precisa as páginas com padrões de escrita INOUT ou MIGR com mais de uma variável de *lock*. A categorização dos padrões de escrita das páginas que estão contidas nos estados dessa figura, dependem da seqüência dos estados de compartilhamento captada por FIESTA, e estão explicadas no próximo capítulo. Dessa forma, as páginas que apresentam essas características podem também se beneficiar das técnicas de tolerância à latência introduzidas no protocolo.

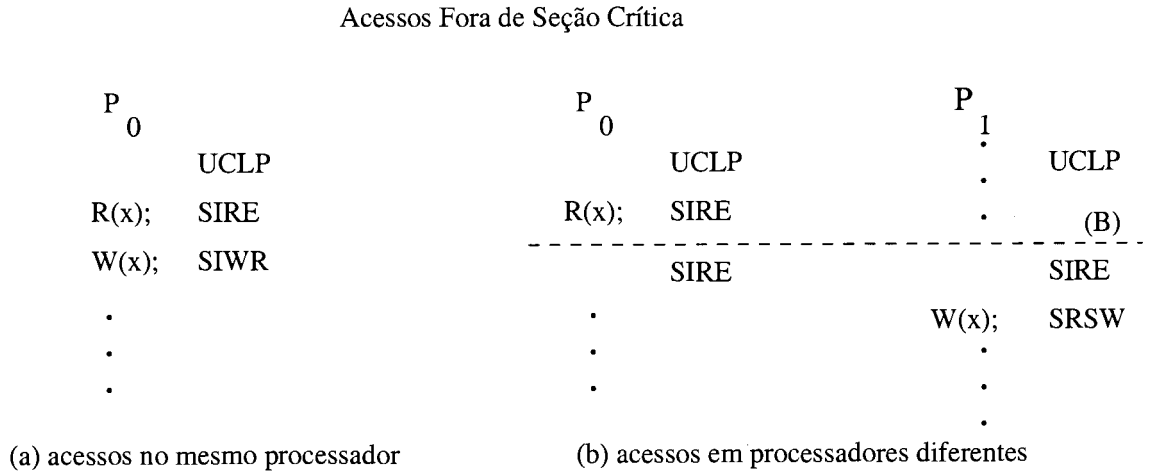

Figura 3.3: Trechos de código simples que exemplificam algumas transições de estado

Na figura 3.3 reexaminamos o exemplo anterior, apresentando agora alguns trechos de código simples que exemplificam transições de estado fora (a e b) e dentro (c e d) de seção crítica, provocadas pelas falhas nos acessos de leitura (**F**) e escrita (**T**). Suponha uma página no seu estado inicial UCLP. A ocorrência de um evento

identificado como falha num acesso de leitura, provoca a transição do seu estado para SIRE. Na ocorrência de um evento de violação de proteção de escrita, fora de seção crítica, pode ocorrer a transição para dois possíveis estados. Transição do estado SIRE para SIWR (a), se o evento ocorreu no processador leitor ou transição para o estado SRSW (b) se o evento ocorreu num outro processador. Similarmente, a ocorrência de um evento **F**, dentro de seção crítica numa página UCLP, provoca a transição do seu estado para SICO. Na ocorrência de um evento **T**, pode ocorrer a transição para dois possíveis estados. Transição do estado SICO para SIPR (c), se o evento ocorreu no processador consumidor ou transição para o estado SPSC (d), se o evento ocorreu num outro processador.

3.2 Implementação de FIESTA

Para manter informações sobre os tipos de estado de compartilhamento que resultam dos acessos dinâmicos às páginas compartilhadas, FIESTA possui, associada a cada página, uma estrutura que armazena o estado corrente da página bem como os conjuntos de processadores relacionados a cada tipo de acesso. Quando ocorre um evento, a identificação do processador é adicionada ao conjunto apropriado ($\{R\}$, $\{W\}$, $\{P\}$ e $\{C\}$). O conjunto R corresponde aos acessos de leitura e o conjunto W aos acessos de escrita, ambos fora de seção crítica. Os conjuntos C e P correspondem também a acessos de leitura e escrita, respectivamente, porém, dentro de seção crítica.

Cada processador possui um estado individual para cada página local. As possíveis diferenças dos estados locais de uma página entre processadores distintos ocorrem temporariamente e são corrigidas quando há necessidade da realização de operações de coerência.

Uma página pode ter estados diferentes em processadores distintos até que eles se comuniquem e seja computada a combinação de seus estados individuais. Num pedido de página ou num pedido de *diffs* esta combinação é realizada entre os processadores envolvidos. Junto com a mensagem de pedido de página ou *diff* é transmitido o estado individual que o processador solicitante possui. O processador que atende ao pedido computa a combinação do estado recebido com seu próprio estado, e ao responder ao pedido envia o estado resultante. Desta forma, ao final do processo, ambos processadores possuirão o mesmo estado para a página.

Numa operação de barreira, a combinação dos estados individuais é realizada pelo processador gerente da barreira. Estes estados são recebidos acoplados à mensagem que informa ao gerente da chegada de cada processador à barreira. Na operação de barreira é computado o estado global de cada página. Ao final da operação de barreira todos os processadores conhecem o mesmo estado para a página.

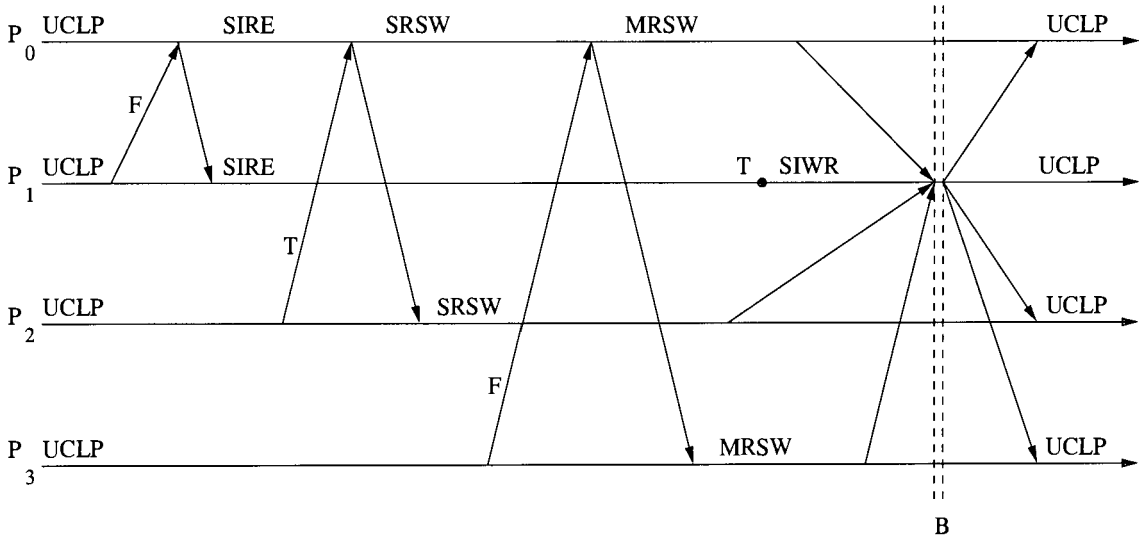

Figura 3.4: Exemplos de combinação de estados de FIESTA

A figura 3.4 ilustra as combinações dos estados para uma página com acessos fora de seção crítica, compartilhada pelos processadores P_0, P_1, P_2 e P_3. Considere o protocolo TreadMarks, sistema em que nos baseamos para realizar nossos estudos. No início da execução, P_0 possui todas as páginas e o estado inicial das páginas é UCLP[1]. Ao ocorrer uma falha de acesso de leitura em P_1, ele a requisita a P_0. Ao atender o pedido, P_0 troca seu estado para SIRE, incluindo a identifição de P_1 no conjunto dos leitores ($\{R\}$). Num outro momento, em P_2 ocorre uma falha num acesso de escrita, e a página é requisitada a P_0. Da mesma forma, ao atender o pedido de P_2, P_0 combina seu estado atual com o estado de P_2 (SRSW), incluindo-o no conjunto dos escritores ($\{W\}$). Em P_3, também ocorre uma falha de acesso de leitura, e o procedimento é idêntico ao de P_1, sendo que agora o estado da página é MRSW. No decorrer da execução, P_1 possui a página válida e ocorre uma violação de proteção de escrita, o que provoca a transição do seu estado de SIRE para SIWR. Na barreira, ao receber as mensagens dos processadores, nas quais vieram acoplados os

[1]O fato de P_0 ter as páginas liberadas para leitura no início da execução é um detalhe de implementação de TreadMarks. Este detalhe faz com que P_0 não seja incluído no conjunto dos leitores até que a página seja invalidada. Se não houver invalidação, então não há necessidade de operações de coerência, porque a página não foi modificada, os acessos terão sido apenas de leitura.

estados individuais das páginas, o gerente (P_1), computa o estado global da página (UCLP) e o envia aos outros processadores juntamente com as mensagens de término da barreira. Dessa forma, todos os processadores conhecerão o mesmo estado para a página.

A implementação de FIESTA não necessita de mensagens extras para a captação das informações, porém acrescenta às mensagens trocadas na barreira ou no pedido de página ou *diffs* uma quantidade relativamente pequena de informações (4 *bytes* por página) sobre o estado de compartilhamento das páginas que cada processador possui.

Além disso, ela pode ser implementada sobre outros protocolos *DSM* que gerem os mesmos eventos para controlar a coerência, independente da unidade de coerência.

Capítulo 4

Categorização dos Padrões de Escritas

Iniciamos este capítulo descrevendo o algoritmo para categorizar cada uma das página da aplicação, baseado na seqüência de estados gerados por FIESTA. Esse algoritmo é denomindado RITMO (*algoRIThM fOr FIESTA*). Em seguida, mostramos através de exemplos, as seqüências que geram as categorizações.

4.1 Categorização

A partir das informações geradas dinamicamente por FIESTA, categorizamos o padrão de escrita das páginas em três grupos distintos que denominamos SW, MW e MIGR. Uma página é categorizada como SW se possuir um único escritor. As páginas categorizadas como MIGR, possuem mais de um escritor, porém somente um deles acessa a página de cada vez. Já as páginas categorizadas como MW possuem vários escritores. Nas três categorizações podem ou não existir leitores[1].

A categorização do padrão de escrita de cada página é realizada monitorando a seqüência de seu estado de compartilhamento. Para páginas com predominância de acessos fora de seção crítica, gravamos a seqüência de estados globais na barreira até encontrarmos os três primeiros estados globais onde ocorreram escritas úteis. Isto é, os estados com escritas úteis são aqueles que ocorrem depois que descartamos o primeiro estado global com escrita da fase[2] inicial de acesso à página,

[1] Como o sistema que escolhemos para nossos experimentos é o sistema TreadMarks, que possui suporte a múltiplos escritores, as páginas com acesso somente de leitura têm sua categorização mantida com o valor *default* inicial do sistema que é MW. Este fato não traz inconsistências porque estas páginas não requerem operações de coerência. Logo, não desperdiçamos oportunidades de melhorar o desempenho.

[2] Definimos uma fase como o intervalo entre duas barreiras.

pois freqüentemente este primeiro estado global não é representativo. Após atingir esta condição, analisamos a seqüência dos estados de compartilhamento e a categorizamos.

Experimentalmente observamos que a seqüência de estados globais gerados até encontrarmos as três escritas úteis são suficientes para demonstrar o padrão de compartilhamento de cada página com predominância de acessos fora de seção crítica, para a grande maioria das aplicações do nosso conjunto de testes. Para páginas com predominância de acessos dentro de seção crítica, gravamos a seqüência de estados parciais na liberação das variáveis de *lock* durante os primeiros oito acessos à página. A escolha deste valor, também foi definida experimentalmente, para dar a oportunidade para todos os oito processadores do sistema realizarem o acesso a página pelo menos uma vez. Da mesma forma que para acessos fora de seção crítica, após atingir essa condição, analisamos a seqüência dos estados de compartilhamento e categorizamos as páginas. Para páginas onde existe tanto acessos dentro quanto fora de seção crítica, analisamos e categorizamos o padrão de compartilhamento de acordo com a primeira condição atendida: três estados globais onde ocorreram escritas ou oito acessos dentro de seção crítica.

Os valores de três estados globais ou oito acessos dentro de seção crítica são independentes das aplicações. Esses valores foram escolhidos após observarmos o comportamento de 17 aplicações diferentes, incluindo as aplicações do nosso conjunto de testes.

Na figura 4.1 destacamos a diferença entre RITMO e os algoritmos de categorização existentes na literatura. Como identificamos precisamente o estado de compartilhamento INOUT através de FIESTA, podemos categorizar o padrão de escrita de uma página em SW, MIGR ou MW. As setas indicam estas possibilidades de categorização. Além disso, páginas acessadas com mais de uma variável de *lock* podem ser categorizadas como MIGR.

4.1.1 Exemplos de Categorização de Páginas

Como visto no capítulo 3, o que caracteriza o estado de compartilhamento são os tipos de acesso realizados na página e o conjunto de processadores que realizam os acessos. Essas informações, armazenadas nos conjuntos R, W, P e C, são analisadas e definem a categorização do padrão de escrita de cada página. Exemplificamos, a seguir, algumas seqüências de estados de compartilhamento que geram as cate-

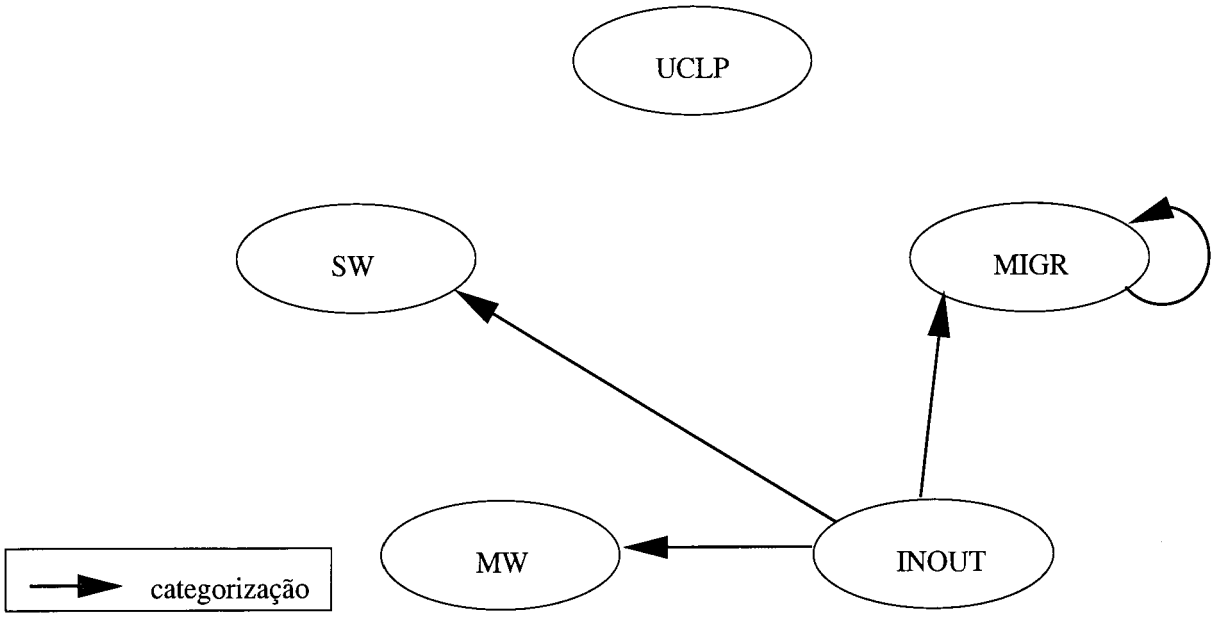

Figura 4.1: Categorização de estados de compartilhamento INOUT

gorizações. Nesses exemplos, mostramos desde as seqüências de estados mais simples que geram categorizações identificadas também por outras estratégias, até as seqüências de estados que geram categorizações identificadas somente por RITMO.

Na figura 4.2 são mostrados três exemplos (a, b e c) de seqüências que permitem identificar a categorização para casos simples. As seqüências (a) e (b) pertencem a duas páginas distintas da aplicação regular BarnesTmk, e demonstram os padrões SW e MW. Essas páginas têm acessos somente fora de seção crítica. A seqüência (c) pertence à aplicação TSP e tem acessos somente dentro de seção crítica.

A página (a) foi monitorada durante quatro fases consecutivas até ter a condição de três estados globais onde ocorrem escritas atendida. Esta página é categorizada como SW porque, observando a seqüência de acessos, verifica-se que só um processador escreve durante uma fase, enquanto outros processadores realizam acessos de leitura.

No exemplo (b), a página foi monitorada durante cinco fases e categorizada como MW. Note que na fase entre as barreiras 2 e 3 não ocorrem acessos de escrita. Nas demais fases, os acessos de escrita são realizados por mais de um processador, enquanto que, os acessos de leitura são realizados por apenas um processador.

A página do exemplo (c) foi monitorada entre duas fases na qual são registrados os oito acessos dentro de seção crítica. Essa página foi categorizada como MIGR.

Páginas com acessos irregulares, ou seja, com acessos dentro e fora de seção crítica

Categorização				
Fora de seção crítica			Dentro de seção crítica	
Barreira	SW	MW	Acessos à lock	MIGR
1	SIWR	SRMW	1	SIPR
2	SIWR	SRMW	2	SPSC
3	SIWR	SIRE	3	MIGR
4	MRSW	SRMW	4	MIGR
5		SRMW	5	MIGR
			6	MIGR
			7	MIGR
			8	MIGR

(a) Página 416 (b) Página 1 (c) Página 3

Figura 4.2: Exemplos de seqüências de estados de compartilhamento que geram as categorizações SW, MW e MIGR

ou com acessos a mais de um *lock* podem gerar diferentes categorizações dependendo da seqüência de estados. A seguir mostramos três exemplos de seqüências de estados para páginas com essas características e que geram categorizações diferentes, SW, MW e MIGR.

A análise que gera a categorização das páginas desses exemplos se baseia nos estados parciais gravados na saída das seções críticas, pois a primeira condição a ser satisfeita, para todas elas, são os oito acessos dentro de seção crítica. A seqüência de estados mostrada reflete a visão que o processador que está liberando o *lock* tem da página.

A figura 4.3 mostra uma seqüência de estados de compartilhamento que gera a categorização SW. Esta seqüência pertence a uma página da aplicação irregular TSP. No primeiro estado gravado só o processador 5 (P_5) escreve na página, como pode ser observado pelo conteúdo do conjunto P. No segundo estado é o processador P_7 que escreve na página. Entre estes dois primeiros estados gravados e os restantes, ocorre um evento de barreira e acessos de leitura fora de seção crítica pelos processadores 5 e 7. No terceiro estado gravado, a página está no estado INOUT e é escrita por P_3. Nos estados seguintes a página é escrita pelos processadores 6, 2, 1, 0 e 4, nesta ordem. Como nenhum processador escreve na página fora de seção crítica, e todas as escritas só ocorrem de modo exclusivo, então essa página é categorizada como

Categorização SW - Página 1

Acessos à lock		{R}	{W}	{P}	{C}
1	SIPR	{ }	{ }	{5}	{ }
2	MIGR	{ }	{ }	{5, 7}	{ }
3	INOUT	{5, 7}	{ }	{3}	{3}
4	INOUT	{3, 5, 7}	{ }	{3, 6}	{6}
5	INOUT	{3, 5, 6, 7}	{ }	{2, 3, 6}	{2}
6	INOUT	{2, 3, 5, 6, 7}	{ }	{1, 2, 3, 6}	{1}
7	INOUT	{1, 2, 3, 5, 6, 7}	{ }	{0, 1, 2, 3, 6}	{0}
8	INOUT	{0, 1, 2, 3, 5, 6, 7}	{ }	{0, 1, 2, 3, 4, 6}	{4}

Figura 4.3: Exemplo de seqüência de estados de compartilhamento para uma página INOUT categorizada como SW

SW. Note que essa página é acessada sob três variáveis de *lock* diferentes.

A figura 4.4 mostra uma seqüência de estados de compartilhamento que gera a categorização MW. Esta seqüência pertence a uma página da aplicação irregular Barnes2. Nos três primeiros acessos só o processador 0 (P_0) escreve na página, como pode ser observado pelo conteúdo do conjunto P. A partir do quarto acesso, a página é escrita tanto dentro quanto fora de seção crítica pelos processadores P_0, P_1 e P_5 (conjunto W e P). No sexto acesso, é registrada uma falha de acesso de leitura fora de seção crítica realizada por P_7. Como três processadores distintos escrevem na página tanto dentro quanto fora de seção crítica, ela é categorizada como MW.

Categorização MW - Página 336

Acessos à lock		{R}	{W}	{P}	{C}
1	SIPR	{ }	{ }	{0}	{ }
2	SIPR	{ }	{ }	{0}	{ }
3	SIPR	{ }	{ }	{0}	{ }
4	INOUT	{ }	{1}	{0}	{ }
5	INOUT	{0}	{1}	{0}	{ }
6	INOUT	{1, 7}	{0,1}	{0}	{ }
7	INOUT	{ }	{1}	{0}	{ }
8	INOUT	{0}	{1, 5}	{0}	{ }

Figura 4.4: Exemplo de seqüência de estados de compartilhamento para uma página INOUT categorizada como MW

A figura 4.5 mostra uma outra seqüência de estados de compartilhamento para

uma página com acessos dentro e fora de seção crítica, porém, para esta página a categorização é MIGR. Uma seqüência de eventos semelhante a apresentada nesse exemplo ocorre na aplicação irregular Water-Nsquared.

Categorização MIGR - Página 9

Acessos à lock		{R}	{W}	{P}	{C}
1	INOUT	{2, 5, 6, 7}	{1}	{1}	{1}
2	INOUT	{2, 5, 6, 7}	{1}	{1}	{1}
3	INOUT	{2, 5, 6, 7}	{1}	{1}	{1}
4	INOUT	{2, 5, 6, 7}	{1}	{1}	{1}
5	INOUT	{2, 5, 6, 7}	{1}	{1}	{1}
6	INOUT	{2, 5, 6, 7}	{1}	{1}	{1}
7	INOUT	{5, 6, 7}	{1}	{1, 2}	{2}
8	INOUT	{5, 6, 7}	{1}	{1, 2}	{2}

Figura 4.5: Exemplo de seqüência de estados de compartilhamento para uma página INOUT categorizada como MIGR

Os estados gravados para análise e categorização desta página foram captados após o primeiro evento de barreira e acessos de leitura e escrita fora de seção crítica. Desde o primeiro estado gravado, a página já está no estado INOUT. O processador P_1, que conhece que outros processadores fizeram acessos fora de seção crítica, escreve na página dentro de seção crítica. Este mesmo processador escreve na página sob variáveis de *lock* diferentes até o sexto acesso. Nos dois estados seguintes que foram gravados, é o processador P_2 que passa a escrever na página. Como o único processador dentro do conjunto W também pertence ao conjunto P, esta página é categorizada como MIGR.

Para cada grupo distinto de páginas é possível empregar técnicas de adaptação diferentes. Por exemplo, nas páginas categorizadas como SW ou MIGR podemos enviar as páginas em avanço (*forwarding*) quando for conveniente e, para páginas categorizadas como MW podemos pedir pelos *diffs* antecipadamente (*prefetching*). As técnicas utilizadas para demonstrar como as informações coletadas dinamicamente por FIESTA e tratadas por RITMO permitem melhorar o desempenho de aplicações quando submetidas a sistemas *software DSM* são descritas no próximo capítulo.

4.1.2 Comparação com Outras Estratégias

Seqüências simples, como as exemplificadas, também podem ser identificadas por outras estratégias de categorização do padrão de escrita de páginas como, por exemplo, as utilizadas nos trabalhos de Amza[7] e Monnerat[64].

O trabalho de Amza identifica padrões de escrita SW ou MW através de mensagens de posse e de notificações de escrita com *version number*. Uma das diferenças principais é que não precisamos de mensagens extras para a identificação dos padrões de escrita. Além disso, páginas com acessos a seções críticas realizados com variáveis de *lock* diferentes são consideradas MW, enquanto RITMO pode categorizá-las como SW, MW ou MIGR.

Existem duas diferenças entre RITMO e o algoritmo de categorização implementado por Monnerat[64]. A primeira diferença é que podemos categorizar uma página onde ocorre uma seqüência de estados de compartilhamento com acessos dentro e fora de seção crítica como SW, MW ou MIGR, enquanto o algoritmo de Monnerat assume que estas páginas são MW. A outra diferença está nas páginas com seqüências de estados de compartilhamento com acessos a seções críticas realizados com variáveis de *lock* diferentes, onde RITMO pode categorizá-las como SW ou MIGR e algoritmo de Monnerat as mantém como MW.

Capítulo 5

Aplicação de FIESTA a Protocolos *Software DSM*

Neste capítulo mostramos como, a partir das informações fornecidas por FIESTA e RITMO, podem ser realizadas ações adaptativas que atuam no protocolo TreadMarks de modo a adequá-lo ao padrão de compartilhamento das aplicações. Discutimos as técnicas implementadas, baseadas no estado de compartilhamento, para tolerar a latência de acesso aos dados remotos, e dentre as existentes escolhemos *single writer/ multiple writers*, *forwarding* e *prefetching*.

5.1 Técnicas de Tolerância à Latência

Realizamos nosso estudo sobre técnicas de tolerância à latência tendo como referência o protocolo *software DSM* TreadMarks e, a menos que explicitamente citado, nas sincronizações de barreira ou com *locks* são executadas as mesmas operações. A diferença está no tratamento das falhas de acesso. As técnicas de tolerância à latência utilizadas neste trabalho são *single writer/ multiple writers* (SW-MW), *forwarding* e *prefetching*. Elas são implementadas de modo transparente ao usuário, são baseadas nas informações fornecidas pelos estados de compartilhamento captados dinamicamente por FIESTA e na estratégia de categorização das seqüências desses estados, e estão detalhadas a seguir.

5.1.1 Adaptação SW-MW

Implementamos a adaptação SW-MW para diminuir o *overhead* do protocolo relativo aos mecanismos de *twining* e *diffing* utilizados para manter a coerência das páginas compartilhadas. Numa falha de página, seja de leitura ou de escrita, além

das operações normais do Treadmarks, é verificada a categorização da página e o atual estado de compartilhamento. O processador requisita a página se as seguintes condições, denominadas condições **P**, são obedecidas:

P1. sua categorização é SW ou MIGR;

P2. só há um processador no conjunto de processadores dominantes;

P3. este processador é coincidente com o processador do estado de compartilhamento presente em W ou P, e,

P4. se a falha ocorreu dentro de seção crítica, é a primeira vez que faz um acesso a uma variável de *lock* ou a variável de *lock* atual é igual a do último acesso.

Caso contrário, o protocolo não sofre nenhuma interferência. As páginas MW são tratadas segundo os mecanismos de *twinning* e *diffing*. Numa mensagem de pedido por página é informado ao processador que vai atender ao pedido se a falha ocorreu num acesso de leitura ou de escrita.

Ao fornecer a página, o processador que atende ao pedido protege a página contra escrita para que sejam geradas novas notificações de escrita se ele voltar a escrever nos dados compartilhados. Além disso, verifica se a falha que gerou o pedido é de escrita. Neste caso, o processador cria um *diff* para que, quando necessário, possa ser formada uma nova versão da página, já que a partir deste momento existem dois processadores escrevendo na página.

Considere que temos, novamente, a seqüência de eventos da figura 4.5 que exemplifica o comportamento de uma página típica da aplicação irregular Water-Nsquared. Após o processador P_0 pedir a posse do *lock a* e invalidar a página, ao invés de requisitar o *diff* a P_1, ele requisita a página. Neste caso, P_0 não tem os *overheads* de espera pela criação e aplicação do *diff*. Porém, P_1 tem que criar o *diff* após enviar a página, pois se outro processador requisitar seus *diffs*, P_1 deve ser capaz de fornecê-lo. No momento em que P_0 sincroniza novamente com P_1 no *lock b*, a página não é invalidada e P_0 pode escrever livremente.

Esta técnica não aumenta o número de mensagens em relação a versão de Tread-Marks original. Na falha de página ou é transferida a página ou os *diffs*. Pode haver um aumento do tamanho médio das mensagens transferidas. Uma desvantagem com esta adaptação pode surgir se somente poucos dados forem escritos dentro da página, pois informações desnecessárias são transferidas (página inteira ao invés de

um *diff* de tamanho muito menor). Neste caso, para evitar o efeito da fragmentação poderíamos adicionar mais uma condição a esta adaptação, considerando o tamanho dos *diffs* gerados durante a categorização e que poderia também ser usado na escolha de pedido de página ou *diff*. Considerando o custo do tratamento dos *diffs*, seria necessário definir um valor limite que determinaria a transferência de página ou *diff*. Esta adaptação sugerida seria uma combinação da técnica SW-MW condicionada à granularidade de acesso.

5.1.2 Técnica de *Forwarding* (FWD)

A técnica de *forwarding* é uma outra técnica utilizada para tolerar os *overheads* relativos aos mecanismos de *twining* e *diffing*. Ela habilita o protocolo a enviar ou não páginas antecipadamente. O envio antecipado das páginas pode ocorrer tanto nas sincronizações de barreira quanto nas sincronizações por *locks*. A diferença entre *forwarding* e a técnica SW-MW está no momento do envio das páginas. No modo SW, ao ocorrer uma falha de acesso no processador leitor, ele requisita a página ao escritor. Já na técnica *forwarding* as páginas são enviadas antecipadamente pelo escritor a um conjunto de processadores, nas operações de sincronização. A diferença entre o protocolo de atualização (capítulo 2) e a técnica de *forwarding* é o envio das páginas de modo seletivo, nesta última. No protocolo de atualização, a página é enviada a todos os processadores que a compartilham. No protocolo com *forwarding* são enviadas as páginas, com antecedência, a um conjunto de processadores com os quais sincroniza, que têm as páginas invalidadas e que se prevê que sofrerão falha de acesso às páginas.

A nossa implementação da técnica de *forwarding* só considera as páginas categorizadas como SW ou MIGR, e pode ser ativada tanto nas sincronizações de barreira quanto nas sincronizações por *locks* e obedece às mesmas condições **P** estabelecidas para a técnica SW-MW. As páginas MW são tratadas segundo os mecanismos de *twinning* e *diffing*. O envio antecipado da página é realizado de modo seletivo considerando um conjunto de previsão. O **conjunto de previsão** é obtido através de uma heurística baseada no estado global da página e no conjunto de processadores das seqüências de estados de compartilhamento utilizadas para a sua categorização. O gerente da barreira, após computar o estado global da página, realiza a união com os conjuntos R, W, P e C, das seqüências de estados de compartilhamento. Os processadores presentes neste conjunto são candidatos a receber a página. Com

esse procedimento estamos incluindo os processadores que passaram a compartilhar a página após ter sua categorização inferida. O processador escritor envia as páginas para os processadores que têm a página invalidada. Note que antes de existir uma categorização de escrita para a página, o conjunto de previsão é exatamente igual a união dos conjuntos R, W, P e C do estado global.

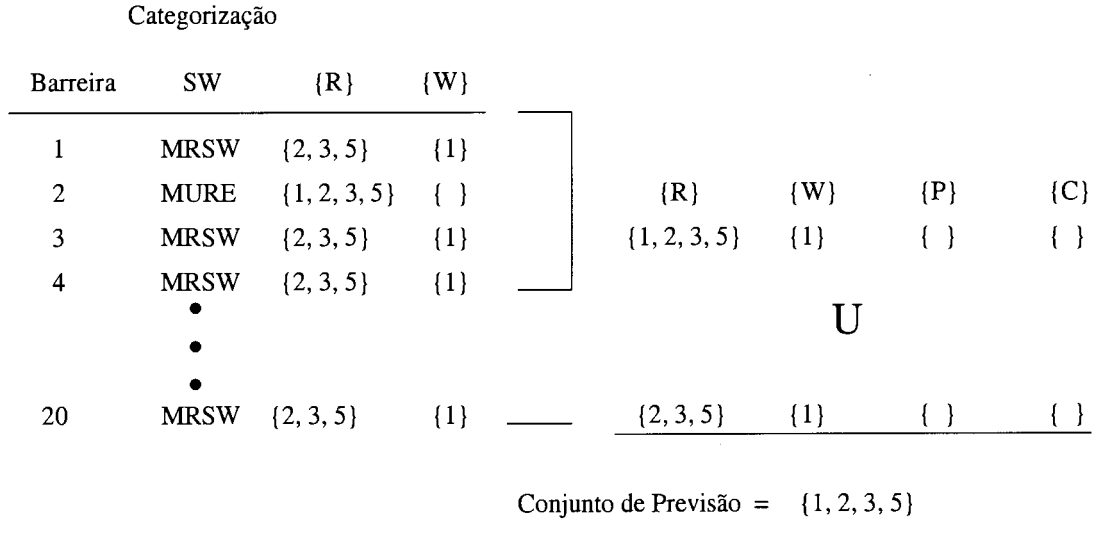

Figura 5.1: Exemplo de conjunto de previsão

A figura 5.1 ilustra como é realizada a união entre os processadores das seqüências de estados de compartilhamento com o estado global de uma página para gerar o conjunto de previsão. Considere como exemplo a seqüência de estados de compartilhamento ilustrada nessa figura. Nela agora estão discrimidados os processadores dos conjuntos R e W. Os conjuntos P e C estão vazios. O conjunto de previsão computado na barreira 20 é a união desses conjuntos com os que representam o estado global da página.

Na saída da barreira, cada processador, para todas as páginas que compartilha, verifica a categorização da página. Se a página é categorizada como SW ou MIGR, e o processador é o único escritor, então, se existem leitores pertencentes ao conjunto de previsão com a página inválida, estas são enviadas antecipadamente. Caso contrário, nenhuma ação é realizada. Uma outra característica dessa implementação é que, se existe mais de uma página a ser enviada para um mesmo processador, estas são agrupadas e enviadas em uma mesma mensagem.

Da mesma forma, num pedido de aquisição de *lock*, o processador *owner*verifica as páginas quando enviar a posse do *lock* ao processador *acquirer*. As páginas que ele escreveu, são enviadas antecipadamente se as condições **P** são atendidas. Caso

contrário, nenhuma ação é realizada.

Caso uma página categorizada como SW ou MIGR, sofra uma falha de página e esta não tiver sido enviada antecipadamente e suas condições **P** são satisfeitas, é enviado um pedido de página ao invés de um pedido de *diffs*. Isto é, é utilizada a técnica SW-MW.

A técnica *forwarding* pode diminuir, também, o número de falhas de página, pois ao receber a página, o processador leitor marca as notificações de escrita como já recebidas e a desprotege para leitura. Além disso, pode aumentar o número de mensagens em relação ao TreadMarks original, se a página é invalidada antes de ser utilizada, e também aumentar o tamanho médio das mensagens transferidas, já que transfere a página inteira.

Com o envio prévio das páginas, o ganho pode não ser o esperado se as páginas são invalidadas antes de serem utilizadas, ou porque não chegam a tempo no seu destino. Neste último caso, o ganho é menor do que o esperado.

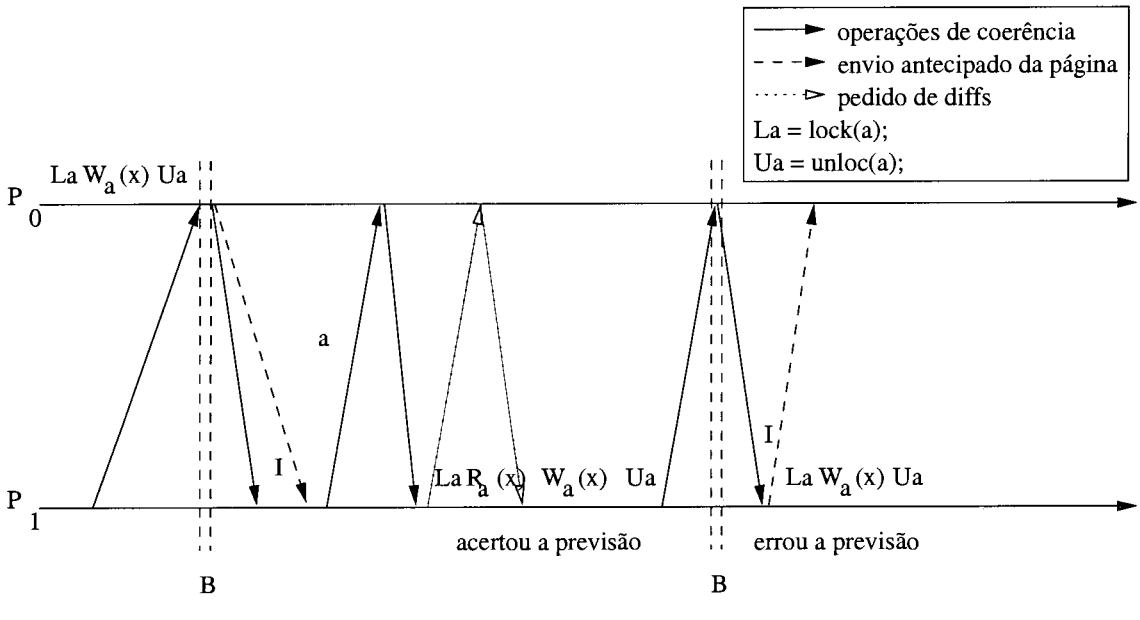

Figura 5.2: O uso da técnica de *forwarding* no protocolo de TreadMarks

A figura 5.2 exemplifica o envio antecipado de uma página na barreira num sistema com dois processadores P_0 e P_1. As setas contínuas representam operações de coerência, as setas tracejadas o envio antecipado da página e as setas pontilhadas o pedido por *diffs* se a técnica de *forwarding* não existisse. P_0 é o gerente de todas as barreiras. Na primeira fase, P_0 escreve na página. Ao chegar a barreira, verifica que as condições **P** são atendidas e envia a página antecipadamente a P_1. P_1 referencia a página e a previsão é correta, o que evita a espera pelo dado compartilhado e a falha

de página. Na barreira seguinte P_1 envia a página antecipadamente a P_0, que não a referencia. Além disso, P_1 volta a escrever na página, o que faz com que a previsão tenha sido incorreta. Note que ao acertar a previsão conseguimos tolerar o tempo de espera pelos dados compartilhados e reduzimos o número de mensagens trocadas. Quando erramos a previsão, enviamos os dados desta página desnecessariamente. A mensagem só terá sido completamente desnecessária se todas as páginas que foram enviadas em conjunto não tiverem sido invalidadas antes de terem sido utilizadas.

5.1.3 Técnica de *Prefetching* (PRF)

A técnica de *prefetching* visa minimizar o *overhead* do acesso a dados remotos, antecipando a busca destes dados. Os dados invalidados por outros processadores, são buscados antecipadamente para que no momento do acesso à pagina, se aumente a chance dos *diffs* já estarem disponíveis.

Neste trabalho, a técnica de *prefetching* é implementada somente na barreira. Nenhuma ação é realizada para falhas que ocorrem dentro de seções críticas. Na técnica de *prefetching* também utilizamos o conjunto de previsão descrito para a técnica de *forwarding*, para requisitar os dados de modo seletivo. A requisição só é disparada se as condições, denominadas condições **D**, são obedecidas:

D1. a página é categorizada como MW;

D2. a página está inválida na memória local, e,

D3. o processador está presente no conjunto de previsão.

Caso contrário, nenhuma ação é realizada, ou seja, as páginas categorizadas como SW e MIGR são tratadas segundo os mecanismos de *twinning* e *diffing*.

Os dados recebidos antecipadamente são armazenados. Quando ocorre uma falha de acesso, é verificado se todos os *diffs* estão disponíveis. Se necessário, são coletados *diffs* adicionais, que não tenham sido solicitados, ou que não tenham sido recebidos ainda. Todos os *diffs* são aplicados à versão da página desatualizada, de modo a torná-la válida. Note que ocorre a falha de acesso mesmo que todos os *diffs* necessários já estejam disponíveis localmente.

A figura 5.3 exemplifica a busca antecipada pelos dados remotos nas barreiras num sistema com dois processadores P_0 e P_1, onde ambos escrevem na página fora de seção crítica. As setas contínuas representam operações de coerência, as setas

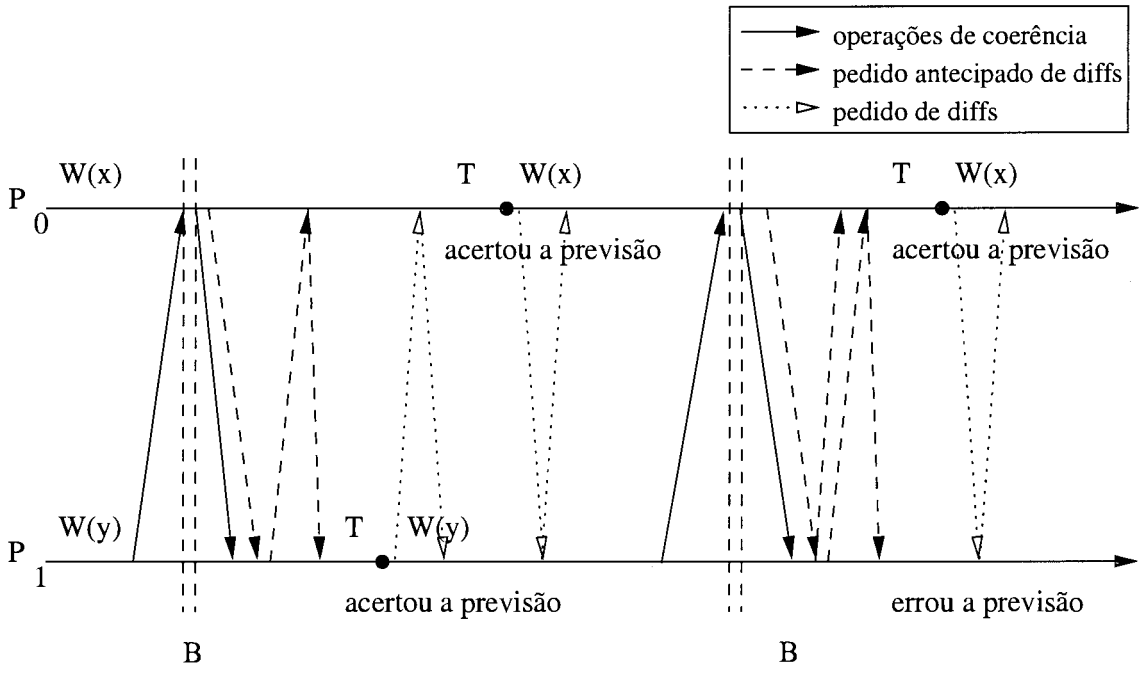

Figura 5.3: O uso da técnica de *prefetching* no protocolo de TreadMarks

tracejadas o pedido antecipado dos *diffs* e as setas pontilhadas o pedido dos *diffs* se a técnica de *prefetching* não existisse.

P_0 é o gerente de todas as barreiras. Na primeira fase, P_0 e P_1 escrevem na página. Ao chegar à barreira, eles verificam se as condições **D** são atendidas e pedem antecipadamente o *diff*, isto é, P_0 e P_1 trocam seus dados modificados localmente. Ambos voltam a escrever na página e a previsão é correta, o que evita a espera pelo dado compartilhado. Na barreira seguinte, as condições **D** são atendidas e ocorre nova troca de dados. Porém, nesta fase, somente P_0 referencia a página. Então a previsão é correta em P_0 e errada em P_1.

Nossa forma de implementação da técnica de *prefetching* não diminui o número de falhas de página. Além disso, pode aumentar o número de mensagens em relação à versão de TreadMarks original. Com a busca antecipada dos *diffs*, o ganho pode não ser o esperado se as páginas são invalidadas antes de serem utilizadas, ou então porque os *diffs* não chegam a tempo no seu destino.

Capítulo 6

Metodologia Experimental

Para atingir um bom desempenho em sistemas *software DSM* é necessário conhecer o potencial de paralelismo de cada aplicação, suas características de comunicação e de compartilhamento de dados.

Neste capítulo apresentamos o ambiente de simulação e as aplicações utilizadas em nossos estudos, cujos resultados serão analisados qunado da aplicação das diversas técnicas implementadas nos próximos capítulos. O conjunto de aplicações utilizado é composto por aplicações ou *kernels* (núcleos) representativos das áreas científica e de engenharia. Essas aplicações foram desenvolvidas com o intuito de auxiliar a análise e a avaliação de desempenho de sistemas paralelos com espaço de endereço compartilhado.

Para caracterizar cada aplicação ou núcleo descrevemos o problema resolvido, suas principais estruturas de dados e como são distribuídas através dos processadores, as operações de sincronização utilizadas nas implementações e o tamanho do problema nas execuções. Além disso, essas aplicações possuem características diferentes em relação à quantidade de dados modificados entre operações de sincronização, e, para cada uma delas, discutimos os possíveis *overheads* gerados, quando os programas são executados em sistemas *software DSM*. Todas as aplicações utilizadas foram escritas em linguagem *C*.

6.1 Ambiente de Simulação

O objetivo dessa tese não é implementar um novo sistema *software DSM*, mas propor e avaliar o potencial de novas técnicas para redução de *overheads* em sistemas *software DSM*. Por isso, optamos por utilizar um sistema simulado para avaliar o potencial de FIESTA, de RITMO e de novas estratégias de implementação das técnicas

de tolerância à latência.

Com a opção de utilizar um sistema simulado evitamos a complexidade inerente de um sistema paralelo real, relativa a questões de implementação ou de depuração. Na implementação é necessário, por exemplo, considerar a otimização do armazenamento de dados na memória. A depuração pode se tornar extremamente laboriosa devido às cargas do sistema que podem gerar temporizações diferentes a cada execução levando a resultados diferentes. Este fato dificulta a correção de erros.

Dessa forma, a opção pelo uso de um sistema simulado nos permite concentrar nas avaliações qualitativas das nossas propostas sem a complexidade inerente ao sistema real. Através de simulação podemos ter maior controle sobre os recursos do sistema e obter mais detalhadamente informações sobre o seu comportamento. Por exemplo, num simulador podemos reproduzir execuções com maior controle do seu estado de execução. Além disso, podemos contabilizar as falhas na *cache* ou na TLB, que têm influência direta no desempenho do sistema, e que são muito difíceis de serem contabilizadas num sistema real. Porém, sistemas simulados têm suas limitações, como, por exemplo, utilizar quantidade de dados de entrada muito grande, freqüentemente necessária para avaliar sistema paralelos, que produz simulações muito longas, ou uma simplificação do sistema real que possa ocultar situações dependentes de temporização.

A escolha do simulador empregado nesta tese se baseou na imediata disponibilidade de um simulador de TreadMarks amplamente testado e utilizado como base em várias outras pesquisas[17, 74, 14].

Esse simulador consiste de duas partes, um *front end* e um *back end*. O *front end*, Mint[83], simula a execução de processadores e invoca o *back end* em todas as referências aos dados. Consideramos que a busca de instruções não causa falhas na memória *cache*. O *back end* simula o sistema de memória de forma bastante detalhada. Ele considera *write buffers* e memórias *cache* com tamanho finito, comportamento de *Table Lookaside Buffer* (TLB), emulação completa do protocolo, custos relativos a transferências na rede de interconexão e de acesso à memória, ambos incluindo os efeitos de contenção.

Simulamos uma rede de estações de trabalho com 8 nós. Cada nó consiste de um processador, um *write buffer*, uma *cache* de dados de primeiro nível mapeada diretamente, um módulo de memória local, e, um roteador de rede em malha, utilizando o roteamento *wormhole*[67]. Consideramos que todas as instruções são executadas

em um ciclo.

A tabela 6.1 mostra os parâmetros utilizados em nossas simulações. Consideramos ainda que todos os tempos são dados em ciclos de processador de 5 ns. Os parâmetros escolhidos para as nossas simulações correspondem a valores compatíveis com processadores e redes de interconexão atuais.

Nome das Constantes do Sistema	Valores
Número de processadores	8
Tamanho da TLB	128 entradas
Tempo de serviço para completar a TLB	100 ciclos
Todas as interrupções	4000 ciclos
Tamanho da página	4K bytes
Cache total por processador	256K bytes
Tamanho do *buffer* de escrita	4 entradas
Tamanho da linha da *cache*	32 bytes
Tempo de ativação da memória	18 ciclos
Tempo de acesso à memória (após ativação)	1.25 ciclos/palavra
Tempo de ativação do PCI	18 ciclos
Tempo de acesso PCI *burst* (após ativação)	3 ciclos/palavra
Largura da rede	16bits (bidirecional)
Overhead de mensagens	400 ciclos
Latência de *switch*	4 ciclos
Latência de *wire*	2 ciclos
Processamento de listas	6 ciclos por elemento
Geração de *twin*	5 ciclos/palavra + acessos à memória
Criação e aplicação de *diff*	7 ciclos/palavra + acessos à memória

Tabela 6.1: Valores dos parâmetros do sistema simulado. 1 ciclo = 5 ns.

6.2 O Conjunto de Aplicações

Para demonstrar o potencial de FIESTA e das técnicas de adaptação e tolerância à latência utilizamos oito aplicações com características distintas em nossos experimentos. As aplicações Barnes2, Ocean e Water-Nsquared pertencem ao SPLASH-2 *suite*[84]. Fazem parte do pacote de distribuição de TreadMarks, BarnesTmk e TSP. A aplicação Em3d foi desenvolvida na Universidade de Berkeley[27] e SOR foi desenvolvida localmente. Nas referências citadas podem ser encontrados mais detalhes sobre esses programas.

6.2.1 Barnes2

Barnes2 pertence ao SPLASH-2 *suite*[84] e simula a interação entre um sistema de corpos em três dimensões sob a influência de forças gravitacionais, por um determinado número de passos, usando o método de N-corpos hierárquico denominado *Barnes-Hut*. Barnes2 representa seu domínio computacional como uma *octree* com folhas contendo informações de cada corpo, e nós internos representando células do espaço. Não é feita nenhuma distribuição especial dos dados dos corpos na memória, porque é difícil com a unidade de compartilhamento do tamanho de uma página. Grande parte do tempo de execução é dispendido em travessias parciais na *octree* (uma por corpo) para computar as forças dos corpos individuais. Os padrões de comunicação são dependentes da distribuição das partículas, que é não-estruturada. A sincronização entre os processos é realizada através do uso de barreiras e de *locks*. Nesta aplicação podemos identificar três fases principais: a fase de construção da árvore, na qual todos os processos armazenam seus corpos nas células, utilizando operações de *locks* na sincronização; a fase que realiza as travessias parciais na *octree* para computar as forças dos corpos individuais; a fase onde os processos atualizam informações sobre as células e os corpos de outros processos, e que ao final computam as novas posições dos corpos. A granularidade dos acessos, de leitura e de escrita, aos corpos e às células são finas. Assim, encontramos em Barnes2 fragmentação e falso compartilhamento. Em nossos experimentos simulamos um sistema com 16K corpos.

6.2.2 BarnesTmk

A aplicação BarnesTmk pertence ao pacote de TreadMarks, e implementa o mesmo problema de simulação da interação de um sistema de corpos, apresentado em Barnes2. A diferença entre essas implementações é a forma de construção da árvore que contém as informações sobre cada corpo. Em BarnesTmk a árvore é construída de modo seqüencial por um único processador. A outra diferença é a utilização principalmente de operações de sincronização de barreira. O único *lock* é utilizado para identificação dos processos no simulador. Esta forma de implementação é otimizada para TreadMarks. Da mesma forma, a granularidade dos acessos de leitura e escrita é fina, o que provoca tanto fragmentação quanto falso compartilhamento, e simulamos um sistema com 16K corpos.

6.2.3 Em3d

A aplicação Em3d simula a propagação de ondas eletromagnéticas em objetos tridimensionais. O problema é formulado como uma computação em um grafo bipartido com arcos direcionados dos nós E, que representam os campos elétricos, para os nós H que representam os campos magnéticos, e vice-versa. A computação modela as trocas nos campos sobre o tempo.

Em3d consiste de uma fase de inicial e uma fase de computação principal. O grafo é construído na fase inicial. Cada processador armazena um conjunto de nós E e de nós H. Os arcos são gerados aleatoriamente com uma fração de travessia dos limites do processador especificada pelo usuário. Na fase principal são computadas as trocas entre E e H. A cada iteração do *loop* principal, cada processador atualiza o potencial eletromagnético de seus nós baseado no potencial dos nós vizinhos. Na atualização dos nós elétricos só são considerados os nós magnéticos. Da mesma forma, na atualização dos nós magnéticos só são considerados os nós elétricos. As operações de sincronização são realizadas somente com barreiras. A granularidade dos acessos de leitura é fina em Em3d, gerando então fragmentação. São simulados 40064 objetos elétricos e magnéticos conectados aleatoriamente a 24 outros objetos com 10% de probabilidade que os objetos vizinhos estejam residentes em diferentes nós. As interações entre os objetos são simuladas por 40 iterações.

6.2.4 FFT

FFT é um núcleo que implementa uma versão complexa 1-D do algoritmo *radix-$\sqrt{n}$ six step FFT*[84]. O conjunto de dados consiste de n pontos de dados complexos que são transformados, e outros n pontos de dados complexos referenciados como *root of unity*. Ambos os conjuntos são organizados como matrizes particionadas $\sqrt{n} \times \sqrt{n}$. Dessa forma, a cada processador é atribuído um conjunto contíguo de linhas que são armazenadas em suas memórias locais. A comunicação ocorre nos três passos de transposição da matriz, e envolve todos os processadores (*all-to-all communication*). Cada processador transpõe uma submatriz contígua de $\sqrt{n/p} \times \sqrt{n/p}$ de cada um dos outros processadores e transpõe uma submatriz localmente. A sincronização entre os processos é realizada através do uso de barreiras e de um *lock* para a identificação dos processos. O padrão de compartilhamento varia com os valores de n e p. Para valores típicos dos parâmetros de entrada, FFT apresenta granularidade de acesso de leitura média, o que causa fragmentação. São simulados dois conjuntos de dados

de 1M pontos, um para ser transformado e o outro como *root of unity*. Cada um destes grupos é organizado como uma matriz de 256 × 256.

6.2.5 Ocean

A aplicação Ocean estuda o movimento das correntes oceânicas. Em cada passo, é ativado e solucionado um conjunto de equações diferenciais parciais. As funções contínuas são transformadas em discretas, e as equações diferença resultantes ativadas e solucionadas. Os dados são particionados em grades quadradas, conceitualmente representadas como *arrays* 4-D, com todas as grades armazenadas contiguamente e localmente nos nós. Ocean usa um método iterativo (*solver multigrade red-black Gauss-Seidel* para solucionar as equações. A sincronização entre os processos é realizada através do uso de barreiras e de *locks*. As atualizações são realizadas localmente nas partições armazenadas contiguamente. A comunicação é gerada nesta aplicação, devido aos acessos remotos de leitura feitos aos dados localizados nas bordas das partições. A granularidade desses acessos de leitura é fina e gera fragmentação. Utilizamos como entrada uma grade 258 × 258.

6.2.6 SOR

SOR é um núcleo que computa equações diferenciais parciais usando um método iterativo baseado em relaxações sucessivas. As relaxações sucessivas são realizadas utilizando uma estratégia que considera as linhas pares e ímpares da matriz de entrada. Cada iteração é dividida em duas fases separadas por barreira.

Na primeira fase, são calculados os valores das linhas pares da matriz como sendo a média das linhas ímpares da matriz. Na segunda fase, da mesma forma que para as linhas pares, são calculados os valores das linhas ímpares da matriz a partir dos valores das linhas pares da matriz. Como a matriz é dividida em fatias (bloco contínuo de linhas) entre os processadores, o compartilhamento de dados ocorre nas bordas de cada fatia. As operações de sincronização são realizadas somente com barreiras. Em SOR, a granularidade dos acessos tanto de leitura quanto de escrita é grossa, portanto não há fragmentação nem falso compartilhamento a menos das bordas. A matriz de entrada tem dimensão 256 × 640.

6.2.7 TSP

Traveling Salesman Problem é uma aplicação que resolve o problema clássico do

caixeiro viajante usando um algoritmo do tipo *branch and bound*, com estratégias de filas de tarefas para balanceamento de carga. O algoritmo assume um mapa totalmente conectado de n cidades, com cada caminho entre duas cidades tendo um peso. O problema consiste em determinar o menor caminho que, saindo de um determinada cidade, percorre todas as outras cidades exatamente uma vez e retorna à cidade original. A sincronização entre os processos é realizada através do uso de barreiras e de *locks*. TSP tem granularidade de acesso fina, o que causa fragmentação. Simulamos um problema com 18 cidades.

6.2.8 Water Nsquared

Esta aplicação avalia forças e potenciais que ocorrem num sistema de moléculas de água em estado líquido. As forças e potenciais são computados usando um método de força bruta da $O(n^2)$. As moléculas de água são armazenadas contiguamente em um vetor de n moléculas, e particionadas entre os processadores em pedaços contíguos de n/p moléculas. Cada iteração é composta por várias fases.

Na fase inicial, o processador 0 inicializa algumas somas globais. Na fase seguinte, denominada fase intramolécula, cada processador computa os valores de deslocamento para o seu conjunto de moléculas. A terceira fase, denominada fase intermolécula, utiliza os valores de deslocamento gerados na fase anterior para computar as forças entre as moléculas. Nesta fase, um processador atualiza suas n/p moléculas e as $(n/2-n/p)$ móleculas seguintes armazenadas em outros processadores. As atualizações das forças das moléculas são realizadas dentro de seção crítica, usando um *lock* por molécula. A quarta fase utiliza os valores das forças calculados na fase intermoléculas para calcular os valores corretos para as moléculas, condições de fronteira e a energia cinética do sistema. Nessa fase, cada processador realiza computações no seu conjunto de moléculas. A sincronização entre os processos é realizada através do uso de barreiras e de *locks*. A granularidade dos acessos, de leitura e escrita, as moléculas são finas, gerando fragmentação e falso compartilhamento em Water Nsquared. Utilizamos como entrada um conjunto de 512 moléculas, e as interações entre as moléculas são simuladas por 10 iterações.

6.3 Diversidade das Aplicações

As aplicações utilizadas nesse estudo diferem bastante no tipo e na freqüência das operações de sincronização. A tabela 6.2 destaca o tamanho dos problemas simulados para cada aplicação, além da utilização das operações de sincronização, barreira (BAR) e *lock* (L), granularidade de acesso e a presença de falso compartilhamento (FC) e fragmentação (FRG). Esta tabela tem como referência o trabalho[42].

Aplicação	Tamanho	Sincronização	Granularidade	*Overhead*
Barnes2	16K corpos	BAR e L	fina	FC e FRG
BarnesTmk	16K corpos	BAR	fina	FC e FRG
Em3d	40064 objetos	BAR	fina	FRG
FFT	1M pontos	BAR e L	média	FRG
Ocean	258 × 258	BAR e L	fina	FRG
SOR	256 × 640	BAR	grossa	-
TSP	18 cidades	BAR e L	fina	FRG
Water-Nsquared	512 moléculas	BAR e L	fina	FC e FRG

Tabela 6.2: Características das Aplicações

Capítulo 7

Análise do Comportamento das Aplicações

As aplicações escolhidas para analisar o potencial de FIESTA e RITMO possuem características distintas também em relação aos seus padrões de escrita. Além disso, podemos diferenciá-las como regulares ou irregulares considerando os acessos realizados dentro e fora de seção crítica.

Definimos como aplicações regulares, aquelas onde todas as páginas compartilhadas do conjunto da aplicação ou têm acessos somente fora de seção crítica ou somente dentro de uma única seção crítica. Já as aplicações irregulares são aquelas que possuem pelo menos uma página ou com acessos dentro e fora de seção crítica ou dentro de seção crítica com mais de uma variável de *lock*.

Nas seções seguintes analisamos cada uma das aplicações, avaliando inicialmente o *overhead* inserido pela máquina de estados e pelo algoritmo de categorização, necessários à implementação de cada técnica. Comparamos a versão original de TreadMarks (TM) com as versões onde são inseridos FIESTA e o algoritmo de categorização (RITMO) do padrão de escrita das páginas. Nessa comparação apresentamos os *overheads* relativos somente à quantidade total de *bytes* transmitidos, já que tanto para a implementação da máquina de estados quanto para o algoritmo de categorização não aumentamos o número de mensagens transmitidas. Somente para as aplicações onde ocorre um aumento perceptível da quantidade total de *bytes* transmitidos é que mostramos o impacto causado no tempo de execução.

Em seguida, destacamos os padrões de escrita das aplicações e quantificamos a influência de cada padrão no desempenho da aplicação, mostrando a porcentagem de falhas de página relativo a cada grupo de páginas categorizado por RITMO. Baseados nesses resultados, a escolha das técnicas de tolerância à latência e adaptação

mais apropriadas é naturalmente identificada. Finalmente, discutimos o *overhead* de memória causado pela inclusão da máquina de estados.

7.1 Aplicações Regulares

7.1.1 BarnesTmk

A aplicação BarnesTmk é uma aplicação regular com predominância de operações de sincronização de barreira. Existe somente uma variável de *lock* utilizada para identificação de processos na versão simulada da aplicação.

Comparando a quantidade total de *bytes* transmitidos da versão TreadMarks original com cada uma das versões que incluem a máquina de estados (FIESTA) e o algoritmo de categorização (RITMO), observamos que o *ovehead* introduzido em ambos os casos é desprezível. Este *overhead* é próximo a 1% e não afeta o tempo total de execução da aplicação.

Em BarnesTmk, FIESTA e RITMO identificam dois tipos de padrões de escrita: único escritor (SW) e múltiplos escritores (MW). A tabela 7.1 mostra a quantidade de páginas em cada um desses padrões e a quantidade total de falhas de acesso relativas a cada um desses grupos.

Quantidade	Categoria				Total
	SW	%	MW	%	
Páginas	156	27	417	73	573
Falhas de Acesso	3917	19	16284	81	20201

Tabela 7.1: Quantidade de páginas e de falhas de acesso relativas a cada padrão de escrita em BarnesTmk

As páginas categorizadas por RITMO como SW correspondem às páginas que contém as estruturas de dados que armazenam os ponteiros das células e dos corpos, e que são acessadas na fase de construção da árvore. As páginas categorizadas como MW armazenam as estrututras das células e dos corpos propriamente ditos e cujos acessos são feitos nas demais fases da aplicação (seção 6.2.2).

Essas informações sobre o padrão de escrita de BarnesTmk sugerem que avaliemos o potencial das técnicas SW-MW, *forwarding* e *prefetching*, para reduzir o tempo de espera pelos dados no protocolo *software DSM*.

Note que a técnica de *prefetching*, se bem sucedida, é a que causará maior impacto de redução dos *overheads* de acessos a dados remotos, porque é no grupo de páginas categorizadas como MW que encontramos a maior quantidade de falhas 81%. Isto é, elas são as principais responsáveis pelas operações de coerência.

7.1.2 Em3d

Em Em3d todas as operações de sincronização são realizadas com barreiras, portanto, só ocorrem acessos fora de seção crítica.

Na aplicação Em3d há um aumento perceptível da quantidade total de *bytes* transmitidos comparando a versão de TreadMarks original com as versões onde estão incluídos FIESTA e RITMO. Esse aumento é um pouco inferior a 10%. Este fato ocorre porque Em3d possui uma grande quantidade de páginas compartilhadas e de barreiras. Nas barreiras é computado o estado global das páginas. Mesmo com o aumento na quantidade de *bytes* transmitidos, o impacto no tempo total de execução é bem pequeno, pois não atinge 1,5% nas nossas simulações.

Em Em3d, FIESTA e RITMO identificam dois tipos de padrões de escrita: único escritor (SW) e múltiplos escritores (MW). A tabela 7.2 mostra a quantidade de páginas em cada um desses padrões e a quantidade total de falhas de acesso em cada um desses grupos.

Quantidade	Categoria				Total
	SW	%	MW	%	
Páginas	480	11	3761	89	4241
Falhas de Acesso	51335	93	3761	7	55103

Tabela 7.2: Quantidade de páginas e de falhas de acesso relativas a cada padrão de escrita em Em3d

As páginas categorizadas por RITMO como SW correspodem às páginas que armazenam os nós elétricos e os nós magnéticos (seção 6.2.3). Já as páginas categorizadas como MW armazenam as informações sobre as dependências entre os nós elétricos e magnéticos e os coeficientes destas dependências (seção 6.2.3).

Em Em3d, a grande maioria das páginas categorizadas como MW têm acessos somente de leitura, e as falhas contabilizadas são falhas de *cold start*. Portanto, essas páginas não causam *overheads* de coerência. Elas têm categorização MW porque permanecem com o valor *default* inicial de categorização para TreadMarks.

A técnica de *prefetching* simplesmente não causaria nenhum efeito, porque apesar de existirem páginas categorizadas como MW a nossa estratégia de implementação não atinge *overheads* causados por *cold start.*

Como em Em3d o grupo de páginas com padrão de escrita SW é o grupo responsável pela maior quantidade de falhas de acesso 93% e pelas operações de coerência, este fato nos sugere que avaliemos o potencial das técnicas SW-MW e *forwarding*, para reduzir o tempo de espera pelos dados.

7.1.3 FFT

Assim como BarnesTmk, a aplicação FFT é uma aplicação regular com predominância de operações de sincronização de barreira, e existe somente uma variável de *lock* utilizada para identificação de processos na versão simulada da aplicação. Na versão real de FFT não existe esta variável de *lock.*

Comparando a quantidade total de *bytes* transmitidos da versão de TreadMarks original com cada uma das versões com a adição de FIESTA e RITMO, observamos que o aumento para ambas versões é de aproximadamente 1%. FFT é uma aplicação com um número de páginas compartilhadas maior do que a aplicação Em3d. Poderíamos esperar que o *ovehead* fosse maior em FFT do que em Em3d. Porém, como não ocorre uma grande quantidade de operações de sincronização de barreira, o *overhead* introduzido tanto por FIESTA quanto por RITMO é desprezível, diferentemente do que ocorreu com Em3d.

Em FFT, também são identificados por FIESTA e RITMO dois tipos de padrões de escrita: único escritor (SW) e múltiplos escritores (MW). A quantidade de páginas em cada um desses padrões e a quantidade total de falhas de acesso correspondente a cada um deles são mostradas na tabela 7.3

Quantidade	Categoria				Total
	SW	%	MW	%	
Páginas	8193	67	4102	33	12295
Falhas de Acesso	83488	95	4132	5	87620

Tabela 7.3: Quantidade de páginas e de falhas de acesso relativas a cada padrão de escrita em FFT

As páginas categorizadas por RITMO como SW correspondem às páginas que armazenam os n pontos de dados complexos que são transformados (seção 6.2.4).

As páginas categorizadas como MW armazenam os n pontos de dados complexos referenciados como *root of unity*.

Da mesma forma que na aplicação Em3d, em FFT as páginas categorizadas como MW têm acessos somente de leitura e suas falhas são de *cold start*. Essas não causam *overheads* de coerência ao longo da execução da aplicação. Elas têm categorização MW porque permanecem com o valor *default* inicial de TreadMarks. Como em Em3d, a técnica de *prefetching* não causaria nenhum efeito se introduzida no protocolo, porque apesar de existirem páginas categorizadas como MW a nossa estratégia de implementação não atinge *overheads* causados por *cold start*.

Na aplicação FFT, as páginas categorizadas como SW são responsáveis pelas operações de coerência, o que sugere que avaliemos o potencial das técnicas SW-MW e *forwarding*, para reduzir o tempo de espera pelos dados.

7.1.4 SOR

A aplicação SOR é outra aplicação onde todas as operações de sincronização são realizadas com barreiras.

Nela, como na aplicação Em3d, ocorre um aumento perceptível na quantidade total de *bytes* transmitidos, embora ainda inferior a 10%, quando comparamos a versão de TreadMarks original com as versões onde FIESTA e RITMO são incluídos. Em SOR o impacto no tempo total de execução é desprezível para ambos FIESTA e RITMO, atingindo 0,5%.

Em SOR, FIESTA e RITMO também identificam dois tipos de padrões de escrita: único escritor (SW) e múltiplos escritores (MW). A tabela 7.4 mostra a quantidade de páginas em cada um desses padrões e a quantidade total de falhas de acesso em cada um desses grupos.

Quantidade	Categoria				Total
	SW	%	MW	%	
Páginas	1024	99,7	3	0,3	1027
Falhas de Acesso	5448	99,6	24	0,4	5472

Tabela 7.4: Quantidade de páginas e de falhas de acesso relativas a cada padrão de escrita em SOR

As páginas categoridas por RITMO como SW são dominantes nessa aplicação e armazenam os dados da matriz (seção 6.2.6).

Na aplicação SOR, assim como em FFT e em Em3d, as páginas categorizadas como SW são as páginas que necessitam de tratamento de coerência, e que possuem a maior quantidade de falhas. Na verdade, somente uma parte dessas páginas categorizadas como SW é que causam operações de coerência. Como a matriz é dividida em fatias por processador, somente as páginas que contém os dados das bordas das fatias pertencentes a cada processador é que necessitam de coerência. Para SOR, também, as informações de FIESTA e RITMO sugerem que avaliemos as técnicas SW-MW e *forwarding*.

7.2 Aplicações Irregulares

As aplicações que utilizamos nesta seção pertencem à classe de aplicações que apresentam padrões de escrita irregulares e que são identificados por FIESTA e RITMO. As estratégias de categorização de padrões de escritas atuais, não conseguem identificar com precisão páginas com estas características. Para cada aplicação irregular mostramos o percentual de páginas que apresentam padrão de escrita irregular e a quantidade total de falhas somente dessas páginas.

7.2.1 Water-Nsquared

A aplicação Water-Nsquared é considerada irregular porque possui páginas onde os acessos são realizados tanto fora quanto dentro de seção crítica, e páginas com variáveis de *locks* diferentes. As operações de sincronização são realizadas através de barreiras e de *locks*.

Comparando a quantidade total de *bytes* transmitidos pela versão TreadMarks original com cada uma das versões com a adição de FIESTA e de RITMO observamos que, para ambos os casos, o *overhead* introduzido é desprezível, sendo inferior a 1%.

Na aplicação Water-Nsquared, FIESTA e RITMO identificam três tipos de padrões de escrita: único escritor (SW), migratório (MIGR) e múltiplos escritores (MW). A quantidade de páginas em cada um desses padrões e a quantidade total de falhas de acesso correspondente a cada um deles são mostradas na tabela 7.5.

As páginas categorizadas por RITMO como SW e MIGR são dominantes nesta aplicação e correspondem às páginas que armazenam as móleculas (seção 6.2.8).

Consideramos que uma página tem padrão de escrita irregular quando ocorrem acessos dentro e fora de seções críticas ou quando ocorrem acessos em seções críticas

Quantidade	Categoria						Total
	SW	%	MIGR	%	MW	%	
Páginas	26	14	159	85	1	0,5	186
Falhas de Acesso	81	0,5	17510	94,4	22	0,1	17613

Tabela 7.5: Quantidade de páginas e de falhas de acesso relativas a cada padrão de escrita em Water-Nsquared

guardadas por *locks* distintos. A tabela 7.6 mostra a quantidade de páginas com padrão de escrita irregular, onde encontramos pelo menos um estado INOUT na seqüência que RITMO utiliza para categorizá-la e a quantidade de páginas que possuem mais de um *lock*. Note que pode haver interseção nesses grupos porque uma página pode ter tanto acessos dentro e fora de seção crítica quanto possuir mais de um *lock* na página. Por isso, optamos por mostrar somente a categorização dentro do grupo de páginas que apresentam estado INOUT. Essa tabela também mostra a quantidade total de falhas de acesso correspondente a cada um desses grupos, que denominamos INOUT e MULT-LOCKS. É importante observar também que primeiro contabilizamos as falhas de acesso para o grupo INOUT e se a página não estiver neste estado contabilizamos as falhas para o grupo MULT-LOCKS.

Quantidade	INOUT				MULT-LOCKS
	SW	MIGR	MW	Total	
Páginas	10	81	1	92	92
Falhas de Acesso	4321				197

Tabela 7.6: Quantidade de páginas e de falhas de acesso relativas aos padrões de escrita irregular em Water-Nsquared

A quantidade total de páginas com padrão de escrita irregular INOUT e MULT-LOCKS correspondem a 50% do total de páginas em Water-Nsquared em ambos os casos. A quantidade total de falhas de acesso geradas por estas páginas correspondem a 26% do total de falhas da aplicação.

Para Water-Nsquared, as informações obtidas com FIESTA e RITMO sugerem que avaliemos as técnicas SW-MW e *forwarding*. Note que, essas técnicas, se bem sucedidas, têm o potencial de causar redução no tempo de espera pelos dados remotos, porque as páginas categorizadas como MIGR são as que possuem a maior quantidade de falhas 94,4% e, portanto, as principais responsáveis pelas operações

de coerência. É importante notar também que as páginas migratórias são 85,5% do total de páginas, os 14,5% das páginas restantes contribuem somente com 0,6% do total de falhas. Como somente 0,1% das falhas corresponde ao padrão de escrita MW, a introdução da técnica *prefetching* causaria efeito mínimo, e por isso não é avaliada.

7.2.2 Barnes2

A aplicação Barnes2 é irregular porque contém páginas com acessos dentro e fora de seção crítica e com variáveis de *lock* diferentes. É implementada com operações de sincronização de barreiras e de *locks*. As sincronizações realizadas através de *locks* são predominantes nessa aplicação.

Como na maioria das aplicações estudadas, em Barnes2, não há um aumento significativo da quantidade total de *bytes* transmitidos quando comparamos a versão TreadMarks original com as versões onde FIESTA e RITMO são incluídos. Esse aumento também é próximo de 1%.

Em Barnes2, assim como em Water-Nsquared, FIESTA e RITMO também identificam três tipos de padrões de escrita: único escritor (SW), migratório (MIGR) e múltiplos escritores (MW). A tabela 7.7 mostra a quantidade de páginas em cada um desses padrões e a quantidade total de falhas de acesso em cada um deles.

Quantidade	Categoria						Total
	SW	%	MIGR	%	MW	%	
Páginas	35	5	699	93	18	2	752
Falhas de Acesso	99458	60	58284	35	8698	5	166440

Tabela 7.7: Quantidade de páginas e de falhas de acesso relativas a cada padrão de escrita em Barnes2

As páginas categorizadas por RITMO como SW e MIGR são dominantes em Barnes2. As páginas SW correspondem às páginas que contém as estruturas de dados que armazenam os ponteiros das células e dos corpos, e que são acessadas na fase de construção da árvore. As páginas categorizadas como MIGR armazenam as estrututras das células e dos corpos propriamente ditos e cujos acessos são feitos nas demais fases da aplicação (seção 6.2.1).

A tabela 7.8 mostra tal qual a tabela 7.6 da aplicação Water-Nsquared a quantidade de páginas com padrão de escrita irregular e a quantidade total de falhas de

acesso correspondente a elas.

Quantidade	INOUT				MULT-LOCKS
	SW	MIGR	MW	Total	
Páginas	26	21	12	59	454
Falhas de Acesso	67725				51

Tabela 7.8: Quantidade de páginas e de falhas de acesso relativas aos padrões de escrita irregular em Barnes2

A quantidade total de páginas com padrão de escrita irregular INOUT e MULT-LOCKS corresponde a 8% e 60%, respectivamente, do total de páginas em Barnes2. A quantidade total de falhas de acesso geradas por estas páginas correspondem a 41% do total de falhas da aplicação.

Na aplicação Barnes2, as páginas categorizadas como SW e MIGR são as páginas que mais necessitam de tratamento de coerência, e que possuem a maior quantidade de falhas 95%. Para Barnes2, também, as informações de FIESTA e RITMO sugerem que avaliemos as técnicas SW-MW e *forwarding* para reduzir o tempo de espera pelos dados remotos. É esperado que o *prefetching* não cause impacto no desempenho da aplicação.

7.2.3 TSP

Consideramos a aplicação TSP como irregular porque possui páginas com acessos dentro e fora de seção crítica e uma única página com padrão de escrita irregular, isto é, protegida por variáveis de *locks* diferentes. As sincronizações são realizadas através de barreiras e de *locks*.

Em TSP, a diferença entre a quantidade total de *bytes* transmitidos pela versão TreadMarks original e pelas versões com a adição de FIESTA e RITMO é desprezível e para ambos os casos, o *overhead* introduzido é inferior a 1%.

A tabela 7.9 mostra a quantidade de páginas e a quantidade total de falhas de acesso para os dois padrões de escrita, MIGR e MW, que FIESTA e RITMO identificam na aplicação TSP.

As páginas categorizadas por RITMO como SW são dominantes nessa aplicação e armazenam as rotas geradas (seção 6.2.7).

Apresentamos na tabela 7.10 informações idênticas a da tabela 7.6 da aplicação

Quantidade	Categoria						Total
	SW	%	MIGR	%	MW	%	
Páginas	24	92	1	4	1	4	26
Falhas de Acesso	1248	97	36	2,5	8	0,5	1292

Tabela 7.9: Quantidade de páginas e de falhas de acesso relativas a cada padrão de escrita em TSP

Water-Nsquared, a quantidade de páginas com padrão de escrita irregular e a quantidade total de falhas de acesso correspondente a elas.

Quantidade	INOUT				MULT-LOCKS
	SW	MIGR	MW	Total	
Páginas	6	1	1	8	1
Falhas de Acesso	479				0

Tabela 7.10: Quantidade de páginas e de falhas de acesso relativas aos padrões de escrita irregular em TSP

A quantidade total de páginas com padrão de escrita irregular INOUT e MULT-LOCKS correspondem a 30% e 4%, respectivamente, do total de páginas em TSP. A quantidade total de falhas de acesso geradas por estas páginas correspondem a 37% do total de falhas da aplicação.

As informações sobre o padrão de escrita de TSP sugerem que avaliemos o potencial das técnicas SW-MW e *forwarding*, porque o padrão de escrita SW é o padrão da maioria das páginas e o responsável por 92% das falhas de página. Em TSP, a técnica de *prefetching* também não influenciaria no desempenho da aplicação se inserida no protocolo.

7.2.4 Ocean

A aplicação Ocean é uma aplicação com predominância de operações de sincronização de barreira, e poucas operações de sincronização são realizadas com *locks*, se comparadas à quantidade de barreiras. Ela é irregular porque contém páginas com acessos tanto dentro quanto fora de seção crítica.

Assim como nas aplicações Em3d, FFT e SOR, em Ocean ocorre um aumento na quantidade total de *bytes* transmitidos quando comparamos a versão de TreadMarks original com as versões onde FIESTA e RITMO são incluídos. Este fato

ocorre porque Ocean é uma das aplicações que têm uma grande quantidade de páginas compartilhadas, e dentre todas as aplicações, a que têm maior quantidade de barreiras. Nas barreiras é computado o estado global das páginas.

Ocean é a aplicação com maior aumento na quantidade de *bytes* transmitidos 18,9%. Porém, observamos que o impacto no tempo total de execução atinge 4,8% e que este impacto ainda pode ser considerado como pequeno.

Em Ocean, FIESTA e RITMO identificam dois tipos de padrões de escrita: único escritor (SW) e múltiplos escritores (MW). A tabela 7.11 mostra a quantidade de páginas e a quantidade total de falhas de acesso para cada um desses grupos.

Quantidade	Categoria				Total
	SW	%	MW	%	
Páginas	3731	99,7	12	0,3	3743
Falhas de Acesso	236712	99	1707	1	238419

Tabela 7.11: Quantidade de páginas e de falhas de acesso relativas a cada padrão de escrita em Ocean

As páginas categorizadas por RITMO como SW são dominantes nessa aplicação e armazenam as grades (seção 6.2.5).

A tabela 7.12 mostra tal qual a tabela 7.6 da aplicação Water-Nsquared a quantidade de páginas com padrão de escrita irregular e a quantidade total de falhas de acesso correspondente a elas.

Quantidade	INOUT				MULT-LOCKS
	SW	MIGR	MW	Total	
Páginas	1	-	1	2	1
Falhas de Acesso	2741				45

Tabela 7.12: Quantidade de páginas e de falhas de acesso relativas aos padrões de escrita irregular em Ocean

A quantidade total de páginas com padrão de escrita irregular INOUT e MULT-LOCKS corresponde a menos de 1% do total de páginas em Ocean, para ambos os casos. A quantidade total de falhas de acesso geradas por estas páginas corresponde a 1% do total de falhas da aplicação.

As informações sobre o padrão de escrita de Ocean, obtidas através de FIESTA e RITMO, mais uma vez, sugerem que avaliemos o potencial das técnica SW-MW

e *forwarding*, porque o padrão de escrita SW é o padrão responsável por 99,3% das falhas de página. Em Ocean, a técnica de *prefetching* não influenciaria o desempenho da aplicação.

7.3 *Overhead* de Memória

Na implementação da máquina de estados utilizamos quatro conjuntos (R, W, P e C) para representar os tipos de acesso realizados e quais processadores realizam os acessos. O armazenamento dessas informações varia de acordo com o tamanho do sistema implementado. Para um sistema de oito processadores, como o nosso sistema simulado, são necessários 32 bits (4 *bytes*) por cada página da aplicação para armazenar as informações desses conjuntos. Num sistema com 16 processadores seriam necessários 64 bits para cada página, e assim por diante. Dessa forma, o *overhead* de memória para o armazenamento da máquina de estados pode variar consideravelmente dependendo da quantidade de páginas compartilhadas.

Considere, por exemplo, duas das aplicações do conjunto que estudamos, TSP e FFT, e que representam os extremos de quantidade de páginas compartilhadas nesse conjunto. Além disso, considere também um sistema com oito processadores, como o que simulamos. A aplicação TSP possui a menor quantidade de páginas compartilhadas (26). O *overhead* de memória dessa aplicação é igual a 104 *bytes* por processador ou 832 *bytes* no total. A aplicação FFT é a que possui a maior quantidade de páginas compartilhadas (12295). Seu *overhead* de memória é igual a 49.180 *bytes* por processador ou 393.440 *bytes* no total. Observe que embora o *overhead* em valor absoluto tenha aumentado o *overhead* introduzido por FIESTA é fixo e igual a 0,78% para ambos os casos.

7.4 Discussão

A tabela 7.13 resume, para cada aplicação, os padrões de compartilhamento dominantes identificados por FIESTA e RITMO (SW, MIGR ou MW) e as técnicas que potencialmente podem reduzir o tempo de espera por dados remotos, SW-MW, *forwarding* (FWD) e *prefetching* (PRF).

Nas nossas estratégias de implementação utilizamos as técnicas de SW-MW e *forwarding* em páginas categorizadas como SW ou MIGR, e a técnica de *prefetching* somente em páginas categorizadas como MW. Assumimos esta estratégia porque

Aplicação	Padrões Dominantes	Técnicas
BarnesTmk	SW e MW	SW-MW, FWD e PRF
Em3d	SW	SW-MW e FWD
FFT	SW	SW-MW e FWD
SOR	SW	SW-MW e FWD
Water-Nsquared	MIGR	SW-MW e FWD
TSP	SW	SW-MW e FWD
Barnes2	SW e MIGR	SW-MW e FWD
Ocean	SW	SW-MW e FWD

Tabela 7.13: Padrões de escrita e técnicas que podem reduzir o tempo de espera pelos dados remotos para cada aplicação

errar a previsão no *prefetching* causa mais *overhead* ao protocolo do que errar a previsão no *forwarding*. O envio de páginas tem custos diferentes do envio de *diffs* se considerarmos as mensagens trocadas e as interrupções.

O custo do *forwarding* é composto pelo envio de uma mensagem do produtor para o consumidor contendo a página, e uma interrupção no consumidor para receber a página. Já o custo do *prefetching* é composto pelo envio de uma mensagem do consumidor para o produtor pedindo os *diffs*, uma interrupção no produtor para atender ao pedido, uma mensagem do produtor para o consumidor enviando os *diffs*, uma interrupção no consumidor para armazenar os *diffs* e, finalmente, uma interrupção para aplicar os *diffs* quando é feito um acesso à página.

Outro motivo que nos leva a essa escolha de implementação é a possibilidade de redução do *overhead* no armazenamento da memória. Utilizando as técnicas SW-MW e *forwarding* necessitamos de menos memória para armazenar *twins* e *diffs* e podemos em alguns casos evitar o *garbage collection*.

Na maioria das aplicações estudadas prevalece o padrão de compartilhamento de escrita de um único escritor, onde ocorrem a maior quantidade de falhas de página, mesmo para algumas aplicações irregulares. A partir desses resultados podemos dizer que os protocolos que implementem alguma técnica de adaptação e de tolerância à latência tal como SW-MW e/ou *forwarding* têm o potencial de melhorar seu desempenho para praticamente todas as aplicações.

A aplicação BarnesTmk é a única aplicação que justifica que avaliemos o potencial do *prefetching*, porque nas demais aplicações não existem páginas MW que têm falhas de página o suficiente para a avaliação, segundo nossa estratégia. Seria

interessante completar a estratégia de *prefetching* para considerar as falhas de acesso geradas por *cold start*, que acontecem por exemplo em Em3d e FFT.

Capítulo 8

Resultados Experimentais

Neste capítulo apresentamos os resultados obtidos com a execução das oito aplicações que constituem o nosso conjunto de testes. Analisamos inicialmente o perfil do tempo de execução de cada aplicação. Em seguida, avaliamos o desempenho das técnicas de tolerância à latência isoladamente e de forma combinada para cada aplicação. Consideramos nas nossas comparações a quantidade total de *bytes* transmitidos, o número das mensagens transferidas e o tempo de execução paralela. Finalizamos o capítulo discutindo nossos resultados.

8.1 Avaliação dos Resultados

8.1.1 Perfil dos *Overheads* das Aplicações

Para muitos sistemas *software DSM*, os principais *overheads* encontrados estão relacionados às latências de comunicação e ao tempo dispendido com operações para manter a memória coerente. Nesta tese, estamos interessados em avaliar a redução dos *overheads* causados pelo acesso a dados remotos, quando introduzimos estratégias de tolerância à latência, baseadas nas informações fornecidas por FIESTA e por RITMO, no sistema *software DSM* TreadMarks.

Para caracterizar a composição dos *overheads* das aplicações-teste, mostramos na figura 8.1 o perfil de execução para cada uma delas. Esses resultados são obtidos executando cada uma das aplicações sobre o sistema TreadMarks simulado com oito processadores. Essa figura apresenta uma visão detalhada do tempo de execução das aplicações através de um diagrama de barras. Todas os tempos estão normalizados.

Cada barra corresponde a uma das aplicações estudadas, e mostra o tempo de execução dividido em tempo de computação (*busy*), tempo de espera pelos dados (*data*), tempos dispendidos nas sincronizações de barreiras e *locks* (*synch*), IPC *over-*

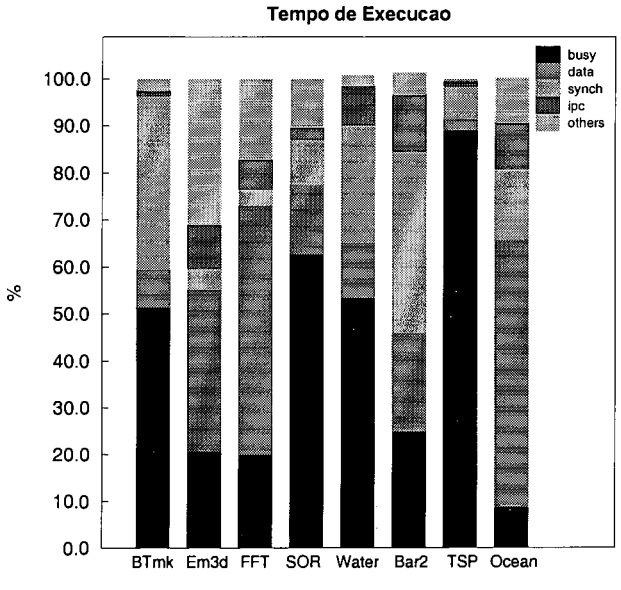

Figura 8.1: Tempo de execução paralela normalizado das aplicações em TreadMarks

head (ipc), e outros *overheads* (*others*). *Busy* representa a quantidade de trabalho útil realizado. *Data* é uma combinação do tempo de processamento de operações de coerência (geração e aplicação de *diffs*, por exemplo) e da latência da rede envolvidos na busca de páginas ou *diffs* como um resultado das falhas de acesso às páginas. *Synch* representa os retardos envolvidos na espera em barreiras e *lock acquires/releases*, incluindo o *overhead* do processamento das notificações de escrita e intervalos. IPC contabiliza o tempo que o processador gasta servindo pedidos que vêm de processadores remotos. A última subdivisão, *others*, é composta pelas latências de falhas na *cache* e de falhas na TLB, e pelos tempos de espera para o *write buffer* esvaziar e de interrupção.

Observando a figura 8.1 vemos que muitas das aplicações sofrem severamente com os *overheads* de acesso a dados remotos e de sincronização. Em3d (35%), FFT (53%), Barnes2 (20%) e Ocean (57%) têm uma grande quantidade do tempo de execução gasto no tempo de espera pelos dados remotos. As aplicações BarnesTmk (36%), Water-Nsquared (25%), Barnes2 (38%) e Ocean (15%) têm tempo de sincronização muito alto. Esses *overheads* influenciam diretamente os *speedups* atingidos por muitas aplicações executadas sobre sistemas *software DSM*.

As técnicas de tolerância à latência e adaptação que implementamos (SW-MW, FWD e PRF) são utilizadas para aliviar os efeitos do *overhead* de acesso a dados

remotos. O potencial de eliminação (valores de *data* mostrado na figura 8.1) que podemos atingir varia entre 2,5% em TSP e 57% em Ocean, assumindo que os *overheads* de sincronização não sejam afetados indiretamente pelas técnicas. Na verdade, as técnicas podem, em alguns casos, reduzir um pouco mais os *overheads* porque podem interferir positivamente nas sincronizações pela melhora do balanceamento de carga.

8.1.2 *Speedup*

A figura 8.2 mostra os *speedups* das aplicações analisadas. Para cada aplicação está mostrado o *speedup* do melhor resultado obtido com uma técnica isolada ou combinação de técnicas. As barras da esquerda de cada grupo correspondem à versão original de TreadMarks, e as barras da direita correspondem a versão da melhor técnica dependendo da aplicação. O nosso conjunto de aplicações tem *speedups* que variam entre 1,8 e 24,7. O *speedup* igual a 1,8 é baixo mas não incomum em aplicações executadas sobre sistemas *software DSM*.

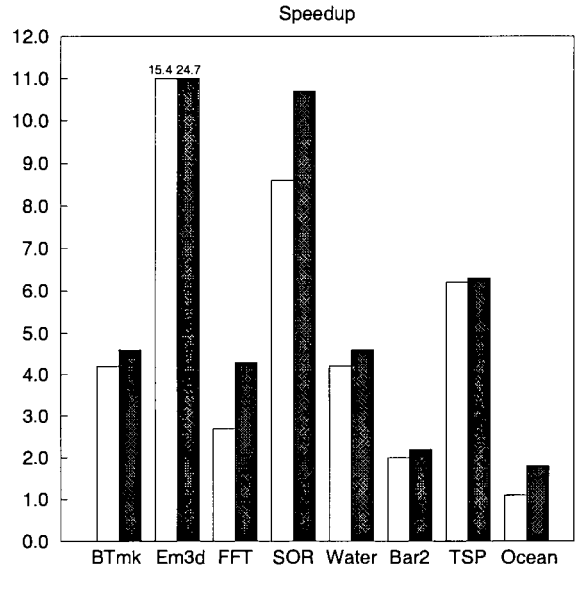

Figura 8.2: *Speedups* das aplicações para os protocolos estudados

Para praticamente todas as aplicações, os melhores resultados foram obtidos com a adição da técnica de *forwarding*. Somente a aplicação BarnesTmk obteve melhor resultado com a combinação das técnicas de *forwarding* e *prefetching*.

As aplicações Em3d e SOR possuem *speedups* superlineares. Ambas têm uma

porcentagem alta do tempo de execução com falhas na *cache*, quando executadas em um único processador. Esses tempos correspondem a 72% em Em3d e 61% em SOR.

A figura 8.2 mostra que Em3d, FFT, SOR e Ocean conseguem melhorar substancialmente o desempenho com a inserção das técnicas. *Forwarding* apresenta ganhos significativos em relação a TreadMarks, para essas aplicações. BarnesTmk, Water-Nsquared e Barnes2 são menos afetadas pela inserção das técnicas porque nelas a parcela de tempo de execução gasta em sincronizações é significativa. O desempenho de TSP praticamente não se modifica porque esta aplicação tem muito pouco *overhead* (11%).

Os resultados mostram que, em geral, a inclusão da técnica *forwarding* permitiu maiores ganhos para as aplicações do que a inclusão de *prefetching* ou simplesmente SW-MW. É importante notar que independente de SW-MW, *forwarding* ou *prefetching*, houve consistência na utilização das técnicas que mostram um potencial de desempenho superior à versão original de TreadMarks.

8.1.3 Quantidade de Falhas de Acesso

Em nossos estudos, utilizamos como base o sistema *software DSM* TreadMarks que utiliza o protocolo de invalidações para manter a coerência da memória. Quando uma página está invalidada, na primeira tentativa de acesso à página ocorre uma falha de acesso. Assim, a quantidade de falhas indica a necessidade de disparar ações de coerência, estando então relacionada aos *overheads* de acesso aos dados. Dessa forma, a redução na quantidade de falhas pode indicar a melhora de desempenho do sistema.

Podemos reduzir a quantidade de falhas em TreadMarks adicionando mecanismos de previsão de quais processadores irão receber os dados modificados, desde que a página seja imediatamente atualizada ao receber os dados. Ao ser imediatamente atualizada a página se torna válida e não sofre falhas de acesso.

As falhas de página podem acontecer nos acessos de leitura ou de escrita. A quantidade de falhas de acesso de leitura, excetuando as de *cold start*, pode indicar a necessidade de se disparar o tratamento de coerência das páginas antecipadamente para se reduzir o *overhead* do protocolo. Uma falha de acesso de leitura envolve não apenas interrupções locais como troca de mensagens entre processadores. As falhas de escrita envolvem somente uma interrupção local. Dessa forma, as quantidades

de falhas de acesso de leitura e de escrita nos permitem inferir o impacto da implementação dessa técnica sobre o desempenho do protocolo.

Das técnicas que implementamos, somente a técnica de *forwarding* pode reduzir a quantidade de falhas de acesso, porque, ao receber uma página antecipadamente, ela é desprotegida para leitura. Na tabela 8.1 comparamos a quantidade de falhas de acesso da versão original de TreadMarks (TM) com a versão onde a técnica de *forwarding* (FWD) foi adicionada, para cada uma das aplicações do nosso conjunto de testes. Essa tabela mostra a quantidade de falhas em acessos de leitura, de escrita, e suas respectivas diferenças absolutas (ABS) e percentuais (%) entre as versões TM e FWD.

Aplicações	Leitura				Escrita			
	TM	FWD	ABS	%	TM	FWD	ABS	%
BarnesTmk	12310	9917	2393	-19	7891	7891	0	0,0
Em3d	36448	5054	31394	-86	18615	18615	0	0,0
FFT	58938	29387	29551	-50	28682	32778	4096	+13
SOR	3288	1416	1872	-57	2184	2212	28	+1
Water-Nsq	10983	9455	1528	-14	6645	7210	565	+8
Barnes2	89402	86818	2584	-3	76113	79466	3353	+4
TSP	971	634	337	-35	326	430	104	+24
Ocean	140924	116686	24238	-17	98725	118768	20043	+17

Tabela 8.1: Percentuais de redução (-) e aumento (+) nas falhas de acesso das aplicações testadas

A tabela mostra que a técnica de *forwarding* reduz as falhas de acesso de leitura para todas as aplicações, sendo que para quatro das oito aplicações, Em3d, FFT, SOR e TSP essa redução é significativa. A redução nas falhas de acesso de leitura variam de 3% para Barnes2 a 86% para Em3d.

As falhas de violação de escrita variam de 0% a 24%. O aumento da quantidade dessas falhas pode acontecer porque ao enviar uma página antecipadamente, a protegemos contra escrita para que sejam geradas notificações de escrita.

Observando então as informações de falhas de acesso contidas nesta tabela, inferimos que a tendência é que, para todas as aplicações, a adição da técnica FWD no sistema TreadMarks, melhore o seu desempenho, se desconsiderarmos os efeitos negativos indiretos que possam ocorrer (por exemplo, aumento do tempo de espera por *locks*).

8.2 Estudo Detalhado das Aplicações

Apresentamos nesta seção uma avaliação quantitativa das técnicas SW-MW, *forwarding* (FWD) e *prefetching* (PRF), tanto de forma isolada quanto combinadas. Para cada aplicação observamos as taxas de acerto de páginas que foram enviadas antecipadamente e que foram utilizadas para a técnica de *forwarding*. Somente para a aplicação BarnesTmk mostramos as taxas relativas ao *prefetching* e sua combinação com as técnicas SW-MW (SM+PRF)[1] e *forwarding* (FWD+PRF), porque é a única aplicação com um percentual de páginas categorizadas como MW que justifica esta avaliação.

A latência de comunicação remota entre processos é um dos fatores determinantes no desempenho de sistemas *software DSM*, baseados em redes de *workstations* ou PCs. Por isso, mostramos também a comunicação gerada, através do número de mensagens e da quantidade de *bytes* transmitidos. Além disso, mostramos a redução no tempo total de execução.

Todas as nossas comparações são realizadas de forma normalizada em relação à versão original de TreadMarks.

8.2.1 Aplicações Regulares

BarnesTmk

A tabela 8.2 mostra uma avaliação comparativa das técnicas de tolerância à latência FWD, PRF e das combinações SM+PRF e FWD+PRF, na aplicação BarnesTmk. Nessa tabela, mostramos, para a técnica FWD, a quantidade total de páginas enviadas antecipadamente e a quantidade dessas páginas que foram acessadas antes de serem invalidadas. Esses envios antecipados de páginas consideramos como úteis. Mostramos apenas os valores para o conjunto de páginas categorizadas como SW, porque em BarnesTmk não existem páginas migratórias.

Para a técnica PRF, mostramos a quantidade total de requisições antecipadas de *diffs* e a quantidade desses *diffs* que foram utilizados antes das páginas serem novamente invalidadas. Essas são consideradas requisições úteis. É importante ressaltar que a nossa estratégia de implementação de *prefetching* só considera as páginas categorizadas como MW.

Observando as informações dessa tabela, para BarnesTmk, mostramos que nossas

[1] Por conveniência utilizamos SM em vez de SW-MW

Técnica	SW			MW		
	Úteis	Total	%	Úteis	Total	%
FWD	2393	2590	92	-	-	-
PRF	-	-	-	6480	8833	73
SM+PRF	-	-	-	6481	8844	73
FWD+PRF	2393	2590	92	6481	8844	73

Tabela 8.2: Eficiência do conjunto de técnicas testadas em BarnesTmk

estratégias de implementação tanto para FWD quanto para PRF têm uma alta taxa de acertos. Essas taxas são de 92% e 73%, respectivamente.

Podemos avaliar o potencial de redução nos *overheads* de comunicação observando os gráficos das figuras 8.3 e 8.4. Esses gráficos mostram o número de mensagens transmitidas e a quantidade de *bytes* transmitidos para cada uma das técnicas testadas em BarnesTmk.

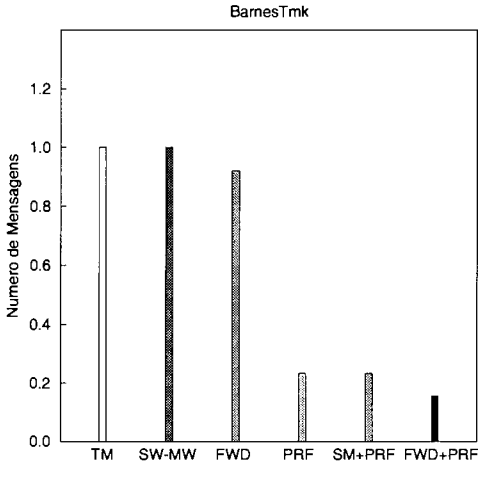

Figura 8.3: Número de mensagens para TM, SW-MW, FWD, PRF, SM+PRF e FWD+PRF em BarnesTmk

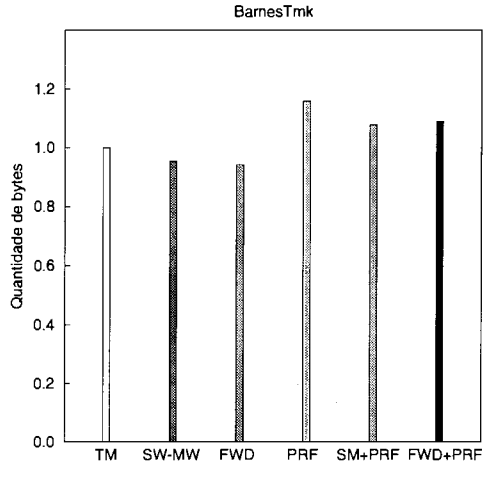

Figura 8.4: Quantidade de bytes transmitidos para TM, SW-MW, FWD, PRF, SM+PRF e FWD+PRF em BarnesTmk

A figura 8.3 mostra que reduzimos consideravelmente o número de mensagens transmitidas quando utilizamos a técnica PRF. Este fato ocorre porque cada processador requisita todos os *diffs* que necessita na saída da barreira e os processadores que atendem ao pedido enviam todos os *diffs* que possuem de forma agrupada numa mesma mensagem. A maior redução de 84% encontramos na combinação das técnicas FWD+PRF. Este fato ocorre porque a nossa estratégia para seleção

dos processadores candidatos a realizarem acessos a página foi bem sucedida. A técnica de adaptação SW-MW não reduz o número de mensagens. A técnica FWD reduz pouco o número de mensagens (8%), porque são poucas as páginas categorizadas como SW em relação às páginas categorizadas como MW. Para ambas as técnicas PRF e SM+PRF, a redução no número de mensagens é igual a 77%.

Na figura 8.4 podemos ver a quantidade de *bytes* transmitidos para cada técnica utilizada em BarnesTmk. As técnicas SW-MW e FWD reduzem a quantidade de *bytes* transmitidos de 5% e 6%, respectivamente, porque as páginas SW são completamente modificadas na fase de construção da árvore. As técnicas PRF, SM+PRF e FWD+PRF aumentam essa quantidade. Este fato ocorre porque na técnica PRF existem páginas que são novamente invalidadas antes de utilizarem os *diffs* pedidos antecipadamente, necessitando então que *diffs* adicionais sejam requisitados. A técnica PRF isoladamente contribui com um aumento de 16%.

A figura 8.5 ilustra os resultados do tempo de execução para BarnesTmk utilizando diagramas de barras. Os resultados estão normalizados para o tempo de execução da versão original de TreadMarks. Todas as comparações são realizadas em relação à essa versão.

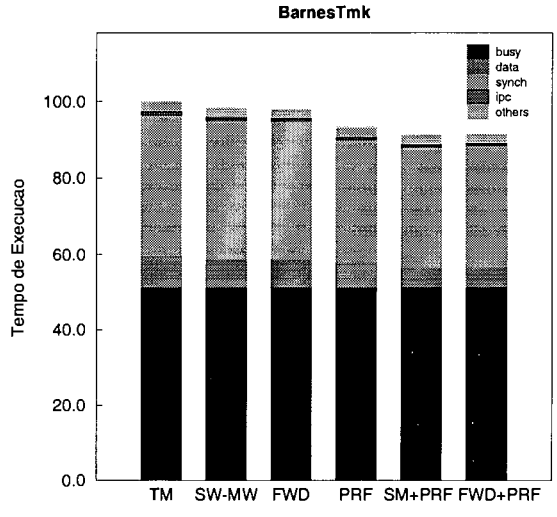

Figura 8.5: Tempo de execução de BarnesTmk para TM, SW-MW, FWD, PRF, SM+PRF e FWD+PRF

As técnicas de tolerância à latência e adaptação que implementamos têm como objetivo principal reduzir o tempo correspondente a *data*. Como esperado pela análise realizada no capítulo anterior, a figura 8.5 mostra que a técnica PRF (12%)

e suas combinações causam mais impacto na redução dos *overheads* de acessos a dados remotos, do que as técnicas SW-MW (1%) e FWD (2%). A redução do tempo de acesso a dados remotos para SM+PRF e FWD+PRF é igual a 14%.

Todas as técnicas reduzem o tempo de execução quando comparadas à versão original de TreadMarks simulado. A combinação das técnicas SM+PRF e FWD+PRF proporcionam a mesma redução no tempo total de execução. Elas reduzem em 9% esse tempo, sendo esta a maior redução obtida entre as técnicas testadas. Este fato ocorre porque nas nossas estratégias de implementação das técnicas FWD e PRF, na saída da barreira, enviamos as páginas ou requisitamos os *diffs* antecipadamente. Um dos efeitos da combinação destas técnicas é aumentar a quantidade de tráfego na rede na saída da barreira, o que anula um pouco o ganho das técnicas. O aumento do tráfego na rede, na saída da barreira, é menor quando combinamos SM+PRF, porque as requisições de página estão distribuídas ao longo das fases.

Indiretamente, reduzimos o tempo de IPC em 8% para SW-MW, 20% para FWD, 25% para PRF e 35% para SM+PRF e FWD+PRF. As reduções obtidas tanto em FWD quanto em PRF e suas combinações, ocorrem por causa do mecanismo de previsão preciso implementado nestas técnicas. Além do tempo de IPC, indiretamente reduzimos também o tempo de sincronização de barreira (14%) porque melhoramos o balanceamento de carga. Em BarnesTmk, o processador responsável por gerar seqüencialmente a árvore, que representa os corpos e células do espaço, obtém os dados remotos necessários em menor tempo. Assim, os demais processadores que atingem a barreira primeiro, esperam menos por esse processador.

Em3d

Na aplicação Em3d somente as páginas categorizadas como SW necessitam de manutenção de coerência ao longo da execução da aplicação, e correspondem a apenas 11% do total de páginas. A tabela 8.3 mostra que obtemos um alto índice de acertos (99%) com a técnica de tolerância à latência FWD, isso porque conseguimos prever, com bastante exatidão, o conjunto de processadores que compartilham as páginas e estas são utilizadas antes de serem novamente invalidadas.

O potencial de redução dos *overheads* de comunicação para Em3d pode ser visto nos gráficos das figuras 8.6 e 8.7. Esses gráficos mostram o número de mensagens e a quantidade de *bytes* transmitidos para as técnicas SW-MW e FWD. Estes gráficos

Técnica	SW		
	Úteis	Total	%
FWD	31434	31860	99

Tabela 8.3: Eficiência da técnica FWD para Em3d

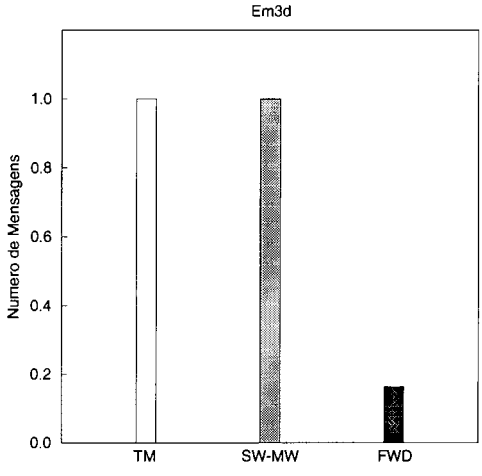

Figura 8.6: Número de mensagens para TM, SW-MW e FWD em Em3d

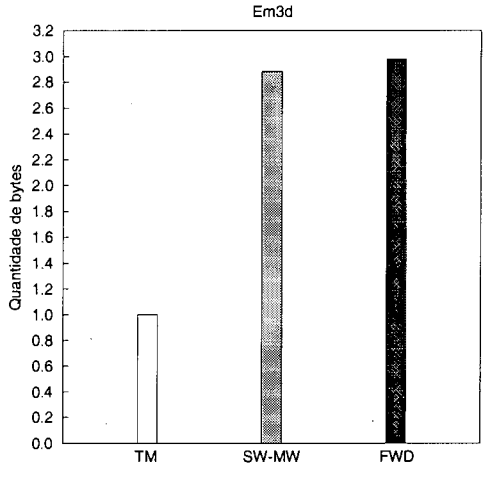

Figura 8.7: Quantidade de bytes transmitidos para TM, SW-MW e FWD

estão normalizados em relação a TreadMarks.

A figura 8.6 mostra que reduzimos consideravelmente o número de mensagens transmitidas quando utilizamos a técnica FWD (84%). Este fato ocorre porque atingimos um alto índice de acertos quando enviamos as páginas antecipadamente. A técnica SW-MW não reduz o número de mensagens.

Segundo o gráfico da figura 8.7, as duas técnicas utilizadas em Em3d apresentam um aumento expressivo na quantidade de *bytes* transmitidos em relação a TreadMarks. Esses aumentos correspondem a 188% para SW-MW e 197% para FWD. Essas técnicas transferem páginas inteiras ao invés de enviar *diffs* de tamanho menor (em média 1011 *bytes*), como ocorre em TreadMarks. Em3d apresenta muita fragmentação.

A figura 8.8 ilustra os resultados do tempo de execução para Em3d utilizando o diagramas de barras. As barras correspondem a versão original de TreadMarks e as técnicas SW-MW e FWD. Como podemos observar nessa figura a técnica SW-MW tem um grande impacto no tempo de espera pelos dados remotos e a técnica FWD, com sua previsão acertada, reduz ainda mais esse tempo comparado ao tempo

de TreadMarks. Uma vez estabelecida a ligação entre os nós elétricos e os nós magnéticos na fase inicial da computação, elas não mudam durante a execução. Na fase principal da computação, os nós elétricos são modificados considerando somente os nós magnéticos, e os nós magnéticos são modificados considerando somente os nós elétricos. A redução no tempo de espera pelos dados remotos atinge 90% em FWD e 64% em SW-MW.

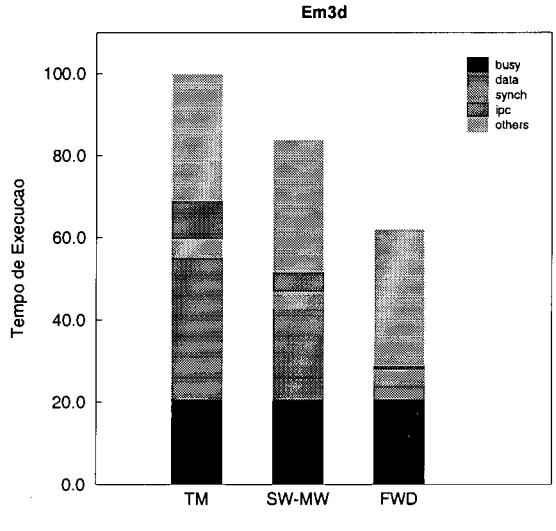

Figura 8.8: Tempo de execução de Em3d para TM, SW-MW e FWD

Indiretamente reduzimos também o tempo de IPC de 97% para FWD e de 87% para SW-MW em relação a TreadMarks. O tempo de sincronização é reduzido em 16% e 7%, para FWD e SW-MW, respectivamente. A técnica FWD é a que apresenta melhor resultado porque é seletiva e eficiente, os processadores que requisitariam as modificações realizadas nos dados compartilhados não precisam mais fazê-lo porque a página já está disponível quando do acesso.

A redução total no tempo de execução da aplicação Em3d para as técnicas FWD e SW-MW corresponde a 38% e 16%, respectivamente. Este diagrama confirma as expectativas de que as páginas categorizadas como SW são responsáveis pelos *overheads* de coerência e a utilização da técnica de tolerância à latência seletiva pode reduzir consideravelmente o tempo de execução. Além disso, confirma também a tendência mostrada através da comparação das falhas de acesso de leitura e escrita, realizada no início deste capítulo. Em Em3d ocorre uma redução de 86% nas falhas de acesso de leitura.

FFT

Na aplicação FFT, assim como em Em3d, somente as páginas categorizadas como SW necessitam de manutenção de coerência ao longo da execução, porém em FFT elas são a maioria das páginas (67%). Na tabela 8.4 mostramos que a técnica de tolerância à latência FWD apresenta um bom índice de acertos (59%).

Técnica	SW		
	Úteis	Total	%
FWD	50567	29959	59

Tabela 8.4: Eficiência da técnica FWD para FFT

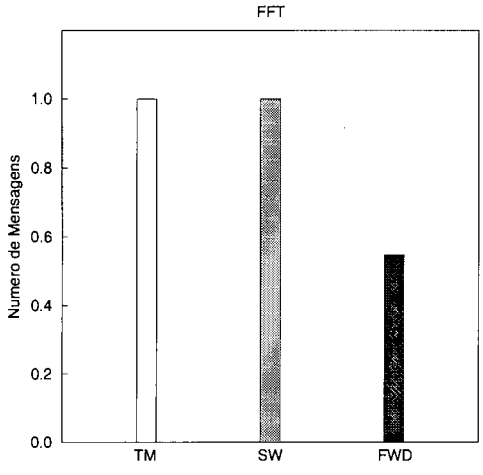

Figura 8.9: Número de mensagens para TM, SW-MW e FWD em FFT

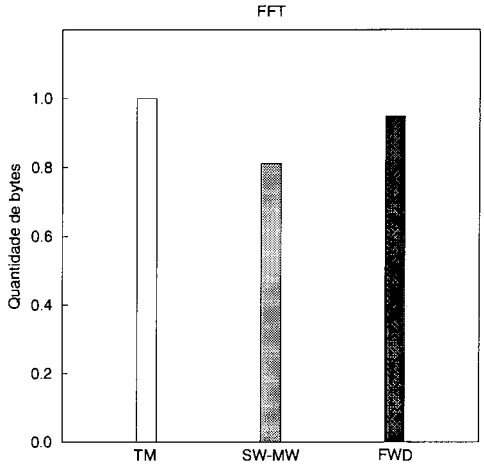

Figura 8.10: Quantidade de bytes transmitidos para TM, SW-MW e FWD

As figuras 8.9 e 8.10 mostram o potencial de redução dos *overheads* de comunicação para FFT. Esses gráficos mostram o número de mensagens e a quantidade de *bytes* transmitidos para TreadMarks e para as técnicas SW-MW e FWD. Comparando com TreadMarks, a técnica SW-MW reduz a quantidade de *bytes* transmitidos em 19%, porém não reduz o número de mensagens. A técnica FWD reduz ambos o número de mensagens (45%) e a quantidade de *bytes* transmitidos (5%) em relação a TreadMarks. A redução na quantidade de *bytes* transmitidos para ambas as técnicas ocorre porque em FFT, uma grande quantidade de dados é modificada dentro da página, e como conseqüência, há pouca fragmentação. Então, o tamanho médio dos

diffs, aproximadamente 6100 *bytes*, gerados por TreadMarks é grande se comparado ao tamanho de uma página.

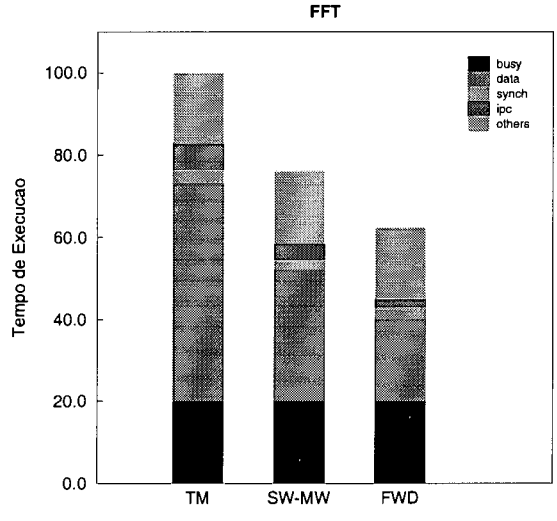

Figura 8.11: Tempo de execução de FFT para TM, SW-MW e FWD

A figura 8.11 mostra o tempo de execução de FFT executando a versão original de TreadMarks, e das técnicas SW-MW e FWD. Conforme podemos observar, os tempos de execução diminuem sensivelmente em relação a TreadMarks tanto para a técnica FWD (32%) quanto para SW-MW (24%). A técnica FWD obtém maior redução do que a técnica SW-MW, porque sua previsão de quais processadores irão acessar as páginas é correta na maior parte das vezes. A redução no tempo de espera pelos dados remotos para FWD e para SW-MW atingem 62% e 38%, respectivamente.

Da mesma forma que ocorre na aplicação Em3d, em FFT, indiretamente reduzimos o tempo de IPC. A redução para FWD é igual a 69% e para SW-MW é igual a 39%.

Na aplicação FFT, reduzimos, também, indiretamente o tempo de sincronização na barreira. Esta redução é de 17% em FWD e de 30% em SW-MW. Este efeito é conseguido por um melhor balanceamento de carga, isto é, os processadores que chegam primeiro a barreira ficam menos tempo parados esperando pelos outros processadores. A redução maior no tempo de sincronização em SW-MW acontece porque os pedidos de página ocorrem ao longo das fases, e na técnica FWD a previsão de enviar as páginas antecipadamente e o envio propriamente dito são contabilizados na barreira, então os processadores gastam mais tempo nela. Além disso, FFT possui

um grande número de páginas compartilhadas.

Em FFT a redução no total do tempo de execução para técnica FWD é de 32% e para a técnica SW-MW é de 24%, em relação à versão de TreadMarks. FWD se mostra como a técnica mais eficiente.

Em FFT, as expectativas sugeridas em relação ao comportamento da aplicação, onde as páginas categorizadas como SW é que necessitam de coerência, e da tendência mostrada através da comparação das falhas de acesso de leitura e escrita, também se confirmam.

SOR

A aplicação SOR é outra aplicação onde todas as páginas que necessitam de coerência são categorizadas como SW. Para SOR, a técnica de tolerância à latência FWD também obtém uma alta taxa de acertos nos envios de página realizados antecipadamente (98%) que são úteis.

Técnica	SW		
	Úteis	Total	%
FWD	2184	2240	98

Tabela 8.5: Eficiência da técnica FWD para SOR

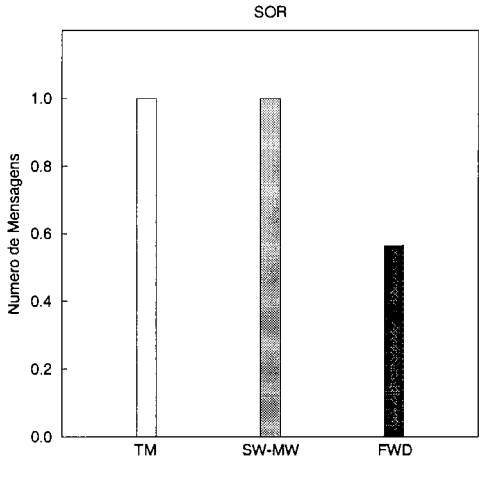

Figura 8.12: Número de mensagens para TM, SW-MW e FWD em SOR

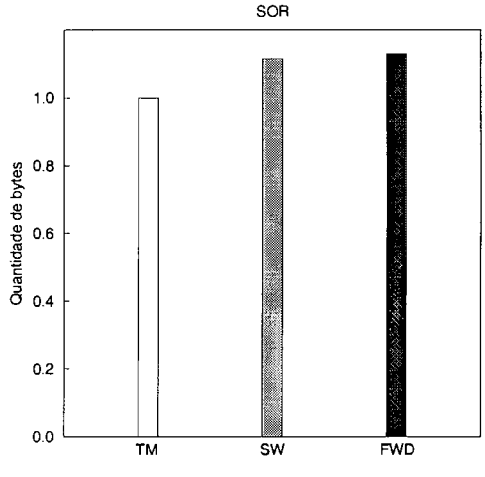

Figura 8.13: Quantidade de bytes transmitidos para TM, SW-MW e FWD

As figuras 8.12 e 8.13 mostram o potencial de redução dos *overheads* de comu-

nicação para SOR. Esses gráficos mostram o número de mensagens e a quantidade de *bytes* transmitidos para TreadMarks e para as técnicas SW-MW e FWD. A aplicação SOR, apresenta comportamento similar a aplicação Em3d em relação ao número e a quantidade de mensagens transmitidos, tanto para a técnica SW-MW quanto para a técnica FWD. Ambas as técnicas aumentam a quantidade de *bytes* transmitidos, em 12% e 13%, respectivamente. A técnica FWD reduz o número de mensagens (44%). Porém, como a granularidade de acesso é maior em SOR, os *diffs* gerados por TreadMarks têm tamanho maior. Então, a diferença entre as quantidades de *bytes* transmitidos é muito menor se comparada com a diferença encontrada em Em3d. Porém, os *diffs* gerados por TreadMarks ainda têm tamanho menor do que uma página, cerca de 3800 *bytes* em média.

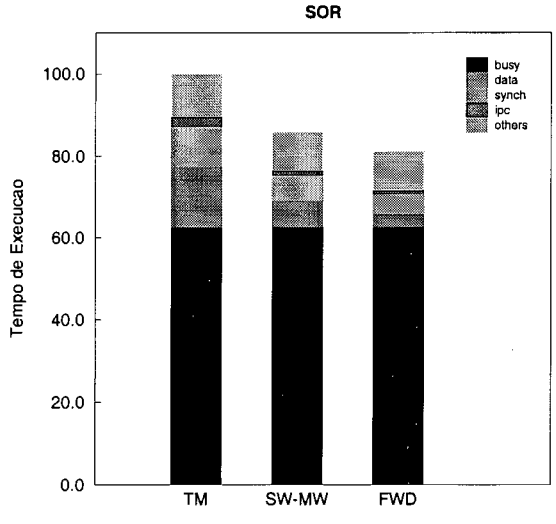

Figura 8.14: Tempo de execução de SOR para TM, SW-MW e FWD

A figura 8.14 ilustra o tempo de execução de SOR executando a versão original de TreadMarks, e das técnicas SW-MW e FWD, utilizando o diagrama de barras.

Como podemos observar, os tempos de execução diminuem em relação a TreadMarks tanto para a técnica SW-MW (14%) quanto para FWD (19%). FWD obtém maior redução do que SW-MW, porque sua previsão de quais processadores irão acessar as páginas é correta. Com essas técnicas de tolerância à latência, a redução no tempo de espera pelos dados remotos é igual a 78% em FWD e igual a 57% em SW-MW.

Em SOR, indiretamente reduzimos o tempo de IPC e o tempo de sincronização de barreira. A redução no IPC para FWD é igual a 62% e para SW-MW é igual a 53%.

Essa redução é conseqüência da previsão acertada do conjunto de processadores que recebem as páginas antecipadamente. Eles não necessitam requisitar estas páginas, logo, não interrompem outros processadores.

As reduções no tempo de sincronização de barreira atingem 50% em FWD e 36% em SW-MW. Diferente do que acontece na aplicação FFT, em SOR, as páginas categorizadas como SW que realmente necessitam de coerência são muito poucas. Então, o tempo dispendido na barreira com a previsão e envio das páginas é pequeno. Somente as páginas que armazenam os dados localizadas nas bordas de cada fatia da matriz atribuída aos processadores é que necessitam de coerência.

Em SOR, a redução no tempo total de execução atinge 19% em FWD e 14% em SW-MW, sendo a técnica FWD também a mais eficiente. Além disso, confirmamos o comportamento da aplicação, onde páginas SW são as que precisam de tratamento de coerência, e a tendência mostrada através da comparação das falhas de acesso de leitura e escrita.

8.2.2 Aplicações Irregulares

As aplicações que utilizamos nesta seção pertencem a classe de aplicações que apresentam padrões de escrita irregulares. FIESTA identifica tais páginas, e após categorizá-las com RITMO empregamos as técnicas de adaptação e tolerância à latência adeqüadamente. As estratégias de categorização de padrões de escrita atuais não conseguem identificar com precisão páginas com estas características e portanto podem deixar de tratá-las de forma apropriada.

Water-Nsquared

Em Water-Nsquared a maioria das páginas ou são categorizadas como SW ou como MIGR. Com a utilização da técnica de tolerância à latência FWD, a quantidade de páginas enviadas antecipadamente que são úteis corresponde a 52% para as páginas SW, e 57% para as páginas MIGR, do total de páginas enviadas em cada um destes grupos. Obtemos estes índices porque o restante das páginas enviadas antecipadamente são invalidadas antes de serem utilizadas. Isto acontece porque cada processador após modificar as suas moléculas sincroniza para modificar as $n/2 - n/p$ moléculas seguintes armazenadas em outros processadores.

O potencial de redução dos *overheads* de comunicação para Water-Nsquared pode ser visto nas figuras 8.15 e 8.16. Esses gráficos mostram o número de mensagens e

Técnica	SW			MIGR		
	Úteis	Total	%	Úteis	Total	%
FWD	168	349	52	1214	2849	57

Tabela 8.6: Eficiência da técnica FWD para Water-Nsquared

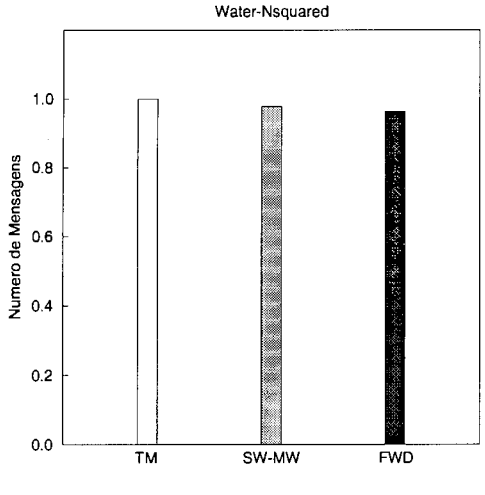

Figura 8.15: Número de mensagens para TM, SW-MW e FWD em Water-Nsquared

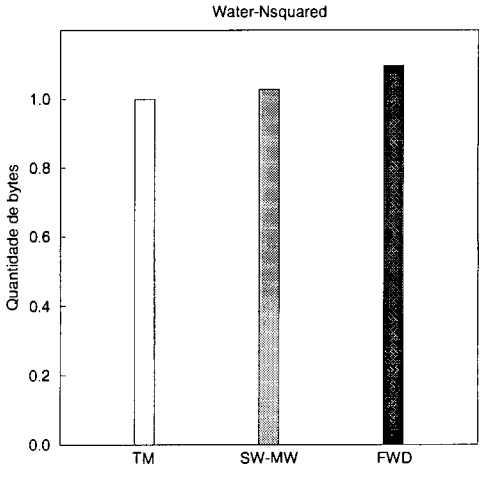

Figura 8.16: Quantidade de bytes transmitidos para TM, SW-MW e FWD

a quantidade de *bytes* transmitidos para TreadMarks e para as técnicas SW-MW e FWD.

Observando a figura 8.15, tanto a técnica FWD quanto a técnica SW-MW reduzem pouco o número de mensagens em relação a TreadMarks. Essas reduções são iguais a 4% em FWD e 2% em SW-MW. Em Water-Nsquared grande parte (aproximadamente dois terços) das mensagens trocadas entre os processadores se deve as sincronizações de barreiras e de *locks* e não por acesso a dados remotos.

Observando a figura 8.16 e comparando com TreadMarks, ambas as técnicas FWD e SW-MW aumentam ligeiramente a quantidade de *bytes* transmitidos, e são equivalentes a 10% e 3%, respectivamente. O aumento pequeno da quantidade de *bytes* transmitidos para ambas as técnicas, ocorre porque, em Water-Nsquared, cada processador modifica todas as suas n/p moléculas, sendo que em cada página podem ser armazenadas aproximadamente seis moléculas. Além disso, cada processador modifica também as $n/2 - n/p$ moléculas seguintes armazenadas em outros processadores.

A figura 8.17 ilustra o tempo de execução de Water-Nsquared executando a versão original de TreadMarks, e das técnicas SW-MW e FWD, através do diagrama em barras. Em Water-Nsquared, as técnicas FWD e SW-MW reduzem o tempo total de execução em 6%. A redução no tempo total de execução não é expressiva porque a maior parte dos tempos de *overhead* desta aplicação é devida às sincronizações e não ao acesso dos dados compartilhados. Assim mesmo, reduzimos consideravelmente o tempo de espera pelos dados remotos em FWD (56%) e em SW-MW (54%).

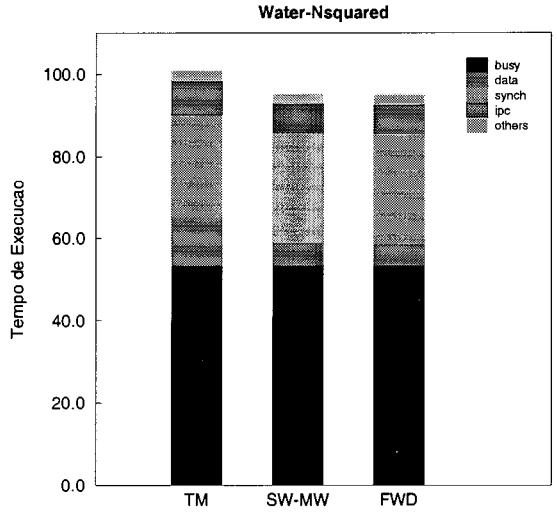

Figura 8.17: Tempo de execução de Water-Nsquared para TM, SW-MW e FWD

Indiretamente, reduzimos os tempos gastos com IPC e incrementamos o tempo de sincronização. A redução no IPC para FWD e SW-MW é igual a 15% em ambos os casos. Isto acontece porque acertamos a previsão ao enviar páginas.

O incremento no tempo de sincronização é de 8% tanto para FWD quanto para SW-MW. Este fato ocorre porque mais processadores ficam parados esperando para ter acesso aos *locks*.

Water-Nsquared apresenta o comportamento sugerido onde as páginas SW e MIGR são as responsáveis pelas ações de coerência e a utilização das técnicas de tolerânia à latência FWD e SW-MW têm potencial de reduzir o tempo de acesso a dados remotos.

Barnes2

Em Barnes2, como pode ser visto na tabela 8.7, obtemos um alto índice de acertos (78%) para as páginas categorizadas como MIGR porque as páginas são acessadas

antes de serem novamente invalidadas. Porém, para as páginas SW esse índice é de 43%. Este é o menor índice de acertos obtido com a utilização da técnica de tolerância à latência FWD. Isso porque, o restante das páginas enviadas antecipadamente são invalidadas antes de serem utilizadas.

Técnica	SW			MW		
	Úteis	Total	%	Úteis	Total	%
FWD	2478	5772	43	377	486	78

Tabela 8.7: Eficiência da técnica FWD para Barnes2

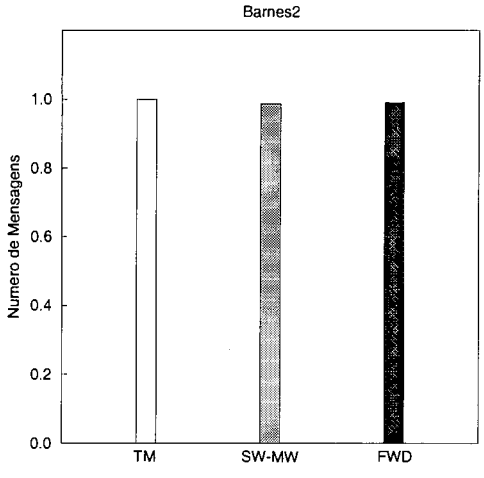

Figura 8.18: Número de mensagens para TM, SW-MW e FWD em Barnes2

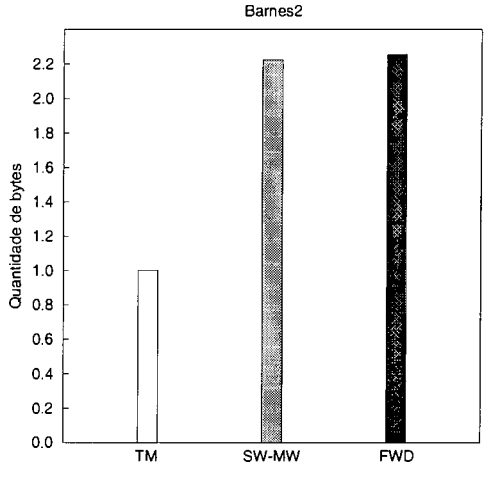

Figura 8.19: Quantidade de bytes transmitidos para TM, SW-MW e FWD

As figuras 8.18 e 8.19 mostram o potencial de redução dos *overheads* de comunicação para Barnes2. Esses gráficos mostram o número de mensagens e a quantidade de *bytes* transmitidos para TreadMarks e para as técnicas SW-MW e FWD. Em Barnes2 podemos observar na figura 8.18, que tanto para a técnica FWD quanto para SW-MW a redução no número de mensagens em relação a TreadMarks é imperceptível. Essas reduções são iguais a 1% para FWD e SW-MW. Esta aplicação é dominada pelas mensagens trocadas entre os processadores por causa das operações de sincronização, principalmente sincronizações com *locks*.

O gráfico da figura 8.19 mostra que as duas técnicas utilizadas em Barnes2 apresentam um aumento expressivo na quantidade de *bytes* transmitidos quando comparadas a TreadMarks. Em FWD, o aumento equivale a 126% e em SW-MW a

123%. TreadMarks envia *diffs* de tamanho muito pequeno (menor do que 200 *bytes* em média) se comparados ao tamanho da página que é enviada pelas técnicas. Os *diffs* gerados por TreadMarks correspondem às modificações realizadas nos corpos e células do espaço. Este comportamento é diferente do encontrado na aplicação Water-Nsquared, onde TreadMarks gera *diffs* de tamanho maior se comparados ao tamanho dos *diffs* gerados em Barnes2.

O tempo de execução de Barnes2, executando a versão original de TreadMarks, e das técnicas SW-MW e FWD, pode ser visto na figura 8.20.

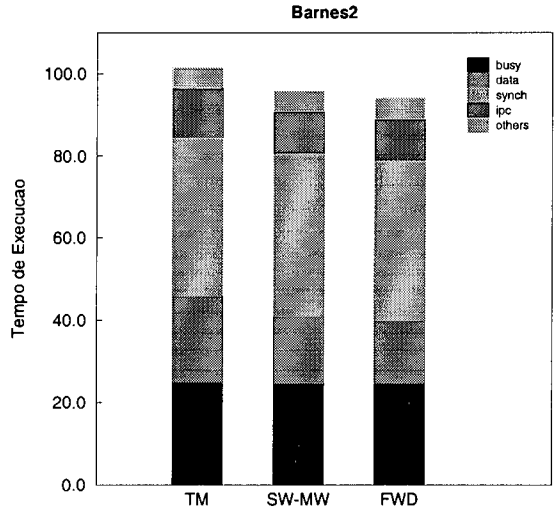

Figura 8.20: Tempo de execução de Barnes2 para TM, SW-MW e FWD

Comparando com a versão de TreadMarks, as técnicas FWD e SW-MW reduzem o tempo total de execução em 7% e 6%. A redução no tempo total de execução de Barnes2 é um pouco mais expressiva do que a redução atingida em Water-Nsquared. Mas em ambas as aplicações, o maior tempo é gasto no *overhead* de sincronização e não no acesso aos dados compartilhados.

Em Barnes2, reduzimos o tempo de espera pelos dados remotos de 28% em FWD e de 23% em SW-MW.

Além disso, indiretamente reduzimos o IPC e incrementamos do tempo de sincronização. A redução no IPC para FWD é igual a 17% e igual a 16% em SW-MW porque acertamos a previsão do conjunto de processadores que recebem as páginas. O incremento no tempo de sincronização corresponde a 1% em FWD e 3% em SW-MW. Barnes2 é uma aplicação onde o tempo de sincronização domina o tempo de execução.

Assim como Water-Nsquared, Barnes2 também apresenta o comportamento sugerido, onde as páginas categorizadas como SW e MIGR são as responsáveis pelas operações de coerência.

TSP

Na aplicação TSP, as páginas que necessitam de coerência são categorizadas como MIGR. A tabela 8.8 mostra que, com a técnica de tolerância à latência FWD, obtemos uma alta taxa de acertos (88%) na previsão do conjunto de processadores que recebem as páginas antecipadamente.

Técnica	MIGR		
	Úteis	Total	%
FWD	337	384	88

Tabela 8.8: Eficiência da técnica FWD para TSP

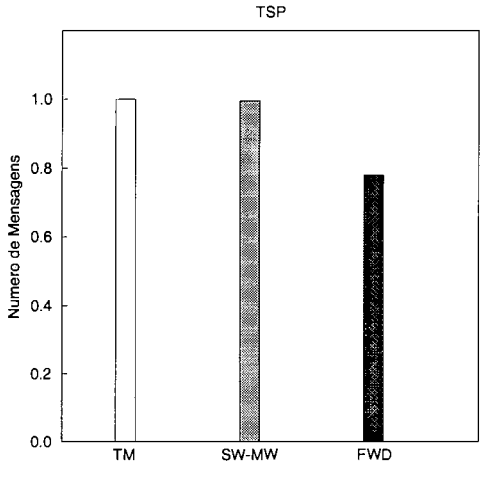

Figura 8.21: Número de mensagens para TM, SW-MW e FWD em TSP

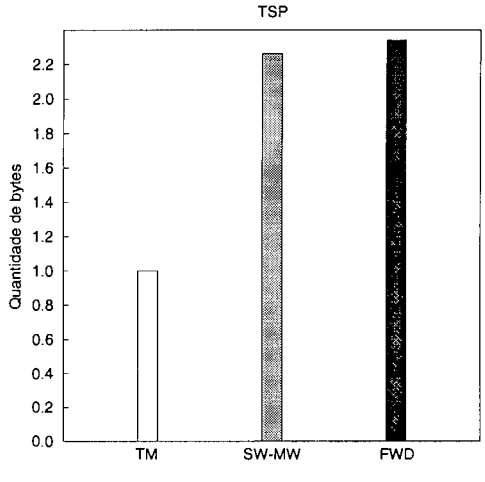

Figura 8.22: Quantidade de bytes transmitidos para TM, SW-MW e FWD

O potencial de redução dos *overheads* de comunicação para TSP pode ser visto nas figuras 8.21 e 8.22. Esses gráficos mostram o número de mensagens e a quantidade de *bytes* transmitidos para TreadMarks e para as técnicas SW-MW e FWD. A figura 8.21 mostra que reduzimos o número de mensagens transmitidas quando utilizamos a técnica FWD (22%). Este fato ocorre porque atingimos um alto índice

de acertos quando enviamos as páginas antecipadamente. Na técnica SW-MW a redução do número de mensagens é imperceptível (0,5%).

Segundo o gráfico da figura 8.22, as duas técnicas utilizadas em TSP apresentam um aumento expressivo na quantidade de *bytes* transmitidos em relação à Tread-Marks. Esses aumentos correspondem a 127% para SW-MW e 134% para FWD. Essas técnicas transferem páginas inteiras ao invés de enviar *diffs* de tamanho menor (em média menores do que 320 *bytes*), como ocorre em TreadMarks.

O tempo de execução de TSP executando a versão original de TreadMarks, e das técnicas SW-MW e FWD, pode ser vistos na figura 8.20.

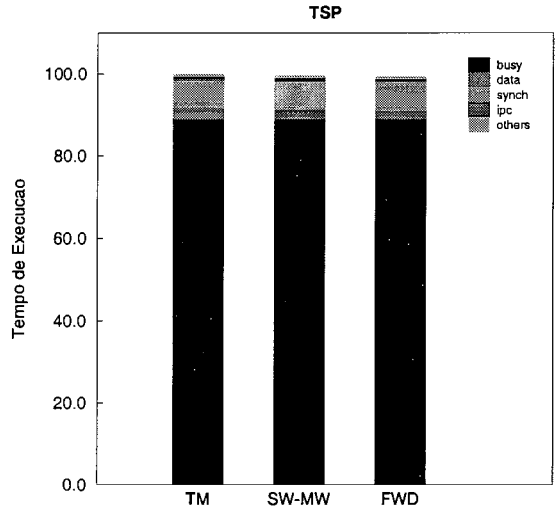

Figura 8.23: Tempo de execução de TSP para TM, SW-MW e FWD

Em TSP, os tempos de execução diminuem de forma imperceptível em relação a TreadMarks tanto para a técnica SW-MW (0,5%) quanto para FWD (1%). Como podemos observar, TSP é a aplicação que possui o maior tempo de computação útil. Todo o tempo gasto em *overheads* não ultrapassa 11% do tempo total de execução.

Na aplicação TSP, as técnicas de tolerância à latência FWD e SW-MW, reduzem o tempo de espera pelos dados remotos de 25% e 5%, respectivamente. Indiretamente reduzimos o tempo de IPC e incrementamos o tempo de sincronização de barreira. A redução no IPC para FWD é igual a 38% e para SW-MW 20%. Essa redução é conseqüência da previsão acertada do conjunto de processadores que recebem as páginas antecipadamente. Os incrementos no tempo de sincronização atingem 6% em FWD e 1% em SW-MW. Nesta aplicação, a parcela de tempo relativo ao acesso aos dados remotos é muito pequena (2,5%), por isso, mesmo as técnicas FWD e

SW-MW reduzindo esta parcela e contabilizando os efeitos indiretos, a redução no tempo total de execução é muito pequena.

Ocean

Com a técnica de tolerância à latência FWD, em Ocean, obtemos um índice de acertos de 67% na previsão dos processadores que recebem as páginas antecipadamente, como mostra a tabela 8.9. Nesta aplicação, as páginas categorizadas como SW é que necessitam de tratamento de coerência.

Técnica	SW		
	Úteis	Total	%
FWD	26869	402883	67

Tabela 8.9: Eficiência da técnica FWD para Ocean

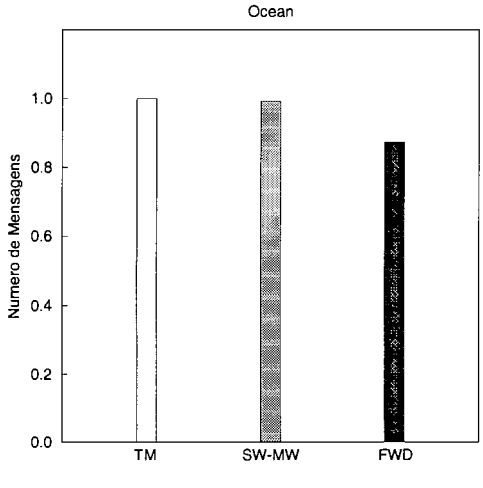

Figura 8.24: Número de mensagens para TM, SW-MW e FWD em Ocean

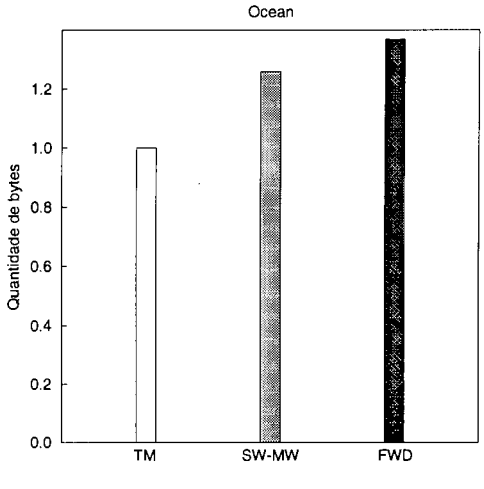

Figura 8.25: Quantidade de bytes transmitidos para TM, SW-MW e FWD

O potencial de redução dos *overheads* de comunicação para Ocean pode ser visto nas figuras 8.24 e 8.25. Esses gráficos mostram o número de mensagens e a quantidade de *bytes* transmitidos para TreadMarks e para as técnicas SW-MW e FWD.

Em Ocean, reduzimos o número de mensagens transmitidas para FWD de 13%, como mostra a figura 8.24. Este fato ocorre porque atingimos um alto índice de

acertos quando enviamos as páginas antecipadamente. A técnica SW-MW reduz pouco o número de mensagens (1%).

O gráfico da figura 8.25, mostra que as duas técnicas utilizadas em Ocean apresentam aumento na quantidade de *bytes* transmitidos em relação a TreadMarks. Esses aumentos correspondem a 26% para SW-MW e 37% para FWD. A diferença entre TreadMarks e as técnicas não é tão grande como em TSP, por exemplo, porque TreadMarks gera *diffs* quase do tamanho de uma página. Esses *diffs* têm em média tamanho de 3915 *bytes*.

O tempo de execução de Ocean sobre a versão original de TreadMarks, e das técnicas SW-MW e FWD, pode ser vistos na figura 8.26.

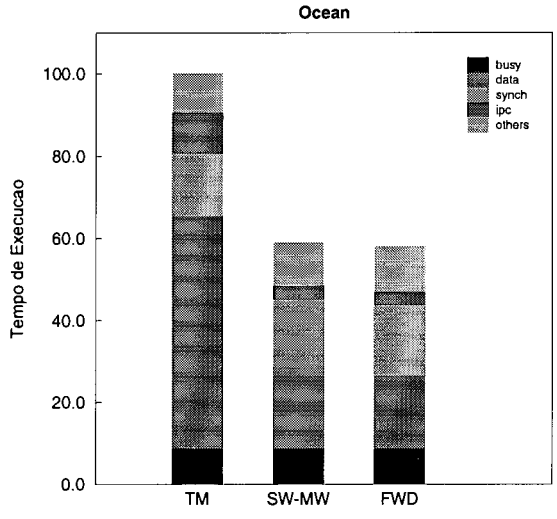

Figura 8.26: Tempo de execução de Ocean para TM, SW-MW e FWD

Como podemos observar, os tempos de execução diminuem em relação a TreadMarks tanto para a técnica FWD (42%) quanto para SW-MW (41%). FWD obtém uma redução ligeiramente maior do que SW-MW, por causa da sua previsão. Com essas técnicas de tolerância à latência, a redução no tempo de espera pelos dados remotos é igual a 69% em FWD e igual a 65% em SW-MW. A diferença entre FWD e SW-MW não é expressiva devido ao tempo dispendido na barreira com a previsão e envio das páginas. Na aplicação Ocean há uma grande quantidade de páginas compartilhadas e de operações de sincronização de barreira.

Em Ocean, indiretamente reduzimos o tempo de IPC e incrementamos o tempo de sincronização de barreira. A redução no IPC é de 68% para FWD e de 66% para SW-MW porque, acertando a previsão do conjunto de processadores que recebem

as páginas antecipadamente, deixamos de interromper outros processadores. Os incrementos no tempo de sincronização de barreira atingem 15% em FWD e 9% em SW-MW. A grande quantidade de páginas compartilhadas e de barreiras, desta aplicação, têm efeito direto no desempenho porque é na barreira que computamos o estado global de cada página, a previsão e o envio das páginas.

Na aplicação Ocean, confirmamos que as páginas categorizadas como SW são as que necessitam de operações de coerência e as técnicas FWD e SW-MW têm potencial para melhorar o desempenho desta aplicação. Além disso, confirmamos também a tendência mostrada através da comparação das falhas de acesso de leitura e escrita.

8.3 Discussão

Os resultados apresentados mostram que, baseados nas informações de FIESTA e de RITMO, a técnica implementada com mecanismo de previsão FWD tem alta precisão.

Testamos a técnica PRF somente na aplicação BarnesTmk. PRF também possui mecanismo de previsão com alta taxa de acertos para essa aplicação e atinge 73%.

A técnica FWD tem taxas de acerto, independente da categorização das páginas, variando de 43% a 99%. Cinco das oito aplicações estudadas têm índices de acerto acima de 70%: BarnesTmk, Em3d, SOR, Barnes2 e TSP.

Considerando a categorização das páginas, as aplicações Water-Nsquared, Barnes2 e TSP são as aplicações que possuem páginas classificadas como MIGR, sendo que as duas primeiras possuem também páginas SW. Water-Nsquared apresenta uma eficiência média nos dois grupos de página, 52% para SW e 57% para MIGR. Já Barnes2, apresenta uma eficiência baixa para páginas SW (43%) e alta para páginas MIGR (78%). TSP tem alto índice de acertos para suas páginas MIGR (88%). As demais aplicações que utilizam a técnica FWD, só têm páginas categorizadas como SW. Três delas, BarnesTmk (92%), Em3d (99%) e SOR (97%), têm uma alta eficiência, e as duas restantes, Ocean (67%) e FFT (59%), têm boa eficiência.

Com relação ao tempo de execução das aplicações com as técnicas implementadas nessa tese, mostramos que nossas estratégias apresentam ganhos para todas as aplicações em relação à versão original de TreadMarks. As reduções variam de

1% em TSP a 42% em Ocean. Para quatro aplicações, Ocean, Em3d, FFT e SOR, obtemos reduções significativas.

Considerando os *overheads* de comunicação, reduzimos, também em relação a TreadMarks, o número de mensagens para todas as aplicações com a técnica FWD. E novamente, em quatro delas, Em3d, FFT, SOR e TSP, a redução é significativa. Em geral, nas aplicações irregulares, os ganhos resultantes são menores.

A quantidade de *bytes* transmitidos só é reduzida na aplicação FFT porque TreadMarks gera *diffs* de tamanho grande se comparados ao tamanho de uma página. Em3d, Barnes2 e TSP têm aumentos substanciais devido a grande fragmentação. Ocean apresenta um aumento médio. Nas demais aplicações o aumento da quantidade de *bytes* transmitidos é pequeno.

Em todas as aplicações, a nossa estratégia seletiva, proporciona redução no *overhead* de acesso a dados remotos. Em todas, a redução é significativa (25 - 89%), quando comparadas ao sistema TreadMarks.

A estratégia seletiva também permite um ganho indireto significativo (15- 97%) no tempo de IPC para todas as aplicações. Porém, indiretamente, reduzimos o tempo de sincronização para as aplicações regulares (14-50%) e aumentamos nas irregulares (1-15%), devido ao balanceamento de carga.

Capítulo 9

Trabalhos Relacionados

Existem muitos trabalhos já desenvolvidos que enfocam sistemas de memória compartilhada distribuída. Estes sistemas podem ser implementados no nível de usuário, de linguagem, do compilador, do sistema operacional ou no nível de *hardware*. Neste capítulo apresentamos, de forma resumida, os trabalhos genericamente relacionados, deixando para discutir apenas os trabalhos mais diretamente relacionados aos temas contidos nessa tese. Iniciamos este capítulo com os sistemas *software DSM*, e em seguida, apresentamos os sistemas *hardware DSM* e híbridos. Finalizamos com outros trabalhos que abordam soluções diversas para melhorar o desempenho de sistemas DSM.

9.1 Sistemas *Software DSM*

Os sistemas *software DSM* foram propostos como uma alternativa ao alto custo dos sistemas *hardware DSM*. Porém, esses sistemas geralmente possuem *overheads* de comunicação e manutenção de coerência, que precisam ser minimizados para torná-los atrativos. Apresentamos primeiramente alguns sistemas *software DSM* para exemplificar a diversidade entre os modelos de consistência, unidades e protocolos de coerência utilizados em suas implementações. Em seguida, discutindo mais detalhadamente os sistemas que implementam técnicas adaptativas e de tolerância a latência mais diretamente relacionados a esta tese.

O primeiro sistema *DSM* totalmente implementado em *software* foi o sistema Ivy[61]. Este sistema foi implementado considerando o modelo de consistência seqüencial, a página como unidade de coerência, suporte a único escritor e o protocolo de invalidação para manter a coerência da memória. O sistema Ivy demonstrou que sistemas *software DSM* poderiam ser uma alternativa viável aos multiproces-

sadores de memória centralizada, embora não tratasse o problema de falso compartilhamento e gerasse uma grande quantidade de comunicação.

Os sistemas *software DSM*, citados a seguir, utilizam diferentes modelos de consistência, unidades e protocolos de coerência e têm abordagens mais diferenciadas daquelas utilizadas nesta tese.

Munin[11] foi o primeiro sistema *software DSM* a implementar um modelo de consistência relaxada, o modelo *Release Consistency* (RC). O armazenamento de escritas realizado por Munin antes de uma operação de *release* resultou numa redução de comunicação substancial se comparado a um sistema DSM similar utilizando o modelo de consistência seqüencial. Além disso, Munin foi o primeiro sistema a empregar o mecanismo de *diffing* para gerenciar modificações nos dados compartilhados e a permitir ao usuário empregar múltiplos protocolos para manipular os dados com diferentes características de acesso. Muitos dos protocolos utilizados por Munin exigem do programador a identificação do padrão de compartilhamento, o que dificulta o seu uso. Em contraste, nesta tese identificamos os padrões de compartilhamento de forma dinâmica e sem a interferência do usuário.

Orca[10] é um exemplo de sistema *software DSM*, com unidade de coerência baseada em objetos que usa suporte de *hardware* para obter um mecanismo de *multicast* eficiente. Orca usa este mecanismo para prover a replicação dos objetos de dados, de forma transparente, em diferentes nós de processamento. Os dados são encapsulados em objetos especificados pelo programador, e não são permitidos dados compartilhados globalmente. O sistema de tempo de execução detecta as escritas a esses objetos, removendo a necessidade de falhas de violação de acessos ou operações de proteção de páginas.

Uma vez que todos os acessos a dados compartilhados ocorre através de mecanismos específicos fornecidos pelo objeto encapsulando o dado, o sistema facilmente detecta dados modificados e não necessita de mecanismos como *diffing*. Os mecanismos de atualização de Orca são eficientes, porém, ele requer suporte de ambos *hardware* e linguagem. As estratégias de implementação que propomos nesta tese não modificam o paradigma de programação de memória compartilhada, o que torna mais fácil a utilização de programas de aplicações já existentes, assim como fornece um ambiente de programação mais difundido.

Como alternativa ao modelo RC, o sistema Midway[12] introduziu o modelo de consistência mais relaxado *Entry Consistency* e o mecanismo de *dirty bits*. O

modelo *Entry Consistency* associa acessos a dados compartilhados a objetos de sincronização. Numa aquisição de um *lock*, *Entry Consistency* somente propaga os dados modificados associados ao *lock*. Os *dirty bits* são utilizados para detectar escritas. Através de um compilador sofisticado é inserido código extra para ativar *flags* de escrita (*dirty bits*). Sua unidade de coerência é o objeto ou região e utiliza atualização como protoloco de coerência. As diferenças principais entre Midway e a nossa proposta é que não nos baseamos em nenhum suporte de compilação ou linguagem e utilizamos o modelo de consistência LRC.

O sistema *Scope Consistency*[43] introduziu o modelo de consistência *Scope Consistency* que tal qual *Entry Consistency* necessita da associação de dados compartilhados com variáveis de sincronização. Essa associação é realizada dinamicamente pelo sistema e de forma implícita. Sua unidade de coerência é a página e utiliza uma combinação de invalidações e atualizações para tentar obter menos tráfego de mensagens e de *overheads* gerados pelas falhas de acesso. Em [43] é avaliada a utilização de atualizações automáticas com suporte de *hardware* para manter a página coerente no processador *home*. Esse sistema também pode ser implementado somente em *software*.

O sistema *software DSM* CRL (*C Region Library*)[46] utiliza o modelo de consistência *Release Consistency*, tem como unidade de coerência regiões, que são áreas de memória contíguas de tamanho arbitrário, e mantém a coerência através do protocolo de invalidações. É o programador quem define as regiões e delimita os acessos através de anotações. CRL é implementado como uma biblioteca, e não necessita de qualquer suporte especial de *hardware*, compilador ou sistema operacional.

CVM[48] foi um dos primeiros sistemas a explorar os benefícios de protocolos de consistência relaxados e *multithreading* no nível da aplicação. CVM é próximo de TreadMarks, emprega suporte a múltiplos escritores, protocolo de consistência *Lazy Release Consistency*, como mecanismo de coerência *default*, mas CVM foi escrito para permitir comparação fácil de desempenho entre diferentes protocolos.

O sistema DSM Shasta[71] permite acesso a dados com granularidade fina e permite a granularidade de coerência variar com a estrutura de dados compartilhados. Shasta implementa sua versão de controle de granularidade fina inserindo código antes de cada *load* ou *store* para verificar se o dado está disponível localmente, comunicando com outros nós se necessário como é feito em Blizzard-S[72]. Diversas técnicas têm sido adicionadas à verificação básica para reduzir o *overhead* de

tempo de execução. Embora o controle de acesso de granularidade fina forneça uma solução parcial ao problema do falso compartilhamento, estudos recentes[8, 86] têm mostrado que várias aplicações podem ter bons ganhos de desempenho com granularidades grossas. Não há um consenso sobre o tamanho da unidade de coerência que seja ideal para todas as aplicações e sistemas.

Brazos[76] é um sistema *software DSM* proposto para uma rede de computadores pessoais executando o sistema operacional Windows NT 4.0. O modelo de consistência de Brazos é baseado numa implementação de *Scope Consistency* e sua unidade de coerência é a página. Esse sistema utiliza *multicast* seletivo para reduzir o tráfego de comunicação relacionado à consistência numa sincronização global. O *multicast* é utilizado tanto para requisitar os dados remotos modificados quanto no atendimento da requisição desses dados. Brazos provê o mecanismo *early update* que utiliza o *multicast* para enviar os dados modificados antes da chegada dos processadores à barreira, baseado num conjunto que contém os processadores que acessam as páginas. Se o processador deixar de acessar a página, após um determinado limite de atualizações desnecessárias, ele é retirado do conjunto e volta ao protocolo de invalidações. Outra característica de Brazos é o seu sistema de tempo de execução *multithreaded* e distribuição das *threads* do nível do usuário nos processadores que estejam disponíveis para computação. Nesta tese não consideramos a técnica de *multithreading* em nossas implementações. As técnicas de *forwarding* e *prefetching* implementadas baseadas nas informações de FIESTA e RITMO podem se beneficar de mecanismos de *multicast*.

O protocolo AEC[74] utiliza o modelo de consistência *Entry Consistency*, sua unidade de coerência é a página e utiliza os protocolos de invalidação e atualização. Esse protocolo utiliza a técnica LAP para prever dinamicamente a ordem de transferência de *locks*, e permite o envio seletivo de atualizações. Com esta técnica tenta-se sobrepor computação útil com os *overheads* de coerência. A atualização realizada por AEC considera o envio antecipado de *diffs* enquanto em nossos estudos consideramos o envio antecipado de páginas na entrada das seções críticas. A técnica LAP proposta em AEC é geral e pode ser utilizada, por exemplo, para antecipar o envio das páginas para a saída das seções críticas.

Muitas pesquisas também tem enfocado diferentes formas de adaptação para protocolos de sistemas de memória compartilhada distribuída. Kim e Vaidya[52] propuseram um esquema adaptativo que automaticamente tenta escolher o protocolo

de coerência apropriado a aplicação. Para isso, coletam dados estatísticos do número de mensagens e da quantidade de *bytes* tranferidos em tempo de execução, para prever qual o protocolo mais conveniente baseado numa função de custo que varia com os dados coletados.

Outros trabalhos também utilizam algum tipo de mecanismo para detectar padrões de compartilhamento. Em [53] foi proposto o mecanismo *Adaptive Migratory Scheme* que é capaz de detectar padrões migratórios no sistema Quarks[51]. Esta estratégia difere de nossa implementação em vários pontos: usa um método estatístico; somente suporta modo único escritor e o tratamento de páginas migratórias é feito de modo simples com o uso de um mecanismo de *self-invalidation*, que permite apenas um leitor por página.

Os sistemas *software DSM* mais próximos das propostas desta tese são os sistemas descritos a seguir. Em ambos, são utilizadas técnicas adaptativas e de tolerância à latência ativadas em tempo de execução.

Amza *et al.*[7] propuseram uma versão de TreadMarks que dinamicamente se adapta entre único escritor e múltiplos escritores, baseado nos padrões de compartilhamento das aplicações. A diferença fundamental entre FIESTA e RITMO propostos nesta tese e a abordagem do trabalho de Amza, é o modo como os padrões de compartilhamento são inferidos. Amza identifica esses padrões através de mensagens de posse e de notificações de escrita com *version number*. As mensagens de posse aumentam a quantidade de mensagens transferidas pelo protocolo, o que não ocorre com FIESTA e RITMO. Além disso, páginas com acessos a seções críticas realizados com variáveis de *lock* diferentes ou com acessos dentro e fora de seção crítica na mesma fase são consideradas MW, enquanto RITMO pode categorizá-las como SW, MW ou MIGR dependendo da seqüência de estados. A categorização distinta dessas páginas permite que possamos utilizar técnicas de tolerância à latência mais apropriadas a essas páginas.

Monnerat e Bianchini[64], concorrentemente a este trabalho, propuseram ADSM, que é um *software DSM* baseado em TreadMarks, que se adapta dinamicamente a diferentes protocolos segundo o padrão de compartilhamento exibido pelas páginas da aplicação. ADSM utiliza um algoritmo de categorização, denominado SPC, para identificar os padrões de compartilhamento. Existem duas diferenças entre RITMO e SPC. A primeira diferença é que RITMO categoriza uma página onde ocorre uma seqüência de estados de compartilhamento com acessos dentro e fora de seção

crítica como SW, MW ou MIGR, enquanto SPC assume que estas páginas são MW. A outra diferença está nas páginas com seqüências de estados de compartilhamento com acessos a seções críticas realizados com variáveis de *lock* diferentes, onde RITMO pode categorizá-las como SW ou MIGR e SPC as mantém como MW.

Mais recentemente em [6] são investigados diferentes tipos de adaptação. Baseados num protocolo com modelo de consistência *Lazy Release Consistency* e na detecção dos padrões de compartilhamento das aplicações, eles comparam o desempenho de diferentes versões que incluem adaptação entre um escritor e múltiplos escritores, agregação dinâmica de páginas em unidades de transferência maiores e adaptação entre protocolos de invalidação e atualização. As adaptações são realizadas automaticamente em tempo de execução. A versão do protocolo com adaptação entre único escritor e múltiplos escritores é semelhante ao trabalho anterior[7]. A versão do protocolo com adaptação entre invalidações e atualizações utiliza os métodos propostos em[64] e[77]. A versão do protocolo que inclui agregação dinâmica tem um algoritmo que monitora as falhas de acessos em cada processador e agrupa as páginas segundo essas falhas. Os *diffs* para todas as páginas do grupo são requisitados na primeira falha de acesso em qualquer página pertencente ao grupo. Em nossos estudos não implementamos agregação dinâmica.

O protocolo *Lazy Hybrid* proposto por Dwarkadas *et al.*[32] adapta-se entre os protocolos de coerência de invalidação e atualização. *Lazy Hybrid* implementa o modelo de consistência *Release Consistency*, fornece suporte a mútiplos escritores e tem a página como unidade de coerência. Esse protocolo utiliza invalidações da mesma forma que TreadMarks, mas aproveita para enviar algumas atualizações na mensagem de liberação da seção crítica. As atualizações enviadas são os *diffs* que o *releaser* tem mas que ainda não foram observados pelo *acquirer*. Várias são as diferenças em relação as nossas propostas. A principal é que na técnica de *forwarding* enviamos antecipadamente, de forma seletiva, páginas enquanto *Lazy Hybrid* envia *diffs*. Além disso, nos baseamos em FIESTA e RITMO para realizar a seleção dos processadores que recebem os dados antecipadamente enquanto *Lazy Hybrid* se baseia no conjunto de processadores dominantes.

9.2 Sistemas *Hardware DSM*

Os sistemas *hardware DSM* permitem que cada processador tenha acesso direto a qualquer posição de memória da máquina. Esses sistemas alcançam bom desempenho para diversas classes de aplicações, porém o projeto é complexo, exige um longo tempo de desenvolvimento, além de apresentarem um alto custo. As diferenças entre os sistemas *hardware DSM* podem ser divididas em cinco grupos principais de acordo com o modo de implementação de coerência de memória: os que implementam coerência de páginas em *hardware*, como DDM[39] e KSR[19]; os que implementam coerência de linhas de *cache* em *hardware*, como DASH[60], Alewife[4], Origin[59], Exemplar[24] e SCI[44]; os que implementam coerência de *caches* em *software*, como T3D[26] e Multiplus[9]; os que utilizam *hardware* programável altamente otimizado para implementação do protocolo de coerência, como FLASH[57], Typhoon[70] e NUMAQ[62] e os que simplesmente não permitem o armazenamento de dados remotos em *caches* como T3E[73]. A seguir apresentamos alguns exemplos desses sistemas.

DASH foi o primeiro sistema a dar suporte de *hardware* ao modelo de consistência *Release Consistency*. Ele é organizado como uma rede de nós multiprocessadores, com coerência entre os nós mantida através de um mecanismo de diretório localizado na *interface* de rede de cada nó. A sua característica principal é o protocolo de coerência de *cache* baseado em diretório distribuído. DASH usa o protocolo de invalidação para manter a coerência, e permite atualizações em modo *pipelined*. A coerência é baseada em linhas de *cache* e portanto não sofre com problemas de desempenho potenciais associados como falso compartilhamento encontrados em sistemas DSM baseados em páginas. Origin é um sistema mais moderno, baseado em DASH, que inclui *hardware* mais sofisticado.

Alewife é um multiprocessador de grande escala que também fornece suporte para memória compartilhada e passagem de mensagens. O modelo de consistência é seqüencial e usa um diretório baseado em coerência de *cache*. Alewife usa uma estrutura de diretório, denominada LimitLESS, que dá suporte a um número pequeno de ponteiros de diretório diretamente em *hardware*. Se a quantidade de ponteiros for insuficiente, o sistema é interrompido e utilizado um mecanismo em *software* para o restante dos ponteiros. Uma outra inovação de Alewife é o uso de chaveamento do contexto de *threads* muito rápido para tolerar a latência de acesso à memória

remota.

A arquitetura de FLASH fornece suporte a uma variedade de modelos de comunicação, incluindo memória compartilhada e passagem de mensagens, através de um controlador programável, denominado MAGIC (*Memory And General Interconnect Controller*). O *chip* MAGIC contém um processador de protocolo que permite ao usuário desenvolver seu próprio protocolo de coerência, fornecendo ainda caminhos de baixa latência entre nós de processamento.

Typhoon[70] suporta memória compartilhada sobre uma rede de passagem de mensagem usando um co-processador para conectar cada processador principal à rede. Todo código do protocolo e da comunicação é manipulado pelo co-processador de comunicação, liberando o processador principal para continuar executando o código da aplicação.

No contexto de *hardware DSM*, vários pesquisadores propuseram técnicas simples para adaptação a padrões de compartilhamento, em especial nos trabalhos de [25, 78] foram avaliadas otimizações no tratamento de dados migratórios em protocolos de coerência. Em[28] foi estudado um protocolo híbrido baseado em invalidações e atualizações, no qual cada processador decide localmente se atualiza ou invalida uma linha de *cache* quando recebe uma mensagem de atualização.

9.3 Sistemas Híbridos

Uma das tendências mais dominantes nas arquiteturas paralelas desenvolvidas recentemente é a combinação de suporte de *hardware* e *software* para implementar memória compartilhada distribuída. Os sistemas SHRIMP[18], CASHEMERe[55], NCP_2[14] e SoftFLASH[33], descritos a seguir, são exemplos de sistemas híbridos que implementam protocolo de coerência em *software*, mas que possuem mecanismos de *hardware* especiais que auxiliam, de forma eficiente, a tarefa do protocolo de coerência.

O multicomputador SHRIMP é composto por PCs Pentium conectados através de uma rede similar à rede de roteamento usada no Intel-Paragon[82]. SHRIMP provê uma facilidade de mapeamento remoto de memória, segundo o qual uma página de memória local pode ser associada a uma página da memória de um outro nó. As escritas feitas a esta página são automaticamente propagadas para a página remota pelo *hardware*. Os processadores remotos então recuperam uma página in-

teira do processador que tem a página atualizada numa violação de acesso como é feito em Munin. Com esta facilidade foram desenvolvidos os protocolos *Automatic Update Releasy Consistency* e *Scope Consistency* que utilizam o conceito de *home nodes*. AURC tem mostrado fornecer benefícios de desempenho substanciais, porém é necessário um *hardware* especializado.

O protótipo CASHEMERe consiste de 32 nós construídos a partir de quatro multiprocessadores DEC 2100 4/233 interconectados através de uma rede *Memory Channel*. *Memory Channel* é ligado em um *slot* PCI, e permite aos processadores acessarem locações de memória diretamente em processadores remotos, de modo similar ao protocolo AURC utilizado em SHRIMP. O *Memory Channel* fornece um mecanismo para implementação de compartilhamento de granularidade fina na arquitetura CASHEMERe.

O sistema NCP_2 é um computador paralelo, em desenvolvimento na COPPE Sistemas/UFRJ, que utiliza um *hardware* simples e de baixo custo como suporte para implementação eficiente de protocolos de coerência de dados em *software*. O suporte de *hardware* consiste em controladores de protocolo que permitem a implementação de técnicas de tolerância à latência de comunicação e a *overheads* de coerência. Cada controlador de protocolo está asssociado a um processador principal e provê como forma de minimizar as perdas de desempenho o processamento da comunicação externa ou pedidos de coerência sem o envolvimento do processador principal, quando possível, busca antecipada de páginas e *diffs* e criação dinâmica de *diffs*.

SoftFLASH é implementado no nível do sistema operacional num *cluster* de multiprocessadores SGI, usando mecanismo de coerência em *hardware* no *cluster* para acessos locais e sistema *software DSM* para acesso a dados remotos.

9.4 Outros Trabalhos

Idealmente compiladores poderiam realizar as tarefas de paralelização de código e interação entre os processos paralelos de modo eficiente, o que permitiria aos programadores a utilização de um modelo de programação seqüencial.

Alguns compiladores para sistemas paralelos são dirigidos para esse objetivo, como HPF[54] e SUIF[40]. Porém, os compiladores paralelos atuais além de serem limitados em suas análises, em geral necessitam que o programador intervenha na geração de código para que se obtenha um bom desempenho. Por exemplo, em

HPF, é necessário que o programador especifique a distribuição dos dados, o que não ocorreria se ele programasse num modelo seqüencial.

Em [31], por exemplo, Dwarkadas *et al.* projetaram e implementaram um sistema *software DSM* que combina um compilador com o sistema de tempo de execução. O compilador insere no código chamadas para informar ao sistema de tempo de execução os padrões de acesso aos dados. O sistema de tempo de execução usa essas informações para agregar comunicação, agregar sincronização e dados numa única mensagem, eliminar *overheads* de consistência e substituir sincronização global por sincronização ponto-a-ponto quando possível.

Os compiladores para sistemas paralelos atuais, em geral, não permitem a geração de códigos eficientes, por isso a grande maioria dos sistemas utiliza programação paralela com paralelismo explícito: passagem de mensagens e memória compartilhada.

Buscando classificar aplicações, estudos anteriores propuseram diferentes conceitos de aplicações regulares e irregulares. Em[42] as aplicações são classificadas segundo o padrão de compartilhamento induzido[1]. Jiang *et al.*[45] consideram a quantidade de produtores por página. Em[17] utiliza-se a distribuição espacial e temporal das falhas de página para classificar as aplicações como regulares ou irregulares. Em nossos estudos classificamos as aplicações segundo os acessos feitos as páginas, dentro e/ou fora de seção crítica e a quantidade de variáveis de sincronização por página.

Vários trabalhos já foram desenvolvidos pesquisando meios para tolerar os *overheads* através de sua sobreposição com tempo de computação ou com outros *overheads*. A seguir citamos alguns trabalhos relativos as técnicas de *prefetching*, *forwarding*, *multithreading*, *pre-diffing* e *diff* dinâmico.

A técnica de *prefetching* já foi implementada tanto no contexto de *hardware* quanto de *software*. Como exemplo de implementações em *hardware* podemos citar[20, 16, 29, 66]. Poucas são as implementações para sistemas *software DSM*[8, 14, 17, 47, 65]. Em [14] propõe-se uma técnica de *prefetching* que usa invalidações para nortear as ações de pedido dos dados compartilhados. Em outro trabalho[17], é proposta uma técnica adaptativa que considera as falhas e os *strides* de acesso às páginas para realizar o *prefetching*. Em ambos trabalhos, as técnicas

[1] O padrão de compartilhamento induzido para uma aplicação é função do seu padrão de compartilhamento inerente (granularidade de palavra) e da granularidade espacial dos acessos de leitura e escrita feito pelos processadores na página.

são baseadas apenas em informações captadas em tempo de execução das aplicações, sem nenhum suporte de compilação ou programação. Em[65] a técnica de *prefetching* é controlada por *software*, porém as chamadas as rotinas que implementam o *prefecthing* são inseridas no código da aplicação. Estas técnicas são implementadas de modo bastante diferente da implementação proposta nesta tese. Realizamos o *prefetching*, baseados nas informações de FIESTA e RITMO, somente para páginas categorizadas como MW.

No trabalho[81] avalia-se a combinação das técnicas de *prefetching* e *forwarding*, no contexto de *hardware DSM*, para evitar falhas de acesso dentro de seções críticas. Sempre que um *lock* é requisitado, é realizado o *prefetch* para todos os dados compartilhados que serão lidos dentro da seção crítica. O *forwarding* é realizado apenas nos casos onde há espera pelo *lock*.

O envio antecipado de atualizações diretamente do produtor para o consumidor de cada dado é examinado[56]. No entanto, a análise de qual processador receberá o dado compartilhado é feita unicamente pelo compilador, o que limita o uso desta técnica a aplicações com padrão de compartilhamento estaticamente determináveis.

A técnica de *mutithreading* não faz parte do contexto desta tese, porém, assim como *prefetching*, ela já foi implementada tanto em sistemas *hardware DSM*[75, 4, 5] quanto em sistemas *software DSM*[34, 80].

A técnica de *pre-diffing* tolera o *overhead* de criação de *diffs* através da sobreposição dos *overheads* de sincronização e de comunicação com a criação deles. Essa técnica também não faz parte do escopo desta tese. Com exemplo de trabalhos que utilizam esta técnica podemos citar[79, 74, 85].

Os *diffs* dinâmicos foram propostos no sistema NCP_2[14]. O controlador de protocolos do sistema monitora o barramento de memória para observar todas as escritas a dados compartilhados. Assim, o controlador pode sobrepor a criação de *diffs* com computação útil, além de eliminar a necessidade de criação de *twins* e de comparações de páginas.

Vários trabalhos estudaram a combinação de diferentes técnicas de tolerância a *overheads* como[14, 16, 65, 38], porém diferente de nossos estudos, nenhum deles combina as técnicas de *forwarding* e *prefetching* no contexto de *software DSM*. Em um trabalho preliminar[23] avaliamos o potencial da utilização dessas técnicas conjugadas num sistema simulado com 16 processadores.

Capítulo 10

Conclusões e Trabalhos Futuros

O objetivo dessa tese foi demonstrar que a discriminação precisa da dinâmica do compartilhamento de páginas em sistemas *software DSM* permite inferir eficientemente os padrões de escrita, não só em aplicações regulares como também em aplicações irregulares ao longo de sua execução. Além disso, com base nessas informações detalhadas, demonstrar que diferentes técnicas adaptativas e/ou de tolerância à latência podem ser inseridas nos protocolos e utilizadas em grupos de páginas distintos, tornando o protocolo adaptável ao comportamento de compartilhamento dinâmico das aplicações.

Nós atingimos esses objetivos através da proposta de uma nova máquina de estados, denominada FIESTA, para sistemas *software DSM*, que, em tempo de execução, capta informações sobre os estados de compartilhamento de cada página, de modo transparente ao usuário.

Os estados gerados por FIESTA são completos para a representação de todos os estados de compartilhamento de uma página em protocolos que controlem a coerência através dos eventos de falha em acessos de leitura e escrita, invalidações, sincronizações por barreiras e *locks*, com modelo de consistência *Lazy Release Consistency*.

Baseados nas seqüências de estados de compartilhamento gerados por FIESTA, desenvolvemos um novo algoritmo de categorização (RITMO), que identifica eficazmente os padrões de escrita tanto em aplicações regulares quanto irregulares. Da mesma forma, a implementação de RITMO é transparente ao usuário.

Avaliamos o potencial de FIESTA e de RITMO inserindo, no protocolo *software DSM* TreadMarks, as diferentes técnicas de adaptação e de tolerância a latência *single writer/multiple writer*, *forwarding* e *prefetching* tanto de forma isolada quanto

de forma combinada, de acordo com a identificação dos padrões de escrita de cada aplicação.

Através da análise detalhada de oito aplicações representativas da área científica, que foram executadas sobre o simulador do sistema *software DSM* TreadMarks e sobre este mesmo simulador com a adição das nossas técnicas, mostramos que a discriminação precisa da dinâmica de compartilhamento de cada página de uma aplicação conjugada com diferentes técnicas de tolerância a latência têm o potencial de melhorar o desempenho de aplicações executadas nesses sistemas. Todas as nossas comparações foram realizadas em relação a versão original de TreadMarks.

Nossos resultados mostraram que FIESTA e RITMO têm *overhead* insignificante para as aplicações utilizadas em nossos estudos, independente de serem regulares ou irregulares. Além disso, nossos resultados demonstraram que, baseados nas informações da FIESTA e de RITMO, as técnicas implementadas com mecanismos de previsão, *forwarding* e *prefetching*, têm alta precisão.

Os ganhos obtidos com a redução dos *overheads* de espera pelos dados remotos e seus efeitos indiretos, utilizando as nossas estratégias foi substancial para a maioria das aplicações estudadas.

Em todas as aplicações obtivemos ganhos positivos de desempenho com o uso apropriado de cada técnica. Este fato confirma que a utilização de FIESTA e de RITMO conjugados com as técnicas de tolerância à latência com mecanismos de previsão, é uma opção que deve ser considerada na implementação de protocolos *software DSM*. Isso porque o potencial demonstrado indica que podemos alcançar bom desempenho sem necessidade de interferência do usuário, suporte de compilação, modificação do modelo de programação ou suporte de *hardware*.

Futuramente, pretendemos implementar nossas propostas num sistema real e realizar estudos complementares para aumentar o desempenho de sistemas *software DSM*, para uma classe mais ampla de aplicações do que a apresentada.

Podemos ainda melhorar o desempenho das classes das aplicações estudadas, se considerarmos algumas soluções para as falhas de acesso iniciais (*cold start*), granularidade de acesso, número de mensagens transmitidas e otimizações aproveitando a precisão do mecanismo de previsão. Pretendemos avaliar a utilização de FIESTA e RITMO para reduzir os *overheads* de sincronização em sistemas *software DSM*. A aplicação de FIESTA e RITMO em sistemas *hardware DSM* e *cluster* de SMPs também se constitutem em futuros caminhos de nossas pesquisas.

Referências Bibliográficas

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

[2] S. Adve and M. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6), June 1993.

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

[4] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proc. of the 22nd An. Int'l Symp. on Computer Architecture (ISCA'95)*, pages 2 – 13, June 1995.

[5] R. Alverson *et al.* The Tera Computer System. In *Proc. of the Int'l Conf. on Supercomputing*, pages 1–16, June 1990.

[6] C. Amza, A. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and Willy Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. In Proc. of the IEEE, Special Issue on Distributed Shared Memory, Spring 1999.

[7] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt Between Single Writer and Multiple Writer. In *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, pages 261 – 271, Febrary 1997.

[8] C. Amza, A. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proc. of*

the 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97), pages 90 – 99, June 1997.

[9] J. S. Aude. The Multiplus/Multiplix Project. In *Anais do Simpósio Brasileiro de Arquitetura de Computadores - PAD*, pages 581 – 586, October 1997.

[10] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEETSE*, 18(3):190–205, March 1992.

[11] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. of the 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, March 1990.

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

[13] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int'l Computer Conference(COMPCON Spring'93)*, pages 528 – 537, Febrary 1993.

[14] R. Bianchini, L. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hinding Communication Latency and Coherence Overhead in Software DSMs. In Proc. of the 7th Int'l Conf. on Architectural Support for Programming Languages and Operanting Systems, pages 198 – 209, October 1996.

[15] R. Bianchini, T. J. LeBlanc, and J. E. Veenstra. Categorizing Network Traffic in Updated-Based Protocols on Scalable Multiprocessors. In *Proc. of the 10th Int'l Parallel Processing Symposium (IPPS'96)*, pages 142–151, April 1996.

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

[17] R. Bianchini, R. Pinto, and C. L. Amorim. Data Prefetching for Software DSMs. In *Proc. of the Int'l Conference on Supercomputing'98*, pages 385 – 392, July 1998.

[18] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of the 21st An. Int'l Symp. on Computer Architecture (ISCA'94)*, pages 142 – 153, April 1994.

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

[20] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. *Proc. of the 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

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

[22] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Information in Distributed Shared Memory Systems. *ACM Trans. on Computer Systems*, 13(3), August 1995.

[23] M. C. S. Castro and C. L. Amorim. Avaliação do Potencial de Técnicas Adaptativas Conjugadas para Software DSMs. In *Anais do Simpósio Brasileiro de Arquitetura de Computadores - PAD*, pages 9 – 19, September 1998.

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

[25] A. L. Cox and R. J. Fowler. Adaptive Cache Coherence for Detecting Migratory Data Shared. In *Proc. of the 20th An. Int'l Symp. on Computer Architecture (ISCA'93)*, pages 98 – 108, May 1993.

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

[27] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programing in Split-C. In *Proc. of Supercomputing'93 (SC'93)*, pages 262 – 273, November 1993.

[28] F. Dahlgren and P. Stenström. Reducing the Write Traffic for a Hybrid Cache Protocol. In *Proc. of the 1994 Int'l Conf. on Parallel Processing (ICPP'94)*, pages 166 – 173, August 1994.

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

[30] M. Dubois, C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. of the 13th An. Int'l Symp. on Computer Architecture (ISCA'86)*, pages 434–442, June 1986.

[31] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Iintegrated Compile-Time/Run-Time Software Distributed Shared Memory Systems. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 186–197, October 1996.

[32] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proc. of the 20th An. Int'l Symp. on Computer Architecture (ISCA'93)*, May 1993.

[33] A. Erlichson, N. Nuckolls, G. Chesson, and J. L. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 210–220, October 1996.

[34] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 201–213, November 1994.

[35] K. Gharachorlo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors. In *Proc. of the*

4th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), pages 245–257, April 1991.

[36] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessor. In *Proc. of the 17th An. Int'l Symp. on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

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

[38] A. Gupta, J. Henessy, K. Gharachorloo, T. Mowry, and W. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proc. of the 18th Annual Int'l Symp. on Computer Architecture*, pages 254–263, May 1991.

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

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

[41] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, pages 14 – 25, Febrary 1996.

[42] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proc. of the 23rd An. Int'l Symp. on Computer Architecture (ISCA'96)*, pages 122–133, May 1996.

[43] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: A Bridge Between Release Consistency and Entry Consistency. In *Proc. of the 8th ACM Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 122 – 133, June 1996.

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

[45] D. Jiang, H. Shan, and J. Pal Singh. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors. In *Proc. of the 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 217 – 229, June 1997.

[46] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP'95)*, pages 213–228, December 1995.

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

[48] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proc. of the 16th Int'l Conference on Distributed Computing Systems*, May 1996.

[49] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th An. Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[50] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the 1994 Winter Usenix Conference*, January 1994.

[51] D. R. Khandekar. *QUARKS: Distributed Shared Memory as a Building Block for Complex Parallel and Distributed System.* PhD thesis, The University of Utah, United States, Mach 1996.

[52] J. H. Kim and N. H. Vaidya. A Cost-Comparasion Approach for Adaptive Distributed Shared Memory. In *Proc. of the Int'l Conference on Supercomputing'96*, pages 44–51, May 1996.

[53] J. H. Kim and N. H. Vaidya. Adaptive Migratory Scheme for Distributed Shared Memory. In *Proc. of the Int'l Conference on Supercomputing'97*, July 1997.

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

[55] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, Dwarkadas S, and M. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networky. In *Proc. of the 24th An. Int'l Symp. on Computer Architecture (ISCA'97)*, pages 157 – 169, May 1997.

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

[57] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. of the 21st An. Int'l Symp. on Computer Architecture (ISCA'94)*, pages 302 – 313, April 1994.

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

[59] J. Laudon and D. Lenoski. The SGI Origin: A CC-NUMA Highly Scalable Server. In *Proc. of the 24th An. Int'l Symp. on Computer Architecture (ISCA'97)*, pages 241–251, June 1997.

[60] D. E. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th An. Int'l Symp. on Computer Architecture (ISCA'90)*, pages 148–159, May 1990.

[61] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, pages 94 – 101, August 1988.

[62] T. Lovett and R. Clapp. STING: A CC-NUMA Computer System for the Commercial Marketplace. In *Proc. of the 23rd An. Int'l Symp. on Computer Architecture (ISCA'96)*, pages 308– 317, May 1996.

[63] L. R. Monnerat and R. Bianchini. Adsm: A Hybrid DSM Protocol that Efficiently Adapts to Sharing Patterns. Technical Report ES-425/97, COPPE/UFRJ, March 1997.

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

[65] T. Mowry, C. Chan, and A. Lo. Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory. In *Proc. of the Fourth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-4)*, pages 300 – 309, Febrary 1998.

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

[67] L. M. Ni and P. K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, 26(2):62–76, Febrary 1993.

[68] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issue and Algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[69] V. S. Pai, P. Ranganathan, S.V. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 12–23, October 1996.

[70] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. of the 21st An. Int'l Symp. on Computer Architecture (ISCA'94)*, pages 325–337, April 1994.

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

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

[73] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 26–36, October 1996.

[74] C. B. Seidel, R. Bianchini, and C. L. Amorim. The Affinity Entry Consistency Protocol. In *Proc. of the 1997 Int'l Conf. on Parallel Processing (ICPP'97)*, pages 208 – 217, August 1997.

[75] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proc. of the 1978 Int'l Conf. on Parallel Processing*, 1978.

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

[77] W. E. Speight and J. K. Bennett. Using multicast and Multithreading to Reduce Communication in Software DSM Systems. In *Proc. of the Fourth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-4)*, pages 312–322, Febrary 1998.

[78] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proc. of the 20th An. Int'l Symp. on Computer Architecture (ISCA'93)*, pages 108 – 118, May 1993.

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

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

[81] P. Trancoso and J. Torrellas. The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding. In *Proc. of the 1996 Int'l Conf. on Parallel Processing (ICPP'96)*, pages 79–86, August 1996.

[82] R. Traylor and D. Dunning. Routing Chip Set for the Intel-Paragon Parallel Supercomputing. In *Proc. of Hot Chips'92 Symposium*, April 1992.

[83] J. E. Veenstra and R. J. Fowler. MINT: A Front end for Efficient Simulation of Shared-Memory Multiprocessors. In *Proc. of the 2nd Int'l Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1994.

[84] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd An. Int'l Symp. on Computer Architecture (ISCA'95)*, pages 24–36, May 1995.

[85] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, October 1996.

[86] Y. Zhou, L. Iftode, K. Li, J. P. Singh, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proc. of the 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 193–205, June 1997.