ADAPTAÇÃO EFICIENTE A PADRÕES DE COMPARTILHAMENTO EM SOFTWARE DSMS

Luiz Rodolpho Rocha Monnerat

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

Aprovada por:

Prof. Ricardo Bianchini, Ph.D.

Prof. Cláudio Amorim, Ph.D.

Prof. Jairo Panetta, Ph.D.

Prof. Sérgio Takeo Kofuji, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
DEZEMBRO DE 1997

MONNERAT, LUIZ RODOLPHO ROCHA

Adaptação Eficiente a Padrões de Compartilhamento em Softwares DSMs [Rio de Janeiro] 1997

IX, 71p, 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1997)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1.Memória Compartilhada Distribuída

I.COPPE/UFRJ II.Título(série)

A Sílvia, Bruno e Rafael

Agradecimentos

Gostaria inicialmente de agradecer aos meus pais por todo o apoio que me deram.

À Petrobras pela oportunidade e pelo apoio. Agradeço a todos os meus colegas de trabalho, em particular aos meus amigos petroleiros Alexandre Korowajczuk, Carlos Augusto, Marcelo Estellita, Paulo Fernando, Paulo Góes, Paulo Osório e Sílvio Sinedino.

Ao Laboratório Nacional de Computação Científica (LNCC), que me forneceu acesso ao IBM SP2.

Ao meu amigo Luiz Maltar pelo apoio dado nos momentos mais complicados.

A Ricardo Bianchini, pela paciência e dedicação.

A Sílvia, por tudo.

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

ADAPTAÇÃO EFICIENTE A PADRÕES DE COMPARTILHAMENTO EM SOFTWARE DSMS

Luiz Rodolpho Rocha Monnerat

Dezembro de 1997

Orientador: Ricardo Bianchini

Programa: Engenharia de Sistemas e Computação

Nesta tese está sendo introduzido o sistema ADSM, um software DSM que se adapta entre diferentes protocolos segundo o padrão de compartilhamento exibido pelas páginas da aplicação paralela. ADSM usa um novo algoritmo, chamado de SPC, para categorizar o tipo de compartilhamento de cada página. SPC categoriza eficientemente as páginas como migratórias, produtor/consumidor e falsamente compartilhadas. Páginas migratórias e produtor/consumidor são tratadas em modo único escritor e podem sofrer atualizações, enquanto páginas falsamente compartilhadas são tratadas em modo múltiplos escritores e sob protocolo de invalidação. Experimentos realizados em um multicomputador IBM SP2 com oito nós mostraram que ADSM é superior a TreadMarks em até 155%, sendo ainda superior a uma versão de TreadMarks que também se adapta a padrões de compartilhamento em até 67%. As principais conclusões da tese são que nossas estratégias de categorização e de adaptação podem melhorar significativamente o desempenho de software DSMs, enquanto ADSM é uma opção eficiente para computação paralela de baixo custo com modelo de programação de memória compartilhada.

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

EFFICIENTLY ADAPTING TO SHARING PATTERNS IN SOFTWARE DSMS

Luiz Rodolpho Rocha Monnerat

December of 1997

Advisor: Ricardo Bianchini

Department: Computing Systems Engineering

This thesis introduces the ADSM system, a software DSM that constantly and efficiently adapts to the parallel application's sharing patterns. Adaptation is based on a dynamic categorization of the sharing experienced by each page. This categorization is made by a novel algorithm (SPC), which is able to categorize pages as migratory, producer/consumer and falsely-shared efficiently. Migratory and producer/consumer pages are managed in single-writer mode and may be updated, while falsely-shared pages are managed in multiple-writer mode and under an invalidate protocol. We performed experiments with eight parallel applications on an 8-node SP2 system. The results show that ADSM can improve the TreadMarks speedups by as much as 155%, while surpassing a version of TreadMarks that also adapts to sharing patterns by as much as 67%. Our main conclusions are that our categorization and adaptation strategies are useful for improving the performance of page-based software DSMs, while ADSM is a highly-efficient option for low-cost parallel computing.

Índice

1 Introdução **1**

1.1 Tese 4

1.2 Roteiro 5

2 O Paradigma de Memória Compartilhada **6**

2.1 O Modelo de Programação de Memória Compartilhada 6

2.2 Modelos de Consistência de Memória 8

2.2.1 Modelo de Consistência Seqüencial 9

2.2.2 Modelos Relaxados de Consistência 9

3 Memória Compartilhada Distribuída em Software **13**

3.1 Protocolos de Atualização ou Invalidação 14

3.2 Suporte a Múltiplos Escritores 15

3.3 TreadMarks 16

3.3.1 Locks 16

3.3.2 Barreiras 17

3.3.3 Falhas de Leitura 17

3.4 TreadMarks Adaptativo 18

4 ADSM **20**

4.1 Uma Visão Geral 20

4.2 Estratégias de Adaptação 21

4.3 Falhas de Acesso 22

5 Caracterização do Compartilhamento das Páginas **24**

5.1 Estados e Posse de Páginas 24

5.2 Transições de Estados 25

5.3 Discussão 27

6 Metodologia **29**

6.1 Ambiente de Testes 29

6.2 Aplicações 29

7 Resultados **33**

7.1 Speedup 33

7.2 Overheads de Comunicação 35

7.3 Overheads de Coerência 38

7.4 Overheads de Memória 39

7.5 Overheads nos Acessos a Dados 41

7.6 Detalhamento dos Tempos de Execução 42

7.7 Discussão 44

8 Trabalhos Relacionados **46**

8.1 Modelos de Programação Amigáveis 46

8.1.1 Modelo (Quase) Seqüencial 46

8.1.2 Modelo Paralelo de Memória Compartilhada 47

8.2 Modelos de Consistência Relaxada 49

8.3 Sistemas Híbridos e Adaptativos 51

8.4 Granularidade da Unidade de Coerência 53

9 Conclusões **55**

Lista de Figuras

1.1 Supercomputadores instalados em 1997 2

2.1 Incremento de um contador. 6

2.2 Incremento de um contador em linguagem assembly 6

2.3 Resultado da incrementação do contador. 7

2.4 Incremento de um contador dentro de seção crítica. 8

3.1 Software DSM. 14

5.1 Diagrama de estados de SPC. 25

7.1 Speedups 34

7.2 Número de bytes transferidos 35

7.3 Número de mensagens transferidas 36

7.4 Número de twins gerados 39

7.5 Número de diffs gerados 40

7.6 Overhead de memória 41

7.7 Número de falhas de acesso 42

7.8 Detalhamento dos tempos de execução 45

Capítulo 1

Introdução

Avanços nas tecnologias de microprocessadores e interconexões vêm tornando ambientes formados por redes de estações de trabalho uma alternativa atraente para computação de alto desempenho. Estes ambientes, também conhecidos como NOWs (do inglês *Network of Workstations*), são, em geral, formados por estações que utilizam os mesmos processadores que constituem a maioria das máquinas paralelas atuais (IBM SP, SGI/Cray Origin e T3E, Sun Starfire, HP/Convex SPP). Com o aparecimento de redes locais de alto desempenho, as NOWs se tornaram computadores paralelos em potencial, em especial para a solução de problemas com demandas moderadas de comunicação.

As NOWs, bem como alguns multicomputadores escaláveis como o IBM SP e o Intel Paragon, não dispõem no entanto de ambientes de programação amigáveis que permitam a sua utilização como uma máquina paralela única. Os ambientes de programação mais utilizados em NOWs, *Parallel Virtual Machine* (PVM) [GBD+94] e *Message Passing Interface* (MPI) [SOHL+96], oferecem ao programador uma solução baseada em troca de mensagens. Ao contrário da tendência atual, que procura fornecer um ambiente mais simples de programação através de linguagens para programação paralela ou do paradigma de memória compartilhada, o modelo de troca de mensagens adiciona uma complexidade extra à já complexa tarefa da programação paralela.

A tendência em direção a sistemas com suporte a modelos amigáveis fica clara a partir de estatísticas coletadas da versão mais recente da lista dos quinhentos maiores supercomputadores (TOP500) [DMS97]. A figura 1.1 mostra a distribuição percentual dos 207 computadores desta lista instalados este ano (1997) em termos

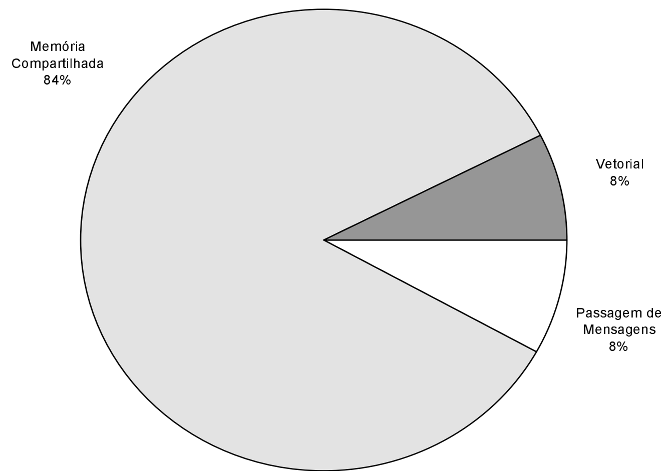

Figura 1.1: Distribuição dos novos supercomputadores instalados em 1997 segundo modelo de programação [DMS97].

de modelo de programação provido pelo hardware. Fica evidente, a partir desta figura, que há atualmente uma forte tendência em direção a máquinas com suporte ao modelo de programação de memória compartilhada. A vertente mais promissora de máquinas deste tipo é baseada em arquiteturas de memória compartilhada distribuída, ou *Distributed Shared Memory* (DSM) [PTM96]. Estes sistemas unem a facilidade de programação do modelo de memória compartilhada à escalabilidade de ambientes distribuídos. *Hardware DSMs* [TM94a, TM94b], por exemplo, usam hardware altamente especializado para implementar o modelo de programação de memória compartilhada em ambientes distribuídos. No entanto, estas soluções são em geral caras e complexas, além de não serem passíveis de implementação em NOWs já existentes.

Em contraste com hardware DSMs, implementações em software, conhecidas como *Software DSMs* [NL91], podem ser implementadas em NOWS e multicomputadores, já que não requerem hardware especial, provendo assim uma solução de baixo custo para computação paralela com paradigma de memória compartilhada. A implementação de software DSMs eficientes não é, no entanto, uma tarefa simples. Além disto, várias aplicações executando sob software DSMs não atingem níveis de desempenho comparáveis com versões usando passagem de mensagens [LDCZ95].

Buscando otimizar o desempenho de software DSMs, vários protocolos têm sido propostos [NL91, SZ90], sendo que, em geral, nenhum protocolo é ideal para todas as aplicações. Por exemplo, de maneira a minimizar os efeitos de falso compartilhamento, é comum em sistemas com unidades de coerência grandes o uso de suporte a múltiplos escritores concorrentes por unidade de coerência [BCZ90]. Esta técnica, no entanto, introduz alguns *overheads* que podem ser evitados nas situações em que efetivamente não ocorra falso compartilhamento. Visto que aplicações podem se beneficiar tanto de protocolos de único como de múltiplos escritores, técnicas que se adaptam a diferentes padrões de compartilhamento se mostram essenciais. Se esta adaptação for feita de maneira eficiente, pode-se alcançar as vantagens dos dois modos de operação, limitando os seus respectivos overheads ao mínimo necessário.

Outro exemplo é a escolha do protocolo para manutenção da coerência dos dados. Protocolos de invalidação são mais amplamente empregados, mas há aplicações que podem se beneficiar do uso de atualizações. Assim, as vantagens dos protocolos de invalidação e atualização podem ser obtidas a partir de um protocolo híbrido, que seja baseado em invalidações para minimizar o tráfico de dados, mas que busque fazer atualizações que sejam potencialmente úteis. Se a seleção das atualizações a serem feitas for eficaz, é possível reduzir significativamente o número de falhas de acesso sem sobrecarregar a banda passante de comunicação do sistema.

Desta maneira, para que um software DSM atinja níveis de desempenho satisfatórios para uma ampla gama de aplicações, ele deve ter a habilidade de se adaptar entre diferentes protocolos de acordo com as características da aplicação que está sendo executada. É importante também que esta adaptação seja feita de modo transparente ao programador, uma vez que a facilidade de programação é o principal atrativo de software DSMs.

Nesta tese está sendo apresentado um novo software DSM adaptativo, batizado de *Adaptive* DSM [MB98], ou simplesmente ADSM, que mantém a coerência da memória a nível de páginas e se adapta entre diferentes protocolos de acordo com o padrão de compartilhamento exibido pela aplicação. Toda a adaptação em ADSM é feita de modo eficiente e transparente ao programador e sem auxílio de compiladores. Os padrões de compartilhamento exibidos pelas aplicações são determinados por um novo algoritmo, batizado de *Sharing Pattern Categorization*, ou simplesmente SPC, que consegue categorizar o padrão de compartilhamento das páginas de maneira

dinâmica, eficiente e transparente.

A categorização feita por SPC permite que ADSM selecione dinamicamente o protocolo a ser utilizado para cada página da aplicação. SPC categoriza as páginas como falsamente compartilhadas, migratórias e produtor/consumidor. Páginas migratórias e produtor/consumidor são tratadas em modo único escritor e podem sofrer atualizações, enquanto páginas falsamente compartilhadas são sempre tratadas em modo múltiplo escritor e sob protocolo de invalidação.

Para avaliar o sistema aqui proposto, foram realizados experimentos com oito aplicações em um multiprocessador IBM SP2 com oito nós. Comparou-se os resultados atingidos por ADSM com aqueles obtidos por TreadMarks [ACD+96b] e por uma versão de TreadMarks que também se adapta a padrões de compartilhamento [ACDZ97]. Os resultados obtidos mostraram que as otimizações introduzidas nesta tese podem melhorar o desempenho de TreadMarks consistentemente em até 155%, sendo ainda superior a TreadMarks Adaptativo em até 67%.

As principais conclusões da tese são que nossas estratégias de categorização e de adaptação podem melhorar significativamente o desempenho de software DSMs, enquanto que ADSM é uma opção eficiente para computação paralela de baixo custo com modelo de programação de memória compartilhada.

1.1 Tese

Esta tese traz como principais contribuições o desenvolvimento, a implementação e a avaliação de novos algoritmos e protocolos para a otimização de software DSMs. A implementação destas otimizações deu origem ao sistema ADSM. Mais especificamente, as contribuições que estão sendo introduzidas são:

- Esta tese apresenta um novo algoritmo, chamado de SPC, que faz a categorização do padrão de compartilhamento de páginas em aplicações executando sob software DSMs, sem a necessidade de mensagens de controle adicionais. O principal algoritmo para categorização dinâmica de padrões de compartilhamento em software DSMs apresentado na literatura [ACDZ97] requer mensagens adicionais para fazer esta categorização e não provê o mesmo nível de detalhe de SPC.

- É também apresentada uma técnica de adaptação entre protocolos de múltiplos e único escritores baseada na categorização feita por SPC. Esta técnica atingiu níveis de desempenho superiores a TreadMarks Adaptativo [ACDZ97], o único software DSM até então proposto com adaptação entre modos único e múltiplos escritores.

- Está sendo apresentada uma nova técnica que permite o uso seletivo de atualizações em software DSMs de maneira eficiente e transparente ao programador. Esta técnica baseia-se também na categorização feita por SPC e permite a adaptação dinâmica entre protocolos de atualização e invalidação em ADSM.

1.2 Roteiro

No capítulo 2 são abordados aspectos relativos ao paradigma de memória compartilhada, mais especificamente o modelo de programação e os modelos de consistência de memória. Os assuntos abordados neste capítulo são, na sua maioria, pertinentes tanto para hardware quanto para software DSMs, e mesmo para máquinas de memória centralizada. No capítulo 3 são discutidas questões da implementação em software de sistemas de memória compartilhada distribuída. São também descritos os sistemas TreadMarks e TreadMarks Adaptativo. No capítulo 4 é apresentado o sistema ADSM, com ênfase nas técnicas de adaptação introduzidas nesta tese, sendo que o algoritmo SPC é apresentado no capítulo 5. O capítulo 6 descreve o ambiente de testes e as aplicações estudadas. Os resultados experimentais são apresentados e discutidos no capítulo 7. No capítulo 8 são discutidos os trabalhos relacionados, enquanto as conclusões da tese são apresentadas no capítulo 9.

Capítulo 2

O Paradigma de Memória Compartilhada

2.1 O Modelo de Programação de Memória Compartilhada

No modelo de programação de memória compartilhada a comunicação entre os processos é feita através de acessos a um espaço de endereçamento único. Todos os processos podem acessar qualquer posição da memória compartilhada, sendo que, para evitar inconsistências, acessos conflitantes [SS88] em geral não podem ocorrer concorrentemente, isto é, não se pode permitir que dois processos escrevam simultaneamente na mesma posição de memória, ou mesmo que um processo leia uma posição que está sendo escrita por outro.

$$I \Leftarrow I + 1$$

Figura 2.1: Incremento de um contador.

$$\begin{aligned} r1 &\Leftarrow I \\ r1 &\Leftarrow r1 + 1 \\ I &\Rightarrow r1 \end{aligned}$$

Figura 2.2: Tradução para linguagem assembly do programa da figura 2.1.

Por exemplo, considere que dois processos desejem incrementar o mesmo contador na memória compartilhada, com um comando numa linguagem de alto nível como ilustrado na figura 2.1. Este comando será traduzido pelo compilador para operações da forma mostrada na figura 2.2, onde *r1* é um registrador. Caso seja permitido que dois processos executem este trecho de programa concorrentemente, resultados imprevisíveis e não determinísticos podem ocorrer. Na figura 2.3 é mostrado

um resultado possível, onde o contador I tem o valor inicial *zero* e, após ser incrementado pelos dois processos, o seu valor final é *um*. Isto ocorre porque cada processo lê a posição de memória I antes que o outro processo armazene o resultado calculado. Assim ambos os processos carregam o valor *zero* no registrador *r1* local. Como conseqüência os dois processos calculam o valor *um* como resultado da soma, e armazenam este resultado na variável I. Na verdade, diferentes execuções paralelas deste trecho de programa podem levar a resultados distintos, o que o caracteriza como não determinístico.

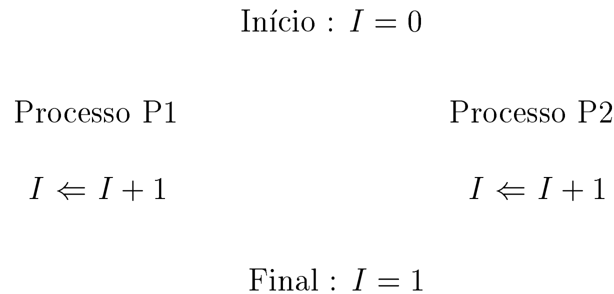

Figura 2.3: Um resultado possível da execução em paralelo do programa da figura 2.1.

De maneira a evitar fenômenos como o exemplificado acima, não se pode permitir a execução concorrente de acessos conflitantes. Para que isto seja garantido, o programador deve inserir operações de sincronização suficientes em seu programa de maneira a forçar a serialização destes acessos. Por exemplo, a classe de programas chamados de propriamente sincronizados, ou *properly-labeled programs* [GLL$^+$90], é formada por programas para os quais, entre qualquer par de acessos conflitantes, existe ao menos uma operação de sincronização entre os processos, de maneira que estes acessos sejam ordenados e não possam executar concorrentemente[1].

As operações de sincronização mais comuns são *locks* e barreiras. As operações de barreira delimitam fases da execução de um programa. Um processo só pode iniciar a fase seguinte depois que todos os processos tiverem atingido a barreira. Desta maneira, garante-se que todos os acessos feitos após a barreira são ordenados em relação àqueles feitos na fase anterior. As operações de lock são usadas para delimitar

[1]Para um programa ser considerado propriamente sincronizado é ainda necessário as operações de sincronização sejam mapeadas em *acquires* e *releases*. Maiores detalhes sobre programas propriamente sincronizados podem ser obtidos em [GLL$^+$90].

trechos do programa, chamados de seções críticas, que só podem ser executados por um processo de cada vez. Uma operação de *lock acquire* inicia uma seção crítica e um *lock release* a finaliza. O programa do exemplo acima poderia tornar-se um programa propriamente sincronizado através do uso de locks. Assim, o trecho de programa mostrado na figura 2.1 se tornaria uma seção crítica, com a adição de operações de lock acquire e release, como mostrado na figura 2.4.

ACQUIRE LOCK 1

$I \Leftarrow I + 1$

RELEASE LOCK 1

Figura 2.4: Incremento de um contador dentro de seção crítica.

É importante ressaltar que a discussão acima é válida somente para programas dos quais se espera um resultado determinístico. Há classes de programas, como por exemplo os que utilizam técnicas de relaxamento caótico, que podem permitir acessos conflitantes concorrentes, não necessitando portanto serem propriamente sincronizados.

2.2 Modelos de Consistência de Memória

Computadores seqüenciais provêem um modelo de consistência de memória muito intuitivo ao programador, onde uma operação de leitura sempre retorna o último valor escrito na respectiva posição de memória. Este modelo simples não pode, no entanto, ser diretamente extrapolado para multiprocessadores com memória compartilhada, já que nem sempre existe uma ordenação trivial e universalmente aceita entre operações executando em diferentes processadores que permita estabelecer qual foi o último valor escrito em uma posição de memória. Na verdade, a definição de último valor escrito pode depender dos mecanismos de sincronização do programa e do modelo de consistência de memória empregado no multiprocessador.

Um modelo de consistência de memória formaliza as regras que regem os acessos à memória compartilhada [AG96]. Estas regras especificam as possíveis ordenações dos acessos, determinando quais os seus resultados possíveis, como por exemplo,

quais os valores uma operação de leitura poderá retornar. Modelos que impõem menores restrições às possíveis ordenações dos acessos são chamados de modelos de consistência relaxada e têm o potencial de permitir melhores níveis de desempenho, em contraste com os modelos de consistência forte, como o modelo de Consistência Seqüencial, que são mais estritos.

2.2.1 Modelo de Consistência Seqüencial

O modelo de consistência mais intuitivo proposto procura estabelecer que a execução de um programa paralelo em um multiprocessador de memória compartilhada seja semelhante à sua execução em um uniprocessador multiprogramado. Este modelo, batizado de Consistência Seqüencial (SC), foi formalizado da seguinte maneira [Lam79]:

Definição 2.1 *Um multiprocessador é seqüencialmente c onsistente 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 respeitassem a ordem estabelecida no programa.*

Para satisfazer esta definição, em um sistema SC um acesso à memória só pode ser realizado após o acesso anterior, seja escrita ou leitura, já tiver sido *visto* por todos os outros processadores. Desta maneira, escritas à memória compartilhada podem ter um efeito similar ao de uma operação de sincronização.

O modelo SC, apesar de intuitivo, é muito restrito e impede a implementação de várias otimizações fundamentais para os processadores atuais. Otimizações comuns como *write-buffers*, *pipelines* de escrita e *overlap* de operações de memória podem violar o modelo de consistência seqüencial.

2.2.2 Modelos Relaxados de Consistência

Com o intuito de viabilizar diversas otimizações nos acessos à memória que são incompatíveis com o modelo SC, foi proposto o modelo de Consistência do Processador (PC) [Goo89]. Este modelo elimina algumas restrições do modelo SC, permitindo, por exemplo, que operações de leitura ultrapassem operações de escrita. O modelo PC garante a execução correta da vasta maioria dos programas escritos segundo

o modelo SC, em especial de programas propriamente sincronizados, e permite a construção de multiprocessadores mais eficientes e menos complexos.

O modelo *Weak Consistency* (WC) [DSB86] foi o primeiro de uma série de modelos híbridos. Estes modelos baseiam-se no fato de, como visto na seção 2.1, ser mandatório o uso operações de sincronização para ordenar acessos conflitantes à memória. Aproveitando-se deste fato, os modelos híbridos só garantem a consistência da memória compartilhada nos pontos de sincronização. Estes modelos são chamados de híbridos [Mos93], por diferenciarem os acessos de sincronização dos acessos ordinários à memória, o que permite o emprego de um modelo de consistência para os acessos ordinários e outro modelo para os acessos de sincronização. Assim, pode-se relaxar as restrições à ordenação dos acessos ordinários, mantendo-se o modelo de programação sob controle através do uso de modelos de consistência forte para os acessos de sincronização [AH90].

No modelo WC as operações de sincronização funcionam como cercas (*fences*) que obedecem ao modelo SC. Novos acessos ordinários à memória podem ser realizados mesmo que hajam outros acessos pendentes, até que uma operação de sincronização seja atingida. A cerca só pode ser terminada quando não houver mais acessos pendentes no processador, assim como o processador não pode emitir novos acessos enquanto houver uma cerca pendente.

Programas propriamente sincronizados executam corretamente sob WC [GLL+90] com potenciais ganhos de desempenho em relação aos modelos SC e PC, uma vez que há mais flexibilidade para reordenação dos acessos à memória.

No modelo *Release Consistency* (RC) [GLL+90], surgido posteriormente, os acessos de sincronização são subdivididos em *acquires* e *releases*. Um acquire corresponde a uma operação de leitura de uma variável de sincronização e um release a uma escrita. Lock acquires e releases são trivialmente mapeados em operações de acquire e release respectivamente. O mapeamento de barreiras é mais complexo e depende da estratégia de implementação utilizada. Implementações que usam gerentes fixos podem ser implementadas como um release seguido de um acquire nos processos que chegam à barreira, além de um acquire seguido de um release no gerente.

A subdivisão das sincronizações em acquires e releases permite ao modelo RC um tratamento diferenciado e mais flexível do que o empregado para as cercas do modelo WC. Um acquire é uma cerca relaxada, que não depende dos acessos prévios

e só bloqueia acessos futuros. Um release atua como uma cerca que não bloqueia acessos futuros, mas que só termina quando não existir acesso anterior pendente. Desta maneira, no modelo RC alterações feitas por um processador Pi em dados compartilhados só são vistas por outros processadores no momento de um release subsequente em Pi. Quando do release, Pi notifica todos os outros processadores do sistema que têm cópias dos dados modificados localmente, como em uma operação de *multicast*.

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 modelo foi inicialmente proposto para hardware DSMs mas têm sido também utilizado para software DSMs [BCZ90].

O modelo *Lazy Release Consistency* (LRC) [KCZ92] é uma versão relaxada e *preguiçosa* do modelo RC, suportando o mesmo modelo de programação. Em LRC a propagação das modificações feitas por Pi para outro processador Pj são postergadas até o momento de um acquire subsequente em Pj. Desta maneira, ao contrário do que ocorre no modelo RC, não há multicast das modificações no momento do release, e a comunicação só é feita entre o processador que faz o release e o que faz o acquire, com uma potencial redução da carga de comunicação. O modelo LRC foi inicialmente proposto para software DSMs, mas pode também ser implementado em hardware DSMs [KSB95].

O modelo *Entry Consistency* (EC) Consistency [BZS93] consegue relaxar a consistência de dados em relação a LRC através da associação de dados a variáveis de lock. Este modelo aproveita a relação entre as variáveis de sincronização (locks) que protegem seções críticas e os dados compartilhados que são acessados dentro destas seções críticas. Desta maneira, em uma operação de acquire de um lock, somente os dados que podem ser acessados dentro da seção crítica são feitos consistentes. Assim, a variável de sincronização que controla o acesso a uma seção crítica atua também como guarda dos dados compartilhados que podem ser acessados dentro desta seção crítica.

O relaxamento proposto pelo modelo EC traz uma potencial redução do número de mensagens de coerência e do impacto de falso compartilhamento, mas afeta o modelo de programação, já que alguns programas propriamente sincronizados podem

exigir modificações para executar corretamente sob este modelo. Programas que usam estratégia de filas de tarefas por exemplo, como Cholesky [SWG91] e Quick-Sort [KDCZ94], podem exigir alterações não triviais para executar corretamente sob o modelo EC [ACD+96a, SBA97, ISL96a]. Apesar da complexidade adicional ao modelo de programação, os ganhos de desempenho observados por implementações de EC sobre LRC [ACD+96a, MB97] não foram no entanto significativos.

Implementações tradicionais de EC [BZS93, CBA96] prejudicam ainda mais o modelo de programação, uma vez que exigem que a associação dos dados às variáveis de sincronização seja feita pelo programador. É importante notar que a definição do modelo EC não estabelece como esta associação é feita, de maneira que a exigência de que o programador as explicite não está associada ao modelo em si, e sim à sua implementação.

Capítulo 3

Memória Compartilhada Distribuída em Software

De maneira a prover a abstração de memória compartilhada em ambiente distribuído como mostrado na figura 3.1, vários software DSMs fazem uso dos mecanismos de proteção de páginas de memória virtual disponíveis na maioria dos sistemas operacionais modernos [PL93]. Páginas que contém dados que não são válidos localmente são protegidas contra leitura. Isto permite que o sistema intercepte acessos a posições da memória compartilhada inválidas localmente (falha de acesso) para assim emitir as mensagens necessárias para buscar estes dados em nós remotos (acesso remoto), de maneira a torná-los válidos.

Buscando melhorar o desempenho através da minimização do número de acessos remotos, software DSMs replicam, quando conveniente, dados compartilhados nas memórias privativas de diversos processadores. Esta replicação traz no entanto o problema de coerência de memória, já que há a necessidade de se manter a coerência das diversas cópias do mesmo dado [LH86, Ste90]. A manutenção desta coerência deve ser feita segundo um modelo de consistência memória. Software DSMs que usam modelos de consistência relaxada podem amenizar problemas de desempenho atrasando e/ou restringindo as tarefas de comunicação e coerência. Vários estudos comprovaram que quanto mais relaxado for o modelo de consistência melhor é o desempenho obtido, por exemplo [GGH91, ZB92, CPS+95, PRAH96, ZIL+97].

A implementação de software DSMs exige ainda a escolha de um protocolo para manutenção da coerência dos dados compartilhados, que pode ser de atualização ou de invalidação. Buscando minimizar os problemas de falso compartilhamento, pode-se também optar pelo emprego de suporte a múltiplos escritores. Neste capítulo

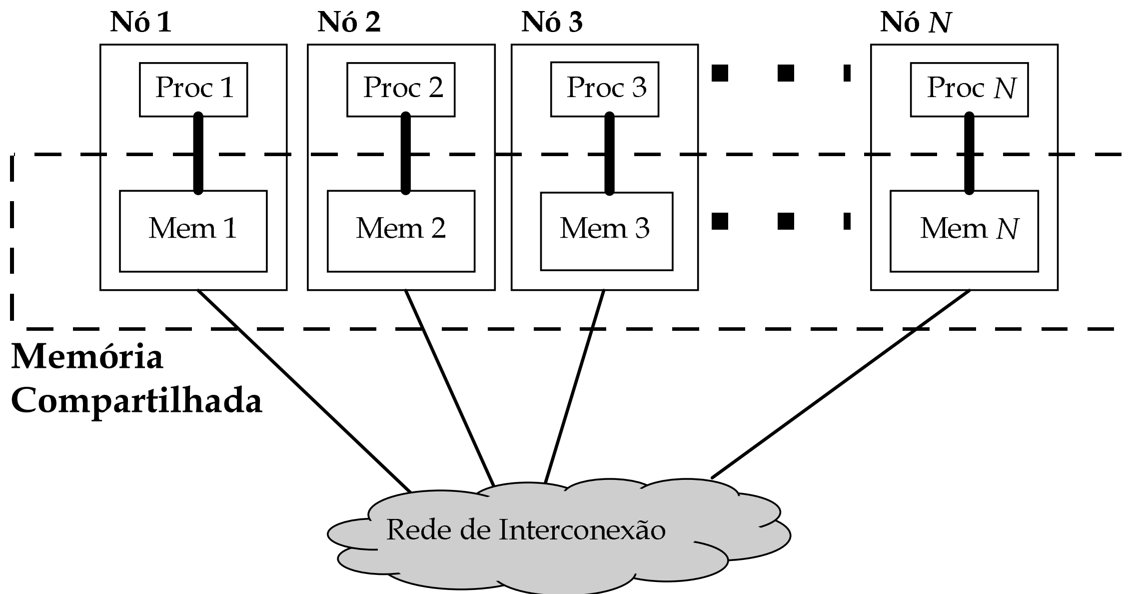

Figura 3.1: Implementação da abstração de memória compartilhada em hardware distribuído.

serão analisados estes protocolos, mostrando as vantagens de sistemas adaptativos, que têm a habilidade de selecionar dinamicamente o melhor protocolo para cada estrutura de dados dentro de uma aplicação. Serão também descritos os sistemas TreadMarks (Tmk) e TreadMarks Adaptativo (ATmk).

3.1 Protocolos de Atualização ou Invalidação

Para manutenção da coerência dos dados é necessário que escritas a posições da memória compartilhada feitas por um processador sejam propagadas para os demais processadores do sistema. Esta propagação de escritas pode ser feita por protocolos de invalidação ou de atualização, como mencionado acima.

No mecanismo de invalidação, uma modificação em um dado compartilhado feita localmente é vista em um processador remoto através de uma mensagem de invalidação. Ao receber a mensagem, o processador remoto invalida o dado modificado, de maneira que um acesso subsequente a este dado gerará uma falha de acesso, para somente então a versão atual do dado ser buscada. Já no protocolo de atualização os dados não são invalidados, uma vez que a mensagem que informa que um determinado dado foi modificado já carrega a sua nova versão, de maneira que em um

próximo acesso a este dado não ocorrerá uma falha de acesso.

Se por um lado o protocolo de atualização minimiza o número de falhas de acesso, por outro ele acarreta uma maior carga de comunicação em relação a protocolos de invalidação, uma vez que muitas vezes são feitas atualizações inúteis, i.e. atualizações a dados que não serão acessados antes que uma nova atualização seja feita [BLV96]. As vantagens de ambas as políticas podem ser obtidas a partir de um protocolo híbrido, que seja baseado em invalidações mas que busque fazer atualizações que sejam potencialmente úteis.

3.2 Suporte a Múltiplos Escritores

Devido ao uso comum de unidades de coerência grandes, em geral páginas de memória virtual, software DSMs podem apresentar graves problemas de desempenho devido a falso compartilhamento. Tais problemas podem ser minimizados através do uso de protocolos com suporte a múltiplos escritores, os quais permitem escritas concorrentes à mesma página, postergando a consolidação das atualizações para uma posterior sincronização entre os processadores.

A técnica mais utilizada para implementação de protocolos com múltiplos escritores é o mecanismo de *twinning and diffing* [BCZ90, CBZ95], que não exige suporte de compiladores. Inicialmente todas as páginas compartilhadas são protegidas contra escrita. Assim, quando um processador tenta atualizar uma página, é gerada uma falha de acesso. O software DSM intercepta esta falha de acesso, faz uma cópia da página (*twin*) e a libera para escrita. Quando se torna necessária a propagação das modificações feitas localmente nesta página, o software DSM faz uma comparação entre o twin gerado e a versão modificada da página e cria um *diff* contendo todas as modificações locais à pagina. De maneira a consolidar modificações feitas por diferentes processadores à mesma página, basta aplicar os diferentes diffs à versão local da página.

É importante notar que atualizações feitas por dois processadores concorrentemente à mesma página devem ser a diferentes posições de memória. Caso contrário, estaria sendo caracterizada uma condição de corrida (*data race*), o que não é permitido em programas propriamente sincronizados, para os quais atualizações à mesma posição de memória devem ser ordenadas através de operações de sincronização. A

ordenação imposta pelas operações de sincronização deve ser preservada quando da aplicação dos diffs à página, para evitar que uma atualização recente seja sobreposta por alguma outra atualização mais antiga à mesma posição de memória.

Se por um lado sistemas com suporte a múltiplos escritores conseguem aliviar problemas de falso compartilhamento, por outro lado esta otimização traz alguns overheads. Os custos para detectar, armazenar e consolidar modificações a dados compartilhados estão sempre presentes, independente de haver ou não falso compartilhamento. Estes custos poderiam ser eliminados para páginas que não são sujeitas a falso compartilhamento, permitindo-se que apenas um processador as atualize de cada vez. Uma vez que aplicações podem se beneficiar de ambas estratégias, técnicas que se adaptam a diferentes padrões de compartilhamento se mostram essenciais. Como diferentes estruturas de dados de uma mesma aplicação podem exibir padrões de compartilhamento distintos, é interessante também que a adaptação seja feita a nível de estrutura de dados (ou páginas) de uma mesma aplicação.

3.3 TreadMarks

TreadMarks [ACD+96b] é um software DSM baseado no modelo LRC, com suporte a múltiplos escritores através do mecanismo de twinning and diffing. A execução de um programa é dividida em intervalos que são iniciados nas sincronizações entre processadores. A cada intervalo é associado um *timestamp*, o que permite a formação de uma ordenação parcial do tipo *happens-before-1* [AH93] entre os diversos intervalos[1].

A coerência dos dados é mantida com uso de invalidações. A cada diff existe um *write notice* associado identificando em que intervalo e por qual processador o diff foi criado. O protocolo de invalidação é implementado através da propagação dos write notices em operações de locks e barreiras. A transferência de dados só é feita em falhas de acesso. As subseções seguintes detalham as operações executadas nas sincronizações por locks e barreiras e nas falhas de acesso.

3.3.1 Locks

À cada variável de lock existe um gerente estaticamente associado. Numa operação de aquisição de lock, o processador *acquirer* envia uma mensagem com seu timestamp

[1] A ordem happens-before-1 é o fecho transitivo da ordem do programa e da ordem das sincronizações. Maiores detalhes sobre a ordem happens-before-1 podem ser obtidos em [AH93].

para o gerente do lock, que a direciona para o último processador que fez release deste lock (*releaser*). Se o lock não está disponível, o pedido é enfileirado para ser servido futuramente.

Quando o lock está disponível, o processador releaser envia para o acquirer uma mensagem contendo todos os write notices que o acquirer ainda não recebeu, i.e. todos os write notices que foram gerados após o timestamp enviado pelo processador acquirer, segundo a ordem parcial happens-before-1. Ao receber a mensagem, o processador acquirer incorpora os write notices recebidos, invalidando as respectivas páginas.

3.3.2 Barreiras

Em uma sincronização por barreiras, cada processador toma conhecimento de todas as modificações feitas pelos demais processadores. Para cada barreira há também um gerente estaticamente selecionado. Ao chegar a uma barreira, cada processador envia ao gerente uma mensagem de chegada, contendo o seu timestamp e todos os write notices gerados localmente que o gerente ainda não recebeu.

A cada mensagem de chegada recebida o gerente incorpora os write notices nela contidos, invalidando as respectivas páginas. Após receber todas as mensagens de chegada, o gerente envia, para os demais processadores do sistema, mensagens de partida contendo os write notices que estes ainda não receberam. Ao receber a mensagem de partida, cada processador incorpora os write notices, invalidando as respectivas páginas.

3.3.3 Falhas de Leitura

Uma falha de leitura exige que o processador busque um conjunto de diffs para tornar a página válida. Como os diffs necessários podem ter sido gerados por diferentes processadores, pode ser necessária a comunicação com mais de um processador para tornar a página válida.

De maneira a minimizar o número de mensagens, o processador procura determinar se existem write notices dominantes entre os pendentes para a página, segundo a ordenação parcial entre os intervalos [2]. Se este for o caso, o processador que gerou

[2] Um diff *d1 domina* um outro diff *d2* da mesma página, quando *d2* precede *d1* segundo a ordem happens-before-1.

o write notice dominante deverá obrigatoriamente ter todos os diffs dominados por este, bastando então uma mensagem a este processador para requerer estes diffs (dominante e dominados). Maiores detalhes sobre TreadMarks podem ser obtidos em [KDCZ94].

3.4 TreadMarks Adaptativo

Em um trabalho recente [ACDZ97], Amza *et al.* propuseram uma versão de TreadMarks que dinamicamente se adapta a protocolos de único (SW) e múltiplos (MW) escritores, conhecido como Adaptive TreadMarks, ou simplesmente ATmk.

O protocolo MW de ATmk usa o mesmo mecanismo de twinning and diffing de TreadMarks. Já o protocolo SW é uma extensão do utilizado pelo sistema CVM [Kel96], e usa o conceito de posse e versões de páginas. Só é permitido um escritor (o dono) por página SW a cada momento. Quando um processador obtém a posse de uma página SW, ele incrementa a versão da página e cria um *owner write notice.* Os owner write notices são similares aos write notices de TreadMarks mas não têm diffs associados, sendo também agrupados em intervalos e propagados em operações de acquire.

Ao incorporar um owner write notice o processador invalida a respectiva página. Assim, em uma subsequente falha de leitura, a nova versão da página será buscada a partir do processador que gerou o último owner write notice recebido, sem que haja transferência de posse.

A posse das páginas é sempre e somente transferida em falhas de escrita. Numa falha de escrita o processador deverá requerer a posse da página ao dono corrente, além da sua versão atual caso ela esteja inválida localmente. Desta maneira, ao contrário do que normalmente ocorre em software DSMs, falhas de escrita a páginas válidas localmente podem implicar em trocas de mensagens.

A adaptação entre protocolos de único e múltiplos escritores é dinâmica e feita a nível de página. Um processador que trata uma página como SW acompanha o seu padrão de acesso, e muda a página para modo MW caso falso compartilhamento seja detectado. Da mesma maneira, um processador muda uma página de modo MW para SW caso seja detectada a ausência de falso compartilhamento.

O princípio básico que norteia a detecção de falso compartilhamento de uma

página SW é: não há falso compartilhamento de uma página se e somente se o processador que sofre uma falha de escrita para a página sabe exatamente quem é o seu dono atual e qual é a sua versão corrente. A ausência de falso compartilhamento para uma página MW é caracterizada quando existe um write notice que domina, segundo a ordem parcial happens-before-1, todos os demais write notices para a página.

Maiores detalhes sobre ATmk podem ser encontrados em [ACDZ97]. Neste trabalho é também apresentada uma versão de TreadMarks que se adapta de acordo com a granularidade das modificações feitas às páginas. Esta versão não foi estudada nesta tese uma vez que ela não trouxe ganhos relevantes em relação à versão que se adapta somente de acordo com o padrão de compartilhamento.

Capítulo 4

ADSM

Neste capítulo é apresentado o sistema Adaptive DSM (ADSM), começando por uma visão geral. As estratégias de adaptação utilizadas por ADSM são detalhadas nas seções posteriores.

4.1 Uma Visão Geral

ADSM foi construído a partir de TreadMarks e, a menos de quando citado, executa as mesmas operações de TreadMarks nas sincronizações por locks e barreiras. Já o tratamento de falhas de acesso é feito de maneira bem diferente pelos dois sistemas. Além disto, ADSM se adapta entre protocolos de invalidação e atualização e entre protocolos de múltiplos e único escritor de maneira eficiente e transparente.

Os dois tipos de adaptação empregados em ADSM são baseados na estratégia de categorização do compartilhamento de páginas proposta nesta tese (SPC), que é descrita em detalhes no próximo capítulo. SPC baseia-se na associação entre cada variável de lock e as páginas que sofrem falhas de acesso dentro das seções críticas protegidas por esta variável.

ADSM determina as páginas que sofrem falhas dentro de seções críticas dinamicamente. Uma falha de escrita dentro de uma seção crítica leva ADSM a criar um write notice associado à página e à variável de lock possuída no momento. No caso de locks aninhados, o write notice é associado ao lock mais recentemente adquirido. Uma falha de escrita fora de seções críticas é representada por um write notice sem lock associado.

Em ADSM existem dois tipos de write notices, aqueles associados a páginas com um único escritor (*SW write notices*) e os associados a páginas com múltiplos

escritores (*MW write notices*), sendo que somente o segundo tipo tem um diff associado.

Write notices são associados a intervalos, os quais são iniciados nas operações de sincronização. Como em TreadMarks, os processadores em ADSM mantém timestamps associados aos intervalos, que formam uma ordenação parcial do tipo happens-before-1. Estes timestamps também permitem a ADSM manter ordenações totais para os intervalos segundo variáveis de lock e segundo processadores.

4.2 Estratégias de Adaptação

A adaptação entre protocolos de múltiplos e único escritores é baseada na categorização das páginas feita por SPC, que as classifica como migratórias, múltiplos escritores e produtor/consumidor. Páginas migratórias e produtor/consumidor são tratadas em modo único escritor (SW), e as demais são tratadas em modo múltiplos escritores (MW).

Assim como em ATmk, em ADSM as páginas MW são tratadas segundo o mecanismo de twinning and diffing, enquanto a coerência de páginas SW é mantida transferindo-se páginas inteiras. Entretanto, em contraste com ATmk, ADSM implementa a adaptação entre modos SW e MW sem a necessidade de mensagens de posse. Além disto, a categorização das páginas feita por ADSM é mais detalhada, permitindo inclusive a adaptação entre protocolos de atualização e invalidação, adaptação esta que não pode ser diretamente implementada em ATmk.

ADSM usa protocolo de atualização seletivo, i.e. de maneira a minimizar a carga de comunicação, a maioria dos tipos de dados é tratada segundo protocolo de invalidação, usando-se atualizações somente para páginas SW que tenham um grande potencial para se beneficiar desta estratégia. Páginas SW que são tratadas segundo protocolo de atualização são páginas migratórias protegidas por lock e páginas produtor/consumidor protegidas por barreiras.

A atualização de páginas SW protegidas por lock acontece em operações de lock acquire. O processador releaser determina, inspecionando as listas de write notices dos intervalos associados ao lock em questão, quais as páginas que foram modificadas sob este lock desde a última vez que o processador acquirer possuiu lock. Tais páginas sofrerão atualização. O processador releaser então envia uma mensagem ao

processador acquirer que, como em TreadMarks, além de transmitir a posse do lock, inclui os write notices associados aos intervalos que o processador acquirer ainda não recebeu. Em ADSM esta mesma mensagem informa também quais páginas sofrerão atualização. Em seguida, o processador releaser envia as novas versões das páginas em mensagens adicionais. ADSM procura agrupar várias páginas em uma mesma mensagem, até o limite permitido pela MTU (*Maximum Transfer Unit*) da rede de interconexão utilizada. Esta estratégia de atualização evita falhas de acesso dentro de seções críticas, diminuindo assim a duração das seções críticas e, por conseguinte, a contenção por locks.

Atualizações de páginas produtor/consumidor são enviadas para todos os consumidores de uma página. Este processo ocorre em sincronizações por barreira e procura fazer as atualizações em paralelo ao overhead de sincronização da seguinte maneira. Assim que um processador determina ser o produtor de uma página produtor/consumidor, ele começa a gravar as identificações dos processadores que pedem cópias da página. Ao chegar a uma barreira, o processador produtor envia a mensagem de chegada ao gerente da barreira e, em seguida, determina quais processadores devem receber as atualizações das suas páginas produtor/consumidor, para então enviar as mensagens de atualização. Cada mensagem de atualização pode transportar uma ou mais páginas, consistindo de um cabeçalho contendo o número das páginas contidas na mensagem, seguido das versões atuais destas páginas.

Apesar dos processadores consumidores não terem que esperar pela chegada das mensagens de atualização, na saída da barreira várias páginas produtor/consumidor estarão atualizadas nos seus consumidores. Assim, esta estratégia de atualização de páginas produtor/consumidor tem o potencial de reduzir os overheads devidos a falhas de leitura sem aumentar a quantidade de dados transmitidos.

4.3 Falhas de Acesso

Falhas de leitura para páginas SW e MW são tratadas de maneira diferente por ADSM. Assim como em TreadMarks, uma falha de leitura para uma página MW exige que o processador busque um conjunto de diffs para torná-la válida. Também como em TreadMarks, uma falha de escrita para uma página MW leva o processador a criar um twin para a página e um MW write notice. ADSM adicionalmente associa

o write notice ao lock em questão, caso a falha tenha ocorrido dentro de uma seção crítica.

Falhas de acesso para páginas SW são tratadas de maneira diferente. Numa falha de leitura para uma página SW, o processador deve obter uma nova cópia da página a partir do processador que criou o SW write notice mais recente recebido localmente.

Falhas de escrita para páginas SW têm também um tratamento especial. Se o processador é o dono (*owner*) da página, i.e. ele é o processador que tem permissão para ser o único escritor da página, não é necessária a criação de um twin. O processador simplesmente cria um SW write notice e desprotege a página contra escritas. Se a falha de escrita ocorreu dentro de uma seção crítica, o write notice gerado será associado ao lock em questão. Como páginas SW só podem ser modificadas pelo dono corrente, caso um processador que não seja o dono de uma página SW tente escrevê-la, ADSM irá trocar o estado da página segundo SPC. O conceito de posse em ADSM e as transições de estados serão apresentados em detalhes no próximo capítulo.

Capítulo 5

Caracterização do Compartilhamento das Páginas

A estratégia para caracterização do compartilhamento das páginas (SPC) é baseada em um diagrama de transição de estados. Como já mencionado, esta técnica usa a associação entre variáveis de lock e os dados por elas protegidos para determinação dos padrões de compartilhamento, dispensando assim mensagens extras.

5.1 Estados e Posse de Páginas

Como mostrado na figura 5.1, uma página em SPC pode estar em um dentre quatro estados: Inicial (INIT), Migratório (MIG), Múltiplos Escritores (MW) e Um Produtor/Múltiplos Consumidores (1PMC), sendo que o último estado engloba também páginas com um produtor e um consumidor (1P1C). O estado MIG é ainda subdividido em MIGi e MIGo, para páginas que sofrem falhas de acesso dentro e fora de seções críticas, respectivamente.

SPC caracteriza como MIGi as páginas que sofrem falhas de acesso dentro de seções críticas protegidas pelo mesmo lock, como MIGo aquelas páginas que somente sofrem falhas de acesso fora de seções críticas e por um processador a cada momento, e como 1PMC as páginas que são sempre modificadas pelo mesmo processador. São categorizadas como MW as páginas que não se enquadram em nenhuma das categorias anteriores. Páginas MIG e 1PMC são consideradas SW, e só podem ser modificadas pelo seu dono corrente, enquanto páginas MW não têm donos.

Enquanto páginas 1PMC têm donos fixos, a posse de páginas MIG deve ser transferida quando apropriado. SPC implementa estas transferências sem a necessidade

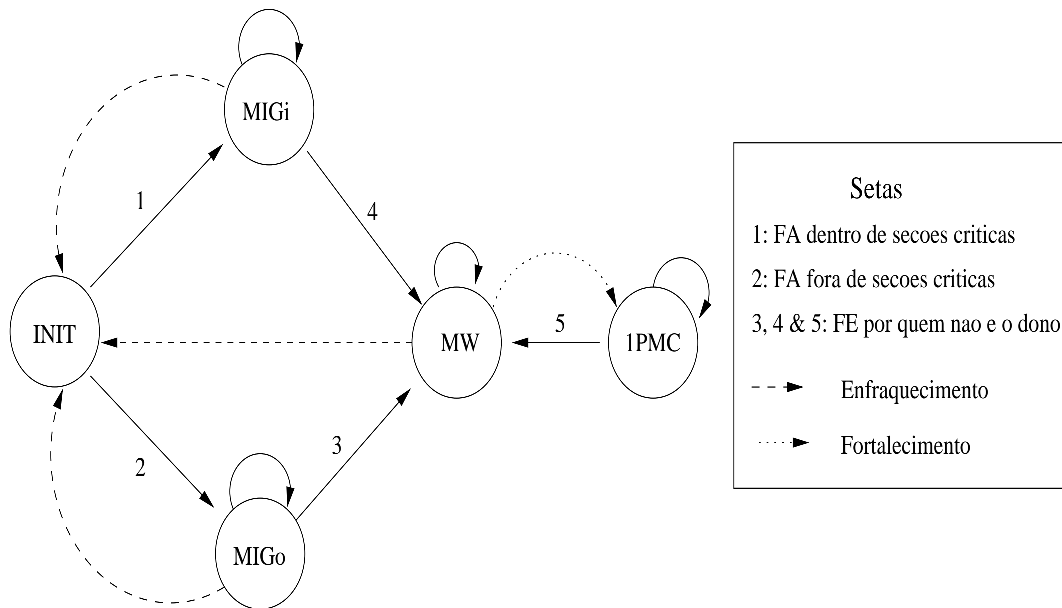

Figura 5.1: Diagrama de estado de páginas para SPC. FA = Falha de Acesso, FE = Falha de Escrita.

de mensagens extras. A posse de uma página MIGi está associada ao lock sob o qual a página tem sofrido falhas de acesso. Assim, quando um processador realiza um acquire de um lock, ele automaticamente obtém a posse das páginas associadas a este lock. Posteriormente, ao fazer release deste lock, o processador perde a posse destas páginas. Esta estratégia consegue determinar com precisão páginas SW migratórias associadas a variáveis de lock.

A posse de páginas MIGo é sempre transferida em pedidos de páginas, i.e. ao servir um pedido de página da qual é o dono, o processador transfere também a sua posse, protegendo-a contra escrita em seguida. Se o processador que está servindo o pedido não for o dono da página, a posse não é transferida. Apesar de simples, esta estratégia se mostrou eficiente na prática.

5.2 Transições de Estados

Inicialmente, todas as páginas estão no estado INIT e possuídas pelo processador 0, além de serem válidas e desprotegidas contra escrita no processador 0 e inválidas nos demais. Na primeira falha de acesso para uma página, o seu estado muda para MIGi (seta 1 na figura 5.1) se a falha ocorreu dentro de uma seção crítica ou MIGo

(seta 2) caso contrário. A posse da página é então atribuída ao processador que sofreu a falha, sendo a partir deste momento transferida segundo os mecanismos já expostos.

Tentativas de escrita a uma página pelo seu próprio dono não alteram o seu estado. Já a tentativa de escrita por um processador que não seja o dono da página levará o seu estado a ser mudado para MW (setas 3, 4 e 5 da figura 5.1), acarretando também a criação de um twin.

De maneira a se adaptar a mudanças dos padrões de compartilhamento, SPC pode em alguns casos "fortalecer" ou "enfraquecer" o estado de uma página. As técnicas de enfraquecimento permitem que SPC reclassifique uma página a partir do estado INIT. Uma página MIG é enfraquecida para INIT, mantendo-se o mesmo dono, em dois casos: a) na primeira falha de acesso sofrida por um processador que não seja o seu dono após uma barreira; ou b) no primeiro pedido de página recebido pelo seu dono após uma barreira, se a última versão criada antes da barreira ainda não tiver sido transmitida para nenhum outro processador.

O estado de uma página MW pode ser fortalecido para 1PMC ou enfraquecido para INIT. Na primeira falha de acesso (ou pedido de página) para uma página MW após a barreira, o processador determina, a partir da lista de write notices, o padrão de acesso da página durante as últimas duas fases do programa (delimitadas pelas três últimas barreiras). Se a página tiver sido modificada por um único processador, o seu estado é fortalecido para 1PMC. Se a página tiver sido modificada sob apenas um lock, o seu estado é enfraquecido para INIT. Em ambos os casos, o dono da página será aquele processador que criou o último write notice. Ao mudar o estado de uma página de MW para INIT ou 1PMC, são liberados todos os diffs e twins associados a esta página, aliviando assim a demanda por memória.

Um página pode ter estados diferentes em dois processadores distintos até que eles se comuniquem. Se a comunicação ocorre através da transferência de write notices, o tipo de write notice é relevante. Incorporar um MW write notice a uma página localmente SW muda o seu estado para MW. Incorporar um SW write notice a uma página SW não muda o seu estado. Incorporar um SW write notice a uma página MW muda o seu estado para INIT, caso não tenha sido recebido nenhum MW write notice para esta página desde a última barreira, o que significa que esta página foi enfraquecida ou fortalecida.

Se os processadores se comunicam via pedido de página as transições de estado são diferentes. O estado de uma página é mudado para MW no processador que serve o pedido (*servidor*) se: a) o estado informado no pedido de página for MW, b) a página é inválida no processador servidor (o que significa que está ocorrendo falso compartilhamento da página), c) o processador *cliente* está escrevendo em uma página que é 1PMC no processador servidor, d) o processador cliente está escrevendo em uma página que é MIGo, mas o processador servidor não é o seu dono corrente, e) o processador cliente sofreu a falha dentro de uma seção crítica mas a página é considerada MIGo pelo processador servidor, ou f) a página é considerada MIGi no processador servidor, mas a falha ocorreu dentro de uma seção crítica protegida por um outro lock. A resposta ao pedido contém não só uma cópia da página, como também os seus novos estado e dono, além do lock associado no caso de páginas MIGi. Ao final deste processo, ambos os processadores cliente e servidor conhecerão o mesmo estado e dono para a página.

A possibilidade de dois processadores atribuírem estados diferentes a uma mesma página pode levar um processador a tratar uma página como MW (computando diffs), enquanto outro processador a considera SW (gerando novas versões). Esta situação não implica em perda de coerência, uma vez que os diversos diffs podem ser condensados com a última versão SW da página, de maneira a consolidar as modificações feitas pelos diferentes processadores. Por exemplo, se um processador Pi criou um diff para uma página e recebe um SW write notice gerado pelo processador Pj, o processador Pi deve, na próxima falha de acesso à página, pedir a nova versão da página ao processador Pj e então reaplicar nesta nova versão os diffs gerados localmente.

5.3 Discussão

Uma característica importante de ADSM é a ausência de proteções contra escrita de páginas em lock acquires e releases. Esta ausência de proteção pode ocultar o fato de que algumas páginas são escritas dentro e fora de seções críticas, ou mesmo dentro de seções críticas protegidas por variáveis de lock diferentes. Uma vez que estas características das páginas estão ocultas, ADSM as classifica como SW, enquanto deveriam ser classificadas como MW de acordo com SPC. Este desvio não é, no

entanto, significativo uma vez que, se estas páginas forem realmente escritas por múltiplos processadores, SPC vai rapidamente observar este fato em pedidos de página ou através da incorporação de write notices. Na implementação de ADSM optou-se por esta redução na precisão da categorização das páginas em prol de um melhor desempenho, uma vez que foi observado que o número de páginas que seria temporariamente incorretamente classificado seria limitado para a maioria das aplicações.

É importante também ressaltar que a categorização descrita é apenas uma boa aproximação do estado real de uma página, uma vez que SPC pode temporariamente categorizar incorretamente uma página SW como MW. Por exemplo, páginas 1PMC são inicialmente incorretamente categorizadas como MW, até que este desvio seja corrigido pelo mecanismo de fortalecimento. Este efeito é resultado da estratégia simples de transmissão de posse empregada, que mostrou claramente trazer mais benefícios (na medida que evita mensagens extras) do que desvantagens, como mostraram os resultados dos experimentos.

Capítulo 6

Metodologia

6.1 Ambiente de Testes

O ambiente utilizado nos experimentos consiste de um multiprocessador IBM SP2 com oito nós *largos* (*wide nodes*) do Laboratório Nacional de Computação Científica (LNCC) no Rio de Janeiro. Cada nó é composto por um processador Power2 de 66MHz com 256 Kbytes de cache de dados, e 1GB de memória, e página virtual de 4KB. A MTU para o switch é 64KB, e os parâmetros de sistema foram ajustados de maneira a maximizar o desempenho do switch, segundo [IBM95]. Os nós são conectados através de um Omega *switch* com banda passante bidirecional de 40MB/s, e por um barramento Ethernet 10baseT. Foram feitos experimentos com ADSM, TreadMarks e ATmk. Todos os três sistemas usam comunicação baseada no protocolo UDP.

6.2 Aplicações

Foram avaliadas oito aplicações científicas paralelas. Seis destas aplicações fazem parte do pacote de distribuição de TreadMarks: 3D-FFT, Integer Sort (IS), Jacobi, SOR, TSP e Shallow. Estas aplicações têm sido utilizadas em várias avaliações de software DSMs, por exemplo [CBZ95, Kel96, ACDZ97, SDH^{+}97, SB98]. Os outros dois *benchmarks* testados são duas versões de migração sísmica 2D pós-estaqueamento [PPA94, Fig95], MigFreq e MigDepth. Na tabela 6.1 são mostrados os tamanhos de problema usados e os tempos de execução seqüencial. Todas as aplicações e protocolos foram compilados usando-se os compiladores *gcc* e *XLFortran* com otimização *-O2*.

Benchmark	Tamanho de Problema	Sincronização	Tempo Seqüencial
3D-FFT	64x64x16, 100 iterações	barreiras	55.90 segs
IS	$N = 2^{23}, Bmax = 2^{15}$, 10 iterações	locks, barreiras	23.30 segs
Jacobi	1024x1024, 100 iterações	barreiras	14.40 segs
MigDepth	512x256	locks	38.00 segs
MigFreq	512x256	locks	42.70 segs
Shallow	1024x256, 50 iterações	barreiras	12.75 segs
SOR	1024x2048, 100 iterações	barreiras	148.0 segs
TSP	18 cidades	locks	20.40 segs

Tabela 6.1: Tamanhos de problema usados nos experimentos.

MigFreq faz migração 2D pós-estaqueamento usando o método $\omega - x$. A paralelização é obtida através de particionamento por freqüências, i.e., a cada processador é atribuído um bloco de freqüências. Com esta estratégia cada processador é capaz de migrar todo o seu bloco de freqüência sem necessidade de comunicação. A comunicação é feita ao final do processamento quando todos os processadores devem acumular as freqüências migradas localmente à seção sísmica final. A serialização desta acumulação é feita com uso de uma variável de lock, sendo que a cada acquire a seção sísmica é toda rescrita. Assim, o padrão de compartilhamento é de páginas migratórias associadas a uma variável de lock. A versão aqui estudada foi desenvolvida a partir da versão de memória compartilhada apresentada em [Fig95] que usa a diretiva *DOACCROSS* para paralelização automática. Esta versão foi modificada de maneira a explicitar o paralelismo de acordo com o modelo de programação de TreadMarks.

MigDepth resolve o mesmo problema que MigFreq usando particionamento por profundidade, ao invés de por freqüência, para criação de trabalho paralelo. Esta técnica expõe uma quantidade maior de paralelismo às custas de uma carga de comunicação muito superior à obtida no particionamento por freqüência. Nesta estratégia, cada processador extrapola a seção sísmica para um determinado conjunto de profundidades. Como a migração de cada profundidade depende da anterior, a computação é feita em modo *pipeline*. Este

pipeline é controlado com uso de locks, e os padrões de compartilhamento observados são de páginas MW e páginas migratórias acessadas em seções críticas. A versão aqui estudada foi também desenvolvida a partir de uma versão que faz uso de paralelização automática [Fig95].

3D-FFT, original do conjunto NAS [BBB+94], resolve um conjunto de equações diferenciais parciais usando FFTs tridimensionais. Toda sincronização é feita por barreiras e o padrão de compartilhamento dominante é 1PMC. Assumindo uma matriz $n_1 \times n_2 \times n_3$, em cada iteração há uma primeira fase onde é feito um 1D FFT em cada um dos n_1 x n_2 vetores, para em seguida ser feito outro 1D FFT em cada um dos n_1 x n_3 vetores. Esta fase é terminada por uma barreira, e a distribuição inicial da matriz é feita de maneira que não haja comunicação de dados entre os processadores nesta fase. Na segunda e última fase de cada iteração, a matriz resultante é transposta em uma outra matriz n_2 x n_3 x n_1, para em seguida ser feito outro 1D FFT em cada um dos n_2 x n_3 vetores. Nesta segunda fase, cada processador lê (consome) dados produzidos por todos os demais processadores na fase anterior, havendo então o padrão 1PMC.

Integer Sort (IS), também original do conjunto NAS [BBB+94], classifica um vetor de N inteiros usando chaves no intervalo [0, Bmax] através a técnica *bucket sort*. As chaves são divididas pelos processadores e cada iteração consiste de três fases separadas por barreiras. Na primeira fase, o processador 0 inicializa o vetor global de *buckets*. Na segunda fase, cada processador conta suas próprias chaves armazenando resultados parciais em um vetor local para, imediatamente antes de chegar à barreira, adicionar os resultados locais ao vetor global de *buckets*. A serialização dos acessos a este vetor global é feita usando-se uma variável de lock: cada processador faz acquire do lock, adiciona os valores locais ao vetor global rescrevendo-o totalmente, e faz o release do lock. Assim, nesta fase o padrão de compartilhamento é caracterizado por páginas migratórias acessadas dentro de seção crítica. Na última fase, todos os processadores lêem o vetor global para classificar suas chaves locais, não havendo nesta fase escritas a dados compartilhados.

Jacobi resolve equações diferenciais parciais usando um método iterativo baseado em relaxações sucessivas. Cada iteração é dividida em duas fases separadas por uma barreira. Na primeira fase, é gerada uma matriz temporária a partir da matriz bidimensional de entrada. Cada elemento da matriz temporária é calculado como sendo a média de seus vizinhos na matriz original. Na segunda fase, a matriz temporária é copiada para a matriz original. Como a matriz é divida em fatias entre os processadores, o compartilhamento de dados ocorre nas bordas de cada fatia e segue padrão 1P1C.

SOR resolve um problema similar ao de Jacobi usando uma estratégia do tipo "vermelho-preto" para realizar as relaxações sucessivas, evitando assim as movimentações de dados de e para a matriz temporária. Cada iteração é composta por duas fases separadas por barreiras. Os valores da matriz vermelha são calculados na primeira fase e os da matriz preta na segunda. Assim como em Jacobi, as matrizes são dividas em fatias entre os processadores, com compartilhamento do tipo 1P1C nas bordas das fatias. De maneira a eliminar efeitos de inicialização, a primeira iteração não foi computada para efeitos de estatísticas.

Shallow é um benchmark originalmente desenvolvido pelo Centro Nacional para Pesquisas Atmosféricas do governo americano, NCAR [Sad75]. O programa resolve equações diferenciais para previsão de tempo usando treze matrizes bidimensionais. Cada iteração e composta por três fases separadas por barreiras, e em cada fase algumas das matrizes são atualizadas. O padrão de acesso varia de acordo com a fase e a matriz sendo acessada. Para o tamanho de problema usado, os padrões dominantes de compartilhamento são 1PMC e MW.

TSP, do inglês *Traveling Salesman Problem*, resolve o problema do caixeiro viajante usando um algoritmo do tipo *branch and bound*, com estratégia de fila 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 preestabelecido. O problema consiste em determinar o menor caminho que, saindo de uma cidade determinada, percorre todas as outras cidades

exatamente uma vez e retorna à cidade original. Os principais padrões de compartilhamento observados são páginas com múltiplos escritores (MW) e páginas migratórias associadas a variáveis de lock (MIGi).

Capítulo 7

Resultados

Neste capítulo são apresentados os resultados obtidos com a execução das oito aplicações testadas. São mostrados separadamente os ganhos de desempenho atingidos por cada uma das duas técnicas de adaptação introduzidas em ADSM, e são comparados estes resultados aos obtidos por TreadMarks e ATmk. São inicialmente discutidos os *speedups*, e em seguida são feitas análises detalhadas do comportamento de cada protocolo de acordo com outras métricas. Ao final são discutidos os resultados obtidos.

7.1 Speedup

Na figura 7.1 são mostrados os speedups das aplicações testadas no multiprocessador SP2 com 8 nós usando o switch de 40 MBytes/seg. Para cada aplicação são mostrados os speedups para, da esquerda para a direita, TreadMarks padrão ("Tmk"), ADSM com adaptação entre único e múltiplos escritores mas sem adaptação para invalidações/atualizações ("SMA"), ADSM completo ("SMA+IUA"), e TreadMarks Adaptativo ("ATmk").

O gráfico mostra que quatro das aplicações testadas (MigFreq, Jacobi, TSP e SOR) atingiram bons speedups com Tmk. SMA trouxe ganhos significativos de desempenho em relação a Tmk para três aplicações: 82% para MigDepth, 52% para IS e 24% para MigFreq. O ganho para FFT foi menor mais ainda significativo (14%). O desempenho das demais aplicações não foi afetado pela adaptação entre único e múltiplos escritores proposta, apesar desta técnica ter reduzido significativamente o overheads de memória e coerência em relação a Tmk. Nas próximas seções serão discutidos os overheads de memória e coerência dos diversos protocolos estudados.

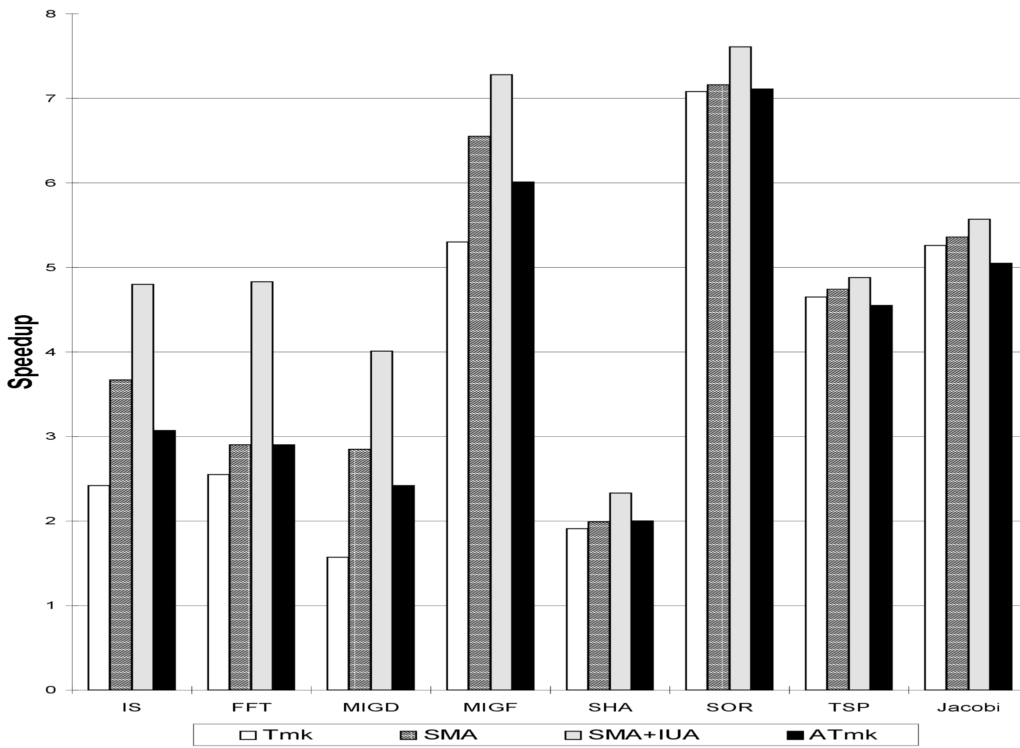

Figura 7.1: Speedups das aplicações para os protocolos estudados

Com a adição de IUA tem-se o protocolo ADSM completo. Todas as aplicações estudadas obtiveram ganhos de desempenho com uso seletivo de atualizações: IS, TSP, MigDepth e MigFreq fazem atualização de dados migratórios protegidos por locks, enquanto FFT, SOR, Shallow e Jacobi usam atualizações para dados produtor/consumidor protegidos por barreiras.

Os resultados mostram que, em geral, a adição de IUA trouxe maiores ganhos de desempenho do que SMA. Os maiores ganhos de ADSM completo em relação a SMA foram para FFT (67%), MigDepth (41%) e IS (31%). O mais importante, no entanto, é que ADSM mostrou desempenho superior a TreadMarks de maneira consistente. A diferença de desempenho entre os dois protocolos variou de 5% para TSP até 155% para MigDepth.

A comparação entre ADSM e ATmk é também favorável, já que ADSM mostrou-se superior em todas as aplicações por pelo menos 7%. Os maiores ganhos de ADSM em relação a ATmk foram para MigDepth (66%), FFT (67%) e IS (56%). Os ganhos obtidos para as outras aplicações são menores mas ainda significativos: 21% para MigFreq, 15% para Shallow, 10% para Jacobi, e 7% para SOR e TSP.

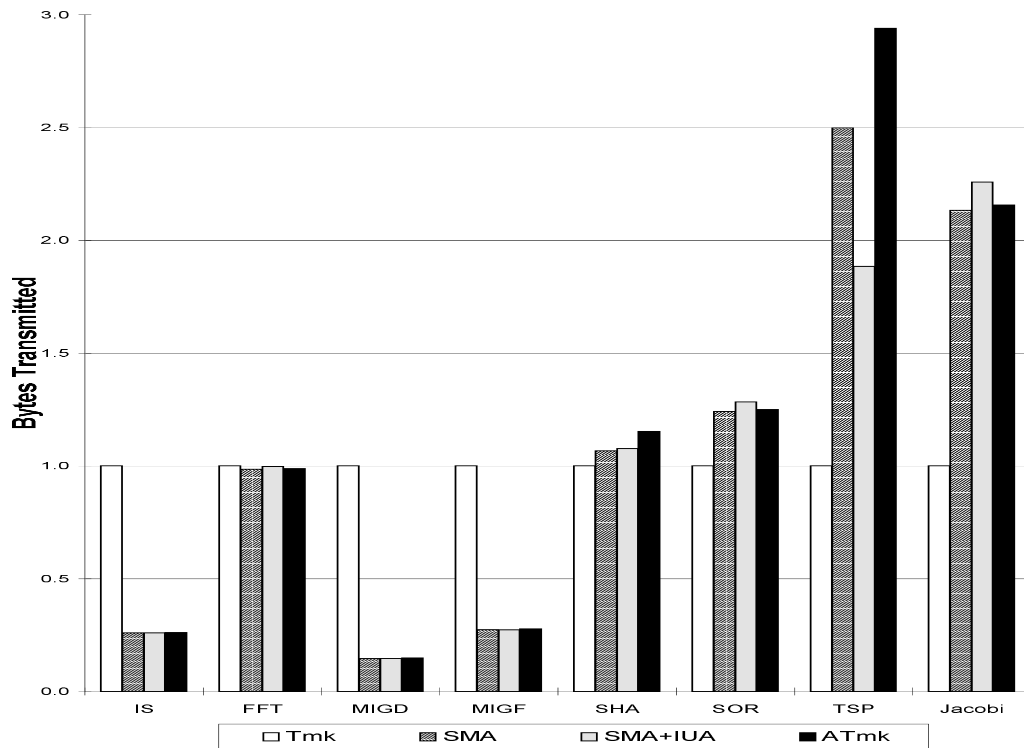

Figura 7.2: Número de bytes transferidos (normalizado em relação a Tmk).

As seções seguintes explicam estes speedups.

7.2 Overheads de Comunicação

As figuras 7.2 e 7.3 mostram o número de bytes e mensagens transmitidos para as aplicações testadas sobre os protocolos em estudo. A ordem das barras é a mesma da figura 7.1. Todas as barras foram normalizadas em relação aos resultados para Tmk.

Os resultados apresentados na figura 7.2 mostram que SMA reduziu drasticamente a quantidade de dados transmitidos para MigDepth (85%), IS (74%) e MigFreq (73%) em relação a Tmk. A razão para estes ganhos é que para estas três aplicações a maioria das páginas SW é completamente rescrita a cada vez que é acessada. Desta maneira, utilizar diffs para estas páginas acarreta desperdício de tempo, memória e, principalmente, comunicação. Este overhead de comunicação ocorre devido ao fenômeno de acumulação de diffs [LDCZ97] causado pelo caráter migratório das páginas destas três aplicações. Em ADSM estes problemas não ocorrem, uma vez que estas páginas são tratadas em modo único escritor.

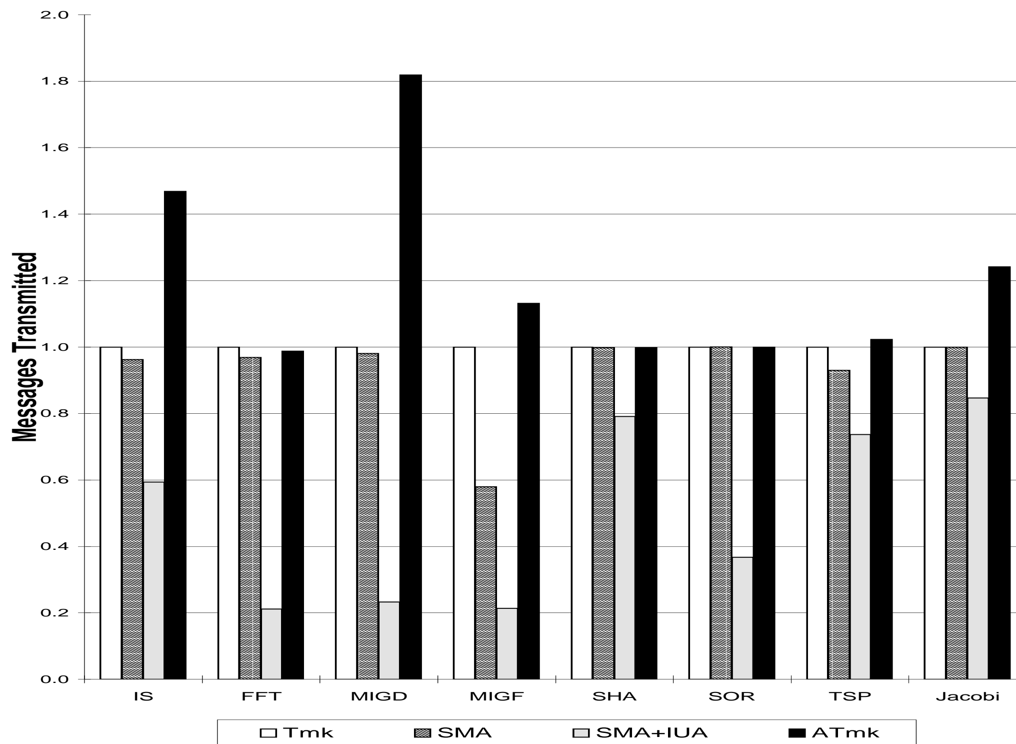

Figura 7.3: Número de mensagens transferidas (normalizado em relação a Tmk).

Para Jacobi, SMA acarreta um aumento significativo da quantidade de dados transmitidos. Apesar de nesta aplicação as páginas serem completamente rescritas sempre que acessadas por um processador, os dados de entrada usados no experimentos (zero para todos os elementos da matriz, a menos das bordas) fazem com que a maioria das escritas não modifique os valores já armazenados nas páginas (zeros), o que leva os diffs a terem tamanhos mínimos. Para este conjunto de dados de entrada específico, a transmissão de diffs gera muito menor tráfego de dados do que a transferência de páginas inteiras. Para SOR e, principalmente, TSP, os diffs gerados por Tmk são também menores que o tamanho de uma página, o que fez com que mais uma vez o uso de protocolo de único escritor aumentasse a quantidade de dados transmitidos para SMA.

A menos de TSP, a adição de IUA pouco interferiu na quantidade de dados transmitidos, o que confirma que as técnicas de atualizações seletivas aqui propostas são efetivas e não comprometem a banda passante do sistema. ATmk transfere basicamente a mesma quantidade de dados que ADSM, exceto TSP.

Para TSP, a adição de IUA diminuiu a quantidade de dados transmitidos em

relação a SMA. Este resultado, a princípio inesperado, ocorreu por que o uso de atualizações em lock acquires levou a uma mudança da caracterização das páginas desta aplicação. Muitas das páginas caracterizadas como MIGi por SMA para esta aplicação são na verdade modificadas dentro e fora de seções críticas. Como explicado na seção 5.2, ADSM não faz proteção de páginas contra escrita no momento das atualizações, o que pode levar páginas com este perfil de acesso a serem caracterizadas como MIGi. Como as páginas atualizadas sobre IUA são protegidas contra escrita no momento do release, SMA+IUA foi então capaz de determinar com mais precisão este padrão de acesso, caracterizando-as então como MW. Uma vez que para TSP os diffs são pequenos, obteve-se como conseqüência uma menor quantidade de dados transmitidos.

ATmk transfere aproximadamente a mesma quantidade de dados que SMA para todas as aplicações, com exceção de Shallow e TSP, para as quais ATmk transfere mais dados. Para estas duas aplicações ATmk categoriza corretamente algumas páginas como SW, enquanto SPC as categoriza erroneamente como MW. Este desvio se torna benéfico em termos de quantidade de dados transmitidos, umas vez que para estas aplicações o tamanho dos diffs é menor que uma página, como já mencionado.

A figura 7.3 mostra que Tmk e SMA transferem aproximadamente o mesmo número de mensagens para todas as aplicações, exceto MigFreq e TSP. Para estas aplicações, SMA reduz em 42% e 7%, respectivamente, o número de mensagens em relação a Tmk. Estas reduções ocorrem porque nestas aplicações há um grande número de falhas de leitura para as quais é necessário buscar um conjunto de diffs além de uma cópia da página. Em Tmk são necessários ao menos dois pares pedido/resposta de mensagens: um para buscar a página e o outro para buscar os diffs. Já em SMA apenas um par de mensagens é necessário, já que estas páginas são SW, não havendo portanto diffs a serem buscados. O mesmo fenômeno ocorre em IS, FFT e MigDepth só que numa extensão muito mais reduzida.

A adição de IUA acarreta grandes reduções nos números de mensagens em relação a Tmk para todas as aplicações. Estas reduções variam de 15% para Jacobi até 79% para FFT e MigFreq. Estes resultados mais uma vez confirmam a eficácia das técnicas de atualização aqui apresentadas quanto à redução da carga de comunicação.

ATmk transfere entre 10% e 95% mais mensagens do que SMA para cinco das aplicações: IS, MigDepth, TSP, MigFreq e Jacobi. Estas mensagens extras são devi-

das unicamente a transferências de posse de páginas. Para MigDepth, por exemplo, SMA transfere 76.697 mensagens, enquanto ATmk transfere 142.205 mensagens, das quais 65.224 são para transferência de posse de páginas. Para IS, SMA transfere 10.192 mensagens, enquanto ATmk transfere 15.545 mensagens, das quais 5.346 são para transferência de posse de páginas.

Em comparação a ADSM, ATmk sempre transfere mais mensagens. A diferença varia entre 26% (Shallow) até um fator de 8 (MigDepth). Em ATmk, cada falha de acesso requer um par pedido/resposta de mensagens para tornar a página válida ou, possivelmente, um par de mensagens para obter a sua posse. Já para ADSM não são necessárias mensagens para transferência de posse, e várias falhas de acesso são evitadas através de atualizações. É importante notar que no ambiente estudado uma única mensagem de atualização em ADSM pode validar até quinze páginas, além de não requerer resposta.

7.3 Overheads de Coerência

As figuras 7.4 e 7.5 mostram respectivamente os números de twins e diffs das diversas aplicações avaliadas sobre os protocolos em estudo. A ordem das barras é a mesma da figura 7.1. Todas as barras foram normalizadas em relação aos resultados para Tmk.

Os resultados mostram que SMA provê drásticas reduções nos overheads de coerência de todas as aplicações em relação a Tmk. As reduções mais significativas ocorreram para IS e MigFreq, para as quais SMA eliminou todos os diffs e twins. Para Jacobi e SOR houve também eliminação total dos diffs. Para as outras aplicações, a redução no número de twins em relação a Tmk variou de 40% (Shallow) até 99% (Jacobi). A redução do número de diffs variou de 35% (Shallow) até 97% (FFT).

A menos de TSP, a adição de IUA não causou alteração no número de twins e diffs. Para TSP, como já visto na seção 7.1, a adição de IUA levou ADSM a caracterizar como MW páginas que anteriormente eram classificadas como MIGi, o que acarretou um aumento do número de twins e diffs.

Os overheads de coerência de ADSM e ATmk são aproximadamente os mesmos para três das aplicações testadas (IS, MigDepth e MigFreq), enquanto ATmk gerou

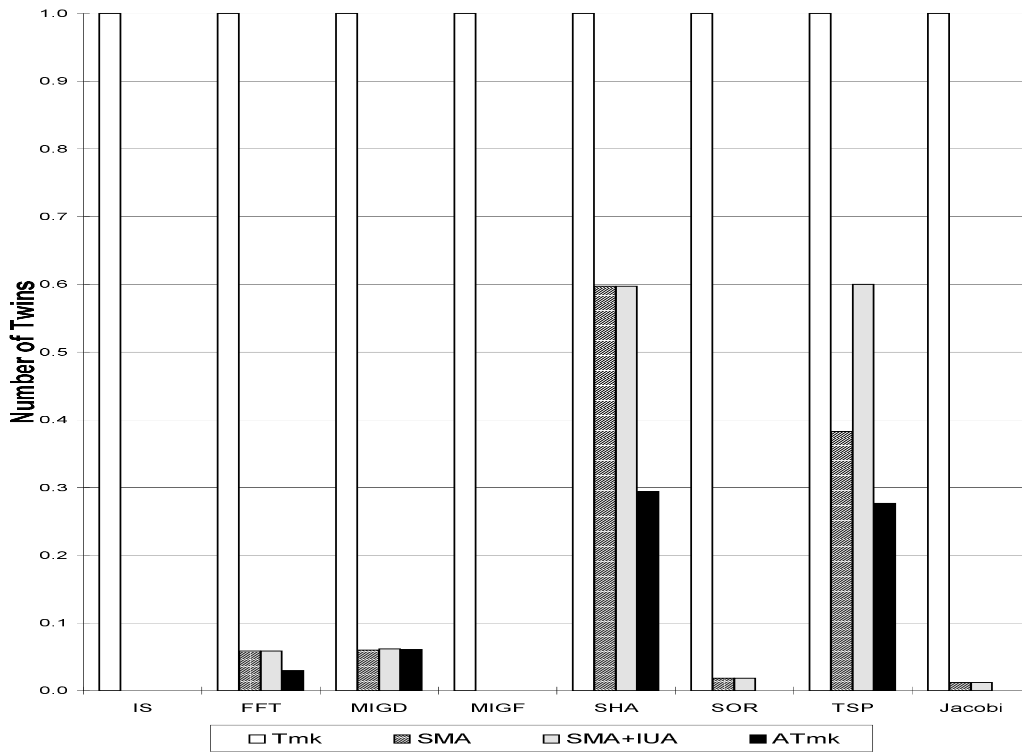

Figura 7.4: Número de twins gerados (normalizado em relação a Tmk).

menor número de twins para FFT, SOR e Jacobi, e menor número de twins e de diffs para Shallow e TSP. Para FFT, SOR e Jacobi, ATmk gera um número muito menor de twins do que ADSM, apesar do número de diffs ser o mesmo para o dois protocolos. Isto ocorre por que SPC categoriza inicialmente algumas páginas como MW, quando na verdade elas são 1PMC. Isto leva a criação de mais twins mas não de diffs, uma vez que SPC corrige este desvio após duas barreiras, antes da geração dos diffs. Para Shallow, a categorização feita por SPC não detecta que algumas páginas poderiam ser tratadas como SW e as mantém como MW, aumentando assim o número de diffs e twins. Para TSP, ADSM não consegue detectar que algumas páginas, apesar de serem atualizadas dentro e fora de seções críticas, têm caráter migratório, caracterizando-as então como MW segundo o diagrama de estados de SPC.

7.4 Overheads de Memória

Na figura 7.6 são mostrados os overheads de memória gerados por cada protocolo estudado. A ordem das barras é a mesma da figura 7.1. Todas as barras foram

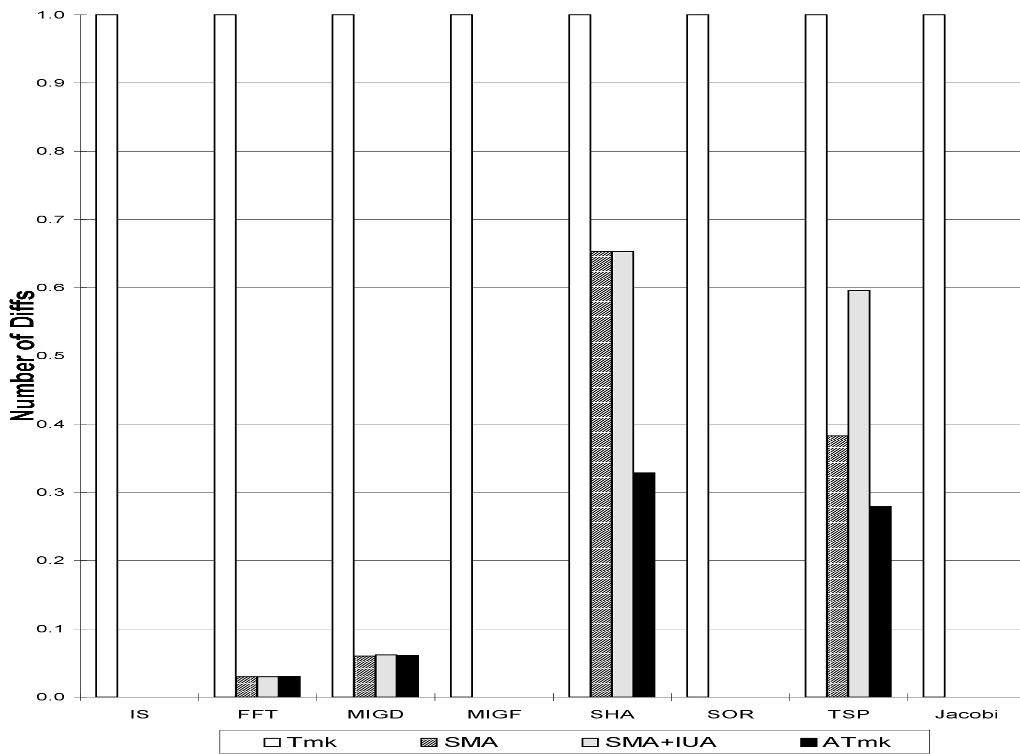

Figura 7.5: Número de diffs gerados (Normalizado em relação a Tmk).

normalizadas em relação aos resultados para Tmk.

Foi considerado como overhead de memória o número de bytes usados para armazenar diffs, twins, intervalos e write notices. É importante ressaltar que todos os experimentos foram realizados sem *garbage collection*. O uso de garbage collection iria reduzir o overhead de memória de Tmk, mas implicaria em mensagens adicionais. Assim, optou-se pela alternativa de melhor desempenho para Tmk. Para os outros protocolos estudados o uso de garbage collection não é necessário, uma vez que o consumo de memória se mantém em níveis aceitáveis. Excetuando Tmk, o maior consumo de memória ocorre para Shallow sobre ADSM, onde o overhead de memória por processador é menor que 4 Mbytes.

A figura mostra que SMA reduz acentuadamente o consumo de memória em relação a Tmk, devido à eliminação de diffs e twins para páginas SW. Conseguiu-se reduções no overhead de memória para todas as aplicações, variando de 56% (TSP) até 99% (IS, MigDepth e MigFreq). A figura mostra também que a adição de IUA não altera o overhead de memória, com exceção de TSP. Como já visto na seção 7.1, para TSP a adição de IUA alterou a caracterização das páginas, o que acarretou

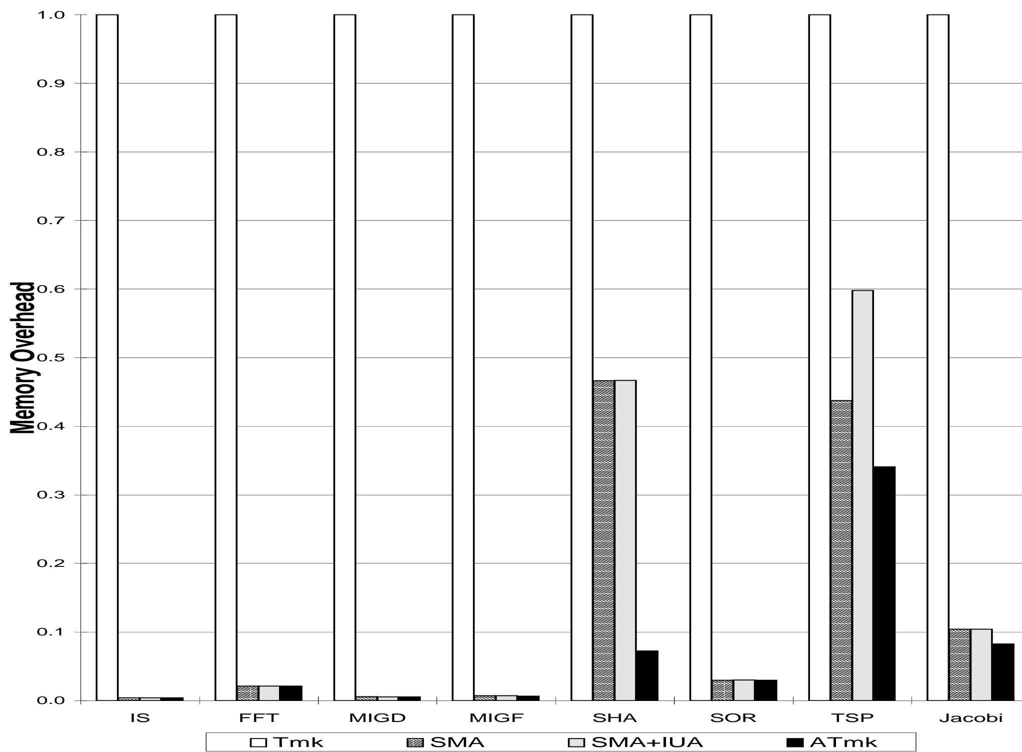

Figura 7.6: Overhead de memória (normalizado em relação a Tmk).

mudança nos overheads de memória.

ATmk consome menos memória do que ADSM para três aplicações: Shallow, TSP e Jacobi. Isto ocorre devido à mais precisa determinação de páginas SW feita por ATmk, que leva a geração de um menor número de diffs e twins, como já analisado na seção anterior. Em média, ADSM reduziu em 85% os overheads de memória em relação a Tmk, enquanto ATmk atingiu uma redução de 93%.

7.5 Overheads nos Acessos a Dados

Na figura 7.7 mostramos os números de falhas de acesso para páginas inválidas (i.e. as falhas que requerem troca de mensagens em Tmk) para cada aplicação sobre dois dos protocolos estudados: Tmk e ADSM. Não são mostrados os resultados para os outros dois protocolos uma vez que estes resultados são idênticos aos de Tmk, já que as técnicas usadas por SMA e ATmk não são capazes de reduzir o número de falhas de acesso.

Os resultados mostrados nesta figura demonstram que a técnica IUA reduz drasticamente o número de falhas em relação aos outros protocolos. As reduções variam

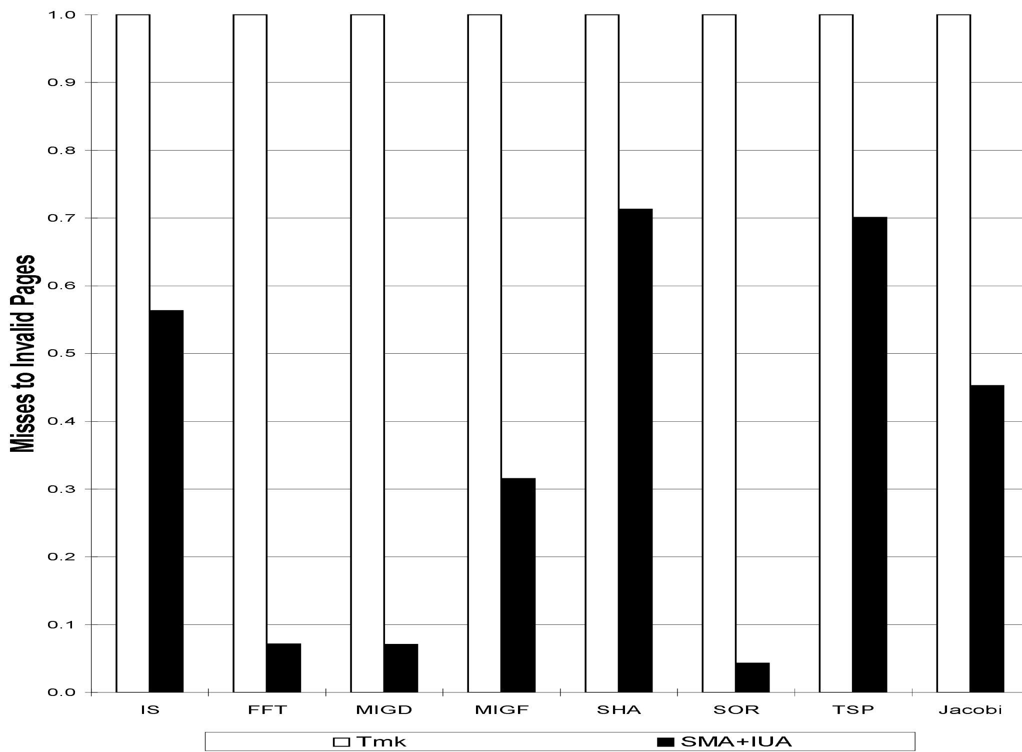

Figura 7.7: Número de falhas de acesso para páginas inválidas (normalizado em relação a Tmk).

de 29% para Shallow até 96% para SOR. As maiores reduções ocorrem para SOR e MigDepth mas tem razões diferentes. Em SOR, a maioria das páginas sofre compartilhamento 1PMC sendo atualizadas em barreiras, enquanto em MigDepth o padrão dominante são páginas migratórias acessadas dentro de seções críticas (MIGi), sofrendo atualizações em lock acquires.

7.6 Detalhamento dos Tempos de Execução

Na figura 7.8 são mostrados os detalhamentos dos tempos de execução das diversas aplicações. Para cada aplicação, apresentamos os tempos de execução para, da esquerda para a direita, Tmk, SMA e SMA+IUA. Todas as barras foram normalizadas em relação aos resultados para Tmk. Cada barra mostra o tempo de execução subdividido em tempo de computação ("Busy"), que inclui cache e TLB misses, overhead de execução do protocolo ("Ovrd"), tempos despendidos aguardando mensagens de sincronização por barreiras e por locks ("Barrier" e "Lock"), e tempo gasto aguardando mensagens em falhas de acesso ("Faults"). O overhead de protocolo

inclui os tempos para proteção de páginas, criação de twins, computação de diffs, execução de código do protocolo e respostas a pedidos remotos (IPC).

A figura mostra que os overheads dos protocolos foram em geral pequenos sob Tmk, o que fez com que as reduções de overhead de até 75% (IS) obtidas por SMA, em virtude da eliminação de diffs e twins, não acarretassem ganhos significativos de speedup. Foram observadas também pequenas variações nos tempos de computação (busy) das diversas aplicações, que são devidas a variações nas taxas de acerto na cache e no TLB.

SMA reduziu o tempo gasto em falhas de acesso para MigDepth (68%), IS (65%) e MigFreq (59%) em relação a Tmk. Estas reduções ocorreram por que, como visto na seção 7.2, o uso de protocolo de único escritor eliminou o problema de acumulação de diffs para páginas MIGi, além de terem sido eliminadas as mensagens para pedidos de diffs em *cold misses* para páginas SW. A eliminação de pedidos de diffs em *cold misses* acarretou também uma redução de 20% no tempo gasto em falhas de acesso para FFT sobre SMA.

As atualizações feitas por ADSM conseguiram reduzir os tempos gastos em falhas de acesso para todas as aplicações em relação a SMA em pelo menos 25% (Shallow), com destaque para MigDepth e SOR com 94% de redução. Como conseqüência, foram também reduzidos os tempos gastos em sincronizações, em especial para IS (43% de redução), MigDepth (30%) e MigFreq (58%), uma vez que nestas aplicações a maioria das falhas de acesso em Tmk ocorre dentro de seções críticas. Estas falhas portanto alongam a duração das seções críticas, aumentando a contenção pelos locks, o que por sua vez agrava o desbalanceamento entre as chegadas dos processadores às barreiras. Para Shallow, FFT e SOR, a adição de IUA reduziu o tempo de sincronização em barreiras, uma vez que a diminuição no número de falhas de acesso proporcionou um melhor balanceamento na chegada às barreiras dos diversos processadores.

É interessante também notar que a estratégia de atualização de páginas 1PMC empregada por SMA+IUA acarretou um aumento do tempo de sincronização por barreiras em Jacobi (+ 25%), devido ao tempo despendido para transmissão das mensagens de atualização, o que foi no entanto compensado pela redução de 40% no tempo gasto em falhas de acesso.

7.7 Discussão

Páginas com um único escritor podem ser freqüentemente encontradas em aplicações paralelas. Os resultados apresentados mostram que seis das aplicações testadas (IS, FFT, MigDepth, MigFreq e Jacobi) são dominadas por páginas SW, enquanto as demais (TSP e Shallow) exibiram uma combinação de páginas SW e MW. As estratégias eficientes para tratamento de páginas SW implementadas em ADSM lhe permitem obter desempenho superior a ATmk e Tmk para a maioria das aplicações dominadas por páginas SW. Para aplicações dominadas por páginas MW, Tmk, ATmk e ADSM obtém desempenho semelhante, já que os dois protocolos adaptativos adicionam um overhead muito pequeno ao tratamento deste tipo de páginas.

SPC produz uma categorização aproximada do padrão de compartilhamento das páginas de uma aplicação. Uma comparação com a categorização precisa feita por ATmk mostra que, para TSP e Shallow, SPC classifica erroneamente algumas páginas como MW. Estes desvios ocorrem para páginas que são na verdade atualizadas por apenas um processador de cada vez, mas que são modificadas dentro e fora de seções críticas (TSP), ou apresentam uma dinâmica de variação de padrões produtor/consumidor e migratório entre diferentes fases do programa que não foi corretamente detectada por SPC (Shallow).

É interessante também notar que a comparação entre SMA e ATmk é dominada pela eliminação das mensagens de posse conseguida por SMA, que trouxe ganhos significativos de desempenho para algumas aplicações. Uma comparação com a versão SW do protocolo CVM [Kel96] seria igualmente favorável a ADSM, uma vez que, assim como ATmk, CVM usa o conceito de mensagens de posse.

A comparação entre ADSM e ATmk mostra um compromisso entre a menor carga de comunicação e o reduzido número de falhas de leitura de ADSM, contra os overheads de memória e coerência ligeiramente inferiores conseguidos por ATmk. Com base nos significativos ganhos de desempenho atingidos por ADSM, pode-se afirmar que a comparação lhe é favorável.

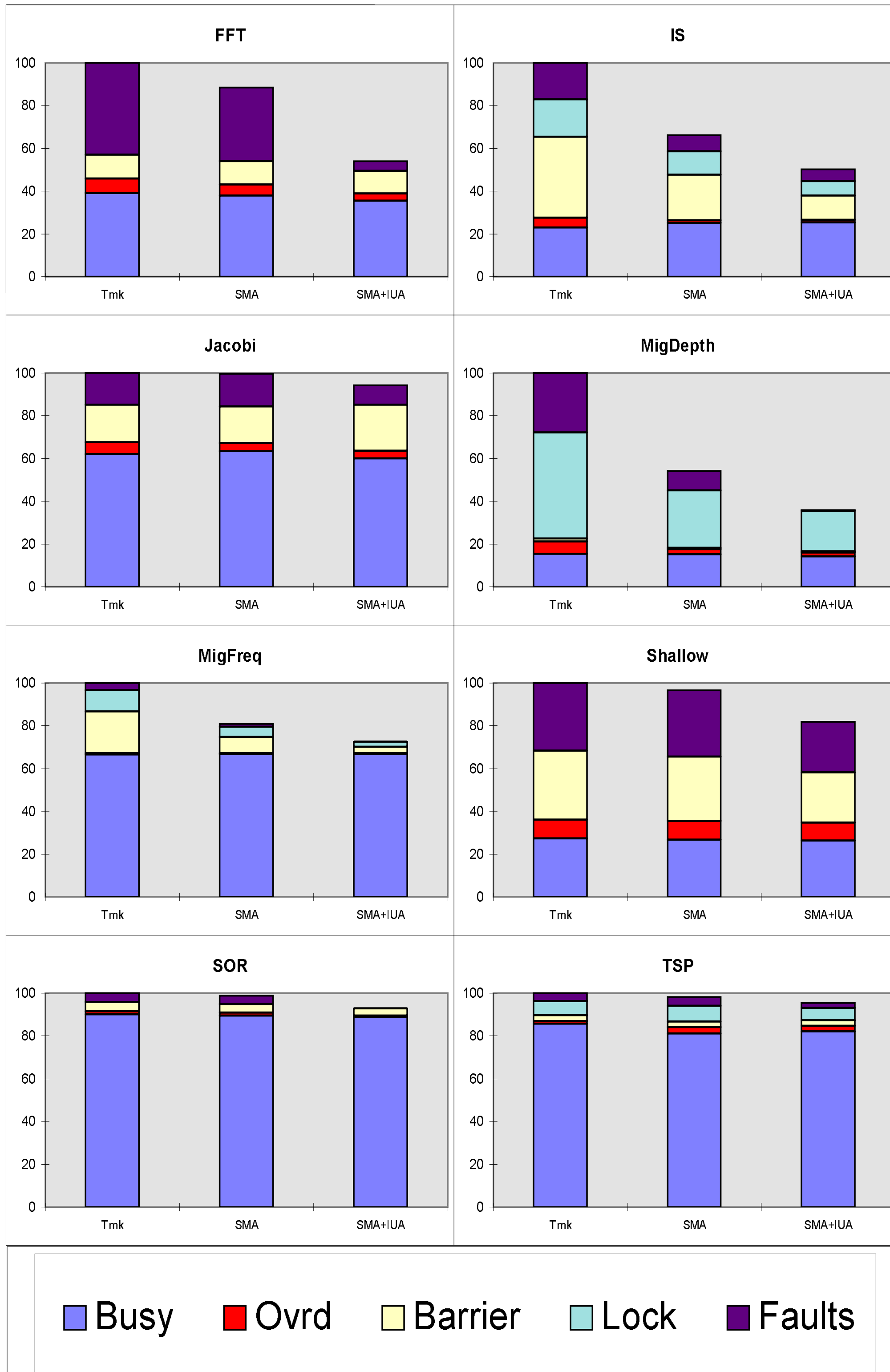

Figura 7.8: Detalhamento dos tempos de execução (normalizado em relação a Tmk).

Capítulo 8

Trabalhos Relacionados

Neste capítulo são discutidos trabalhos e sistemas relacionados às propostas apresentadas nesta tese. A seção 8.1 discute outras formas de implementação do modelo de memória compartilhada. A seção seguinte considera os trabalhos relacionados à consistência relaxada de memória. A seção 8.3 trata de outros sistemas híbridos ou adaptativos, enquanto que a última seção trata tópicos relacionados à granularidade da unidade de coerência em sistemas de memória compartilhada.

8.1 Modelos de Programação Amigáveis

8.1.1 Modelo (Quase) Seqüencial

Compiladores para computação paralela, tais como HPF [KLS+96] e SUIF [HAM+95], provêem um modelo de programação bastante amigável, uma vez que o programador em geral não precisa, por exemplo, explicitar as tarefas de criação, finalização e sincronização de processos. No entanto, devido às limitações intrínsecas a esse tipo de compilador, é freqüentemente necessário que o usuário insira anotações ditando diretrizes ao compilador, o que complica a programação.

HPF é um exemplo interessante. Em HPF, o programador deve, por exemplo, definir a distribuição dos dados entre os diversos processadores. Isso simplifica a geração de código para máquinas em que processadores se comunicam através de passagem de mensagens, o alvo mais freqüente dos compiladores HPF. Mesmo com auxílio de diretrizes fornecidas pelo programador, a geração automática de código paralelo baseado em passagem de mensagens continua sendo uma tarefa muito complexa e nem sempre os resultados obtidos são satisfatórios.

O uso de software DSMs como plataforma alvo de compiladores para computação paralela tem sido objeto de vários estudos recentes, por exemplo [DCZ96, KT97, CDLZ97, CL97], uma vez que esta estratégia pode facilitar o desenvolvimento destes compiladores, além de tornar o desempenho final obtido menos dependente da distribuição de dados feita pelo programador.

Em um trabalho recente [CDLZ97], Cox *et al.* mostraram que códigos gerados pelo compilador Forge-SPF [App96b] com TreadMarks [ACD+96b] como alvo atingiram resultados entre 40% inferiores e 89% superiores aos obtidos com o uso do compilador Forge-HPF [App96a] para seis aplicações executando em um multicomputador IBM SP2, mostrando-se portanto como uma opção competitiva. Uma vez que as otimizações apresentadas nesta tese foram capazes de melhorar substancialmente o desempenho de TreadMarks, acreditamos que o seu uso poderá tornar esta estratégia ainda mais atraente e consistente.

8.1.2 Modelo Paralelo de Memória Compartilhada

Vários tipos de sistemas suportam o modelo de programação de memória compartilhada, i.e. um espaço de endereçamento único acessível por todos os nós do sistema; desde os sistemas de memória compartilhada centralizada, p.e. [Enc87, LT88, Dig96], até os que implementam uma memória logicamente compartilhada sobre memórias fisicamente distribuídas (DSM) [PTM95].

ADSM pertence a classe dos sistemas DSM implementados em software. Essa classe é bastante grande e inclui sistemas DSM implementados a nível de linguagem (e.g. Split-C [CDG+93] e Orca [BBH+97]), compilador (e.g. Shasta [SGT96, SG97]), biblioteca de execução (e.g. ADSM, TreadMarks, CRL [JKW95] e Shared Regions [SGZ93]) e sistema operacional (e.g. SoftFLASH [ENCH96]). Sistemas DSM implementados a nível de biblioteca de execução, tais como ADSM, têm uma vantagem muito importante sobre outros tipos de implementação: tais sistemas não dependem de um software básico sofisticado, podendo ser utilizados com linguagens, compiladores e sistemas operacionais convencionais, tais como C, gcc e Solaris.

Nem todos os sistemas DSM são exclusivamente baseados em software, no entanto. Existem ainda sistemas que se baseiam em software mas utilizam suporte de hardware, e sistemas DSM implementados em hardware (hardware DSMs). A seguir discutimos essas duas estratégias alternativas.

Nos sistemas DSM baseados em software com suporte de hardware, a manutenção da coerência continua a ser feita primordialmente por software, mas o hardware adicional permite a obtenção de melhores níveis de desempenho. Este tipo de aproximação é empregada nos sistemas SHRIMP [BLA+94, BDFL96], CASHMERe [KS96b, SDH+97] e NCP_2 [BKP+96, ABHM+97].

O multicomputador SHRIMP é composto por PCs Pentium conectados através de uma rede similar à usada no Intel Paragon [TD92]. SHRIMP provê uma facilidade de mapeamento remoto de memória, segundo o qual uma página de memória local pode ser associada à uma página da memória de um outro nó. Escritas feitas a esta página local são automaticamente propagadas para a página remota pelo hardware. Aproveitando-se desta facilidade foram desenvolvidos os protocolos AURC [IDFL96] e ScC [ISL96a], que são, respectivamente, baseados nos modelos LRC e EC e utilizam o conceito de *home nodes*. Tais protocolos atingem desempenho superior ao de TreadMarks [IDFL96, ISL96a, ZIL96, BKP+96].

CASHMERe usa suporte de hardware similar ao de SHRIMP através da rede *Memory Channel* [Gil96]. Apesar do uso de suporte de hardware especial, CASHMERe apresentou desempenho similar ao de TreadMarks [KHS+97], devido à maior carga de comunicação gerada por seu protocolo. ADSM não usa hardware especial e mostrou desempenho significativamente superior ao de TreadMarks, mostrando-se portanto como uma alternativa eficiente e barata se comparado com SHRIMP e CASHMERe.

O multicomputador NCP_2 [ABHM+97], que está sendo desenvolvido na COPPE Sistemas/UFRJ, provê modelo de programação de memória compartilhada em uma rede de estações com processadores PowerPC conectadas via a rede de interconexão Myrinet [BCF+95]. O NCP_2 usa um controlador de protocolo [SHMM+97] de maneira a reduzir ou esconder vários atrasos envolvidos na comunicação entre processadores e na manutenção da coerência dos dados. Mais especificamente, este controlador é capaz de gerar diffs automaticamente e sem a necessidade de twins, permite afastar as tarefas básicas de comunicação e coerência do processador, podendo ainda responder a pedidos remotos sem interromper o processador local. Tais características reduzem algumas das principais fontes de atrasos em software DSMs. É importante notar que as estratégias usadas em ADSM e no sistema NCP_2 são complementares, na medida que, enquanto ADSM pode eliminar os overheads para

manutenção de coerência (diffs e twins) para páginas SW, o controlador de protocolo do NCP_2 é capaz de gerar diffs com overhead mínimo para páginas MW.

O uso de software DSMs baseados em redes de multiprocessadores SMPs tem também sido estudado como uma alternativa híbrida para construção de DSMs, como por exemplo nos trabalhos [CDK+94, KS96a, ENCH96, Kof97, SBIS98]. Neste tipo de estratégia, a coerência da memória compartilhada é mantida em dois níveis: por hardware (e.g. *snooping*) dentro de cada nó, e por software entre os diversos nós. ADSM pode ser modificado de maneira a manter a coerência em dois níveis, mantendo as estratégias adaptativas nesta tese introduzidas.

Sistemas DSM exclusivamente baseados em software ou com suporte de hardware geralmente alcançam bom desempenho (a um custo relativamente baixo), mas apenas para uma classe restrita de aplicações. Sistemas DSM baseados em hardware, no entanto, apesar de mais caros, podem executar eficientemente uma gama muito maior de aplicações. Hardware DSMs vêm sendo desenvolvidos há vários anos; desde os pioneiros Ultracomputer [GGK+83] e RP3 [PBG+85] até os modernos Origin [LL97], Exemplar [BA97] e T3E [Sco96]. Todos esses sistemas têm em comum o fato de permitirem o acesso direto a qualquer das memórias da máquina, mas suas diferenças definem três grupos principais de sistemas: aqueles que implementam a coerência de caches em software (e.g. T3D [Cra93] e Multiplus [Aud97]), aqueles que utilizam hardware programável altamente otimizado para implementação de protocolos de coerência (e.g. FLASH [KOH+94] e Typhoon [RLW94]), e aqueles que implementam a coerência de caches em hardware (e.g. DASH [LLW+92], Alewife [ABC+95, CLB+96], Origin e Exemplar.

8.2 Modelos de Consistência Relaxada

O modelo de consistência de memória seqüencial é o mais natural para os programadores e portanto foi o primeiro a ser implementado, tanto em sistemas DSM baseados em software quanto em hardware. Ivy [Li88], um sistema que utiliza protocolo de único escritor baseado em invalidações e modelo de consistência seqüencial, foi o primeiro software DSM. Em termos de hardware DSMs, os pioneiros Ultracomputer e RP3 não utilizavam caches, mas também proviam o modelo de consistência seqüencial.

Desde o projeto e discussão desses sistemas pioneiros, vários sistemas DSM passaram a utilizar modelos relaxados de consistência como forma de otimizar seu desempenho. Munin [BCZ91, Car95] foi o primeiro software DSM a suportar múltiplos escritores e usar um modelo de consistência relaxada (Release Consistency). A partir de Munin, um grande numero de software DSMs, como por exemplo ADSM, TreadMarks e Midway [ZSB94], passou a utilizar modelos de consistência relaxada. ADSM e TreadMarks são baseados em Lazy Release Consistency, um modelo bastante relaxado e, portanto, eficiente. Sistemas baseados em Entry Consistency [BZ91] e Scope Consistency [ISL96a], tais como Midway, AEC [SBA97], ECP [CBA96], Brazos [SB97] e uma versão preliminar de ADSM [MB97], são mais relaxados que ADSM, mas complicam sensivelmente o modelo de programação.

ADSM tem em comum com os sistemas baseados em Entry Consistency e Scope Consistency o fato de que variáveis compartilhadas acessadas em seções críticas são associadas às variáveis de lock que as protegem. No entanto, a associação de dados a locks feita por estes sistemas é necessária para garantir correção do programa (o que exige que esta associação seja precisa), enquanto a versão atual de ADSM usa uma associação aproximada apenas para permitir a caracterização de padrões de compartilhamento e obter ganhos de desempenho.

É interessante notar que, recentemente, Keleher [Kel96] comparou os ganhos de desempenho alcançados por modelos de consistência relaxada e por protocolos de múltiplos escritores. Os resultados por ele obtidos mostram que relaxar o modelo de consistência é mais importante do que suportar múltiplos escritores por página. Keleher também mostrou que protocolos de único escritor são uma alternativa interessante para algumas aplicações devido à significativa redução do overhead de memória, apesar de não terem desempenho satisfatório para aplicações sujeitas a falso compartilhamento. ADSM usa modelo de consistência relaxada, e se adapta eficientemente entre protocolos de único e múltiplos escritores, conseguindo assim eliminar overheads desnecessários de memória sem prejudicar o desempenho para aplicações sujeitas a falso compartilhamento.

A maior parte dos hardware DSMs modernos também relaxa a consistência da memória, mas utiliza protocolos de único escritor quando a coerência é implementada em hardware. DASH foi o primeiro hardware DSM a utilizar um modelo relaxado de consistência, Release Consistency. Multiprocessadores mais recentes (e.g. Con-

vex/SPP, T3D e T3E) também implementam modelos relaxados de consistência, a não ser em raras exceções como no caso da máquina Origin.

8.3 Sistemas Híbridos e Adaptativos

A maior parte dos sistemas híbridos ou adaptativos tem sido implementada em sistemas DSM baseados em software, uma vez que tais sistemas efetivamente permitem uma flexibilidade e sofisticação maiores no tratamento dos dados compartilhados e na sua coerência. Assim, começamos nossa discussão desses sistemas a partir dos software DSMs e depois passamos aos hardware DSMs.

Em um trabalho recente [ACDZ97], Amza *et al.* propuseram o sistema TreadMarks Adaptativo, descrito na seção 3.4, que é o mais semelhante ao aqui apresentado. Nesta tese comparou-se a sua implementação mais importante (ATmk) contra ADSM. Os resultados obtidos mostram que ADSM obtém melhor desempenho devido à menor carga de comunicação, mesmo quando não são usadas atualizações seletivas. Adicionalmente, ADSM faz uma caracterização mais detalhada do compartilhamento das páginas, o que viabilizou o desenvolvimento e implementação de técnicas adaptativas de atualização seletiva, permitindo a ADSM atingir níveis de desempenho significativamente superiores aos de ATmk.

Assim como ADSM, Munin suporta modos de único e múltiplos escritores por página e mantém a coerência com um protocolo híbrido de atualizações e invalidações. Entretanto, ao contrário de ADSM, em Munin a seleção do protocolo a ser utilizado é baseada em anotações feitas pelo programador. A dependência de anotações do programador dificulta a tarefa de programação, contrariando assim uma das motivações básicas de software DSMs. Esta tarefa fica ainda mais difícil quando os padrões de compartilhamento variam ao longo da execução do programa ou dependem dos dados de entrada utilizados. Além disto, nem sempre o programador tem uma idéia clara do padrão de compartilhamento das estruturas de dados da aplicação. ADSM se adapta automaticamente entre diferentes protocolos de maneira a não dificultar a tarefa de programação e ser capaz de determinar variações no padrão de compartilhamento das páginas.

O protocolo *Lazy Hybrid* (LH), estudado por Dwarkadas *et al.* em [DKCZ93], também usa uma estratégia híbrida de atualizações e invalidações. Em uma operação

de lock acquire em LH, o processador releaser envia para o processador acquirer os diffs associados aos write notices das páginas que o releaser tem cópia. Além disto, ao chegar a uma barreira, cada processador envia diffs para todos os processadores que têm cópias das páginas modificadas localmente. ADSM difere de LH na medida que atualizações só são usadas para tipos específicos de dados: páginas migratórias associadas a variáveis de lock em operações de lock acquire, e páginas 1PMC em barreiras. Assim, ao contrário de LH, o uso de atualizações em ADSM não acarreta aumento relevante da quantidade de dados transmitidos, já que só são feitas atualizações para páginas que potencialmente podem se beneficiar desta técnica. Como conseqüência, ADSM obteve expressivos ganhos de desempenho, enquanto os benefícios trazidos pelo protocolo LH foram limitados, acarretando inclusive degradações de desempenho para algumas aplicações [KCDZ95].

Sistemas baseados em Entry Consistency e Scope Consistency, como AEC e Brazos, também utilizam estratégias híbridas para manter a coerência dos dados. As estratégias de atualização usadas nestes sistemas são também distintas das usadas em ADSM, já que SPC permite a ADSM ser mais seletivo nas atualizações de dados associados a locks, além de aplicar atualizações a alguns tipos de dados protegidos por barreiras.

Concorrentemente ao nosso trabalho, Speight e Bennett [SB98] desenvolveram uma técnica, chamada de *Early Updates*, ou simplesmente EU, para o sistema Brazos. Esta técnica usa facilidades de *multicast* existentes em redes de interconexão específicas (e.g. Ethernet), para fazer atualizações de páginas que são acessadas por vários nós logo após uma operação de barreira, procurando assim evitar o tráfico que resultaria de várias operações de multicast concorrentes. IUA difere significativamente de EU em vários pontos. Em especial, IUA não usa e não procura otimizar operações de multicast, além de ser capaz de tratar um espectro de compartilhamento de dados muito mais amplo.

Também concorrente ao nosso trabalho, foi proposto em [KV97] o mecanismo *Adaptive Migratory Scheme*, ou simplesmente AMS, que é capaz de detectar padrões migratórios de páginas no sistema Quarks [Kha96]. Esta estratégia difere de SPC em vários pontos: 1) AMS usa um método estatístico, enquanto SPC é capaz de fazer uma caracterização mais rápida através de um diagrama de estados; 2) AMS só suporta modo único escritor, enquanto as estratégias utilizadas em ADSM são

mais gerais, suportando inclusive múltiplos escritores por página; 3) o tratamento de páginas migratórias em AMS é feito de maneira simples e ineficiente com uso de um mecanismo de *self-invalidation* [LW95], que permite apenas um leitor por página (SWSR), enquanto os mecanismos mais avançados utilizados em ADSM permitem múltiplos leitores por página migratória (SWMR). Além disto, em [KV97] não é apresentado o desempenho global do protocolo, de maneira que fica difícil avaliar precisamente o quão eficientes são estas técnicas.

Vários pesquisadores propuseram técnicas simples para adaptação a padrões de compartilhamento no contexto de hardware DSMs, em especial nos trabalhos [CF93, SBS93, DS94]. Cox e Fowler [CF93] e Stenstrom, Brorsson e Sandberg [SBS93] avaliaram otimizações no tratamento de dados migratórios em protocolos de coerência. Dahlgren e Stenstrom [DS94] estudaram um protocolo híbrido baseado em invalidações e atualizações no qual cada processador toma uma decisão local de atualizar ou invalidar uma linha de cache quando recebe uma mensagem de atualização. Em contraste com estas propostas, ADSM implementa técnicas muito mais sofisticadas de categorização de compartilhamento e de adaptação entre protocolos do que seria viável ou eficiente em hardware DSMs.

Hardware DSMs que permitem técnicas mais complexas de adaptação o fazem, em geral, com algum auxílio de software. O multiprocessador Alewife, por exemplo, provê facilidades que permitem a implementação em software dos mecanismos de coerência, possibilitando assim a implementação de protocolos mais complexos. Da mesma maneira, a arquitetura FLASH pode ser ajustada de forma a utilizar técnicas adaptativas, através do uso dos processadores de protocolo da máquina.

Também com auxílio de software, Trancoso e Torrelas [TT96] propuseram uma otimização para hardware DSMs, na qual um processador ao fazer release de um lock envia ao processador acquirer as modificações feitas dentro da seção crítica, tendo assim um efeito similar ao obtido pela técnica de atualização de páginas associadas a locks apresentada nesta tese. Esta otimização é, no entanto, implementada pelo compilador, e requer instruções especiais de hardware que não estão disponíveis nas arquiteturas comerciais atuais.

8.4 Granularidade da Unidade de Coerência

ADSM mantém a coerência a nível de páginas como inicialmente proposto por Ivy, o que pode induzir a ocorrência de falso compartilhamento em aplicações com granularidade fina de acesso à memória compartilhada [ISL96b].

Vários sistemas, como por exemplo Orca [BBH+97] e CRL [JKW95], mantém a coerência da memória compartilhada a nível de objetos, o que evita problemas de falso compartilhamento. No entanto, esse tipo de aproximação dificulta a programação, já que nem sempre é fácil e/ou natural encapsular dados compartilhados em objetos. ADSM ameniza os problemas com falso compartilhamento através do uso de modelo de consistência relaxada e do suporte a múltiplos escritores, evitando assim prejudicar o modelo de programação.

Assim como os hardware DSMs com coerência de caches, os sistemas Shasta e Blizzard-S [SFL+94] aliviam o problema do falso compartilhamento utilizando uma granularidade muito mais fina para o tamanho da unidade de coerência. O controle de acesso as unidades e feito por instruções extras incluídas no código executável, o que envolve atrasos adicionais às operações de *load* e *store*. Apesar desta estratégia reduzir os efeitos negativos do falso compartilhamento, resultados publicados recentemente [ACRZ97] mostram que aumentar a granularidade da unidade de coerência traz ganhos de desempenho para várias aplicações, na medida que as perdas devidas ao aumento do falso compartilhamento são contrabalançadas por ganhos decorrentes do *prefetching* induzido pelo uso de uma unidade de coerência grande. Estas conclusões são coerentes com resultados publicados em [ZIL+97], onde um protocolo baseado em páginas mostrou desempenho superior a um protocolo com granularidade fina de acesso para a maioria das aplicações avaliadas. Por outro lado, um terceiro estudo recente [BS97], mostrou no que algumas aplicações podem se favorecer da diminuição do tamanho da unidade de coerência, o que demonstra que não há um tamanho de unidade de coerência que seja ideal para todas as aplicações e sistemas.

Capítulo 9

Conclusões

Nesta tese foi proposto o sistema Adaptive DSM (ADSM), que se baseia na técnica Sharing Pattern Categorization (SPC) para adaptar-se entre protocolos de único e múltiplos escritores e entre estratégias de invalidação e atualização.

A análise de ADSM mostrou que as otimizações introduzidas nesta tese conseguem reduções significativas dos overheads de comunicação, coerência e memória em comparação com TreadMarks. A comparação contra o recente e eficiente TreadMarks Adaptativo mostrou que ADSM obteve melhor desempenho para todas as aplicações, como conseqüência da menor carga de comunicação gerada.

Em resumo, as principais contribuições foram a introdução, implementação e avaliação de SPC, das estratégias de adaptação, e de ADSM. ADSM mostrou-se um software DSM eficiente, enquanto SPC e as estratégias de adaptação se mostraram técnicas eficientes que podem ser usadas em outros software DSMs.

Referências Bibliográficas

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

[ABHM+97] C. Amorim, R. Bianchini, M. Hor-Meyll, M. De Maria, R. Pinto, R. Santos, and L. Whately. The NCP_2 Distributed Shared-Memory System - Project Status. In *Anais do IX Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, pages 573–576, October 1997.

[ACD+96a] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA-2)*, pages 26–37, February 1996.

[ACD+96b] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[ACDZ97] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA-3)*, pages 261–271, February 1997.

[ACRZ97] C. Amza, A. L. Cox, K. Ramajamni, and W. Zwaenepoel. Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 90–99, June 1997.

[AG96] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[AH90] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*, pages 2–14, May 1990.

[AH93] S. V. Adve and M. D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[App96a] Applied Parallel Research, Inc. *FORGE High Performance Fortran User's Guide*, Version 2.0 edition, 1996.

[App96b] Applied Parallel Research, Inc. *FORGE Shared Memory Parallelizer User's Guide*, Version 2.0 edition, 1996.

[Aud97] J.S. Aude. The Multiplus/Mulplix Project. In *Anais do IX Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, pages 581–586, October 1997.

[BA97] T. Brewer and G. Astfalk. The Evolution of HP/Convex Exemplar. In *Proceedings of the 42nd IEEE International Computer Conference (COMPCON Spring '97)*, pages 81–86, February 1997.

[BBB+94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[BBH+97] H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, K. Ruhl, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. *Journal of Parallel and Distributed Computing*, February 1997.

[BCF+95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[BCZ90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, March 1990.

[BCZ91] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Using Multi-Protocol Release Consistency. In A. I. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, number 563 in Lecture Notes in Computer Science, pages 56–60. Springer-Verlag, July 1991.

[BDFL96] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li. Protected, User-level DMA for the SHRIMP Network Interface. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA-2)*, pages 154–165, February 1996.

[BKP+96] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 198–209, October 1996.

[BLA+94] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual Inter-*

national Symposium on Computer Architecture (ISCA'94), pages 142–153, April 1994.

[BLV96] R. Bianchini, T.J. LeBlanc, and J.E. Veenstra. Categorizing Network Traffic in Update-Based Protocols on Scalable Multiprocessors. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, pages 142–151, April 1996.

[BS97] A. Bilas and J. P. Singh. The Effects of Communication Parameters on End Performance of Shared Virtual Memory Clusters. In *Proceedings of Supercomputing'97*, November 1997.

[BZ91] 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, School of Computer Science, Carnegie-Mellon University, September 1991.

[BZS93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE International Computer Conference (COMPCON Spring'93)*, pages 528–537, February 1993.

[Car95] J. B. Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, 29(2):219–227, September 1995.

[CBA96] M.C. Carneiro, R. Bianchini, and C. Amorim. Implementação e Avaliação de Entry Consistency. In *Anais do VIII Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, pages 235–244, August 1996.

[CBZ95] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[CDG+93] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing'93*, pages 262–273, November 1993.

[CDK+94] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*, pages 106–117, April 1994.

[CDLZ97] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers. In *Proceedings of the 11st International Parallel Processing Symposium (IPPS'97)*, pages 474–482, April 1997.

[CF93] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 98–108, May 1993.

[CL97] S. Chandra and J. R. Larus. Optimizing Communication in HPF Programs for Fine-Grain Distributed Shared Memory. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 100–111, June 1997.

[CLB+96] F. T. Chong, B-H. Lim, R. Bianchini, J. Kubiatowicz, and A. Agarwal. Application Performance on the MIT Alewife Machine. *IEEE Computer*, 29(12):57–64, December 1996.

[CPS+95] M.C. Carneiro, R. Pinto, C. Seidel, A. Silva, G. Silva, and C. Amorim. Desempenho Simulado de Modelos Fracos de Consistência de Memória. In *Anais do VII Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, 1995.

[Cra93] Cray Research, Inc. *CRAY T3D System Architecture Overview*, HR-04033 edition, September 1993.

[DCZ96] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 186–197, October 1996.

[Dig96] Digital Equipment Corporation. *AlphaServer 8200/8400 Systems: Technical Summary*, 1996.

[DKCZ93] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 144–155, May 1993.

[DMS97] J. Dongarra, H. Meuer, and E. Strohmaier. TOP500 Supercomputer Sites. Technical Report UT-CS-97-365, University of Tennessee, June 1997.

[DS94] F. Dahlgren and P. Stenstrom. Reducing the Write Traffic for a Hybrid Cache Protocol. In *Proceedings of the 1994 International Conference on Parallel Processing (ICPP'94)*, volume I, pages 166–173, August 1994.

[DSB86] M. Dubois, C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA'86)*, pages 434–442, June 1986.

[Enc87] Encore Computer Corporation. *Multimax Technical Summary*, 1987.

[ENCH96] A. Erlichson, N. Nuckolls, G. Chesson, and J. L. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 210–220, October 1996.

[Fig95] P. Figueiredo. Exploitation of Parallelism in Seismic Migration. Master's thesis, University of Illinois at Urbana-Champaign, April 1995.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM - A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, MA, 1994.

[GGH91] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 245–257, April 1991.

[GGK+83] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[Gil96] R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[GLL+90] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

[Goo89] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

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

[IBM95] IBM. *SP Installation Guide*, GC23-3898-01 edition, August 1995. Table 3, page 52.

[IDFL96] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA-2)*, pages 14–25, February 1996.

[ISL96a] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release consistency and Entry Consistency. In *Proceedings of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, June 1996.

[ISL96b] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 122–133, May 1996.

[JKW95] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, pages 213–228, December 1995.

[KCDZ95] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An Evaluation of Software-Based Release Consistent Protocols. *Journal of Parallel and Distributed Computing*, 29(2):126–141, September 1995.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[KDCZ94] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conferencerence*, pages 115–131, January 1994.

[Kel96] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the 16th International Con-*

ference on Distributed Computing Systems (ICDCS-16), pages 91–98, May 1996.

[Kha96] D. R. Khandekar. QUARKS: Distributed shared Memory as a Building Block for Complex Parallel and Distributed Systems. Master's thesis, Department of Computer Science, The University of Utah, March 1996.

[KHS+97] L. I. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, Jr., S. Dwarkadas, and M. L. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 157–169, June 1997.

[KLS+96] C. Koelbel, D. Loveman, R. Schreider, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1996.

[Kof97] S.T. Kofuji. The Integrated Hypersystems Project. In *Anais do IX Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, pages 577–580, October 1997.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*, pages 302–313, April 1994.

[KS96a] M. Karlsson and P. Stenstrom. Performance Evaluation of Cluster-Based Multicomputer Built from ATM Switches and Bus-Based Multiprocessor Servers. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA-2)*, pages 4–12, February 1996.

[KS96b] L. I. Kontothanassis and M. L. Scott. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA-2)*, pages 166–177, February 1996.

[KSB95] L. I. Kontothanassis, M. L. Scott, and R. Bianchini. Lazy Release Consistency for Hardware-Coherent Multiprocessors. In *Proceedings of Supercomputing'95*, December 1995.

[KT97] P. Keleher and C-W. Tseng. Enhancing Software DSMs for Compiler-Parallelized Applications. In *Proceedings of the 11st International Parallel Processing Symposium (IPPS'97)*, pages 490–499, April 1997.

[KV97] J-H. Kim and N. H. Vaidya. Adaptive Migratory Scheme for Distributed Shared Memory. In *Proceedings of the 11st ACM International Conference on Supercomputing*, July 1997.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LDCZ95] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message-Passing vs. Distributed Shared Memory on Networks of Workstations. In *Proceedings of Supercomputing'95*, December 1995.

[LDCZ97] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the Performance Differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, 43(2):208–217, 1997.

[LH86] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing (PODC'86)*, pages 229–239, August 1986.

[Li88] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.

[LL97] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 241–251, June 1997.

[LLW+92] D. E. Lenoski, J. Ludon, K. Gharachorloo W-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LT88] T. Lovett and S. Thakkar. The Symmetry Multiprocessor System. In *Proceedings of the 1988 International Conference on Parallel Processing (ICPP'88)*, pages 163–170, August 1988.

[LW95] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 48–59, June 1995.

[MB97] 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.

[MB98] L.R. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998. To appear.

[Mos93] D. Mosberger. Memory Consistency Models. *ACM Operating Systems Review*, 27(1):18–26, January 1993.

[NL91] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[PBG+85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. L. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing (ICPP'85)*, pages 764–771, August 1985.

[PL93] K. Petersen and K. Li. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. In *Proceedings of the 7th International Parallel Processing Symposium (IPPS'93)*, pages 49–55, April 1993.

[PPA94] S. S. Pinheiro, J. Panetta, and C. Amorim. Investigação de Paralelismo na Migração Sísmica. In *Anais do VI Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, 1994.

[PRAH96] 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 *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 12–23, October 1996.

[PTM95] J. Protic, M. Tomasevic, and V. Milutinovic. A Survey of Distributed Shared Memory Systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS-28)*, volume I, pages 74–84, January 1995.

[PTM96] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel and Distributed Technology*, 4(2):63–71, summer 1996.

[RLW94] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*, pages 325–337, April 1994.

[Sad75] R. Sadourny. The Dynamics of Finite-Difference Models of the Shallow-Water Equations. *Journal of Atmospheric Sciences*, 32(4), April 1975.

[SB97] E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the USENIX Windows NT Workshop*, August 1997.

[SB98] E. Speight and J. K. Bennett. Using Multicast and Multithreading to Reduce Communication in Software DSM Systems. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998. To appear.

[SBA97] C. Seidel, R. Bianchini, and C. Amorim. The Affinity Entry Consistency Protocol. In *Proceedings of the 1997 International Conference on Parallel Processing (ICPP'97)*, pages 65–78, August 1997.

[SBIS98] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM protocols for SMP clusters: Design, Simulations, Implementation and Performance. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998. To appear.

[SBS93] P. Stenstrom, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 108–118, May 1993.

[Sco96] S. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 26–36, October 1996.

[SDH+97] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and Michael Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, October 1997.

[SFL+94] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 297–307, October 1994.

[SG97] D. J. Scales and K. Gharachorloo. Towards Transactionsarent and Efficient Software Distributed Shared Memory. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, October 1997.

[SGT96] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174–185, October 1996.

[SGZ93] H. S. Sandhu, B. Gamsa, and S. Zhou. The Shared Region Approach to Software Cache Coherence on Multiprocessors. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'93)*, pages 229–238, July 1993.

[SHMM+97] R. Santos, M. Hor-Meyll, M. De Maria, R. Pinto, L. Whately, R. Bianchini, and C. Amorim. Implementação e Avaliação do Controlador de Protocolo do NCP_2. In *Anais do IX Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, pages 93–106, October 1997.

[SOHL+96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, 1996.

[SS88] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[Ste90] P. Stenstrom. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[SWG91] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[SZ90] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, May 1990.

[TD92] R. Traylor and D. Dunning. Routing Chip Set for the Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips'92 Symposium*, April 1992.

[TM94a] M. Tomasevic and V. Milutinovic. Hardware Aproaches to Cache Coerence in Shared-Memory Multiprocessors, Part I (Basic Issues). *IEEE Micro*, 14(5):52–59, October 1994.

[TM94b] M. Tomasevic and V. Milutinovic. Hardware Aproaches to Cache Coerence in Shared-Memory Multiprocessors, Part II (Advanced Issues). *IEEE Micro*, 14(6):61–66, December 1994.

[TT96] P. Trancoso and J. Torrelas. The Impact of Speeding up Critical Sections With Data Prefetching and Forwarding. In *Proceedings of the 1996 International Conference on Parallel Processing (ICPP'96)*, volume I, pages 79–86, August 1996.

[ZB92] R. N. Zucker and J-L. Baer. A Performance Study of Memory Consistency Models. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 2–12, May 1992.

[ZIL96] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, October 1996.

[ZIL+97] 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 *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 193–205, June 1997.

[ZSB94] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 87–100, November 1994.