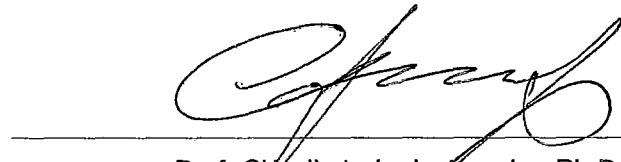
PROPOSTA E AVALIAÇÃO DE ESTRATÉGIAS DE PREVISÃO DE ACESSO A DADOS EM SISTEMAS *SOFTWARE DSM*

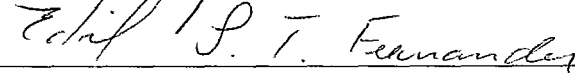
Elcio José Pineschi

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 E COMPUTAÇÃO.

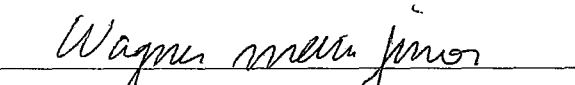
Aprovada por:



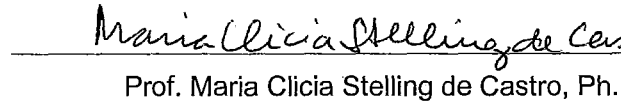
Prof. Cláudio Luís de Amorim, Ph.D.



Prof. Edil Severiano Tavares Fernandes, Ph.D.



Prof. Wagner Meira Júnior, Ph.D.



Prof. Maria Clicia Stelling de Castro, Ph.D.

RIO DE JANEIRO, RJ – BRASIL

JUNHO DE 2002

PINESCHI, ELCIO JOSÉ

Proposta e Avaliação de Estratégias de Previsão de Acesso a Dados em Sistema Software DSM [Rio de Janeiro] 2002

VII, 89 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2002)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 – Sistemas Operacionais

2 – Memória Compartilhada Distribuída

I. COPPE/UFRJ II. Título (série)

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

PROPOSTA E AVALIAÇÃO DE ESTRATÉGIAS DE PREVISÃO DE ACESSO A DADOS EM SISTEMAS *SOFTWARE DSM*

Elcio José Pineschi

Junho/2002

Orientador: Cláudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Nesta tese propomos e avaliamos novas técnicas de previsão de acesso a dados em sistemas *paged-based software DSM* (pSDSM), as quais podem ser divididas em duas classes: técnicas baseadas em preditores de desvios condicionais e técnicas baseadas na máquina de estados finitos FIESTA. Os resultados experimentais mostram que técnicas de ambas as classes podem atingir altas taxas de acertos na maioria das aplicações testadas. Também introduzimos dois novos mecanismos para aliviar o tempo de espera por dados remotos, que propagam os dados antecipadamente a partir de previsões de acessos futuros feitas por técnicas baseadas em FIESTA. O primeiro mecanismo propaga escritas de modo tal que atualizações possam atuar sobre qualquer tipo de página compartilhada da aplicação, incluindo páginas com múltiplos escritores concorrentes. O segundo mecanismo propaga os dados de forma síncrona, permitindo reduzir o custo de interrupções devido às operações de comunicação. Além disso, as atualizações recebidas são imediatamente aplicadas, evitando futuras falhas de acesso. Implementamos e avaliamos esses mecanismos em *TreadMarks* pSDSM, usando vários *benchmarks*. Em geral, os resultados obtidos nos permitem concluir que as estratégias de previsão de acesso a dados remotos que propomos podem contribuir para aumentar significativamente o desempenho dos atuais sistemas pSDSM.

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

DESIGN AND EVALUATION OF DATA ACECESS PREDICTION SCHEMES IN SOFTWARE DSM SYSTEMS

Elcio José Pineschi

June/2002

Advisor: Cláudio Luís de Amorim

Department: Computer Systems Engineering

In this thesis we propose and evaluate new prediction techniques for data access in software DSM (SDSM) systems. These techniques can be divided into two classes: techniques based on branch-condition predictors and techniques based on the FIESTA finite-state machine. Our experimental results show that techniques within both classes can attain high hit rates in most of applications we tested. Also, we introduced two new mechanisms that anticipate data propagation in accord with predictions of future accesses made by FIESTA-based techniques so as to alleviate the waiting time for remote data access. The first mechanism propagates writes in such a way that updates can act upon any type of shared page within an application, including pages that have multiple concurrent writers. The second mechanism propagates data in synchronous mode, allowing to reduce the costs of interrupts due to communication operations. In addition, incoming memory updates are immediately applied to corresponding pages, avoiding future access faults. We implement and evaluate these mechanisms in TreadMarks SDSM using several application benchmarks. Overall, the results allow us to conclude that the prediction strategies for remote data accesses we propose can contribute to increase the performance of current page-based SDSMs significantly.

Dedico este trabalho à memória

de Enedina Pedrosa Fraga (a "Dina")

Agradecimentos

Agradeço:

à meu orientador Cláudio Amorim por sua orientação, atenção, dedicação e compreensão nos momentos em que tanto precisei.

à meus pais Elcio e Teresa e à minha irmã Silvia. Sem vocês eu nunca teria chegado até aqui.

à Sigried, por sua paciência para ouvir e para esperar.

aos meus colegas de laboratório Edison, Lauro, Raquel, Lobosco e Carla. A convivência ao longo destes anos não poderia ter sido melhor. Sua experiência e amizade só me fizeram crescer.

à todos os meus professores pelos valiosos conhecimentos que me foram ensinados.

à Cristiana por me trazer a este mundo fascinante da pesquisa e da ciência. Nunca me esquecerei de seu apoio incondicional e sua amizade.

à Clicia por estar sempre disposta a me ajudar e por acompanhar este trabalho até o fim.

ao meu grande amigo Guga pelas intermináveis conversas sobre os mesmos assuntos.

à Igor pela amizade e pelas longas conversas no almoço que ajudaram a ultrapassar os momentos mais difíceis desta jornada.

à Ricardo, pela confiança e incentivo.

à CAPES, pelo suporte que forneceu para a realização deste trabalho.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Em resposta à crescente demanda por desempenho computacional, em particular pelas aplicações científicas de grande escala, diversos sistemas de computação paralela têm sido desenvolvidos ao longo do tempo.

Atualmente, redes de computadores pessoais (PCs), denominadas *clusters*, vêm se tornando uma alternativa para computação paralela de baixo custo capaz de suprir esta demanda por desempenho. Isto porque, este tipo de arquitetura pode ser implementado através da utilização de *hardware* convencional, amplamente disponível.

Tradicionalmente, a programação de *clusters* é feita utilizando o modelo de programação de passagem de mensagem. Neste modelo o programador deve explicitamente gerenciar a distribuição de dados entre os processadores. Em contraste, os sistemas multiprocessadores com modelo de programação baseado em memória compartilhada são mais convenientes. Isto porque a comunicação entre os processadores pode ser realizada com simples operações de leitura e escrita a memória. O programador, porém, precisa sincronizar o acesso aos dados compartilhados.

Os sistemas *software* **DSM** (*Distributed Shared Memory*) provêm ao programador do *cluster* um modelo de programação baseado em memória compartilhada. A abstração de memória compartilhada é implementada sobre uma biblioteca de passagem de mensagem. Além disso, a abstração esconde do programador toda a comunicação remota dos dados entre processadores, conforme ilustrado na Figura 1.1. Além de fornecerem um modelo de programação conveniente, os sistemas *software* DSM possibilitam a utilização direta de aplicações

desenvolvidas para multiprocessadores com memória compartilhada, o que representa uma redução de custos para o desenvolvimento de aplicações paralelas.

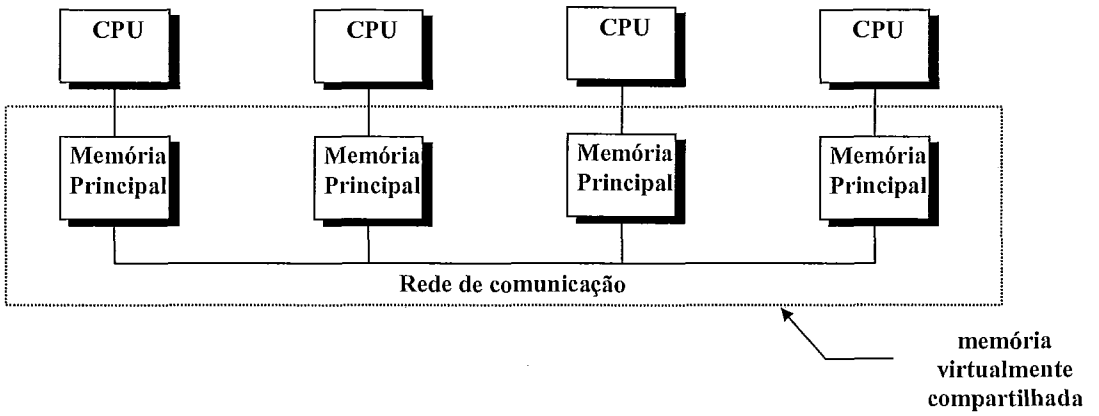


Figura 1.1 Sistema DSM provê a ilusão de uma memória compartilhada

Apesar do crescente aumento de desempenho nas tecnologias de redes de comunicação, os sistemas *software* DSM ainda sofrem muito com os altos custos de comunicação e manutenção da coerência dos dados.

A redução desses custos é então uma das principais preocupações dos projetistas de sistemas *software* DSM. Esta redução de custos permitirá que os sistemas *software* DSM se tornem uma solução viável para suprir o alto poder computacional exigido pelas aplicações científicas e comerciais, com excelente relação custo-benefício.

Várias formas para melhorar o desempenho em sistemas *software* DSM já foram desenvolvidas. Por exemplo, para minimizar o alto custo de acesso a dados remotos, devido à alta latência na rede de comunicação, os sistemas DSM costumam replicar os dados nas memórias privativas dos processadores do *cluster*. Apesar de ser eficiente a replicação dos dados compartilhados, os sistemas DSM necessitam implementar protocolos de coerência de memória. Os protocolos de coerência evitam que os dados replicados tenham valores diferentes entre as memórias privadas dos processadores. Desta forma, protocolos garantem uma visão única da memória para todos os processadores.

Os sistemas DSM podem, então, ser caracterizados pelos protocolos de coerência, tamanho da unidade de coerência e suporte aos escritores. Cada um destes pontos está abordado a seguir.

Tipicamente, sistemas *software* DSM se utilizam do suporte do sistema de memória virtual dos processadores para manterem a coerência dos dados compartilhados. Nesses sistemas, a unidade utilizada para manter a coerência de memória é a **página**. Entretanto, a grande quantidade de dados que pode residir em uma página, ocasiona dois problemas, o falso compartilhamento e a fragmentação. O **falso compartilhamento** ocorre quando dois ou mais processadores utilizam dados distintos que residem em uma mesma página, e pelo menos um dos processadores realiza operações de escrita. Este falso compartilhamento ocasiona a comunicação desnecessária destas páginas. A **fragmentação** ocorre quando apenas parte da página foi modificada por um dos processadores. Porém, toda a unidade de coerência deve ser transmitida pela rede aos outros processadores, aumentando a quantidade de dados trafegados.

O suporte a **múltiplos escritores**, é uma técnica que permite a redução dos custos associados à fragmentação e ao falso compartilhamento. Essa técnica permite a distinção exata das posições de memória escritas pelos processadores em uma página. Além disso, possibilita a propagação apenas das modificações efetuadas nas páginas. No contexto de sistemas *software* DSM essas modificações são chamadas de *diffs*. Elas representam as diferenças encontradas nas páginas antes e após as modificações.

O sistema *software* DSM utiliza os mecanismos de proteção de páginas para distinguir exatamente as posições de memória modificadas. Inicialmente as páginas são protegidas contra escritas. Na primeira tentativa de escrita à página, o sistema realiza uma cópia da página. Em seguida, ele desprotege a página permitindo que novas escritas ocorram livremente. No momento de propagação das modificações a página é comparada com sua cópia, palavra por palavra, para a criação dos *diffs*. O custo adicional introduzido pela técnica de múltiplos escritores, normalmente compensa os *overheads* causados pelo falso compartilhamento. Porém, é desnecessário para páginas onde esse fenômeno não ocorre.

Visando a redução da comunicação na rede, grande parte dos sistemas *software* DSM utilizam **invalidações** para propagarem as modificações. Quando ocorrem modificações nas páginas compartilhadas, os protocolos DSM baseados em invalidações, enviam notificações para os outros processadores, avisando que determinadas páginas foram modificadas. Essas notificações ocupam tipicamente poucos *bytes*. Somente quando algum processador necessita dos dados, os

mesmos são requisitados. Em contraste, os protocolos baseados em **atualizações** enviam sempre os dados modificados, a umentando o tráfego na rede.

Assim como o suporte a múltiplos escritores, os protocolos baseados em invalidações se utilizam dos mecanismos de proteção de páginas. Neste caso, quando um processador recebe uma invalidação, a página é protegida contra leitura. Na ocasião de uma tentativa de acesso a dados nesta página, ocorrerá uma falha de acesso. A falha é interceptada pelo sistema DSM para pedir a um ou mais processadores remotos que enviem suas modificações. Novamente, o custo introduzido pela técnica compensa o *overhead* de comunicação na maioria dos casos. Mas poderia ser evitado se o padrão de compartilhamento da aplicação fosse conhecido com antecedência. Por exemplo, as modificações poderiam ser propagadas apenas para processadores que conhecidamente vão utilizá -las.

Outra solução tradicional para redução dos *overheads* de comunicação e coerência, tem sido o desenvolvimento **de modelos de consistência de memória relaxados**. Esses modelos de memória estabelecem um compromisso entre desempenho e o modelo de programação. Eles introduzem um aumento de complexidade na interface do usuário buscando atingir um melhor desempenho. Essa complicação adicional no modelo de programação permite que o sistema DSM possa ser mais seletivo no que diz respeito à propagação dos dados entre os processadores. A idéia é reduzir a quantidade de comunicação na rede. É importante que esta complicação seja mantida num nível mínimo, pois simplificar o modelo de programação é um dos principais propósitos dos sistemas DSM.

A melhora de desempenho atingida pela implementação das técnicas mencionadas, apesar de significativa, não é satisfatória. Para melhorar ainda mais o desempenho dos sistemas *software* DSM outras alternativas vêm sendo avaliadas. Dentre estas, podemos destacar o suporte de hardware e os sistemas adaptativos, abordados a seguir.

Os sistemas DSM híbridos são basicamente constituídos pela adição de um **suporte de *hardware*** ao *hardware* convencional do *cluster*. Esse *hardware* específico auxilia no suporte as operações de coerência e comunicação. Como exemplo podemos citar o computador paralelo NCP2 (Núcleo de Computação Paralela) da COPPE/UFRJ[9]. No NCP2, esse *hardware* específico se encontra na forma de um controlador de protocolo associado a cada um dos nós da rede. Esse controlador de protocolo evita que o processador principal pare de executar a

aplicação (processamento útil) para tratar algumas das ações de coerência, e da comunicação entre processadores. Os processadores podem simplesmente disparar pedidos ao controlador e continuar sua operação normalmente. Em caso de pedidos de atualização dos dados feitos por processadores remotos o controlador os trata sem interromper o processador local.

Os DSMs híbridos permitem uma melhora significativa no desempenho com relação a um pequeno custo adicional do *hardware*. Contudo, estes sistemas se tornam menos genéricos tendo em vista a necessidade de um *hardware* específico.

A hipótese por trás dos **sistemas adaptativos** é a de que não existe um conjunto único de técnicas capazes de proporcionar um desempenho ótimo para todas as aplicações. Grande parte das técnicas de redução de custos de sistemas *software* DSM citadas anteriormente, funcionam bem para a maioria das aplicações. Entretanto, elas introduzem outros custos colaterais, já que mesmo em uma única aplicação, podemos observar padrões de compartilhamento diferentes dependendo do conjunto de dados. Neste intuito, os sistemas adaptativos buscam identificar os padrões de compartilhamento de dados da aplicação e tratá- los através de protocolos apropriados para aumentar o desempenho do sistema.

Os sistemas adaptativos se dividem basicamente entre os que são baseados em informações **estáticas** e os baseados em informações **dinâmicas**. Nos sistemas baseados em informações estáticas, o programador da aplicação deve classificar as estruturas de dados do programa segundo alguns padrões de compartilhamento suportados pelo sistema DSM em questão. O sistema pode então escolher o protocolo mais eficiente para manter a coerência dos dados de acordo com essas anotações feitas pelo programador. Esses sistemas requerem um suporte adicional por parte do compilador (*linkers* ou pré-processadores) além de colocarem um trabalho adicional para o programador da aplicação. Além disso, não podem detectar mudanças nestes padrões de compartilhamento em tempo de execução. Podemos destacar como exemplo o sistema *Munin*[12].

Os sistemas baseados em informações dinâmicas por sua vez são aqueles que se adequam as características das aplicações considerando informações observadas em tempo de execução. Estes sistemas permitem que diferentes adaptações possam ser realizadas de acordo com os padrões de compartilhamento dinâmico dos dados da aplicação.

Os sistemas adaptativos dinâmicos vêm recebendo uma atenção especial dos pesquisadores da área de *software* DSM. Esses sistemas apresentam um melhor desempenho global em relação aos sistemas híbridos e aos sistemas adaptativos estáticos. Eles têm a vantagem de não precisar de qualquer *hardware* adicional, compilador especial ou complicação para o programador. Como exemplo podemos citar os sistemas ADSM[25] e HAP[30]. Esses sistemas requerem, contudo, informações adicionais sobre o comportamento da aplicação, coletadas em tempo de execução.

Tradicionalmente, os sistemas adaptativos adaptam o protocolo de acordo como um conjunto pré-estabelecidos de padrões, conhecidos com antecedência. Por causa disto, estes sistemas costumam apresentar bom desempenho para aplicações que possuam padrões de compartilhamento **regulares**, dentro do universo de padrões capazes de serem identificados pelo sistema.

Apesar do bom desempenho em aplicações regulares, esses sistemas tratam de maneira conservadora as aplicações, ou conjuntos de dados de aplicações, que possuam padrões de compartilhamento irregulares e que não façam parte do conjunto de padrões que o sistema é capaz de identificar.

No intuito de melhorar o desempenho de aplicações de caráter **irregular**, Castro[14] desenvolveu uma máquina de estados finita, denominada **FIESTA** (*FInitE STAte machine*), e um algoritmo de categorização, denominado **RITMO** (*algoRIThM fOr FIESTA*).

FIESTA permite monitorar o comportamento de compartilhamento de dados entre processadores de forma dinâmica. A informação de compartilhamento de dados é representada pelo que denominamos de estados de compartilhamento. O **alto nível de detalhe** das informações contidas nos estados de compartilhamento permite uma maior precisão na escolha das ações adaptativas por parte do protocolo. O protocolo então se beneficia das características especí ficas da aplicação. Além do nível de detalhe, FIESTA propaga as informações sobre estados de compartilhamento por todos os processadores do sistema.

Com base nas informações da máquina de estados, o algoritmo de categorização de RITMO pode classificar os dados da aplicação de forma mais detalhada, capturando padrões irregulares que não são identificados por outras estratégias adaptativas. Ainda assim, RITMO classifica os dados da aplicação

utilizando categorias pré-estabelecidas. Apesar dos resultados desse trabalho serem bastante promissores, eles foram obtidos através de simulação.

Nesta tese propomos e avaliamos novas técnicas mais precisas de previsão de acesso a dados. Elas visam a detecção de **padrões genéricos** de compartilhamento no comportamento da aplicação, não necessariamente conhecidos com antecedência. Em contraste, a grande maioria das estratégias de adaptação encontradas na literatura buscam a detecção de padrões pré-estabelecidos conhecidos *a priori*.

As estratégias de previsão propostas se dividem em dois tipos. O primeiro, possui um conjunto de estratégias de previsão de acesso a dados baseados em preditores de desvios condicionais, semelhante aos encontrados nas arquiteturas modernas de processadores. Para estes preditores, diversos autômatos de previsão de desvios condicionais bem conhecidos foram avaliados. O segundo tipo de preditor é baseado em informações obtidas pela máquina de estados FIESTA.

Para avaliar a eficácia das estratégias de previsão baseadas em FIESTA, foi desenvolvido um novo protocolo adaptativo baseado em atualizações seletivas. Este protocolo foi implementado sobre o sistema *software* DSM *TreadMarks*, que é baseado em invalidações. A técnica de atualizações seletivas propaga as modificações de forma precisa. Assim, ele evita o custo imposto pelo protocolo de invalidações de *TreadMarks*. Esta técnica tem como objetivo minimizar o tempo de espera por dados remotos na aplicação, pois este é um dos principais custos de sistemas *software* DSM.

Devido a disponibilização da informação global sobre os estados de compartilhamento, a propagação das atualizações pode ser realizada de forma síncrona. Desta forma são evitados os custos adicionais de interrupções para as operações de comunicação das atualizações seletivas. Devido ao alto nível de detalhe das informações disponibilizadas por FIESTA, as atualizações são imediatamente aplicadas no momento em que são recebidas, evitando subseqüentes falhas de acesso. Além disso, a propagação das modificações é realizada através de *diffs*, eliminando o problema de fragmentação. O mais importante, é que a técnica pode abranger todas as páginas compartilhadas da aplicação, inclusive páginas com múltiplos escritores. Todo trabalho foi implementado sobre um sistema *software* DSM real e bastante difundido - *TreadMarks*.

Através de nossos experimentos, mostramos que nossas estratégias de previsão baseadas em estados de compartilhamento apresentam altas taxas de acerto. Este fato ocorre tanto para as aplicações regulares quanto para as aplicações irregulares. Devido a esta acurácia na previsão do padrão de compartilhamento, nossa técnica de atualização seletiva possibilitou a redução no tempo de execução das aplicações. Em alguns casos, as reduções atingiram 60% do tempo de execução das aplicações, em relação ao protocolo original. Além disso, não degradam o desempenho das aplicações que não obtiveram redução significativa, se comparadas com a versão original de *TreadMarks*. Isso mostra a importância de técnicas precisas na previsão de acesso a dados, para reduzi r os *overheads* de sistemas *software* DSM.

Contribuições da Tese

As principais contribuições desta tese são:

- Desenvolvimento de novas estratégias genéricas e precisas para a previsão de acesso a dados compartilhados;
- Adaptação de estratégias de previsão bem conhecidas para desvios condicionais, para serem utilizadas na previsão de acesso a dados em sistemas *software* DSM;
- Desenvolvimento de um novo protocolo adaptativo, que realiza a atualização seletiva de *diffs* de modo síncrono para reduzir o *overhead* de espera por dados remotos;
- Técnica que beneficia todas as páginas compartilhadas da aplicação, inclusive páginas que sofram de falso compartilhamento;
- A redução de interrupções para a propagação dos dados e das falhas de acesso às páginas compartilhadas;
- Implementação distribuída da máquina de estados FIESTA e das demais técnicas propostas em um *software* DSM real e amplamente difundido.

Organização

O trabalho está organizado da seguinte forma. No Capítulo 2, apresentamos as principais características dos sistemas DSM de uma maneira geral, abordamos o problema de coerência de memória, e alguns dos modelos de consistência que resolvem este problema. São também abordadas questões relativas a sistemas DSM implementados em *software*, mais especificamente, ao problema de falso compartilhamento e protocolos múltiplos escritores, a questão atualizações *versus* invalidações, e aos possíveis níveis de implementação dos sistemas *software* DSM. Além disso, enfocamos alguns aspectos da implementação do sistema *TreadMarks*, que foi o sistema utilizado como base para nosso trabalho. No Capítulo 3 descrevemos a máquina de estados finita FIESTA e sua implementação em um ambiente distribuído, assim como nossas duas estratégias de previsão de acesso a dados. No Capítulo 4 apresentamos nossa metodologia experimental e os resultados experimentais em relação a nossas estratégias de previsão. No Capítulo 5 apresentamos nossa técnica de atualização seletiva e avaliamos os resultados experimentais obtidos nessa técnica. No Capítulo 6 citamos alguns trabalhos relacionados. Finalmente, no Capítulo 7, destacamos nossas conclusões a respeito do trabalho realizado e apresentamos algumas direções de trabalhos futuros.

Capítulo 2

Sistemas *Software* DSM

Neste capítulo abordamos os principais conceitos e questões relacionadas à implementação de sistemas *software* DSM. Algumas destas questões, já abordadas no capítulo de introdução, serão examinadas mais detalhadamente. Primeiramente, discutimos a necessidade da replicação dos dados e dos modelos de consistência que garantem que a memória permaneça coerente. Tratamos da questão da granulosidade de coerência utilizada nos sistemas *software* DSM, assim como de alguns problemas relacionados a este aspecto, como a fragmentação e o falso compartilhamento. Em seguida, é explicado como os protocolos múltiplos escritores contornam estes problemas através dos mecanismos de *diffs* e *twins*. Abordamos também a questão como os dados podem ser propagados entre os processadores através de atualizações ou invalidações, e dos possíveis níveis de implementação de sistemas software DSM. Examinamos o funcionamento do sistema *TreadMarks*, que serviu de base para nosso trabalho. Finalmente, descrevemos algumas das formas de adaptação que já foram implementadas sobre *TreadMarks*.

2.1. Problema de Coerência

Uma maneira possível que um sistema DSM tem de compartilhar os dados de um programa entre processadores, é dividir os dados do programa pelo número de processadores. Os dados então são distribuídos entre as diversas memórias locais, mantendo cada processador ciente da localização destes dados. Isto pode ser feito através de uma tabela que indique qual o processador cuja memória local contém determinados dados. Quando um processador executa um acesso a um dado que esteja localizado em sua memória privada, o acesso acontece normalmente. Porém, quando houver a necessidade de acesso a um dado que esteja em uma memória remota, este dado deverá ser pedido ao processador cuja memória local contém

este dado. O processador que possui o dado irá então envi á-lo ao processador que realizou o pedido. Acessos a dados remotos ocorrem sempre desta maneira.

Uma outra possibilidade é permitir que os dados permaneçam locais em cada processador. Assim, possíveis acessos posteriores acontecem de maneira mais eficiente. No momento em que outro processador necessitar dos dados, eles serão transmitidos ao outro processador. Esse esquema se mostra mais eficiente porque possibilita um menor número de mensagens trocadas, e uma maior quantidade de acessos locais.

Em qualquer uma destas duas formas de implementação, apenas um processador possui determinado conjunto de dados num determinado instante de tempo. Isso garante o funcionamento correto dos programas. Porém, não consideram a possibilidade de permitir que diversos processadores tenham acesso ao mesmo tempo a um mesmo conjunto de dados. Por exemplo, aos dados com acesso somente para leitura. Neste caso reduziríamos significativamente o número de mensagens para acesso aos dados compartilhados.

O que se deseja é aumentar, o máximo possível, a quantidade de acessos locais em relação aos acessos remotos. Isso acarreta não só em um aumento na velocidade do tempo de acesso aos dados como também um aumento no grau de paralelismo da aplicação. Este efeito é atingido através da **replicação** dos dados compartilhados entre os diversos processadores, onde cada processador, agora, contém a cópia de todos os dados do programa. O problema dessa replicação é que no momento em que um processador efetuar uma modificação num dado compartilhado, esta modificação passa a não ser vista pelos outros processadores. Certamente, este procedimento levaria a uma execução errada do programa. É preciso, então, replicar os dados sem que a coerência da memória seja perdida.

Para tratar este problema de coerência da memória compartilhada, foram propostos vários modelos de consistência de memória na literatura[2]. Alguns destes modelos estão abordados a seguir.

2.2. Modelos de Consistência

A primeira questão a ser considerada é "quando" um processador deve ver as modificações feitas por outros processadores. Um modelo de consistência de memória especifica formalmente a ordem em que os acessos a dados compartilhados devem ser vistos pelos processadores de sistema.

2.2.1. Consistência Seqüencial

O modelo mais simples e intuitivo de consistência é chamado consistência seqüencial. Este modelo foi formalizado por Lamport[23] e diz que: "Um sistema é dito seqüencialmente consistente se o resultado de qualquer execução é o mesmo que se as operações de todos os processadores fossem executadas em alguma ordem seqüencial, e as operações de cada processador na ordem descrita pelo programa".

O modelo de consistência seqüencial, entretanto, restringe muito o paralelismo potencial do sistema. Um acesso a um dado compartilhado só pode ser realizado se o acesso anterior tiver sido observado por todos os outros processadores, o que deixa o processador paralisado por um período de tempo.

Garantir a consistência de memória empregando um modelo menos restritivo pode, portanto, ter grande influência no desempenho do sistema.

É possível relaxar as restrições impostas pelo modelo seqüencial quando se deixa explícito no programa os pontos em que a consistência de dados se faz necessária. Isto pode ser realizado através do uso de variáveis de sincronização. As primitivas de sincronização mais comumente utilizadas em programação paralela com memória compartilhada são os ***locks*** e as **barreiras**.

Os *locks* delimitam seções críticas de código que podem ser executadas por apenas um processador de cada vez. Os *locks* garantem exclusão mútua em determinados trechos do programa. Para entrar em uma dada seção crítica, o processador deve realizar uma operação de *lock* na variável de sincronização responsável por esta seção crítica. Caso haja algum outro processador executando a seção crítica neste momento, o processador que realizou a operação de *lock* deverá aguardar até que esta seção crítica seja liberada. Por outro lado, se a seção crítica estiver liberada, o processador passa a executar o código da seção crítica,

indicando para os outros processadores que esta se encontra ocupada. Uma vez terminada a seção crítica, o processador executa então uma operação *unlock*, o que torna a seção crítica novamente liberada para outros processadores.

Uma barreira é um mecanismo de sincronização que divide a execução do programa em fases. Quando um processador chega numa barreira, ele deve esperar que todos os outros processadores também cheguem a barreira. Ao final, depois que todos os processadores atingiram a barreira, eles são liberados e uma nova fase de computação começa.

Em programas que utilizam estes tipos de sincronização, podem ser empregados modelos mais *relaxados* de consistência de memória. Os modelos *release consistency* e *lazy release consistency* estão descritos a seguir.

2.2.2. Release consistency

O modelo *release consistency* foi desenvolvido por Gharachorloo *et al*[16]. Ele garante que a memória é consistente apenas em determinados pontos de sincronização. Os pontos de sincronização referem-se aos acessos às variáveis de sincronização. Esses acessos são divididos em dois tipos: *acquires* e *releases*. Um *acquire* indica que o processador está iniciando uma operação que pode depender de valores gerados por outro processador. A execução de um *release* indica que o processador está terminando uma operação que gerou valores dos quais outro processador pode depender.

Por exemplo, as primitivas de sincronização *lock* e *unlock* e barreira podem ser modeladas como operações de *acquire* e de *release*. A primitiva *lock* é modelada como um *acquire* enquanto que a primitiva *unlock* é modelada como um *release*. A chegada numa barreira corresponde a uma operação de *release* e a saída a um *acquire*.

As condições formais para que um sistema esteja consistente segundo o modelo *release consistency* são:

- Antes de executar um *release*, todos os acessos a dados compartilhados anteriores devem ter sido observados por todos os outros processadores;

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

terminado quando escritas posteriores em *s* realizadas por outros processadores não afetam o valor lido no *acquire*.

Segundo estas condições, no momento da execução de uma operação de *release* é necessário garantir que as escritas anteriores já foram observadas por todos os outros processadores. Ou seja, a saída de uma seção crítica sinaliza para os outros processadores que a seção crítica está livre, e o que todas as modificações realizadas dentro dela devem ser vistas por todos os processadores. Além disso, quando um processador executar um *acquire* e entrar em uma seção crítica, as escritas realizadas por outros processadores devem estar visíveis localmente.

O modelo r*elease consistency* requer significativamente menos comunicação do que um modelo consistente seqüencialmente. Sua interface é muito similar, dado que a sincronização explícita normalmente está presente em programas paralelos para garantir execução determinística.

2.2.3. Lazy Release Consistency

O modelo *lazy release consistency* é baseado no modelo *release consistency*, podendo ser considerado uma implementação *lazy* ou "preguiçosa" do mesmo. Este modelo foi proposto por Keleher *et al*[21]. No modelo *release consistency* todas as modificações são transmitidas a todos processadores no momento do *release*. Pois não há nenhum meio de conhecer antecipadamente se eles realmente vão utilizá-las. Enviar todos os dados para todos os processadores é a garantia de execução correta do programa.

No modelo *lazy release consistency* as modificações feitas por um processador são vistas pelo outro no momento do próximo *aquire* e não do *release*, quando um processo executa um *acquire*, além da garantia de exclusão mútua, o processo recebe também as atualizações do processo que por último executou um *release*. Na operação de *release*, nenhuma informação é propagada, apenas a seção critica é liberada.

Isso pode resultar em uma grande diminuição do número de mensagens, conseqüentemente melhorando o desempenho da aplicação. A Figura 2.1 exemplifica essa situação.

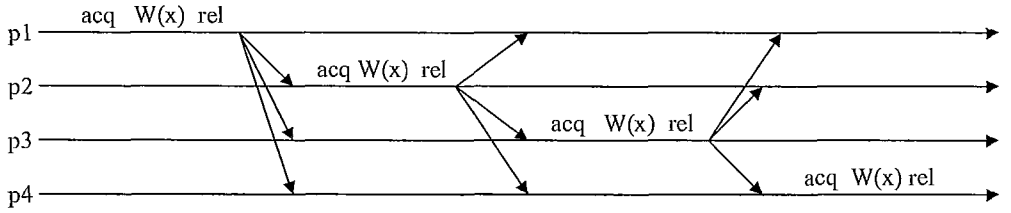


Figura 2.1 Tráfego de mensagens no modelo *release consistency*

Neste exemplo, quatro processadores (*p1* a *p4*) atualizam uma variável *x* dentro de uma seção crítica num sistema DSM que implementa o modelo *release consistency.*

Após *p1* ter conseguido o acesso a esta seção crítica, ele faz uma escrita na variável *x*. Mais tarde no *release*, essa informação é propagada para os outros processadores, *p2*, *p3* e *p4*. O próximo processador a entrar na seção crítica é *p2*. Ele pode eventualmente observar (ler) o valor escrito por p1, modificar *x* novamente e deixar a seção crítica. Quando isso acontece *p3* e *p4* recebem uma nova atualização de *x* que sobrepõe a primeira, e assim por diante. Ao final *p4* terá recebido três atualizações da variável *x*, sendo que destas, duas totalmente inúteis.

Observe agora o mesmo exemplo num sistema DSM implementando *lazy release consistency*, apresentado na Figura 2.2.

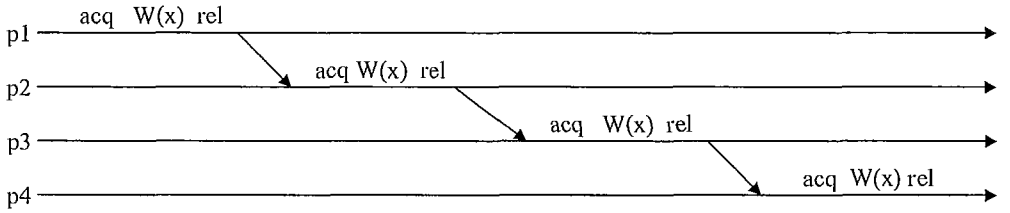


Figura 2.2 Tráfego de mensagens no modelo *lazy release consistency*

Agora as atualizações só são propagadas no *acquire*. Assim, as atualizações só são transmitidas ao processador que realmente precisa delas, reduzindo bastante o tráfego de mensagens. Uma outra grande vantagem disso, é que no caso de um processo executar um *acquire* em uma mesma variável de sincronização na qual ele próprio foi o último a executar um *release*, nada precisa ser feito. Além disso, no *acquire*, as modificações enviadas podem ser agrupadas juntamente com a concessão do acesso a seção crítica numa única mensagem. Podemos concluir então que *lazy release consistency* reduz não só a quantidade de mensagens, como

também a quantidade de dados trocados entre processadores em relação ao modelo *release consistency*. *TreadMarks* foi o primeiro sistema DSM a utilizaro modelo *lazy release consistency*.

2.3. Fragmentação e Falso Compartilhamento

Sempre que há a necessidade de manter a coerência entre memórias locais, é necessário detectar quando modificações acarretam em perda da coerência. Em algum momento é preciso que dados sejam transmitidos de um processador para o outro para que a coerência seja mantida. Para isso, deve-se determinar qual o tamanho da porção de memória a ser utilizada para que a coerência possa ser realizada da forma mais eficiente.

Em sistemas *hardware* DSM os blocos da memória *cache* ou até mesmo palavras de memória são bons candidatos para resolver o problema. Porém, em sistemas *software* DSM não é possível utilizar blocos de *cachê*. O *software* simplesmente não tem controle sobre as falhas de *cache*.

A solução adotada pelos sistemas DSM implementados em *software* foi a de utilizar o suporte de *hardware*, já existentes em muitas máquinas, responsável pelo gerenciamento de memória virtual. Desta forma, ele usa a **página** como unidade de compartilhamento. Os sistemas *software* DSM se aproveitam do fato de que em praticamente todas as arquiteturas com memória virtual, uma parte do gerenciamento desta memória é sempre feita por software. Por exemplo, um *bit* de página inválida pode ser manipulado pelo sistema operacional para que um posterior acesso ocasione uma falha de página. Isso possibilita que o sistema operacional busque a página no disco. Da mesma forma, um sistema DSM pode usar este *bit* para invalidar páginas que foram modificadas fazendo com que acessos àquela página resultem numa falha. Com isso o sistema DSM pode buscar a página atualizada remotamente. Mesmo em sistemas *software* DSM totalmente implementados no nível usuário é possível se utilizar granulosidade de página visto que muitos sistemas operacionais (como por exemplo, os sistemas tipo UNIX) permitem o controle de alguns *bits* de proteção de aplicações sobre suas páginas.

Esse aumento de granulosidade em *softwares* DSM, além de possibilitar um melhor desempenho nas trocas de mensagens via rede, também tem a vantagem de explorar de forma mais acentuada a localidade de referência exibida pelos programas. Por outro lado, o aumento na granulosidade acarreta outros problemas.

Um deles, a **fragmentação**, acontece porque mesmo que um processador necessite das modificações feitas em apenas uma variável, toda a página que contém aquela variável deverá ser buscada pela rede. Um outro problema, menos óbvio, é conhecido como **falso compartilhamento**.

O problema de falso compartilhamento ocorre quando variáveis compartilhadas não relacionadas estão localizadas na mesma página. Esse problema é melhor apresentado através de um exemplo, conforme ilustra a Figura 2.3.

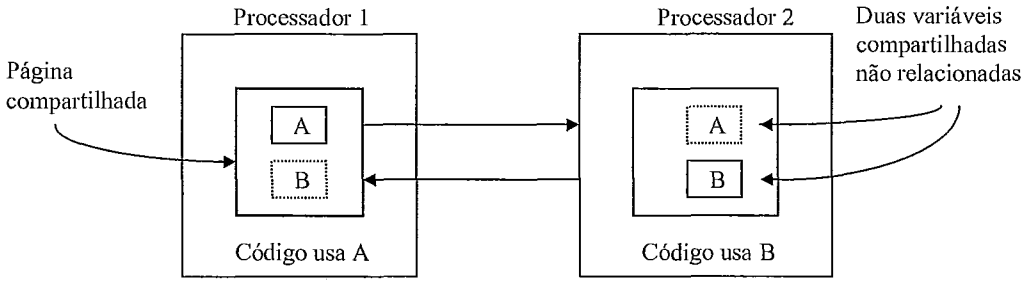


Figura 2.3 Falso compartilhamento numa página contendo duas variáveis não relacionadas

No exemplo as variáveis *A* e *B*, embora não relacionadas, foram alocadas numa mesma página. O processador 1 faz uso intenso da variável *A*, enquanto que o processador 2 faz o mesmo com *B*. Um sistema DSM usando granulosidade de página considera como se os dois processadores estivessem realmente compartilhando dados, quando que na verdade, eles não estão. Um exemplo prático, típico de aplicações científicas paralelas, de falso compart ilhamento poderia ser o de vários processadores empenhados em processar um vetor ou uma matriz de dados, que teria seu processamento dividido entre os diversos processadores. O problema acontece sempre que parte do vetor ou da matriz usada por processadores diferentes é alocada na mesma página. Quanto maior o tamanho da unidade de compartilhamento, maior a probabilidade deste fato ocorrer. Quando a granulosidade é de blocos de *cache* ou unidades menores, o falso compartilhamento, além de mais difícil de acontecer, tem efeitos negativos muito menores.

2.4. Protocolos com Suporte a Múltiplos Escritores

Um sistema pode migrar a página de um processador para outro para que o processador 1 enxergue as modificações feitas no processador 2 e vice versa. Este sistema é dito **Único-Escritor** (*Single-Writer*). Esta migração é desnecessária se

considerarmos o exemplo ilustrado na Figura 2.3, pois o processador 1 não está usando *B* enquanto que o processador 2 não usa *A*. Nos sistemas com único escritor, apenas um processador pode atualizar o conteúdo de uma página por vez. O processador que efetuou a última escrita na página (o único que tem a página atualizada) é chamado de **proprietário** (*owner*) da página. Quando um processador precisa modificar uma determinada página, deve pedi-la ao seu proprietário. Este, então, envia a página ao processador que fez a requisição juntamente com sua posse. Esse processador agora, passa a ser o novo proprietário da página e tem livre direito de escrita sobre ela. Esse efeito "*ping-pong*" sobre a página aumenta o número de mensagens trocadas desnecessariamente e o tempo de espera em cada processador influenciando negativamente no desempenho do sistema.

Para resolver este problema foram criados os protocolos **Múltiplos-Escritores** (*Multiple-Writers*). Estes sistemas permitem que dois ou mais processadores escrevam em uma página ao mesmo tempo, desde que as escritas sejam realizadas em regiões diferentes da página.

Neste protocolo, inicialmente, todas as páginas são replicadas pelos diversos processadores e devidamente protegidas contra escrita. Os acessos às páginas compartilhadas são controlados através dos *bits* de proteção do *hardware* de memória virtual. Enquanto apenas acessos de leituras estão sendo efetuados, estes são realizados sem que nenhuma ação especial seja necessária. Assim que ocorre a primeira tentativa de escrita, um procedimento de falha de página é disparado e o sistema DSM será chamado. Este fará, então, uma cópia idêntica da página em questão, chamada de ***twin***, que fica armazenada. Em seguida, o sistema desprotege a página permitindo que a escrita que causou a falha e subseqüentes escritas aconteçam sem nenhuma intervenção do sistema DSM. Isto pode estar acontecendo simultaneamente na mesma página em diversos processadores diferentes. Quan do as atualizações dessa página precisarem ser transmitidas, como por exemplo em um pedido remoto da página, o sistema fará então uma comparação, palavra por palavra, entre a página e seu respectivo *twin*. As diferenças encontradas possibilitarão saber quais mudanças ocorreram naquela página e principalmente, aonde elas aconteceram. Estas diferenças são colocadas numa estrutura de dados, chamada ***diff***, e transmitidas ao processador remoto.

2.5. Propagação das Informações

Uma outra questão importante, além de quando as informações entre processadores devem ser trocadas (modelos de consistência) é *como* essas informações são propagadas. Existem duas formas de se propagar as modificações feitas pelos processadores: atualizações e invalidações.

Na propagação por atualização, um processador envia para os demais, as modificações feitas por ele no momento das sincronizações.

Na propagação por invalidações, o processador responsável pelas modificações envia apenas uma notificação aos processadores remotos, de quais páginas foram modificadas por ele. Certamente tal notificação, por exemplo, o número da página, é consideravelmente menor do que as atualizações feitas pelo processador. O que se faz agora é propagar invalidações ao invés das atualizações.

Quando os processadores remotos recebem essas notificações eles devem imediatamente invalidar, usando os *bits* de proteção das páginas, todas as páginas relacionadas na notificação. Essa invalidação significa que o conteúdo da página não mais é válido, pois suas atualizações ainda não foram recebidas.

Assim que um dos processadores tentar efetuar um acesso àquela página, ocorre a falha de página. O sistema DSM é chamado e as modificações então serão requisitadas. Em sistemas que permitem múltiplos escritores, a criação de *diffs* pode ser também postergada até este momento. Só as variáveis realmente necessárias, aquelas que forem referenciadas, serão transmitidas, e somente para aqueles processadores que as utilizam.

Mesmo com um aumento do número de mensagens, e um custo adicional da interrupção de falha de página e do pedido das modificações, o uso de invalidações em sistemas *software* DSM é quase sempre vantajoso. Isso por causa da grande redução da quantidade de fluxo de dados na rede. Só são transmitidos dados aos processadores que realmente utilizam dados compartilhados.

2.6. Nível de Implementação

A priori, a implementação de sistemas DSM, está basicamente dividida entre implementar o sistema em *hardware* ou em *software*. Porém uma questão particular

de sistemas DSM implementados em *software* é em que nível essa camada de *software* deve se situar.

Certamente essa camada deve estar situada em uma camada inferior a das aplicações que se utilizem do sistemas. Dentre as diversas possibilidades estão: bibliotecas (nível usuário), sistemas operacionais, linguagens de programação ou possíveis modificações em compiladores, pré-processadores ou *linkers* de linguagens já existentes.

Basicamente essa escolha envolve uma relação entre o custo do sistema e seu desempenho. De modo geral, quanto mais próxima do *hardware* a implementação, maior seu desempenho e maior também seu custo. Conforme descemos a hierarquia, os sistemas ficam também cada vez menos genéricos e difíceis de serem utilizados ou implementados em outras plataformas.

2.7. TreadMarks

TreadMarks[22] é um sistema s*oftware* DSM desenvolvido na universidade de *Rice*, para ser executado em redes de estações de trabalho que utilizam sistemas operacionais UNIX comuns. Ele é implementado totalmente no nível de usuário, não necessitando de quaisquer modificações em sistemas operacionais ou compiladores especiais.

TreadMarks utiliza os serviços do sistema operacional UNIX para criação remota de processos, comunicação entre processos e gerenciamento de memória. A comunicação entre processos é feita através dos protocolos UDP/IP[1]. Como estes protocolos não tem confiabilidade, *TreadMakrs* implementa seu próprio protocolo no nível do usuário para garantir que as mensagens sempre cheguem a seus destinos.

O sistema *software* DSM *TreadMarks* utiliza um protocolo com suporte para múltiplos leitores e múltiplos escritores, implementa o modelo *lazy release consistency* e propaga as informações utilizando invalidações.

[1] User Datagram Protocol / Internet Protocol

2.7.1. Interface de *TreadMarks*

Como *TreadMarks* é totalmente implementado no nível do usuário, todos os procedimentos do sistema estão na forma de uma biblioteca que deve ser ligada a aplicação para ser utilizada. O sistema, como requer o modelo *lazy release consistency*, fornece ao programador as operações de sincronização *acquire* e *release* como chamadas de procedimentos, `Tmk_lock_acquire` e `Tmk_lock_release`. Além disso, é fornecido também um procedimento, `Tmk_barrier`, que implementa a operação de barreira. Essas três operações, formam basicamente a interface do programador com o sistema DSM.

Internamente o sistema trata toda a troca de mensagens entre os processadores para manter a ilusão de uma memória compartilhada ao programador.

Apesar destes três procedimentos constituírem basicamente a interface com o sistema, estes não são os únicos, outros procedimentos também existem. Porém estes realizam tarefas de propósito mais específico, melhor dizendo, cuidam principalmente da inicialização e término dos processos.

Um importante procedimento de inicialização é o `Tmk_startup`. Este procedimento é o responsável pela criação de processos em máquinas remotas e de estabelecer a comunicação entre estes. Antes da execução de `Tmk_startup` há somente um processo rodando em um processador. Quando essa rotina termina, já existem diversos processos distintos em processadores dif erentes, prontos para a iniciar a execução da aplicação.

Antes, porém, é preciso alocar a memória compartilhada a ser utilizada. Este procedimento deve ser realizado por apenas um processador utilizando a chamada `Tmk_malloc` que é similar à chamada *malloc* da biblioteca padrão C. Da mesma forma, existe uma chamada de liberação da memória compartilhada `Tmk_free`. Após a alocação da memória compartilhada, ela deve ser distribuída entre todos os processadores envolvidos. Para isso é utilizado um outro procedimento chamado `Tmk_distribute`.

Para que o programador possa identificar os processadores durante a execução, *TreadMarks* numera os processos e disponibiliza uma variável global `Tmk_proc_id`, cujo valor é o número do processo em questão. Processos podem

então utilizar esta variável, para conhecerem o seu identificador em tempo de execução.

Depois de todo o procedimento de inicialização, que inclui a criação de processos e alocação de memória, a memória alocada por `Tmk_malloc` pode ser utilizada pela aplicação como se fosse realmente um único espaço de endereçamento global. A aplicação segue, então, sua execução normal, se utilizando das operações de sincronização (*locks* e barreiras) sempre que for necessário obter dados compartilhados coerentes.

No final da do programa da aplicação o programador deve utilizar a rotina `Tmk_exit`, que será chamada por cada um dos processos para terminar a computação.

2.7.2. Lazy Release Consistency

A implementação de *TreadMarks* do modelo *lazy release consistency*, divide a execução do programa em **intervalos** representados por índices. A cada vez que um processo executa um acesso de sincronização, seja este um *lock* ou uma barreira, um novo intervalo começa e o índice do intervalo é incrementado. Intervalos num mesmo processador estão ordenados pela ordem do programa. Já intervalos em processadores diferentes são parcialmente ordenados. Como em sistemas distribuídos não há uma temporização global, a relação de **precedência** entre processos se torna relativa. Em *TreadMarks* diz-se que um intervalo num processador *p* precede um intervalo num processador *q* se o intervalo de *q* começa com o *acquire* correspondente ao *release* que terminou o intervalo em *p*.

A ordenação parcial de intervalos entre processadores é mantida em cada processador por um vetor de números inteiros, chamado de *vector timestamp* que contém uma entrada para cada processador. Cada uma das entradas representa os índices dos intervalos. A entrada índice *p* no *vector timestamp* do processador *p* é igual ao intervalo corrente deste processador. Todas as outras entradas correspondentes aos outros processadores contém o intervalo mais recente daquele processador conhecido por *p*.

2.7.3. Locks

Em *TreadMarks* cada variável de sincronização de *lock*, é associada a um processador gerente (*manager*). Todos os pedidos de *lock* são feitos a esse gerente.

A função dele é encaminhar, se necessário, o pedido de *lock* ao processador que por último realizou acesso ao *lock*. O gerente centraliza os pedidos de um mesmo *lock*.

No pedido de *lock* o processador deve informar todos os intervalos de todos os processadores que ele conhece. Como mencionado, esta informação se encontra de forma completa em seu *vector timestamp*, que é então enviado junto com o pedido do *lock*. O último processador que teve a posse do lock compara o vetor enviado com o seu próprio vetor de *timestamps* para conhecer quais intervalos de quais processadores o processador que está requisitando *lock* ainda não conhece. Ele então envia todas as notificações de escrita (*write-notices*) das páginas, ocorridas nestes intervalos. Note que devido a *TreadMarks* utilizar um protocolo baseado em invalidações, apenas as notificações são enviadas neste momento. Estas notificações estão associadas aos respectivos processadores e intervalos onde ocorreram.

Quando finalmente for concedido o acesso ao *lock* ao processador que o requisitou, este deverá invalidar as páginas para as quais as notificações foram enviadas. Qualquer tentativa de acessar estas páginas resultará em uma falha de página.

2.7.4. Barreiras

As operações de sincronização de barreiras são semelhantes às operações de sincronização de *locks*, com a diferença que todos os processadores estão envolvidos na sincronização. O que ocorre então é que no final da operação cada processador recebe todas as notificações de escrita, para t odos os intervalos que ele ainda não conhece. Os processadores incorporam as modificações da mesma maneira como fazem nas operações de *lock*.

Assim como no *lock*, cada variável de sincronização de barreira é associadoaa um processador gerente. Este processador fica responsável por todo o processamento envolvido no evento de barreira. Todos processadores se comunicam com o gerente, que ao final do evento, se comunica novamente com todos os demais processadores. Ele é o responsável pela recepção, comparação e disseminação dos intervalos.

2.7.5. Falhas de Acesso

No início da computação somente o processador zero contém as páginas com os dados do programa. Todos os demais processadores contêm estas respectivas páginas inválidas. Se numa falha de acesso, independente da falha ser causada por uma leitura ou escrita, o processador não tem a página presente ainda ele a pede ao processador 0. Isso só acontece no início do processamento, após essa fase inicial, apenas *diffs* são trocados (devido a característica múltiplos escri tores do protocolo).

Sempre que ocorre uma falha devido a uma escrita, um *twin* da página é criado. Além disso, se o conteúdo da página não estiver válido, é necessário que se faça, antes da criação do *twin*, os pedidos (a processadores remotos) e a aplicação dos *diffs* para que a página se torne válida. Ao final, a página é totalmente desprotegida, e os acessos posteriores ocorrem livremente. Em falhas de leitura a página está inválida e apenas são pedidos e aplicados os *diffs* correspondentes. Neste caso só a proteção contra leitura é retirada, a página permanece protegida contra escrita.

2.8. Sistemas Adaptativos

Os sistemas adaptativos implementados sobre *TreadMarks*, empregam principalmente duas formas de adaptação. São elas a adaptação Único Escritor / Múltiplos Escritores e a adaptação Invalidação / Atualização. Estas formas de adaptação estão abordadas a seguir.

2.8.1. Único Escritor / Múltiplos Escritores

A adaptação único-escritor/múltiplos-escritores buscam evitar o custo imposto pelos protocolos múltiplos-escritores para as páginas onde o fenômeno do falso compartilhamento não ocorra.

Para realizar a adaptação entre um e vários escritores, o protocolo constantemente verifica quais páginas apresentam escritas concorrentes em um intervalo, realizadas por mais de um processador. Esse conjunto de páginas é tratado segundo os mecanismos de *twins* e *diffs*. As demais páginas, que não apresentam escritas concorrentes, são tratadas segundo um protocolo único escritor, sendo transferidas inteiramente entre os processadores.

2.8.2. Invalidação / Atualização

Os protocolos baseados em invalidação normalmente reduzem o tráfego na rede quando comparados aos protocolos baseados em atualizações. Desta forma, eles buscam aumentar o desempenho das aplicações. Contudo, os protocolos baseados em invalidações introduzem um tempo, ainda alto, de espera por dados remotos causado pela necessidade da falha de acesso, pedido e espera pela resposta das modificações.

Os sistemas que implementam a adaptação invalidação/atualização buscam caracterizar os padrões de compartilhamento das páginas das aplicações. A caracterização é utilizada para propagar as atualizações, eliminando assim o custo dos pedidos a dados remotos sem, contudo, incorrer em um aumento de tráfego na rede. Isso é possível, uma vez que, se o padrão de compartilhamento das páginas é determinado, o protocolo pode evitar as atualizações desnecessárias e, em conseqüência disto, evitar o aumento de tráfego na rede.

As páginas onde não se consegue uma caracterização precisa, ou cujo padrão de compartilhamento mude de acordo com o tempo, são tratadas através de invalidações.

Capítulo 3

Estratégias de Previsão de Acesso a Dados

Neste capítulo descrevemos duas técnicas de previsão de acesso a dados para sistemas *software* DSM. Uma delas baseada em estados de compartilhamento das páginas, e a outra, baseada em autômatos de previsão de desvios condicionais. Cada uma destas técnicas está descrita a seguir.

Uma grande parte dos sistemas adaptativos encontrados na literatura visa otimizar os sistemas através da detecção de padrões conhecidos *a priori*[27]. Estas estratégias buscam, dentro de um conjunto fixo de padrões pré-definidos, identificar padrões específicos de compartilhamento dos dados da aplicação. Como exemplo podemos citar os padrões **produtor-consumidor e migratório**. No padrão produtor-consumidor um processador realiza as operações de escrita no dado e um ou mais processadores efetuam operações de leitura ao dado. No padrão migratório cada processador de um conjunto de processadores, lê e escreve no dado de forma alternada.

Em contraste com essas técnicas, estamos buscando estratégias genéricas de previsão de acesso a dados. Neste caso, não existe um conjunto de padrões conhecidos a princípio, e as estratégias visam detectar quaisquer padrões de compartilhamento exibido pela aplicação. Isso pode ser vantajoso, pois é possível detectar e prever padrões de compartilhamento que não estejam pré-estabelecidos.

Suponha que, uma determinada página da aplicação possa apresentar um padrão onde algumas fases têm características produtor-consumidor e que outras fases apresentam um comportamento migratório. Além disso, suponha que entre estes períodos, algumas fases apresentam múltiplos escritores. Se esse comportamento se repetir, uma estratégia de previsão genérica pode observar toda a seqüência de fases como um padrão único. Essa estratégia permite então uma

maior precisão na determinação do padrão e conseqüentemente uma maior taxa de acertos. Neste mesmo exemplo, uma técnica tradicional poderia observar cada um dos "sub-padrões" isoladamente, sem perceber que estes fazem parte de um padrão maior. Essa visão limitada do padrão de compartilhamento pode levar a repetidas tentativas de aprendizado. Este fato degrada a precisão da técnica, ou até mesmo ignorar que exista algum padrão de compartilhamento na página.

Para reforçar nossa tese, alguns exemplos se fazem necessários. Nas tabelas que mostramos a seguir, a coluna **R** indica os processadores leitores e a coluna **W** os processadores escritores. Nas linhas são mostradas as diversas fases da aplicação, ou seja, o intervalo entre duas barreiras. Nesses exemplos, foram utilizados apenas quatro processadores.

A Tabela 3.1 mostra uma seqüência de estados de compartilhamento para uma página da aplicação *Conjugate Gradient* (CG). Este padrão de compartilhamento da página se repete sucessivamente, durante várias fases da aplicação. Podemos observar que, apesar de apresentar múltiplos escritores, o padrão de compartilhamento da página é bastante previsível. Sistemas adaptativos mais conservadores normalmente negligenciam as páginas com múltiplos escritores, perdendo assim oportunidades para melhorar o desempenho das aplicações. Neste exemplo, um protocolo capaz de realizar atualizações em páginas com múltiplos escritores, poderia utilizar a previsão para enviar antecipadamente as modificações dos processadores 0 e 1 ao final da fase de escrita para os processadores 1, 2, 3 e 0, 2, 3 respectivamente. Estas atualizações evitam subseqüentes falhas de páginas e pedidos de modificações.

R	W
{ 2, 3 }	{ 0, 1 }
{ 0, 1, 2, 3 }	{ }
{ 0, 1, 2, 3 }	{ }

Tabela 3.1 Seqüência de estados de compartilhamento em uma página da aplicação CG.

A Tabela 3.2 ilustra uma seqüência de estados de compartilhamento retirada da aplicação *Multigrid* (MG). Esta seqüência de estados também se repete durante algumas fases da aplicação. Neste exemplo temos uma página com um único produtor e múltiplos consumidores. Porém, esta simples categorização da página, associada ao conjunto de processadores consumidores pode causar algumas ineficiências. Isto porque ela despreza o fato de haverem dois sub-padrões na

página. Em um destes sub-padrões, apenas o processador 0 é consumidor e no outro sub-padrão, são consumidores os processadores 0, 2 e 3. Um protocolo que se utiliza apenas da classificação produtor-consumidor para realizar atualizações, pode acabar enviando mensagens adicionais de modificações aos processadores 2 e 3, ou incorrer em novas tentativas de aprendizado.

R	W
{0, 2, 3}	{}
{0, 2, 3}	{}
{0, 2, 3}	{1}
{}	{1}
{0}	{}
{0}	{}
{0}	{1}
{}	{1}

Tabela 3.2 Seqüência de estados de compartilhamento em uma página da aplicação MG

Também podemos notar no exemplo da Tabela 3.2 que existe um intervalo de uma fase entre a primeira escrita feita pelo processador 1 e as demais leituras. Através do exemplo não é possível afirmar se o processador 1 realiza escrita nas duas fases onde ele aparece como escritor, ou apenas na primeira delas. Contudo, pode ser importante enviar atualizações apenas na fase onde as leituras se iniciam e não ao fim da fase de escrita. Isso evita que não haja a necessidade de envio de novas atualizações a cada fase de escrita.

R	W
{}	{3}
{}	{2, 3}
{}	{1}
{0, 2, 3}	{}
{2, 3}	{0}

Tabela 3.3 Seqüência de estados de compartilhamento em uma página da aplicação IS

Nosso último exemplo é retirado de uma das páginas da aplicação *Integer Sort* (IS). Esta aplicação possui um padrão de compartilhamento essencialmente migratório. Porém, a Tabela 3.3 ilustra uma seqüência de estados de compartilhamento de uma das páginas da aplicação, onde ocorrem múltiplos escritores concorrentes. Este tipo de padrão é comum em algumas aplicações e ocorre normalmente em fronteiras de compartilhamento de dados entre processadores. Um protocolo adaptativo, com uma atuação conservadora em relação à páginas com múltiplos escritores, poderá realizar sucessivas tentativas de

categorizar a página como migratória. Porém, ele sempre encontrará dificuldade de definir o padrão na fase onde ocorrem as escritas concorrentes. Mais uma vez, um protocolo capaz de antecipar o envio de modificações para páginas com múltiplos escritores poderá, neste caso, enviar antecipadamente as modificações dos processadores 2 e 3 para o processador 1 com base na previsão precisa do padrão da página.

Por outro lado, a detecção de padrões genéricos pode apresentar como desvantagens um tempo de aprendizado inicial maior. Além disso, apresenta a necessidade de armazenar uma maior quantidade de informações de estado, por períodos mais extensos. Por isso é importante avaliar o custo introduzido pelas técnicas de predição genéricas, uma vez que grande parte das aplicações apresenta padrões simples de compartilhamento. Idealmente, a estratégia deve ser genérica o suficiente para capturar padrões complexos, e eficiente o suficiente para não introduzir um alto custo em aplicações regulares.

3.1. FIESTA

FIESTA (*FInitE STAte machine*) é uma máquina de estados finita definida por[14]. Seu objetivo é captar informações detalhadas a respeito do compartilhamento das páginas efetuado por processadores em sistemas DSM. Para isso, FIESTA define o **estado de compartilhamento** de uma página como sendo a união de quatro conjuntos denominados *R*, *W*, *P* e *C*.

O conjunto ***R*** é conhecido como o **conjunto dos leitores** e é constituído por processadores que efetuaram leituras à página fora de seções críticas. O conjunto ***W*** por sua vez é o **conjunto dos escritores**. Ele é constituído por todos aqueles processadores que efetuaram escritas à página também fora de seções críticas. Os conjuntos ***P*** e ***C*** são chamados de **conjunto dos produtores** e **conjunto dos consumidores**, respectivamente. O conjunto *C* contém todos os processadores que realizaram acessos de leitura na página dentro de seção crítica. Finalmente, o conjunto *P* contém todos os processadores que efetuaram acessos de escrita na página dentro de seção crítica.

Os processadores são incluídos ou excluídos do estado de compartilhamento conforme a ocorrência de cinco tipos de eventos que acontecem em uma página durante a execução da aplicação. Os tipos de eventos são: falhas de leitura fora de

seções críticas, falhas de leitura dentro de seções críticas, falhas de escrita fora de seções críticas, falhas de escrita dentro de seções críticas e invalidações.

Se um processador provocar um evento de leitura fora de seção crítica em uma dada página, este processador será incluído no conjunto *R* (conjunto dos leitores) do estado de compartilhamento da página. Por outro lado, se o evento for uma leitura dentro de seção crítica, o processador deverá ser incluído no conjunto *C* (conjunto dos consumidores) da página em questão. O mesmo acontece para os eventos de escrita. Se um processador causa um evento de escrita fora de seção crítica em uma página, este deverá ser incluído no conjunto *W* (conjunto dos escritores) do estado de compartilhamento da página. Caso o evento seja uma escrita dentro de seção crítica, este será incluído no conjunto *P* (conjunto dos produtores) da página. No caso da ocorrência de um evento de invalidação, o processador que provocou o evento será retirado do conjunto do estado de compartilhamento onde estava antes da ocorrência do evento (se ele estava em algum dos conjuntos).

3.1.1. Implementação Distribuída de FIESTA

Em um ambiente distribuído, cada processador mantém uma visão própria do estado de compartilhamento da página. Além disso, eventos provocados por um processador não são imediatamente observados por outros processadores do sistema. O problema é como manter a consistência do estado de compartilhamento da página através dos processadores do sistema, para que estes observem um mesmo estado de compartilhamento da página, em alguns instantes determinados da execução. Essa é uma tarefa semelhante a do próprio sistema DSM como um todo, cuja finalidade é de manter a coerência da memória compartilhada da aplicação.

Para cumprir sua finalidade, o sistema DSM divide a aplicação em intervalos. Um novo intervalo começa a cada nova sincronização realizada pela aplicação. No decorrer da execução, o sistema DSM guarda todos os eventos de escrita que ocorrem nas páginas, e os associa aos intervalos nos quais estes ocorreram. Nas sincronizações, um processador pode então informar a outros processadores os eventos de escrita que estes ainda não conhecem, baseados nos intervalos que cada um destes processadores observa dos outros processadores do sistema.

Para a implementação distribuída de FIESTA, precisamos simplesmente estender o mecanismo do sistema DSM para nossos cinco eventos de interesse.

Precisamos observar não só eventos de escrita, mas também os eventos de leitura. Mais do que isso, é necessário discernir se estes eventos estão ocorrendo dentro ou fora de seções críticas. Feito isso, estes eventos são distribuídos entre os processadores de uma maneira idêntica a do sistema DSM, que distribui seus eventos de interesse, nas sincronizações, utilizando o mecanismo de intervalos. Nestes instantes os processadores têm conhecimento de eventos que ocorreram em outros processadores.

Os eventos adicionais, que são propagados por FIESTA, são simplesmente apensados nas mensagens contendo os eventos originais do protocolo. Com isso o número de mensagens do protocolo se mantém inalterado. O que ocorre é apenas um aumento no tamanho das mensagens trocadas nas sincronizações.

A conseqüência desta forma de implementação é que os estados de compartilhamento das páginas definidos por FIESTA ficam consistentes juntamente com as páginas compartilhadas da aplicação nos instantes de sincronização.

3.2. Previsão Baseada em Autômatos de Previsão de Desvios

As técnicas de previsões de desvios implementadas nas arquiteturas de processadores são bem conhecidas e têm produzido excelentes resultados[31]. Assim, nesta segunda estratégia de previsão, estamos interessados em investigar em que medida as técnicas de previsão de desvios, podem também ser eficazes na previsão de falhas de páginas em sistemas *software* DSM.

A avaliação das técnicas de previsão de desvios também possibilita a comparação das estratégias de previsão baseadas em FIESTA com técnicas bem conhecidas e desenvolvidas com o propósito específico de realizar previsões. FIESTA, ao contrário, coleta uma quantidade muito maior de informações que permite implementar funções que não sejam simplesmente a de realizar previsões. Nas seções que seguem, vemos como o maior detalhamento das informações pode ser aproveitado por técnicas de tolerância a latência na busca de dados remotos, que se utilizem das previsões de acesso a dados compartilhados.

É importante notar que apesar do fenômeno de falha de página e de fluxo de controle apresentarem dois problemas de previsão distintos, observamos que as técnicas de previsão de desvios podem potencialmente ser aplicadas com sucesso na previsão de falha de acesso a dados em sistemas *software* DSM.

A idéia é simplesmente assumir que cada falha de acesso à uma dada página se comporta como um desvio condicional. Isto é, ela pode ocorrer ou não em função da história das ocorrências anteriores quando na mesma situação de desvio. Em outras palavras, a falha de acesso a uma página num determinado processador, em uma dada fase da computação acontecerá ou não, em função da história passada de falhas da página. É importante observar que esta correlação entre desvios condicionais e acessos à páginas só é possível uma vez que as aplicações em sistemas DSM são divididas em intervalos discretos, que constituem as fases da aplicação. Uma vez feita essa analogia, as técnicas de previsão de desvio se aplicam quase que diretamente na previsão de falhas de páginas.

Os preditores de desvios condicionais recebem como entrada os resultados dos desvios condicionais executados pela aplicação. Eles produzem como saída a previsão dos resultados da próxima execução desses desvios, ou seja, se eles serão realizados ou não. Em nosso caso, o preditor tem como entrada as falhas de páginas realizadas pela aplicação e prevê se uma falha de acesso ocorrerá na próxima fase.

Neste tipo de estratégia, cada processador realiza previsão apenas de seu próprio padrão de acesso a dados. Isso porque estas previsões são feitas utilizando apenas as falhas de página locais de cada processador. Esse tipo de preditor é melhor utilizado em conjunto com técnicas que necessitem apenas de informações locais, como por exemplo, a técnica de *prefetching*. Nessa técnica, cada processador pede antecipadamente as páginas que este julgue necessárias, aos outros processadores. Neste caso, a estratégia de previsão pode determinar as melhores páginas a serem buscadas com antecedência.

Nesse trabalho, testamos alguns autômatos de previsão de desvio, além de esquemas de previsão de dois níveis conforme descritos em [31]. Uma vez que este trabalho representa uma primeira tentativa de se utilizar preditores de desvios para previsão de acesso a dados compartilhados em sistemas *software* DSM, foram escolhidos autômatos clássicos utilizados na previsão de desvios condicionais. Estes autômatos servem de referência nos trabalhos de previsão de desvios. O preditor *Cosmos*[27] realizou uma avaliação do preditor de dois níveis de Yale Patt[31] aplicado para o problema de previsão de acesso a dados em multiprocessadores com coerência de *cache*. Apesar deste trabalho não realizar nenhuma interferência no protocolo, ou seja, utilizar os resultados das predições para redução do tempo de execução das aplicações, as taxas de acerto obtidas

foram promissoras. Neste sentido estamos avaliando também a aplicação dos preditores de dois níveis em sistemas *software* DSM.

Os autômatos utilizados estão representados na Figura 3.1 sob a forma de diagramas de transição de estados. Note que para cada um dos processadores do sistema e para cada uma das páginas da aplicação, existe um autômato associado.

Nos diagramas, os círculos representam estados e as setas representam transições. Dentro de cada estado, temos um número que identifica o estado e, separado por uma barra, o resultado da previsão deste estado. Este pode prever que a falha ocorrerá (T) ou não (N). Ao lado da seta temos o evento que provocou a transição de estados no autômato, que pode indicar que a falha realmente ocorreu nesta fase (T) ou não (N).

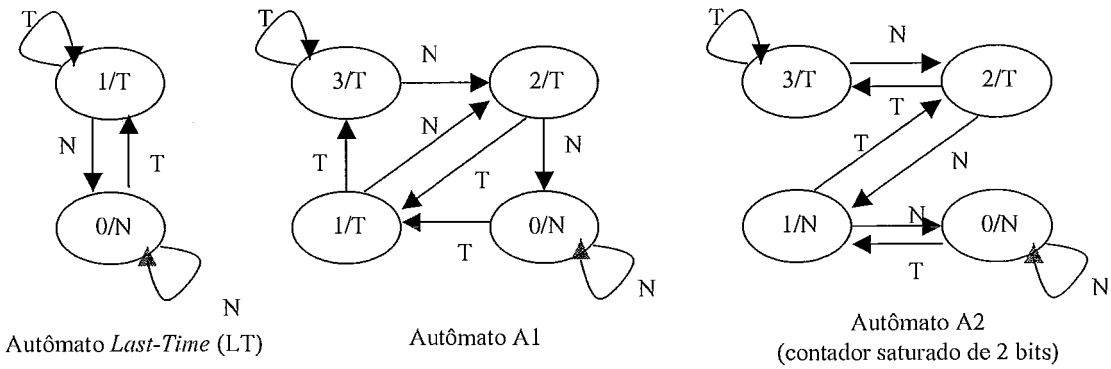


Figura 3.1 Autômatos utilizados na previsão de desvios

O autômato *Last-Time* só armazena o resultado da última fase da aplicação. Se nesta última fase o processador provocou uma falha de acesso, o autômato prevê que esse processador voltará a provocar uma nova falha de acesso na próxima fase. Caso o processador não tenha provocado uma falha de acesso na fase presente, a previsão é a de que este processador não provocará uma falha nesta página.

O autômato A1 guarda informação da história dos últimos dois acessos à página. Somente quando não ocorrer falha durante duas fase seguidas, esse autômato prevê que uma falha não ocorrerá. Caso contrário, a previsão ficará sempre a favor da ocorrência de falha.

O autômato A2 é conhecido com *contador saturado de 2 bits*[29]. Em cada fase onde ocorreu uma falha na página, o contador é incrementado, e a cada fase onde não ocorreu uma falha, o mesmo é decrementado. Porém, esse contador nunca é maior que 3 nem menor que 0. Enquanto o valor do contador é zero ou um, a previsão indica que uma falha não ocorrerá na próxima fase. Caso contrário, quando o contador for 2 ou 3, a previsão é positiva em relação a falha de página.

Além de utilizarmos os autômatos já mencionados de maneira simples, realizamos também experimentos com os mesmos autômatos utilizando esquemas de previsão de dois níveis. Este esquema está descrito a seguir.

Seguindo o mecanismo usado para os preditores de desvios, temos no primeiro nível um histórico dos últimos ***k*** acessos realizados à página pelo processador em questão. No segundo nível, temos o comportamento das últimas ***s*** ocorrências deste padrão.

A implementação deste preditor possui o seguinte esquema. Para cada página, e para cada processador, existe uma palavra de memória de *k bits* chamada de **Palavra de Histórico de Acesso a Página (PHAP)**. Esta palavra é utilizada para armazenar o histórico de acesso à página nas últimas *k* fases do processador. A cada fase, a palavra de memória sofre uma operação de *shift* para a esquerda. O seu *bit* mais à direita é atualizado com 1 ou 0 dependendo se o processador provocou uma falha de acesso ou não na página, respectivamente, durante esta fase. PHAP é utilizada para indexar um vetor de 2^k entradas, chamado de **Tabela de Padrões de Histórico (TPH)**. Cada entrada da TPH implementa um autômato finito. Ela guarda informações sobre as últimas *s* ocorrências do histórico correspondente a essa entrada. Este autômato também é atualizado com o resultado do acesso à página nesta fase. Este esquema é ilustrado na Figura 3.2.

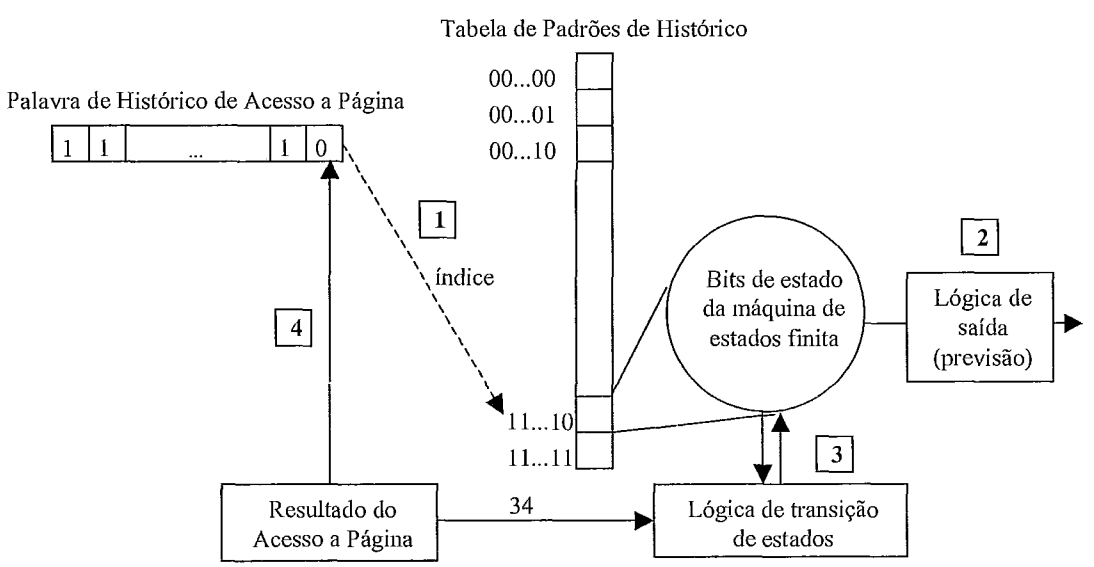


Figura 3.2 Esquema de previsão utilizando dois níveis

A previsão e atualização das tabelas ocorrem da maneira ilustrada na Figura 3.2. Ao final de uma barreira, a previsão para a próxima fase é realizada usando o conteúdo da PHAP para indexar a tabela TPH, conforme indicado no passo 1. A lógica de saída da máquina de estados é aplicada aos *bits* de estado da entrada correspondente (passo 2). A saída da máquina é a própria previsão. Quando a barreira termina, o processador inicia o processamento de uma nova fase. Toda falha de página que ocorre durante a fase é guardada em um *bit* chamado de **RAP (Resultado de Acesso à Página)** associado à cada uma das páginas. No final da fase se inicia uma nova barreira. Neste início de barreira, a lógica de transição de estados da máquina de estados é utilizada para atualizar os *bits* de estado da entrada da TPH. A TPH é indexada pela PHAP da barreira anterior, conforme indicado no passo 3 da Figura 4.5. A atualização é realizada utilizando a informação armazenada no RAP. Após a atualização da THP, a própria PHAP é atualizada, realizando-se uma operação de *shift* à esquerda do RAP com a PHAP (passo 4). Esse processo se repete em cada fase da aplicação.

3.2.1. *Overhead* de Memória

O *overhead* de memória das estratégias de previsão não deve ser negligenciado. Para que as previsões possam ser acuradas, é necessário armazenar uma grande quantidade de informações sobre o passado.

Nas estratégias baseadas em preditores de desvios, esse *overhead* depende do autômato utilizado, e também, se a estratégia utiliza ou não um esquema de dois níveis. O autômato LT precisa apenas de 1 bit para armazenar informação sobre a fase anterior, por página compartilhada da aplicação. Os autômatos A1 e A2 precisam ambos de 2 bits por página para implementação dos quatro estados. Para os esquemas de dois níveis, o *overhead* de memória por página OH_m pode ser dado pela expressão $OH_m = k + 2^k \times OH_a$, onde k é o número de bits da PHAP e OH_a é o overhead de memória do autômato utilizado. Por exemplo, a estratégias de dois níveis com uma PHAP de 8 bits (k = 8) e utilizando o autômato A2 ($OH_a = 2$) utiliza 1032 bits por página. É importante observar que essa quantidade de memória armazena apenas o padrão de acesso local da página. Para capturar o padrão dos outros processadores, é preciso que cada processador mantenha uma estrutura dessa para cada um dos N processadores do sistema, o que resulta em um *overhead* N vezes maior.

3.3. Previsão Baseada em Estados de Compartilhamento das Páginas

Como vimos na seção 3.1, FIESTA fornece informação global sobre o estado de compartilhamento de cada página. Para cada página, FIESTA mantém atualizados quatro conjuntos *R*, *W*, *P* e *C* de leitores, escritores, produtores e consumidores da página, respectivamente. Podemos observar que se estes conjuntos forem amostrados nos pontos de sincronização (por exemplo, nas operações de barreira) da aplicação, podemos então descrever o padrão de acesso a dados da aplicação através desta seqüência de estados de compartilhamento captada. A estratégia de previsão baseada em estados de compartilhamento visa a detecção de padrões repetitivos de seqüências de estados de compartilhamento das páginas. A amostragem é realizada em operações de barreira, de modo que, uma vez observado um conjunto de estados de compartilhamento, os estados seguintes possam ser estimados.

Ao contrário das estratégias baseadas em preditores de desvios, as estratégias baseadas em estados de compartilhamento utilizam informação global para realizarem as previsões. Por isso, esse tipo de técnica pode ser utilizado para dar suporte a técnicas que necessitem de informações globais. Um exemplo é a técnica de atualizações seletivas. Nesse tipo de técnica, um processador envia antecipadamente aos demais as modificações efetuadas nas páginas, sem a necessidade de pedidos. Para que as atualizações propagadas sejam de fato úteis, é importante que haja uma boa previsão global de acessos futuros.

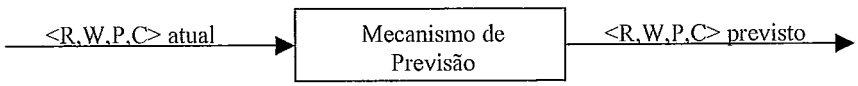


Figura 3.3 Interface do mecanismo de previsão

A estratégia de previsão de estados de compartilhamento possui o seguinte comportamento. Para cada página compartilhada, existe um mecanismo de predição como ilustrado na Figura 3.3. A cada operação de barreira da aplicação, esse mecanismo é alimentado com o estado global de compartilhamento da página dado pelos conjuntos *R*, *W*, *P* e *C* fornecidos por FIESTA. Como resultado, o mecanismo fornece a previsão do estado de compartilhamento para a próxima fase. Uma fase é definida como o intervalo entre duas barreiras.

Esse mecanismo é implementado através de uma máquina de estados finita, conforme mostra a Figura 3.4. Esta máquina de estados é composta de quatro estados, denominados: **Aprendendo**, **Verificando, Predizendo e Ocioso**.

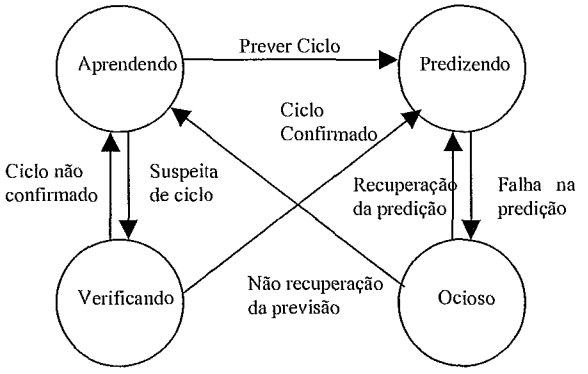


Figura 3.4 Máquina de estados para previsão de estados de compartilhamento

A máquina inicia no estado **Aprendendo**. Neste estado os conjuntos de compartilhamento fornecidos ao mecanismo são armazenados em seqüência. A cada nova operação de barreira, o algoritmo de previsão compara o estado global de compartilhamento com os estados de compartilhamento previamente armazenados na seqüência. Com isso, o algoritmo visa encontrar ciclos de estados de compartilhamento das páginas da aplicação. A idéia de procurar ciclos é baseada no fato das aplicações científicas paralelas serem formadas tipicamente por um conjunto de *loops*, muitas vezes aninhados, com operações de barreiras separando estes *loops* em fases. Em geral, esse comportamento cíclico no padrão de acesso a dados pelos processadores, aumenta o grau de previsibilidade de falhas de acesso. Enquanto permanecer no estado aprendendo a previsão é sempre um conjunto vazio, ou seja, não há previsão.

Uma vez que se identifique uma repetição no padrão de compartilhamento, ou seja, há uma "suspeita de ciclo", a máquina passa para o estado **Verificando**. Neste estado, o algoritmo de previsão tenta confirmar o ciclo, comparando os estados de compartilhamento globais, com os estados de compartilhamento armazenados na seqüência. Se o ciclo não se confirmar, a máquina retornará ao estado **Aprendendo**. Caso contrário, se o ciclo se confirmar pelo menos uma única vez, a máquina mudará seu estado para **Predizendo**. A quantidade de vezes que a máquina verificará a existência de um ciclo é um dos parâmetros do mecanismo de predição. A previsão no estado verificando também é dada por um conjunto de compartilhamento vazio.

No estado **Predizendo**, o algoritmo percorre a seqüência de estados de compartilhamento gravada, de forma circular. Enquanto o estado de compartilhamento, dado pelos conjuntos *R*, *W*, *P* e *C* da seqüência gravada, estiver *semelhante* ao estado global de compartilhamento da página em uma determinada fase, o próximo estado de compartilhamento da seqüência é retornado como a previsão do padrão de compartilhamento da página para a próxima fase, como ilustrado na Figura 3.5.

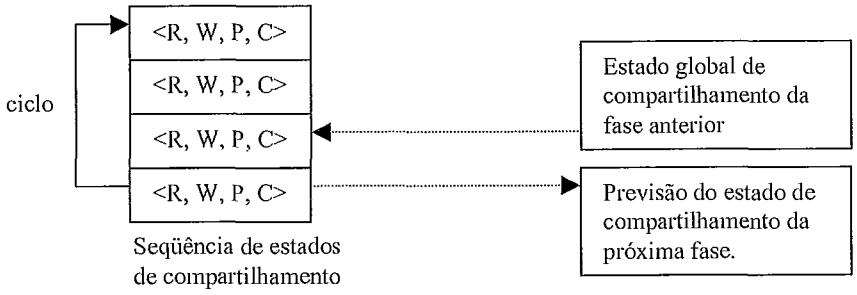


Figura 3.5 Mecanismo utilizado para previsão de estados de compartilhamento

Para estabelecer o critério de semelhança são necessários dois parâmetros. Um deles é chamado de limite positivo do conjunto de previsão e o outro é o limite negativo do conjunto de previsão. O limite positivo informa quantos elementos do conjunto de compartilhamento previsto não fizeram acesso à página. O limite negativo informa quantos elementos que não faziam parte do conjunto de previsão, fizeram acesso à página. Enquanto os erros de previsão estiverem dentro dos limites, a máquina continua no estado predizendo. Caso contrário, a máquina entra no estado **Ocioso**.

A máquina de estados permanecerá no estado ocioso até que um estado de compartilhamento coincida novamente com algum dos estados gravados na seqüência. Caso isso não ocorra em um ciclo completo do padrão de compartilhamento, a máquina retorna para o estado aprendendo. Os limites de conjunto de previsão servem como filtros para que o sistema de predição não tente seguir eventuais ruídos que surjam no estado de compartilhamento, e que sejam de caráter apenas temporário. Assim, os três parâmetros importantes para o preditor são: o limite de verificação e os limites positivo e negativo do conjunto de previsão. Variando estes limites podemos estabelecer, qual o melhor conjunto de parâmetros, para um determinado conjunto de aplicações.

3.3.1. *Overhead* de Memória

A estratégia baseada em estados de compartilhamento possui uma tabela de histórico onde cada entrada armazena um estado de compartilhamento. Cada estado de compartilhamento, por sua vez, possui informação sobre os conjuntos *R*, *W*, *P* e *C*. Em um sistema com 8 processadores, é preciso utilizar 8 bits para cada um desses conjuntos, fazendo com que cada estado de compartilhamento possua 32 bits.

Uma vez que a tabela utilizada em nossos experimentos possui 16 entradas, a mesma possui um overhead de memória de 512 bits. É preciso ainda de dois ponteiros para marcar o início e fim da tabela, que funciona como um *buffer* circular. Cada ponteiro deve ter o tamanho de 4 bits, para indexar a tabela de 16 entradas. Além disso, são precisos mais 2 bits para indicar o estado corrente da máquina de estados que implementa essa estratégia, totalizando um *overhead* de memória de 522 bits por página da aplicação. Esse overhead é praticamente metade daquele calculado para a estratégia baseada em preditores de desvio utilizada como exemplo. Ao contrário das estratégias baseadas em preditores de desvios, esse *overhead* inclui a captura do padrão de todos os processadores do sistema.

Capítulo 4

Avaliação dos Preditores

4.1. Ambiente Experimental

Nesta seção descrevemos os experimentos utilizados para avaliar nossas estratégias de previsão de acesso a dados remotos. Elas são baseadas em preditores de desvio e estados de compartilhamento implementadas sobre o sistema *software* DSM *TreadMarks*. Em uma primeira etapa, avaliamos os resultados das previsões de acesso a dados isoladamente. Esta avaliação é análoga ao estudo de previsão de desvios. Desta forma, avaliamos nossas estratégias somente em termos de taxas de previsões, sem nenhuma interferência adicional no protocolo de coerência. Posteriormente descrevemos os experimentos utilizados para a avaliação de uma técnica de redução do tempo de espera por dados remotos. Esta técnica faz uso das informações da estratégia de previsão de estados de compartilhamento.

Para nossos experimentos utilizamos sete aplicações. Todas as aplicações fazem parte do pacote de distribuição de *TreadMarks*. Como a nossa técnica foi implementada somente para as sincronizações de barreiras, utilizamos apenas as aplicações que possuem operações de sincronização com barreiras. São elas: 3D-FFT, *Gauss*, CG, IS, SOR, *Barnes-Hut* e MG. Estas aplicações estão descritas na seção 5.1.

Nosso ambiente experimental consiste em um *cluster* de PCs com 8 nós. O sistema operacional utilizado é o *Linux*. Cada nó de processamento possui dois processadores *Pentium III 650MHz* com 512 *Mbytes* de memória RAM. A interconexão entre os nós de processamento foi feita por uma rede *myrinet* com uma taxa de transferência de 1.23 *Gbits*/s.

Os experimentos relativos à avaliação das previsões destacam quatro tipos de informação, todas são percentuais em relação ao total de falhas de página da aplicação. Estas informações são apresentadas através de um gráfico de barras, dividido em quatro componentes. A primeira divisão positiva do gráfico informa o percentual de acertos de previsão em relação ao total de falhas de página da aplicação. Esse número representa o percentual de falhas de página cuja ocorrência foi prevista pelo algoritmo. A segunda divisão mostra o percentual de falhas de *cold-start*, que representam as falhas iniciais em cada página da aplicação. Na terceira divisão, temos o percentual de falhas imprevistas pelo preditor. Esses três percentuais somados totalizam 100%, representando assim, todas as falhas de página da aplicação. O quarto tipo de informação se encontra na parte inferior do gráfico e nos mostra o percentual, também em relação ao total de falhas da aplicação, de falhas que foram previstas erroneamente pelo preditor e que não ocorreram na realidade. A este percentual chamamos de taxa de erros. Observe que esse percentual pode ultrapassar 100%, pois o número de falhas previstas erroneamente pode ultrapassar o total de falhas da aplicação.

Podemos dizer que nossas métricas de avaliação são conservadoras em dois aspectos. Primeiro, ao contrário dos resultados apresentados por estudos de previsões de desvios, que apresentam as taxas de acertos em relação ao número total de desvios (realizados ou não) executados pela aplicação, nossos resultados apresentam taxas de acertos em relação apenas ao total de falhas de acessos. Eles não consideram o total de acessos (ocorridos ou não) da aplicação. Resumidamente, podemos dizer que falhas que não ocorrem e que são corretamente previstas como não ocorridas, não são contabilizadas como acertos. Isso reduz drasticamente as taxas de acertos apresentadas. Porém, torna os resultados mais realistas do ponto de vista de sistemas *software* DSM, onde cada processador normalmente realiza acesso a um conjunto restrito de total de páginas da aplicação.

Um segundo aspecto conservador dos resultados de previsão consiste no fato de que, toda vez que um determinado processador que está no conjunto de previsão não realiza acesso a página, é contabilizado um erro de previsão. Essa contabilização parece óbvia e análoga ao caso de um desvio previsto como realizado não ser realizado. Porém, ela não considera que, no caso de um envio antecipado das modificações, este processador possa vir a utilizá-las posteriormente, em uma outra fase da aplicação, usufruindo integralmente dos

benefícios da previsão. Não existe um fenômeno similar a este em predição de desvios condicionais. Mesmo assim, preferimos registrar os resultados de previsão de forma isolada e precisa, não contaminando as taxas de erros com detalhes de implementação. Por isso, as taxas de erros reportadas são conservadoras em relação a real utilização do envio antecipado de modificações.

Os experimentos realizados com objetivo de avaliar a técnica de redução do tempo de espera por dados remotos, compara o sistema *TreadMarks* original com uma versão de *TreadMarks* onde nossa técnica de atualizações seletivas foi implementada. A técnica utiliza as informações da estratégia de previsão de acesso a dados baseada em estados de compartilhamento. As métricas utilizadas nesta comparação foram o tempo de execução, o número de mensagens enviadas, o número de *bytes* trafegados, o número de falhas de páginas, e o número de modificações pedidas em relação ao número de modificações enviadas antecipadamente através de atualização. Através desta última métrica, podemos quantificar na prática o grau de utilidade das modificações antecipadas, em contraste com a taxa de erros, que fornece uma medida teórica do pior caso.

4.1.1. Descrição das Aplicações

3D-FFT

A aplicação 3D-FFT resolve um conjunto de equações diferenciais parciais usando FFTs (*Fast Fourier Transforms*) diretas e inversas em três dimensões. Toda a sincronização é realizada através de barreiras. A aplicação utiliza como estrutura de dados uma matriz de dimensões *N1xN2xN3*. Em cada iteração existem duas fases. Na primeira fase é computada uma 1D-FFT em cada um dos *N1xN2* vetores, para em seguida ser realizado outro procedimento 1D-FFT em cada um dos *N1xN3* vetores. Na segunda fase, a matriz é transposta em uma outra matriz *N2xN3xN1*, para que seja realizado outro procedimento 1D-FFT em cada um dos *N2xN3* vetores. A massa de dados da matriz de entrada utilizada em nossos experimentos possui dimensão 64x64x16 e foram realizadas 100 iterações.

Gauss

A aplicação *Gauss* resolve um sistema de equações lineares utilizando o método de eliminação Gaussiana. Toda a sincronização é realizada através de barreiras. Em nossos experimentos utilizamos uma matriz de entrada de 1024x1024.

CG

Conjugate Gradient (CG) é uma aplicação que calcula uma aproximação do menor autovalor de uma matriz simétrica positiva grande e esparsa, através do método de gradiente conjugado. Utilizamos uma matriz de 14000x14000 elementos, sendo que 2030000 não nulos. Utilizamos 15 iterações para nossos experimentos.

IS

A aplicação IS (*Integer Sort*) ordena uma seqüência de chaves utilizando a técnica *bucket sort*. As chaves são distribuídas igualmente entre os processadores do sistema. As principais estruturas de dados são um vetor de chaves e um vetor que indica a densidade das chaves a serem ordenadas. A computação é dividida em *N* fases, onde *N* é o número de processadores executando a aplicação. A cada fase os processadores atualizam uma porção do vetor de densidades com informações relativas ao conjunto de chaves que lhe pertencem. Utilizamos um vetor com 2^{24} chaves de valor máximo 2^{15} para nossos experimentos, e a aplicação é repetida 10 vezes.

SOR

SOR é uma aplicação que resolve equações diferencias parciais utilizando um método iterativo baseado em relaxações sucessivas. Este método utiliza uma estratégia "rubro negra" para realizar as relaxações. Ele divide uma matriz de entrada em suas linhas pares e ímpares. Cada iteração é composta por duas fases separadas por barreiras. Na primeira fase, são calculados os valores das linhas pares da matriz como sendo a média das linhas ímpares da matriz. Na segunda fase, as linhas ímpares são, então, calculadas como médias das linhas pares. A massa de dados de entrada utilizada em nossos experimentos é uma matriz de dimensão 2000 x 1000 e é executada durante 200 iterações.

Barnes-Hut

A aplicação *Barnes-Hut* simula a iteração de um sistema de corpos sob a influência de forças gravitacionais em um espaço tridimensional, usando um método chamado *Barnes-Hut Hierarquical N-Body*. O algoritmo utiliza uma árvore hierárquica (*octree*) como sua principal estrutura de dados. A simulação divide o

espaço tridimensional em células. Os nós internos representam as células, enquanto as folhas da árvore representam os corpos. A cada passo do algoritmo, a árvore é refeita e percorrida uma vez para cada corpo, para que sejam computadas as novas força e posições destes corpos. O número de corpos utilizado em nossos experimentos é igual a 2^{15} corpos com intervalos discretos de 0,025 segundos, durante 3 segundos.

MG

A aplicação *Multigrid* (MG) utiliza uma computação *multigrid* para obter uma solução aproximada para um problema de *Poisson* escalar em uma grade discreta de três dimensões, com condições de contorno periódicas. Em nosso experimento foi utilizado uma grade de dimensões 64 x 64 x 64 em 100 iterações.

A Tabela 4.1 relaciona de maneira sumarizada cada uma das aplicações utilizadas em nossos experimentos e suas respectivas entradas.

Aplicações	Entradas
FFT	64 x 64 x 16, 100 iterações
Gauss	1024 x 1024
IS	2^{24} chaves, comprimento de chave de 2^{15}, 10 iterações
SOR	2000 x 1000, 200 iterações
CG	14000 x 14000 elementos, 2030000 não nulos, 15 iterações
Barnes-Hut	2^{15} corpos, intervalos de 0.025s , 3s
MG	64 x 64 x 64, 100 iterações

Tabela 4.1 Aplicações utilizadas e suas entradas

4.2. Previsões Baseadas em Algoritmos de Previsão de Desvios

Os resultados experimentais das diferentes estratégias de previsão são apresentados a seguir. Cada barra dos gráficos representa um algoritmo de previsão diferente. Para representar cada um destes algoritmos utilizamos a nomenclatura **LT, A1 e A2** para representar, respectivamente, os algoritmos de previsão de desvio que usam os autômatos *Last-Time*, A1 e A2. Os esquemas que utilizam dois níveis serão prefixados com a notação **2-**, seguidos do autômato utilizado e terminando com o sufixo **–k**, onde *k* representa o tamanho em *bits* da PHAP utilizada. Exemplificando, a sigla: 2-LT-4 significa o esquema de previsão de dois níveis que utiliza o autômato *Last-Time* e uma PHAP de 4 *bits*.

3D-FFT

A Figura 4.1 mostra o gráfico contendo os resultados da aplicação 3D-FFT. Podemos observar que quase todos os algoritmos produzem uma ótima taxa de acertos de previsão de falhas de página, em torno de 95%.

As exceções são os autômatos LT e A2 de um nível. Isso pode ser explicado pelo fato desta aplicação ter, em grande parte, acessos alternados às páginas durante as fases. Se em uma fase um processador provocou uma falha de acesso à página, na outra fase isso não ocorre e vice-versa. Este tipo de padrão não é captado por nenhum dos dois autômatos. O autômato LT faz a previsão baseada na última fase e uma vez que esta é sempre diferente da fase seguinte, as taxas de acerto são nulas. A taxa de erro por sua vez, não aumenta, pois quando a previsão é positiva em relação a falha em um determinado processador, não existe nenhum *diff* a ser enviado para este processador nesta fase. Isso porque todos os *diffs* já foram buscados na falha não prevista da fase anterior. Já o autômato A2 fica oscilando entre os estados 0 e 1, e nunca chega a ter uma previsão positiva.

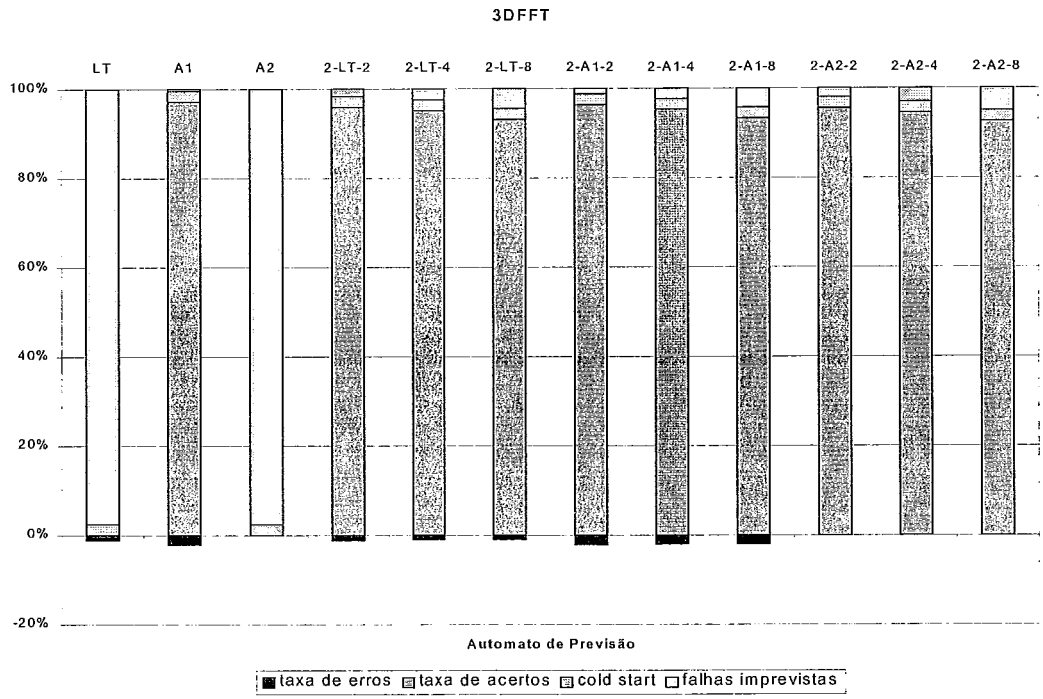


Figura 4.1 Taxas de previsão para autômatos de previsão de desvio em 3D-FFT

Gauss

Na aplicação *Gauss*, todos os algoritmos se comportaram de maneira semelhante com relação às taxas de acerto, atingindo cerca de 40% das falhas de páginas. Nesta aplicação, uma grande parte do total de falhas é devida a *cold-start* (31%) por causa da inicialização da massa de dados. Outra grande parcela das falhas não previstas ocorre porque cada linha da matriz só é efetivamente compartilhada uma vez durante a aplicação. A grande parcela das falhas previsíveis ocorre em uma das páginas da aplicação que armazena a coluna do elemento *pivot*. A cada iteração, um dos processadores calcula e armazena a coluna do *pivot* nesta variável, que é lida em uma fase seguinte por todos os outros processadores. As taxas de acertos atingiram aproximadamente 40% para a maioria dos algoritmos, entretanto as versões baseadas no autômato A2 (A2, 2-A2- 2, 2-A2-4 e 2-A2-8) foram melhores do que os demais já que apresentaram taxas de erros praticamente nulas, como pode ser observado na Figura 4.2.

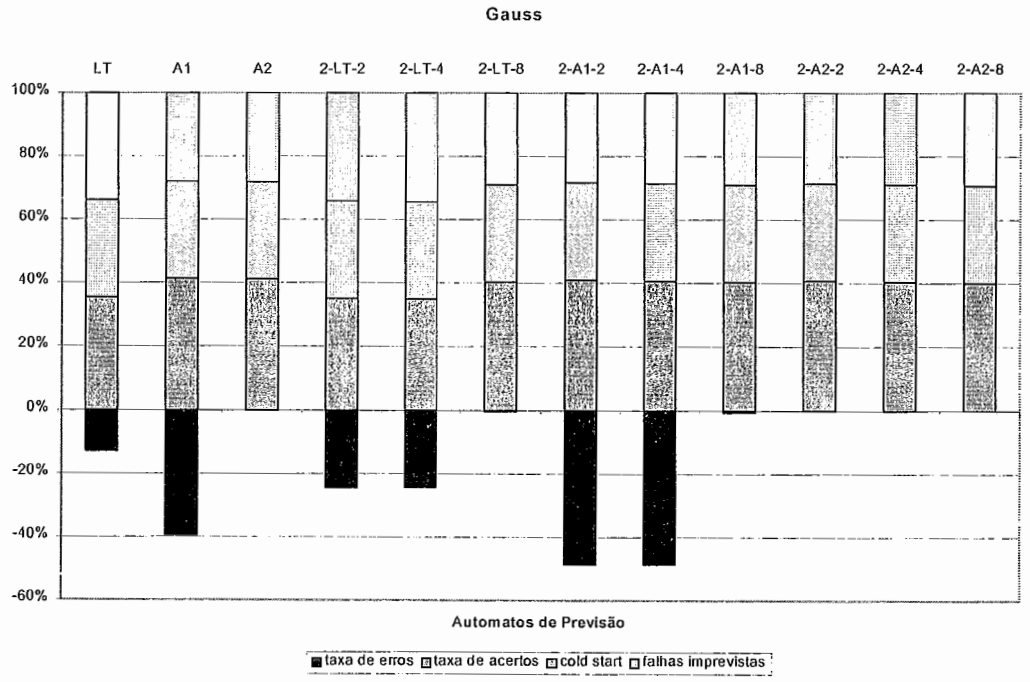


Figura 4.2 Taxas de previsão para autômatos de previsão de desvio em *Gauss*

CG

A aplicação CG apresentou ótimas taxas de acertos, atingindo 99% em alguns casos, para todos os algoritmos de dois níveis. Os autômatos de apenas um nível apresentaram um desempenho bastante ruim. Os resultados para esta aplicação são mostrados na Figura 4.3. Para todos os algoritmos as taxas de erros foram praticamente nulas. Este fato ocorre porque o padrão de acesso ás páginas da aplicação é bastante regular.

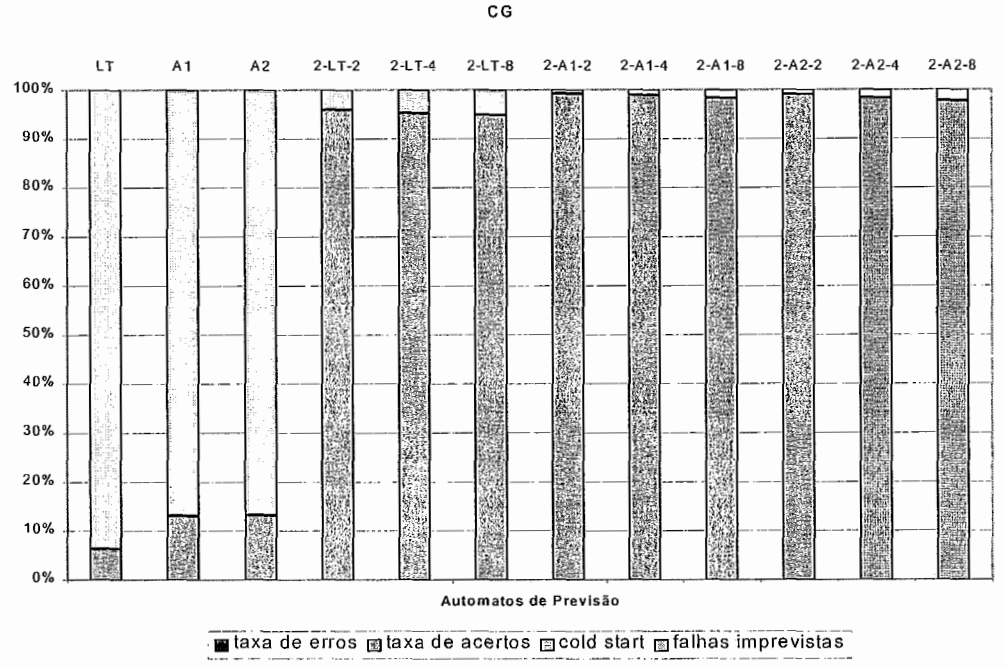


Figura 4.3 Taxas de previsão para autômatos de previsão de desvio em CG

IS

A aplicação IS possui um padrão de compartilhamento migratório. Isto significa que em cada fase da aplicação um dos processadores do sistema faz uso da página por vez. Esse comportamento ocorre uma vez que o vetor de densidades de chave é particionado entre os processadores e cada um atualiza as partições de maneira alternada. Isso explica por que os algoritmos de dois níveis 2-LT, 2-A1 e 2-A2 com uma PHAP de 8 bits produziram melhores resultados do que os demais. Uma PHAP com o tamanho igual ao número de processadores é capaz de armazenar informação completa sobre o padrão de acesso à página neste caso. Esses algoritmos apresentaram taxas de acertos em torno de 85% e taxas de erros praticamente nulas como mostra a Figura 4.4.

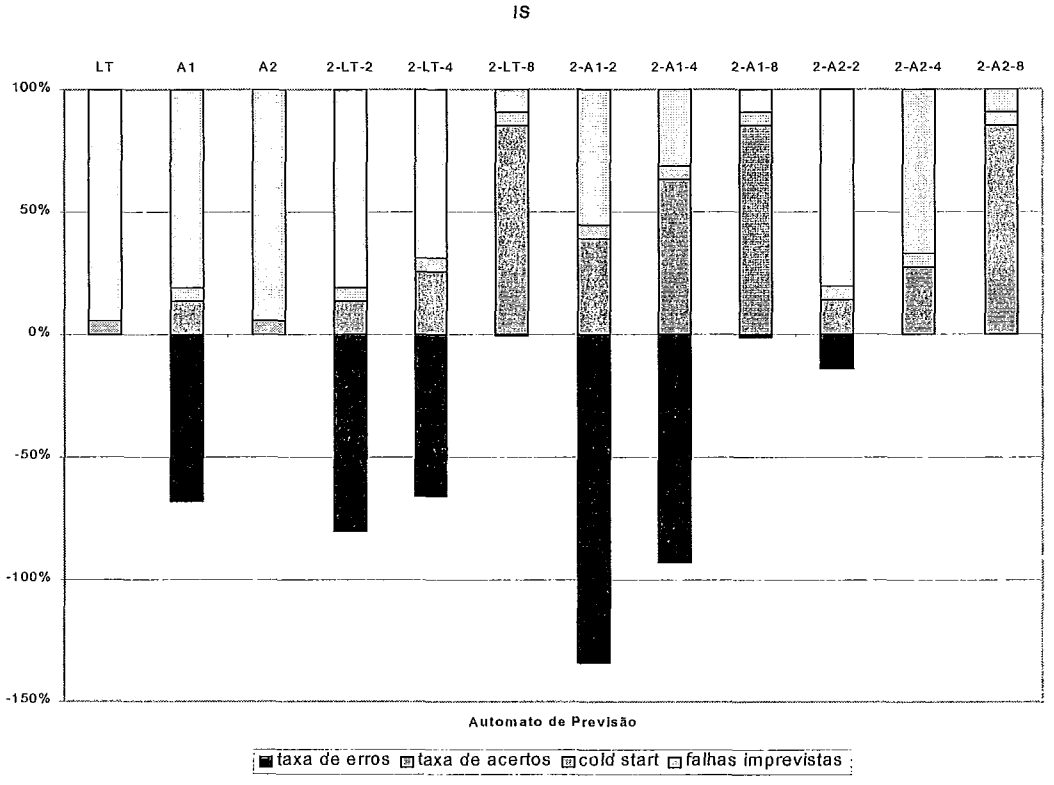


Figura 4.4 Taxas de previsão para autômatos de previsão de desvio em IS

SOR

SOR é uma aplicação que possui um alto grau de paralelismo. Uma vez que as primeiras falhas ocorrem, os acessos que se seguem são essencialmente a dados locais. O compartilhamento só ocorre nas linhas de fronteira entre a porção da matriz destinada a cada processador. Isso explica o grande percentual de falhas de *cold-start* (em torno de 63%) em relação as demais categorias. O restante das falhas é bastante regular e previsível. Todos os algoritmos apresentam resultados bastante satisfatórios e semelhantes, como pode ser observado na figura 4.5. Os autômatos LT e A2 apresentaram um desempenho um pouco inferior aos demais.

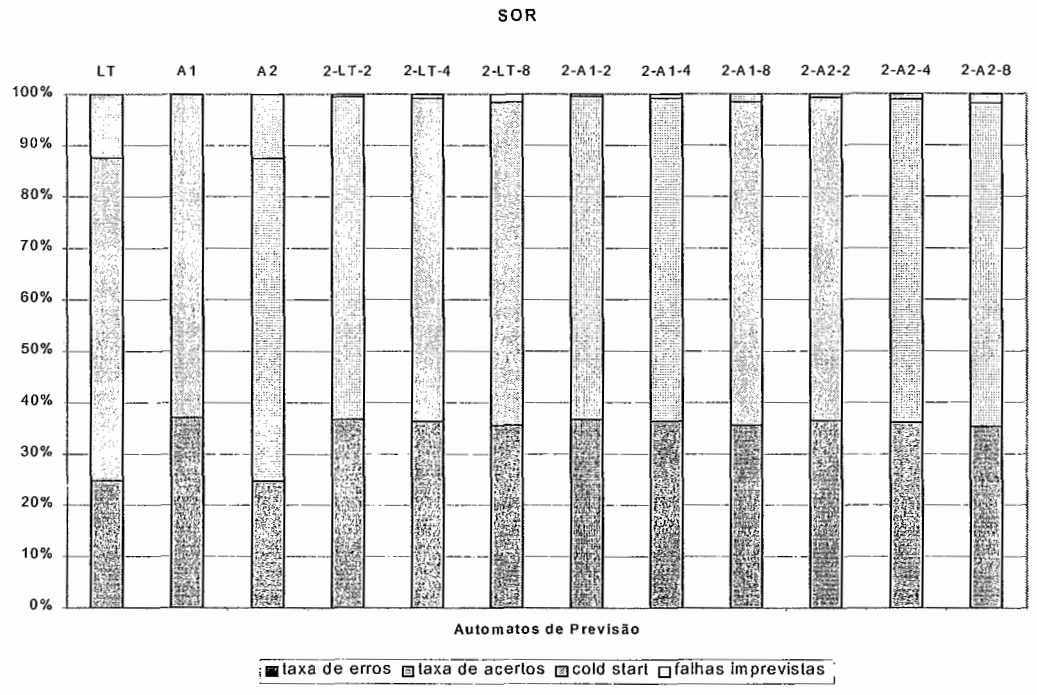


Figura 4.5 Taxas de previsão para autômatos de previsão de desvio em SOR

Barnes-Hut

Em *Barnes-Hut*, as taxas de acertos variaram de 10% no algoritmo LT à 65% nos algoritmos 2-LT-2 e 2-A1-2. Os autômatos de apenas um nível obtiveram um desempenho notadamente inferior aos de dois níveis. Os autômatos LT e A1 apresentaram menores taxas de acerto em relação aos algoritmos de dois níveis e, os três autômatos de um nível apresentaram as taxas de erro mais elevadas. Note que muitos dos algoritmos que apresentaram taxas de erros pequenas, também tiveram taxas de acertos baixas.

Podemos inferir que, para uma taxa de acertos máxima a melhor opção é o algoritmo 2-A1-2. Esse algoritmo atinge a maior taxa de acertos, juntamente com o autômato 2-LT-2. Porém, apresenta uma menor taxa de erros de apenas 4%, contra 10% do autômato 2-LT-2.

No entanto, se uma menor taxa de erros for desejada o algoritmo 2-A2-2 pode ser considerado como a melhor solução. Isto porque ele apresenta uma taxa de acertos de aproximadamente 55%, com uma taxa de erros insignificante. Os resultados podem ser observados na Figura 4.6.

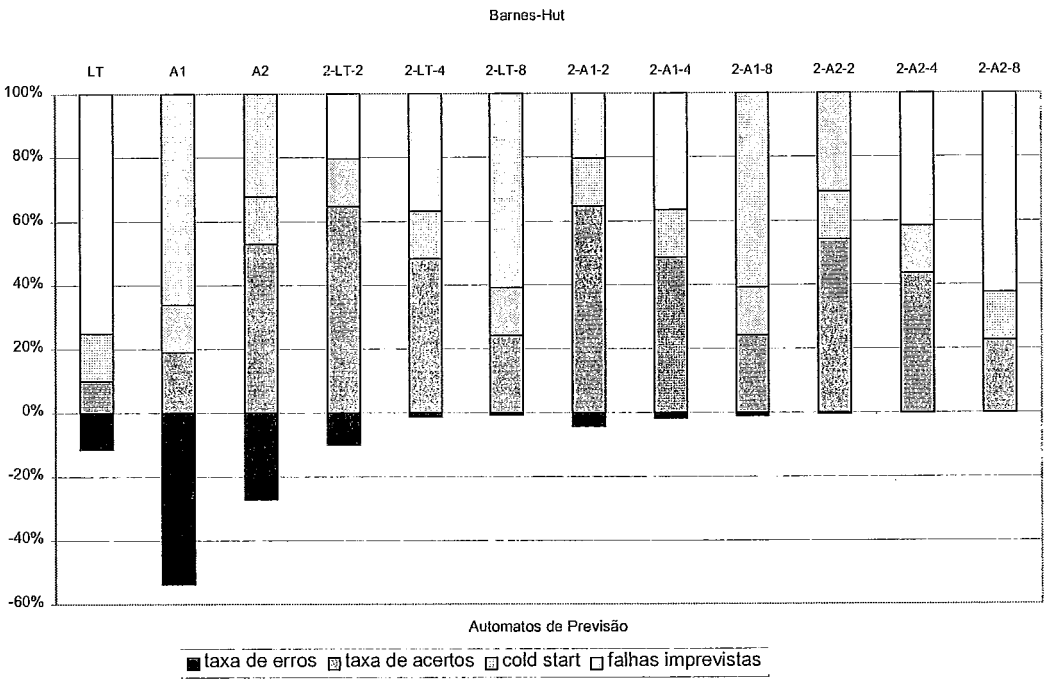


Figura 4.6 Taxas de previsão para autômatos de previsão de desvio em *Barnes-Hut*

MG

Em MG, nenhum dos algoritmos apresentou resultados que pudessem ser considerados bons. Todos eles apresentaram baixas taxas de acerto, atingindo no máximo 48%. Além disto, alguns algoritmos ainda apresentaram altas taxas de erro, atingindo mais de 40%, como no caso do algoritmo 2-A1-2.

O algoritmo que apresentou o melhor resultado foi o 2-A2-8, atingindo uma taxa de acertos de aproximadamente 45% e com taxa de erros insignificante, já os algoritmos 2-LT-8, 2-A1-2, 2-A1-4 e 2-A1-8 obtiveram também taxa de acertos próxima a 45%, porém com taxa de erros bem maiores variando entre 15 e 50%. Estes resultados se encontram na Figura 4.7

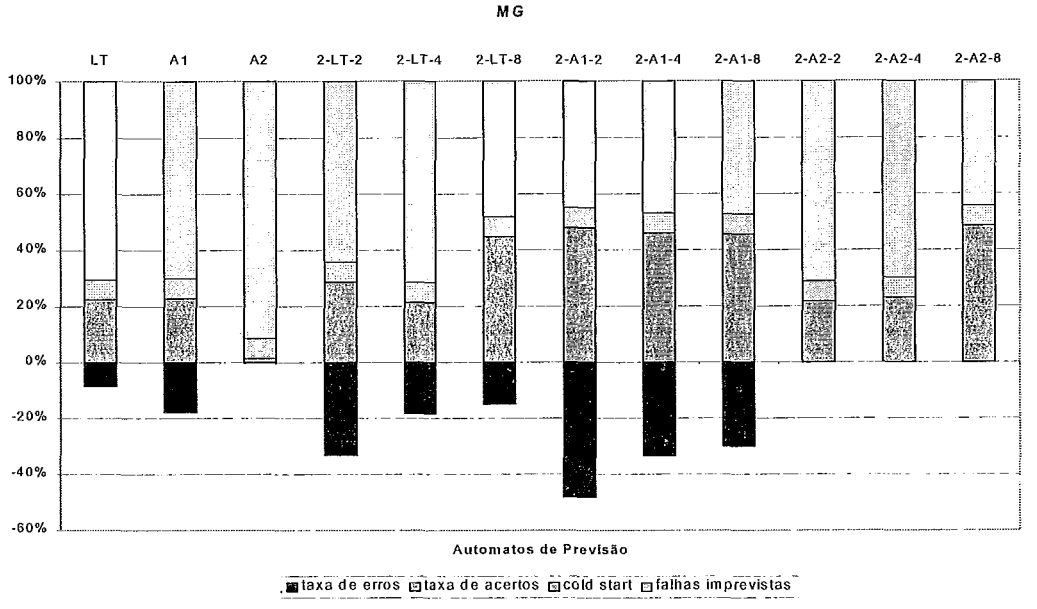


Figura 4.7 Taxas de previsão para autômatos de previsão de desvio em MG

4.2.1. Discussão

A Figura 4.8 resume os resultados das técnicas de autômatos de previsão. Nesta figura estão mostradas as barras representativas dos autômatos que obtiveram as melhores taxas de acerto, para cada uma das aplicações. Nos caso onde exista mais de um autômato com a taxa de acerto máxima, é escolhido aquele como a menor taxa de erros. É importante ressaltar que muitos outros critérios de seleção são possíveis. Como exemplo, poderíamos ter priorizado a redução das taxas de erros, ou até mesmo o número de *bits* gasto na implementação dos autômatos. Esse critério depende, em grande parte, de alguns parâmetros da arquitetura como latência da rede, capacidade de processamento, memória disponível, entre outros. Um outro critério pode ser o tipo de adaptação que será inserida no protocolo utilizando-se do resultado destas previsões, e também da implementação específica de um determinado protocolo.

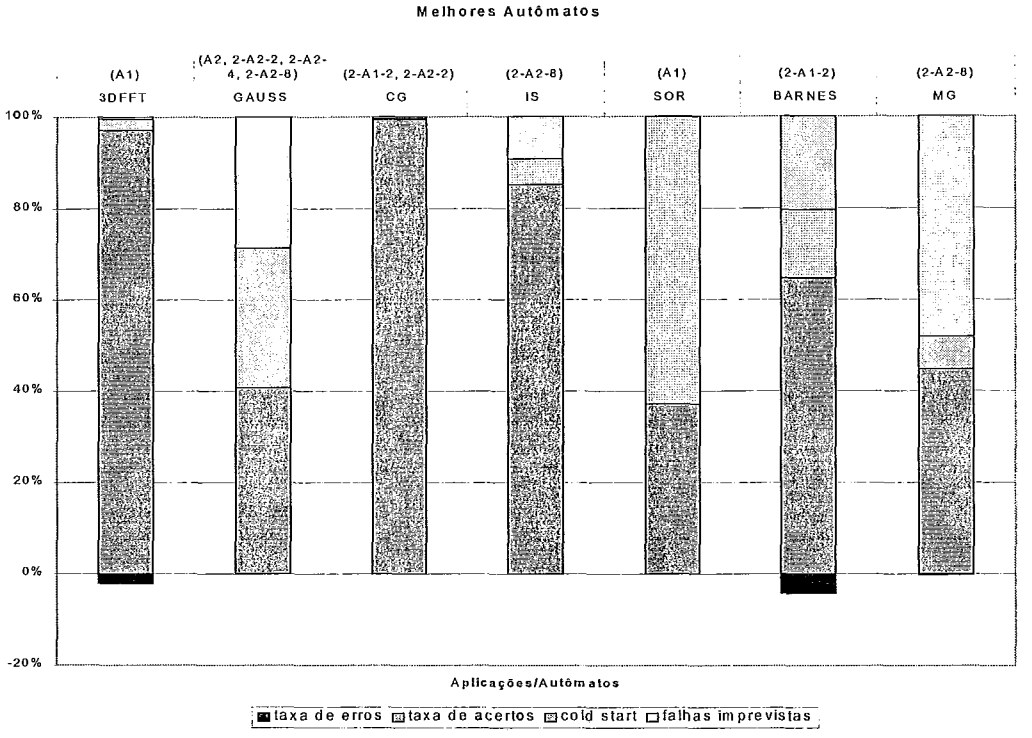


Figura 4.8 Taxas de previsão para os melhores autômatos de previsão de desvio por aplicação

Podemos verificar através da Figura 4.8 que para grande parte das aplicações (3D-FFT, *Gauss*, CG, IS, SOR) a maioria das técnicas selecionadas se comportou de maneira muito eficiente, exceto em *Gauss* e SOR. Este fato se deve a 30% de falhas imprevistas em Gauss e SOR com 63% de falhas de *cold-start*.

Porém, em quase todos os casos as estratégias de dois níveis apresentaram resultados superiores, principalmente em relação a redução nas taxas de erros. Essa situação se acentua para as aplicações mais irregulares como *Barnes-Hut* e MG.

Na aplicação IS pode ser observada a influência do número de bits da PHAP no resultado das previsões. Verificamos que as melhores previsões ocorreram justamente quando o tamanho da PHAP coincide com o período (medido em fases) do padrão de acesso às páginas.

Embora não haja uma configuração claramente superior em relação as demais, a configuração 2-A2-8 apresentou um resultado bastante satisfatório em todas as aplicações. A exceção foi a aplicação *Barnes-Hut*, onde seu resultado foi bem inferior a outros algoritmos, como por exemplo a configuração 2-A1-2.

4.3. Previsão Baseada em Estados de Compartilhamento das Páginas

Para a estratégia baseada em estados de compartilhamento de páginas adotamos as seguintes convenções em relação aos parâmetros de previsão. Os parâmetros são indicados pela palavra **PxNyVz**, onde *x* é o limite positivo do conjunto de previsão, *y* representa o limite negativo do conjunto de previsão, e *z* o limite de verificação.

3D-FFT

Os resultados de previsão da aplicação 3D-FFT se encontram na Figura 4.9. Por ser muito regular, a variação dos parâmetros de previsão não tem muita influência na acurácia da previsão de falhas de acesso. Todas as configurações apresentam uma alta taxa de acertos (em torno de 96%) e taxas de erros insignificantes.

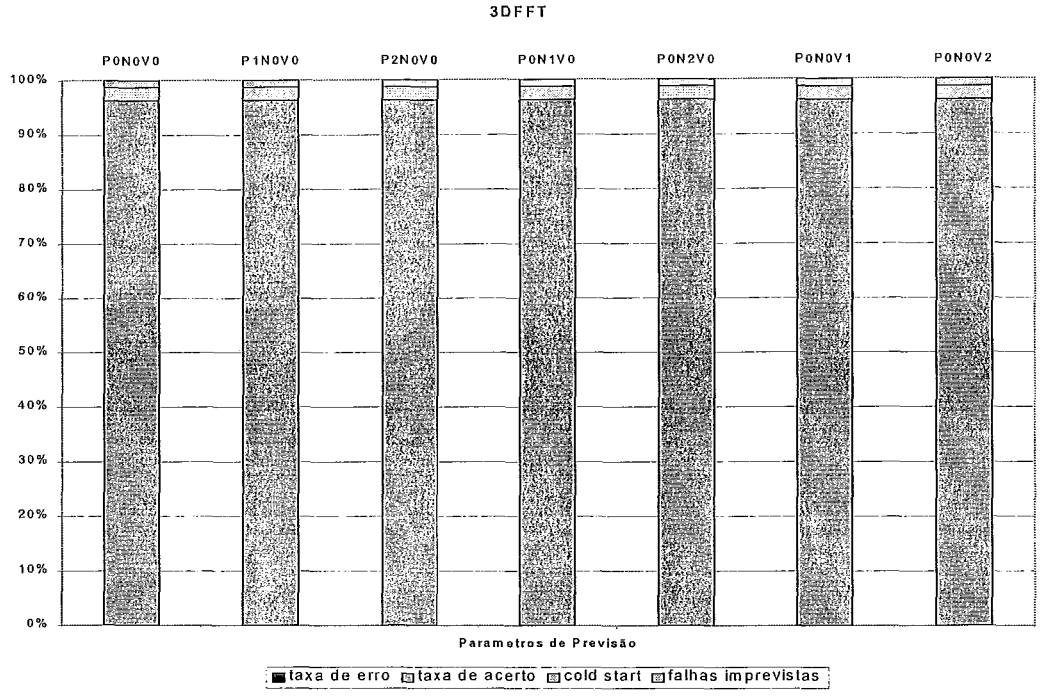


Figura 4.9 Taxas de previsão de estados de compartilhamento para 3D-FFT

Gauss

Em *Gauss*, o limite positivo do conjunto de previsão e o limite de verificação surtiram efeitos no resultado das previsões. Alterando-se o limite positivo de zero para um, a taxa de acertos cresceu de 2% para 43% com um pequeno prejuízo na taxa de erros que aumentou de 7% para 8%. Acima de um o aumento deste limite não surtiu efeito nas previsões. Os resultados se encontram ilustrados na Figura 4.10.

O limite negativo do conjunto de previsão não apresentou qualquer alteração nos resultados das previsões.

O limite de verificação também surtiu efeitos positivos nos resultados das previsões, atingindo uma taxa de acertos de 37% com a redução completa da taxa de erros.

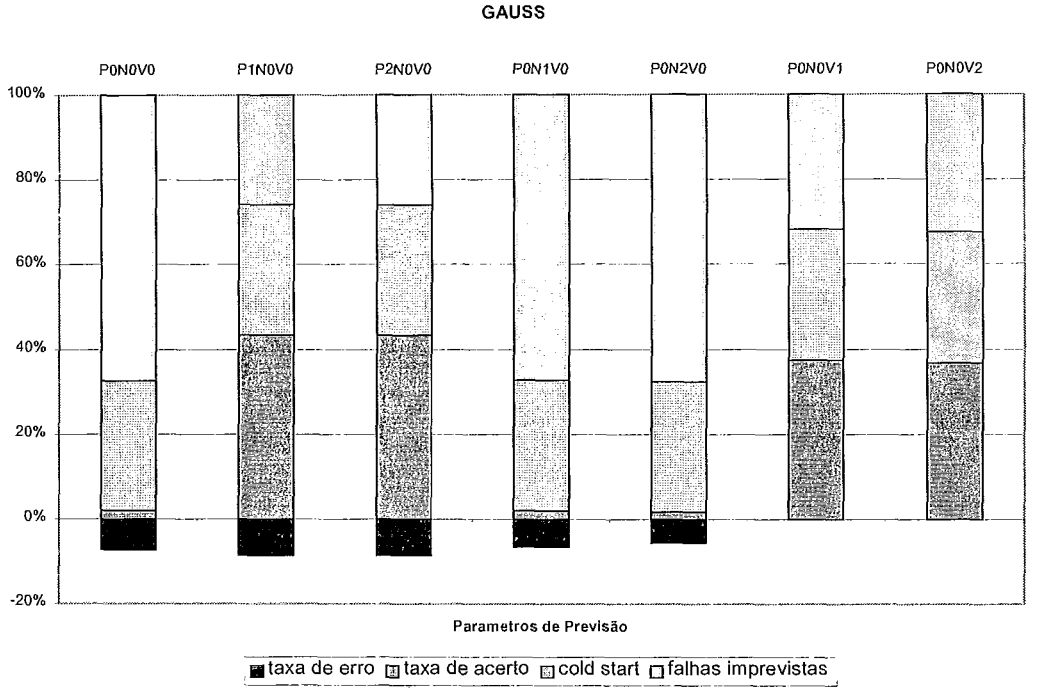


Figura 4.10 Taxas de previsão de estados de compartilhamento para *Gauss*

CG

A aplicação CG também é uma aplicação bastante regular em termos de acessos a dados. Por isso, todos as configurações de parâmetros apresentam ótimos resultados como pode ser observado na Figura 4.11.

Um aumento no limite de verificação produz um acréscimo na taxa de falhas previstas de 93 para 99% de acertos. A alteração dos demais parâmetros não surtiu efeito.

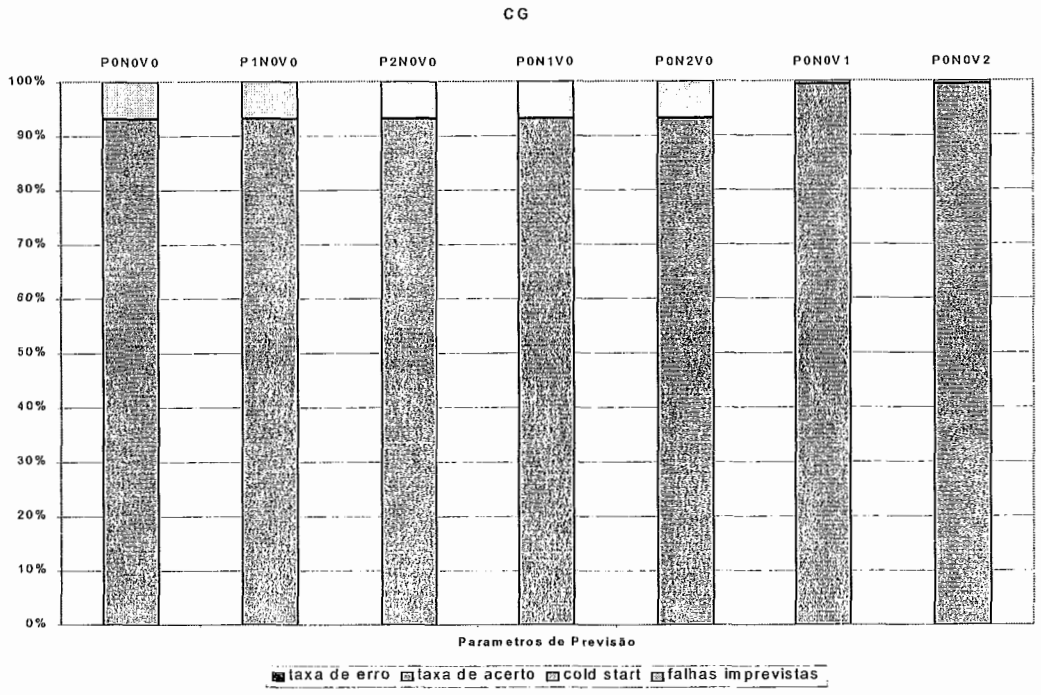


Figura 4.11 Taxas de previsão de estados de compartilhamento para GG

IS

Na aplicação IS todas as configurações apresentaram bons resultados, atingindo 89% de taxa de acertos. Os resultados podem ser vistos pela Figura 4.12. O aumento do limite de verificação nessa aplicação resultou em uma diminuição da taxa de acertos para 83%. Isso ocorre porque o número total de falhas da aplicação é pequeno, e o tempo gasto na verificação do padrão (que já está correto) implica em um aumento nas falhas ocorridas.

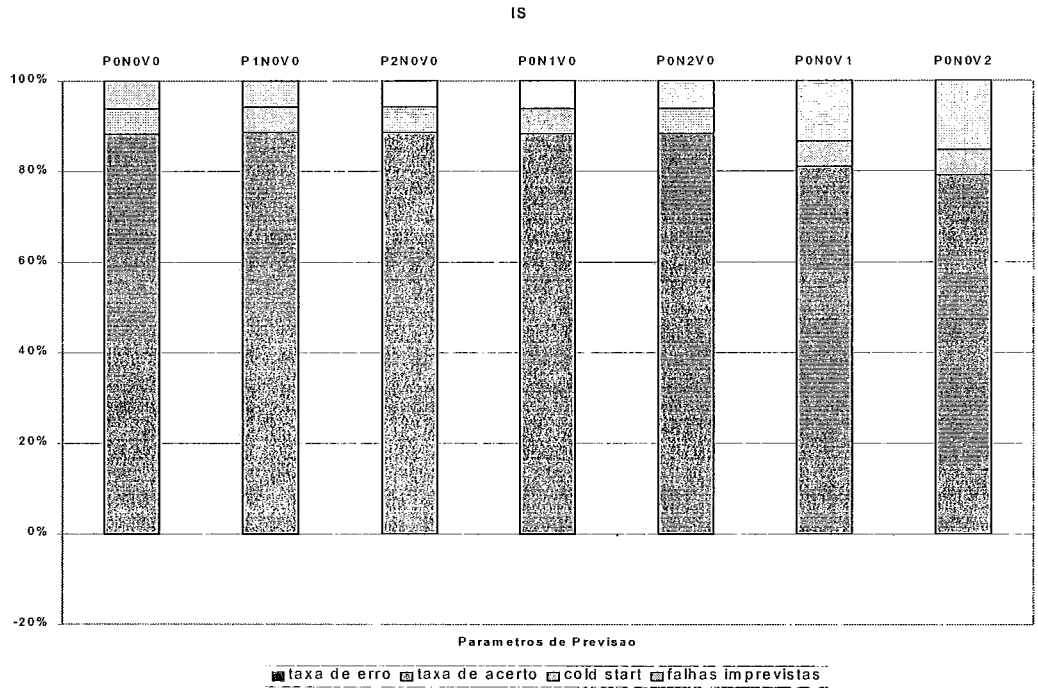


Figura 4.12 Taxas de previsão de estados de compartilhamento para IS

SOR

Na aplicação SOR todas as configurações apresentaram taxas de acertos bastante elevadas, considerando que a maioria das falhas se deve a *cold-start*. Pela sua regularidade de acesso a dados, a variação nos parâmetros não provocou nenhum efeito nos resultados das previsões como pode ser observado pela Figura 4.13.

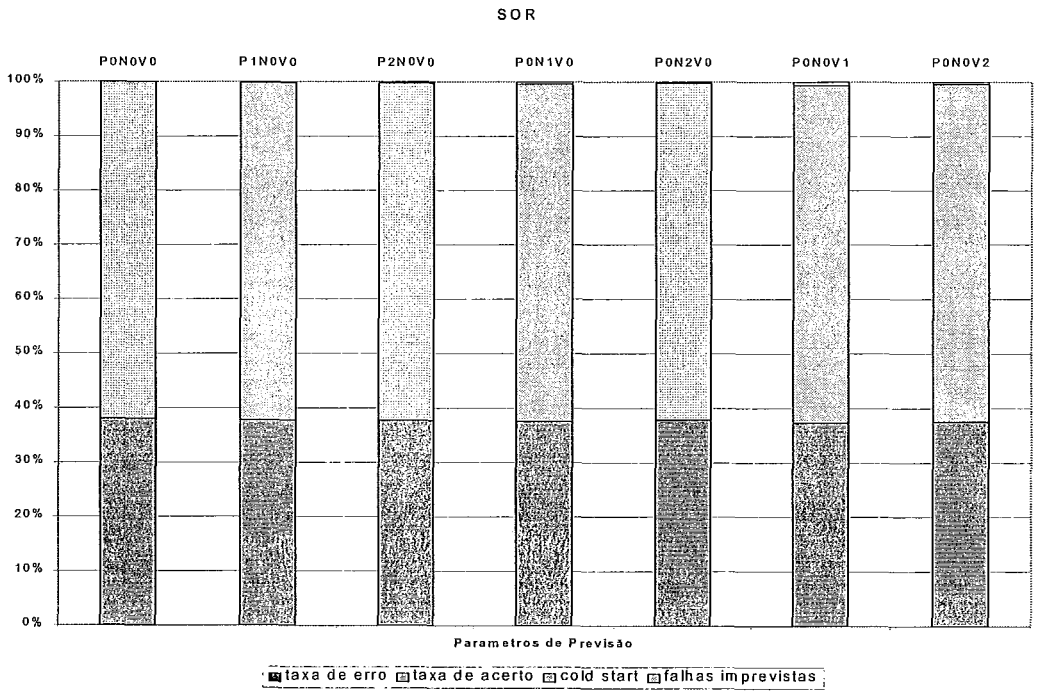


Figura 4.13 Taxa de previsão de estados de compartilhamento em SOR

Barnes-Hut

Para *Barnes-Hut* os limites positivo e negativo do conjunto de previsão não produziram efeito em nenhuma das taxas de acertos. O limite de verificação, por outro lado, se mostrou efetivo tanto no aumento da taxa de acertos (que atingiu 63%), quanto na redução das taxas de erros (praticamente nula). Os resultados podem ser vistos na Figura 4.14.

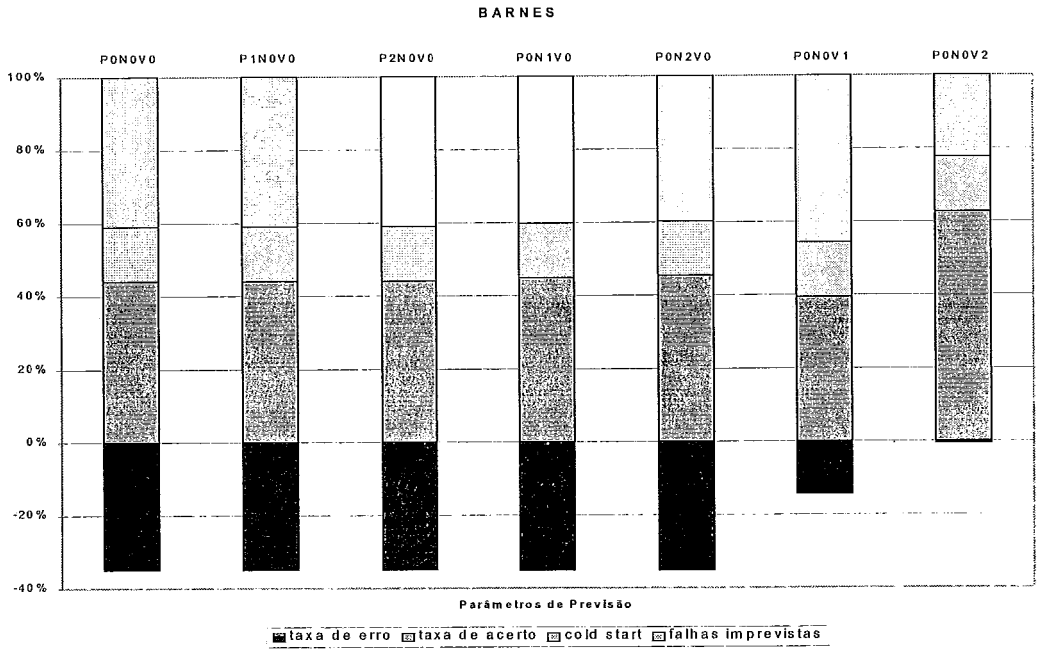


Figura 4.14 Taxas de previsão de estados de compartilhamento para *Barnes-Hut*

MG

Em MG, os resultados das previsões não foram muito satisfatórios. Quanto maior as taxas de falhas previstas pelas diferentes configurações, maiores eram suas taxas de erros. Taxas estas que ultrapassam 100% do total de falhas da aplicação. Esses resultados podem ser vistos na Figura 4.15.

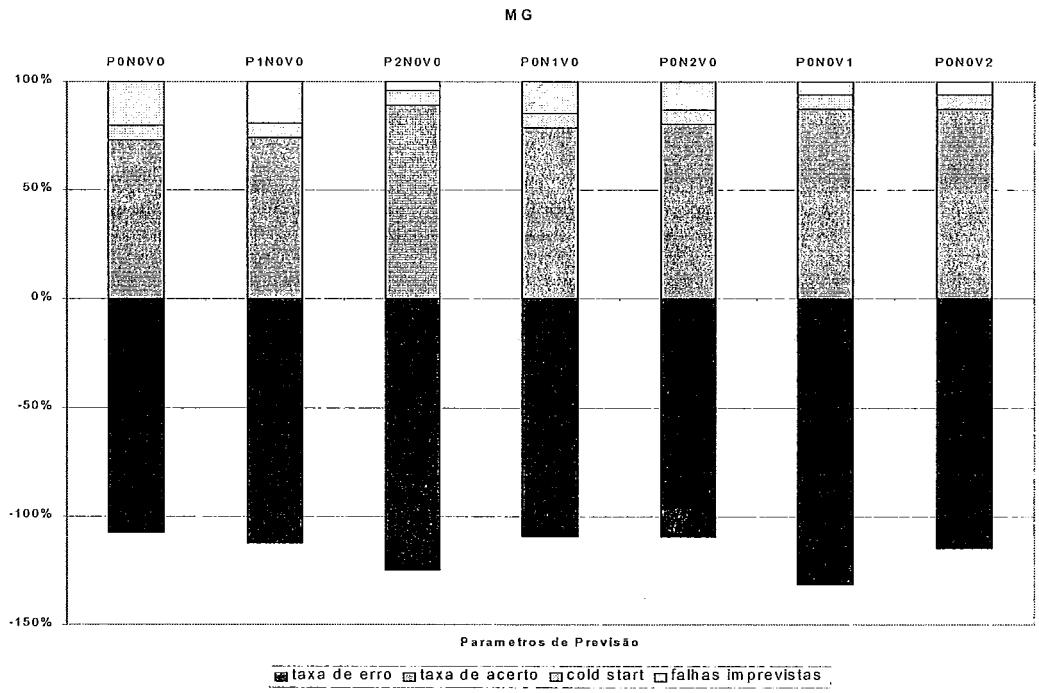


Figura 4.15 Taxas de previsão de estados de compartilhamento para MG

Em algumas páginas da aplicação MG alguns estados de compartilhamento se repetem durante a ocorrência do padrão, dificultando assim a captura do mesmo. Em alguns casos, os níveis de verificação utilizados não são suficientes para contornar este problema. Este fenômeno ocorre da seguinte maneira. Uma vez que a estratégia de previsão acredita que encontrou um ciclo, se inicia a fase de previsão. Como este ciclo não é realmente um padrão completo, em algum momento ocorrerá uma falha na previsão e a estratégia mudará seu estado para **Ocioso**. No entanto, durante esta fase, a estratégia irá encontrar novamente o estado de compartilhamento da fase de previsão anterior, que se repete durante a ocorrência do padrão real, e retornará para o estado **Predizendo.** Esse comportamento resulta em uma seqüência de previsões erradas, o que aumenta a taxa de erros observada nesta aplicação.

4.3.1. Discussão

A Figura 4.16 apresenta, para cada aplicação, a melhor combinação de parâmetros para o preditor de acesso baseado no estado de compartilhamento das páginas. Os parâmetros apresentados na Figura 4.16 foram escolhidos segundo o mesmo critério de seleção utilizado para os autômatos de previsão de desvios. Esse critério prioriza o aumento da taxa de acerto em primeiro lugar e a diminuição das taxas de erro em seguida.

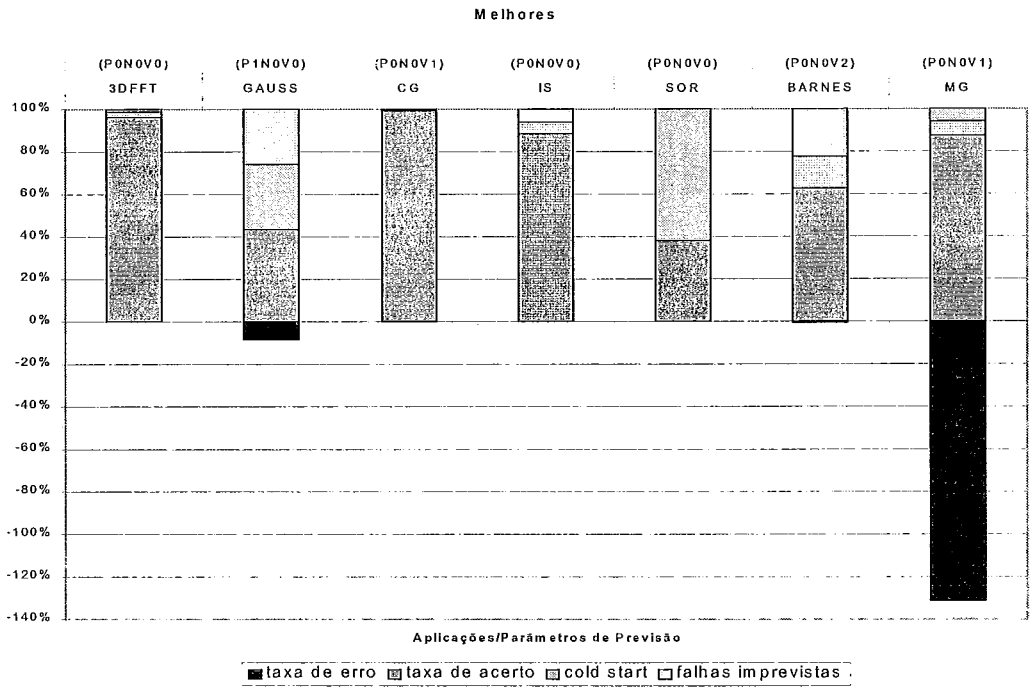


Figura 4.16 Taxas de previsão de estados de compartilhamento utilizando-se os melhores parâmetros por aplicação

Para o algoritmo de previsões baseado em estados de compartilhamento, o limite positivo e o limite de verificação, apresentaram aumentos nas taxas de acertos sem contudo aumentar as taxas de erros, com exceção de algumas aplicações.

Isso mostra que é melhor ignorar certos ruídos do padrão de acesso à página e continuar com a previsão corrente do que esperar reencontrar um padrão novamente. Podemos constatar também que o tempo gasto inicialmente na verificação do padrão, pode ser recompensado no decorrer da aplicação, devido à captura de um padrão de acesso à página mais preciso, resultando em maiores taxas de acertos.

O limite negativo do conjunto de previsão, por sua vez, não afetou muito o resultados das previsões para nosso conjunto de aplicações.

Diferente dos autômatos de previsão de desvio, os três parâmetros da estratégia de previsão baseado em estados de compartilhamento foram avaliados separadamente. Isso significa que estes parâmetros podem ser combinados para produzirem resultados ainda melhores. Podemos estimar que uma configuração do tipo P1N0V2 deverá ser a melhor combinação de parâmetros para o algoritmo baseado em estados de compartilhamento de páginas, para este conjunto de aplicações.

4.4. Análise Comparativa das Estratégias de Previsão de Acesso a Dados

Para grande parte das aplicações, as duas estratégias de previsão apresentaram resultados bastante semelhantes no que diz respeito às suas melhores configurações. Para a aplicação 3D-FFT, a melhor configuração da estratégia baseada em estados de compartilhamento de páginas (P0N0V0), segundo o critério estabelecido, apresentou uma pequena redução na taxa de erro, compensada por uma pequena elevação na taxa de acertos, em relação ao melhor autômato de previsão (A1). O mesmo se deu na aplicação IS, porém em menor proporção. Na aplicação *Gauss*, ocorreu justamente o oposto. A configuração P1N0V0 da estratégia baseada em estados de compartilhamento apresentou um aumento na taxa de erro compensado por um pequeno aumento da taxa de acertos em relação às melhores configurações das estratégias baseadas em preditores de desvio. Nas aplicações CG e SOR os resultados dos dois tipos de estratégia foram praticamente idênticos quando comparamos suas melhores configurações.

Em algumas aplicações a situação foi um pouco diferente. Na aplicação *Barnes-Hut*, a estratégia baseada em estados de compartilhamento apresentou uma pequena redução na taxa de erros, acompanhada também de uma pequena redução na taxa de acertos. Já na aplicação MG foram observadas as maiores discrepâncias. Para as estratégias baseadas em estado de compartilhamento, todas apresentaram elevadas taxas de erro que giravam em torno de 100 a 130% em relação ao total de falhas da aplicação. Porém, conseguiu-se atingir uma taxa de acerto de praticamente 100% para uma taxa de erro de aproximadamente 130%. No caso das estratégias baseadas em preditores de desvio, ocorreu uma redução na taxa de

acertos, que atingiu aproximadamente 50% no melhor caso. Esta redução na taxa de acertos foi acompanhada de uma redução na taxa de erros que não passou de 40% na maioria dos casos. Em particular, os algoritmos que utilizam tabelas de dois níveis, reduziram as taxas de erros a uma quantidade insignificante.

Capítulo 5

Técnica de Atualizações Seletivas

Os principais *overheads* encontrados em sistemas *software* DSM se devem à comunicação entre processadores e as ações de coerência. A Figura 5.1 mostra o impacto destes *overheads* no *speedup* das aplicações avaliadas em nosso trabalho, executando sobre a versão original de *TreadMarks*. Pela figura podemos ver que todas as aplicações estudadas apresentam um desempenho muito abaixo do ideal.

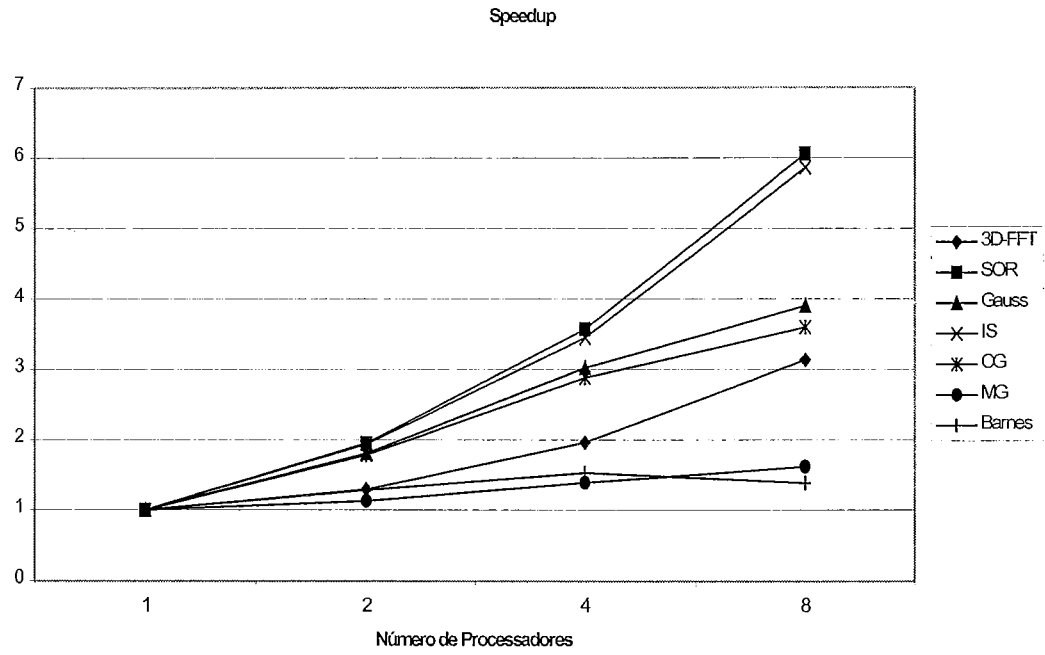


Figura 5.1 *Speedup* **das aplicações avaliadas em** *TreadMarks* **original**

A Figura 5.2 apresenta o tempo de execução das aplicações, normalizado e dividido em categorias. A categoria *comp* representa a computação útil realizada pelos processadores. Porém, alguns *overheads* de *hardware* como falhas na TLB e

na *cache* ainda se encontram nesta categoria. A criação de *twins* também foi incorporada neste tempo. A categoria *sync* representa o tempo despendido esperando e processando informações de sincronização em barreiras. A categoria *data* representa o tempo gasto esperando por dados que se encontram em outros processadores. A categoria *ipc*, por sua vez, representa o tempo gasto por um processador servindo as requisições de outros processadores.

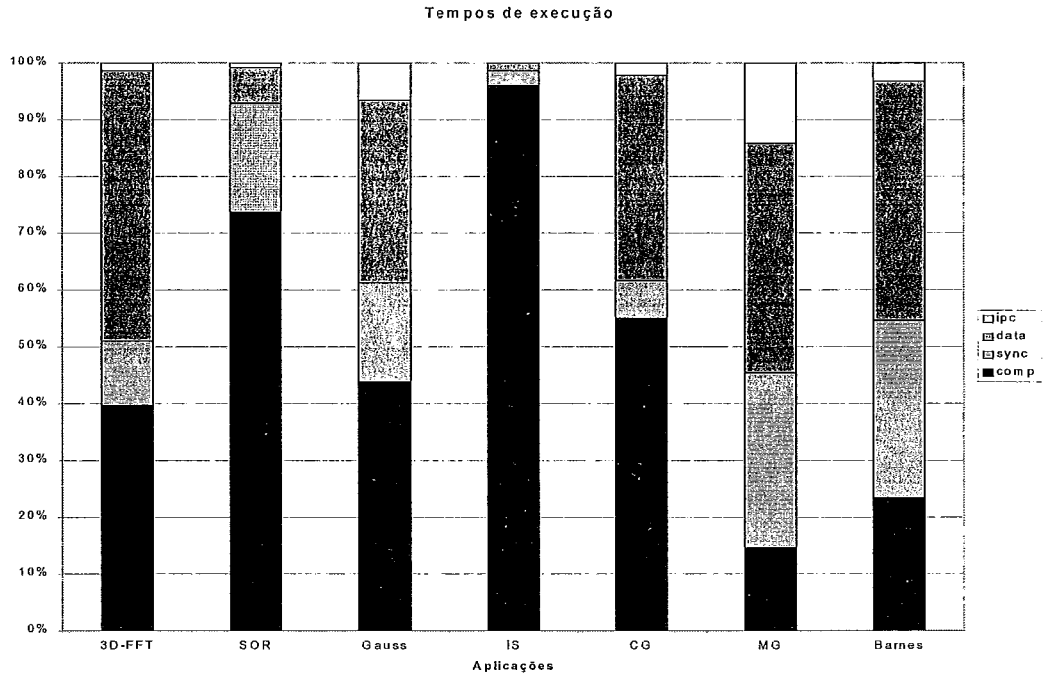


Figura 5.2 Tempos de execução em *TreadMarks* original, dividido em categorias

Pela Figura 5.2 podemos observar que, em muitos casos, os *overheads* causados pelo protocolo de coerência chegam a superar o tempo gasto pela aplicação realizando computação útil. Isso explica os baixos *speedups* exibidos pelas aplicações.

Podemos observar também que, na grande maioria das aplicações, o tempo gasto com espera de dados (categoria *data*) domina os demais *overheads*. Além disto, é valido ressaltar que uma grande parte do tempo da categoria *ipc* se deve ao tempo despendido pelo processador servindo as requisições de dados.

É sabido que os protocolos baseados em invalidações são normalmente mais vantajosos do que os protocolos baseados em atualizações, pois permitem uma redução no tráfego na rede. Porém, uma vez que se consegue determinar o

padrão de compartilhamento das páginas, é possível se beneficiar de um protocolo baseado em atualizações.

A técnica de atualizações seletivas pode ser ainda vantajosa em relação à técnica de *prefetching*, pois não necessita de mensagens de pedido a dados. Isso evita o tempo de espera que ocorre entre o pedido e recebimento dos dados, possibilitando um maior potencial de ganho de desempenho para a técnica de atualizações.

Neste sentido, utilizamos nossa estratégia baseada em previsão de estados de compartilhamento para implementar atualizações seletivas das modificações efetuadas nas páginas. Com isso podemos aliar o benefício dos protocolos baseados em invalidação, que somente buscam dados quando são necessários, com os benefícios dos protocolos baseados em atualizações, que evitam as falhas de páginas e os subseqüentes pedidos das modificações. A escolha da estratégia de estados de compartilhamento surge naturalmente, uma vez que está é mais apropriada para a implementação de técnicas que utilizam informações globais do que as estratégias baseadas em preditores de desvio.

Para avaliarmos o impacto real de nossas estratégias de previsão nos tempos de execução das aplicações, utilizamos uma nova técnica de atualização seletiva de *diffs* nos pontos de sincronização. Essa técnica está implementada da seguinte forma. Nos eventos de sincronização de barreira, a estratégia de previsão indica o conjunto das páginas que provavelmente sofrerão acesso na próxima fase, em cada um dos processadores. O algoritmo de atualização é constituído de duas etapas. Na primeira etapa cada processador, com base na informação de previsão e nos *write-notices* acumulados ao longo da execução, envia *diffs* para outros processadores. Na segunda etapa, um procedimento similar irá ocorrer no recebimento dos *diffs*. Ao serem recebidos, os *diffs* são aplicados e as páginas são desprotegidas para leituras.

Esse procedimento objetiva que, quando cada processador inicia a fase de recebimento, os *diffs* enviados pelos outros processadores já se encontram disponíveis em áreas de sua memória local, reservadas para o armazenamento de pacotes da rede. Desse modo, as operações de recebimento retornam os dados imediatamente, ou seja, sem bloquear o processo. O procedimento assume que a massa de dados da aplicação esteja razoavelmente balanceada entre os

processadores, o que é uma hipótese bastante plausí vel para a grande maioria das aplicações paralelas.

Além disso, a nossa técnica de atualizações é síncrona, isto é, não utilizamos interrupções. Isto é possível devido à informação global mantida entre os processadores. Especificamente, a propagação de eventos de leitura e seus respectivos intervalos, permite que cada processador saiba exatamente todos os *diffs* que deve receber tanto como enviar, em cada fase da aplicação. Desta forma, tanto o número de falhas de páginas como os subseqüentes pedidos de *diffs* são reduzidos permitindo uma diminuição no tempo de espera por dados remotos. Mais ainda diminui o número de interrupções e o tempo de processamento, uma vez que a atualização destas páginas será agora realizada de maneira síncrona e somente nos momentos de sincronização de barreira. A desvantagem é que aumentamos o tempo das operações de sincronização.

Em resumo, avaliamos nossas estratégias de previsão na redução dos custos relativos a espera por dados (*data*) e comunicação entre processos (*ipc*) em detrimento do custo de sincronização.

Existe, porém, uma dificuldade na implementação da técnica de atualização seletiva das páginas. Uma vez que as páginas atualizadas são desprotegidas, as falhas são evitadas. Contudo uma vez que os acessos às páginas atualizadas não mais acarretam falhas, a estratégia de previsão poderá erroneamente retirar processadores do conjunto de previsão. Por conta disso, esses processadores não mais seriam atualizados e voltariam a provocar falhas de acesso. Essa interferência entre a técnica de atualização e as estratégias de previsão se repete de maneira cíclica, degradando a acurácia da previsão e, com isso, o tempo de execução das aplicações. Esse problema se manifesta não somente em técnicas de atualizações seletivas, mas também em outras técnicas que necessitam realizar a previsão de acesso a dados, como por exemplo, nas técnicas de *prefetching*[10].

Em geral, as implementações de técnicas de redução de espera por dados encontradas na literatura[10] não aplicam as modificações no momento em que a mensagem de atualização ou o resultado de um busca antecipada chega em seu destino. Ao invés disso, as páginas são mantidas protegidas e as modificações só são aplicadas à página no momento em que ocorre uma falha de acesso. Com isso se evita que o controle de acesso às páginas seja perdido. Além disso, evita também a possível perda de tempo devido a aplicação de modificações desnecessárias. A

desvantagem dessa abordagem é que as falhas de leitura não são evitadas, e o processador é interrompido durante a execução da aplicação para aplicar as modificações.

Manter as falhas se torna de vital importância para a implementação de técnicas onde as previsões são baseadas em informações locais. Em nosso caso, a previsão é feita com base em informações globais do estado de compartilhamento da página. Isto nos permite evitar as falhas de leitura nas páginas sem, contudo, perder totalmente o controle sobre seu estado de compartilhamento.

Para evitar as falhas de página e ao mesmo tempo contornar o problema de interferência entre a técnica de atualização e nossa estratégia de previsão baseada em estados de compartilhamento, o estado **Predizendo** da máquina de estados que implementa a estratégia de previsão foi alterado. A modificação está na comparação entre o estado global de compartilhamento da página e o estado de compartilhamento previsto para a página. Esta comparação visa determinar se nossa previsão está de acordo com a realidade do padrão de compartilhamento exibido pela página. Para que a previsão não seja muito afetada pelas atualizações, é efetuada uma união entre os conjuntos dos leitores e consumidores do estado de compartilhamento de previsão com os respectivos conjuntos do estado de compartilhamento global da página antes da comparação. Com es te procedimento estamos supondo que os conjuntos de previsão de leitores e consumidores realmente efetuaram acessos a página. Ou seja, os estados de compartilhamento só serão considerados diferentes se o conjunto de escritores ou produtores se modificar, ou se um processador diferente daqueles contidos no conjunto de previsão dos leitores ou consumidores efetuar acessos à página. É importante observar que as falhas de escrita, para criação de *twins*, não são evitadas pela técnica de atualização. Estamos util izando, desta forma, nossa informação global para implementar uma técnica de atualizações que evite as falhas de página, possibilitando assim, maior eficiência.

Apesar de nossa técnica de atualizações possuir algumas estratégias de otimização agressivas como a eliminação de falhas de página e de interrupções, ela possui alguns aspectos conservadores na implementação. Em particular, a implementação avaliada envia uma mensagem para cada conjunto de *diffs* de uma página. Como isso se perde a oportunidade de agr egar os *diffs* de diferentes páginas em uma única mensagem de atualização. Este fato reduziria, mais ainda o número total de mensagens enviadas pelo protocolo. Nesse trabalho, estamos mais

interessados em uma primeira avaliação das estratégias propostas do que em certos detalhes de implementação.

5.1.1. Avaliação da Técnica de Atualização Seletiva

Tempos de Execução

A Figura 5.3 mostra os tempos de execução das aplicações separadas em duas barras para cada aplicação. As barras à esquerda correspondem a versão de *TreadMarks* original. As barras à direita correspondem aos tempos de execução utilizando-se a técnica de atualização seleti va juntamente com a estratégia de previsão baseada em estados de compartilhamento. Foi acrescentada a categoria *update* que corresponde ao tempo gasto transmitindo e aplicando as atualizações seletivas. Todas as aplicações foram executadas com os parâmetros de previsão **P1N0V2** que foi considerado a melhor combinação de parâmetros em relação aos resultados de taxas de previsão.

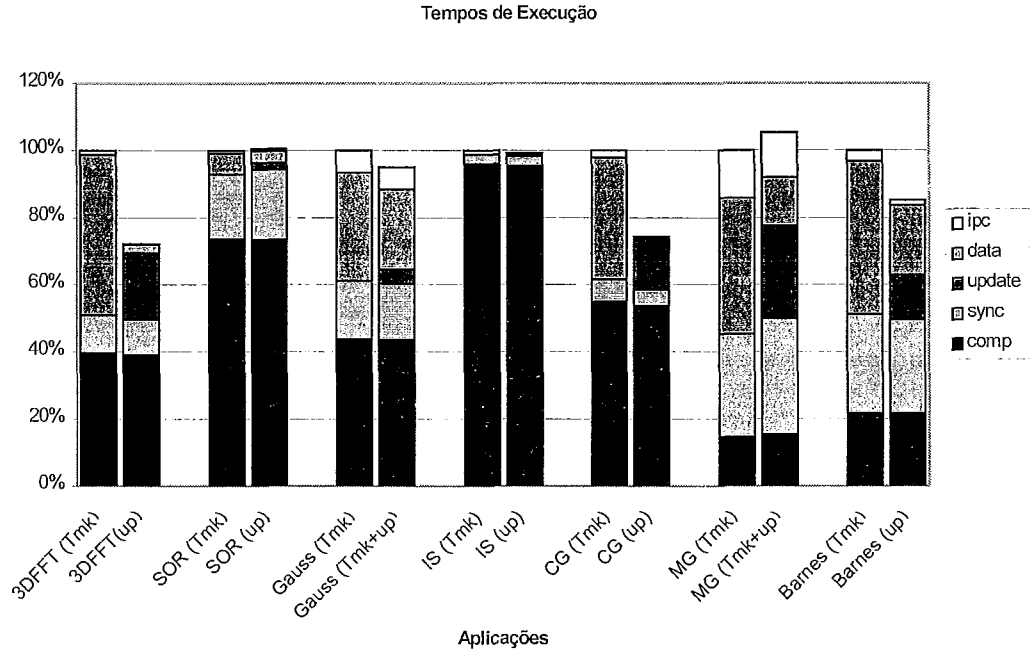


Figura 5.3 Comparação entre os tempo de execução de *TreadMarks* puro e com atualizações

Pela Figura 5.3 podemos ver que a aplicação 3D-FFT obteve um ótimo desempenho, atingindo uma redução do tempo de execução de aproximadamente 40% em relação à versão original. Isso se deve principalmente a enorme regularidade desta aplicação. A regularidade que nos permite obter excelentes taxas

de acerto, aliado ao grande *overhead* de espera por dados apresentado por esta aplicação, o que propicia ganhos de desempenho substanciais. Nossa técnica de atualização se mostrou eficiente em reduzir o tempo de espera por dados sem gerar outros *overheads* significativos.

A aplicação SOR apresentou uma leve degradação de desempenho ao ser adicionada a técnica de atualizações. O que pode ser constatado pela Figura 5.3. A aplicação apresenta poucas oportunidades de melhoria de desempenho uma vez que os *overheads* de espera por dados são bastante pequenos. A maioria das falhas da aplicação são compulsórias. Porém, as poucas páginas realmente compartilhadas apresentam um padrão de acesso único-escritor único-leitor altamente previsível.

Nossas expectativas iniciais eram de pequenos ganhos em relação a esta aplicação ao contrário da degradação que observamos. Ao verificarmos o problema mais cautelosamente pudemos constatar que a aplicação estava se beneficiando da característica de criação de *diffs* preguiçosa do protocolo original. Isso ocorria da seguinte forma. Após a barreira, o escritor da página mantém a mesma desprotegida até que seus *diffs* sejam requisitados. Entre a barreira e o pedido dos *diffs*, a página sofre novas escritas que passam desapercebidas ao protocolo, já que o acesso a página se encontra liberado. Após o pedido de *diffs* feito pelo leitor da página, os *diffs* são criados e a página é finalmente desprotegida. Após ser protegida, não ocorrem novas escritas na página. Este fenômeno é bastante dependente da temporização dos eventos.

Com a adição da técnica de atualizações, as páginas são protegidas antes da saída da barreira. Com isso ocorre uma nova falha de escrita quando o processador escreve nestas páginas na saída da barreira. Este fato praticamente dobra o número de falhas de escrita. Conseqüentemente, ocorre a geração de *twins*, destas páginas. Ao final da fase, novos *diffs* são gerados e novas mensagens de atualização são enviadas. Esse fenômeno é melhor percebido quando analisamos mais adiante o número total de *diffs* enviados.

A aplicação *Gauss* apresentou uma pequena redução no tempo de execução, como mostra a Figura 5.3. Muitas das falhas da aplicação são devido a *cold-start* como pudemos constatar pelos resultados das previsões. Além disso, a maioria das páginas apresenta apenas uma falha no decorrer da execução da aplicação. Devido a isso, os ganhos proporcionados pela técnica de atualização

foram mínimos, porém propiciaram uma pequena melhora no desempenho da aplicação.

A aplicação IS apesar de apresentar uma ótima taxa de acertos e uma taxa de erros baixíssima, não apresentou um ganho de desempenho significativo. Isso se deve ao fato desta aplicação ter um bom *speedup,* um *overhead* de espera por dados muito pequeno e um número reduzido de iterações. Estes fatores acabam por diminuir o ganho esperado com a técnica de atualização. As páginas da aplicação seguem basicamente um padrão migratório. Nessa aplicação, cada processador escreve exclusivamente na página em uma fase da aplicação, e uma vez que todos escreveram, acontece uma fase onde todos os processadores lêem a página. Esse padrão se repete durante as fases da aplicação, propiciando as altas taxas de acerto observadas.

A aplicação CG também apresentou uma ótima redução do tempo total de execução, como mostra a Figura 5.3. Essa melhora no desempenho da aplicação era esperada devido as ótimas taxas de acerto apresentadas na previsão dos estados de compartilhamento. A precisão da detecção do padrão de acesso a dados desta aplicação propiciou uma grande redução no tempo de espera por dados remotos. A redução foi de aproximadamente 40% do tempo total de execução da aplicação. Também podemos observar uma pequena redução no tempo de sincronização, o que a princípio ia de encontro a nossas expectativas. Pudemos constatar que isso ocorreu devido ao maior balanceamento na chegada a barreira, uma vez que a computação entre as barreiras não era mais interrompida por mensagens de pedidos a dados.

A aplicação MG apresentou um aumento no tempo de execução, como podemos ver na Figura 5.3. Neste caso, o tempo gasto com as atualizações não foi suficiente para compensar as reduções ocorridas no tempo de espera por dados. As altas taxas de erro de previsões, observadas na Figura 4.15, afetaram negativamente a aplicação. Porém, a pequena degradação no desempenho da aplicação não foi proporcional as altas tax as de erro de previsões observadas. Este fato pode ser melhor explicado na análise de *diffs* atualizados e requisitados, realizada mais adiante.

A aplicação *Barnes-Hut* apresentou uma boa redução no tempo de execução em relação ao protocolo original, como pode ser visto na Figura 5.3. Porém, para atingir estes resultados, a entrada desta aplicação deve que ser bastante reduzida

(de 2^{15} corpos para 2^{10} corpos) para que a perda de mensagens não degradasse muito o desempenho da aplicação.

Bytes Trafegados

A Figura 5.4 apresenta, para cada aplicação, o percentual de *bytes* trafegados pelo protocolo que utiliza atualizações seletivas em relação ao total de *bytes* trafegados pelo protocolo original. Por estes resultados, esperamos avaliar o *overhead* de nossa técnica em relação ao acréscimo produzido no tamanho das mensagens para propagação de informações, e também, em relação ao envio desnecessário de *diffs*.

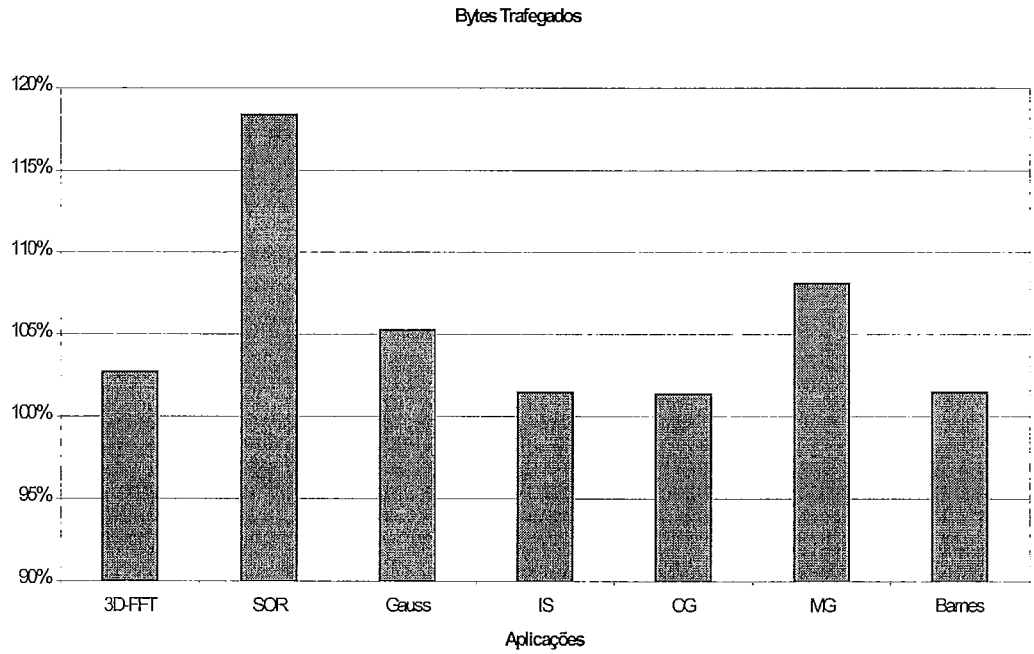


Figura 5.4 Proporção do total de bytes transferidos em relação a *TreadMarks*

Podemos observar que a maioria das aplicações apresentou um acréscimo muito pequeno em relação total de *bytes* transferidos, ficando entre 100% e 105% do total de bytes transmitidos pelo protocolo original. As exceções foram as aplicações MG e SOR. Esta última apresentou um acréscimo significativo no total de *bytes* transmitidos.

Na aplicação 3D-FFT, o total de *bytes* transferidos sofreu um acréscimo insignificante. Isso mostra que o *overhead* causado pela máquina de estados e pela técnica de predição de estados de compartilhamento se manteve muito baixa.

A aplicação SOR apresentou um aumento do total de *bytes* trafegados de aproximadamente 20% em relação ao protocolo original. Conforme mencionado anteriormente, isso ocorreu devido ao efeito da eliminação da característica de criação de *diffs* preguiçosa por parte da técnica de atualizações seletivas. Esse efeito acarreta um aumento no número de *diffs* enviados, o que explica esse aumento no total de bytes transmitidos. O *overhead* da máquina de estados pouco contribui para este aumento.

A aplicação MG também sofreu um acréscimo significativo no total de *bytes* trafegados. Este acréscimo se deve a um aumento no número de *diffs* enviados. Diferente da aplicação SOR, o motivo deste aumento no número de *diffs* enviados se deu por causa das elevadas taxas de erros de previsão de acesso a dados encontradas nesta aplicação. Taxas de erros estas que chegam a 100% do total de falhas ocorridas.

Na aplicação *Gauss*, o número total de bytes transmitidos não sofreu um acréscimo significativo, ficando em torno de 5% do total de *bytes* transmitidos originalmente. Esse acréscimo se deve basicamente a propagação de eventos da máquina de estados (FIESTA). Pudemos constatar que essa aplicação apresenta um número elevado de eventos de leitura e escrita em relação às demais, o que provoca um pequeno aumento no *overhead* de FIESTA.

As aplicações IS, CG e *Barnes-Hut* apresentaram aumentos insignificantes no número de *bytes* transmitidos em relação ao protocolo original.

Número de Mensagens

A Figura 5.5 apresenta o percentual do número de mensagens transmitidas pelo protocolo utilizando atualizações seletivas, em relação ao protocolo original. Esta métrica tem o objetivo de avaliar o impacto da redução do número de mensagens de pedidos de *diffs*, por acertos de previsão. Além disso, avaliar o impacto de um possível aumento no número de mensagens de *diffs* enviadas devido a previsões erradas.

Podemos observar que em todos os casos obtivemos uma redução do número total de mensagens transmitidas em relação ao protocolo original. Estas reduções variam de aproximadamente 50%, como no caso de 3DFFT e CG, a aproximadamente 10%, como nos casos de *Gauss* e SOR.

Na aplicação 3D-FFT, o número de mensagens gerados sofreu uma grande redução que atingiu aproximadamente 50% do total de mensagens transmitidas pela versão original de *TreadMarks*. Esta redução é resultado da eliminação quase que completa das mensagens de pedidos de *diffs* gerados pelo protocolo original, propiciada pelas altas taxas de acerto de previsão de acesso a dados.

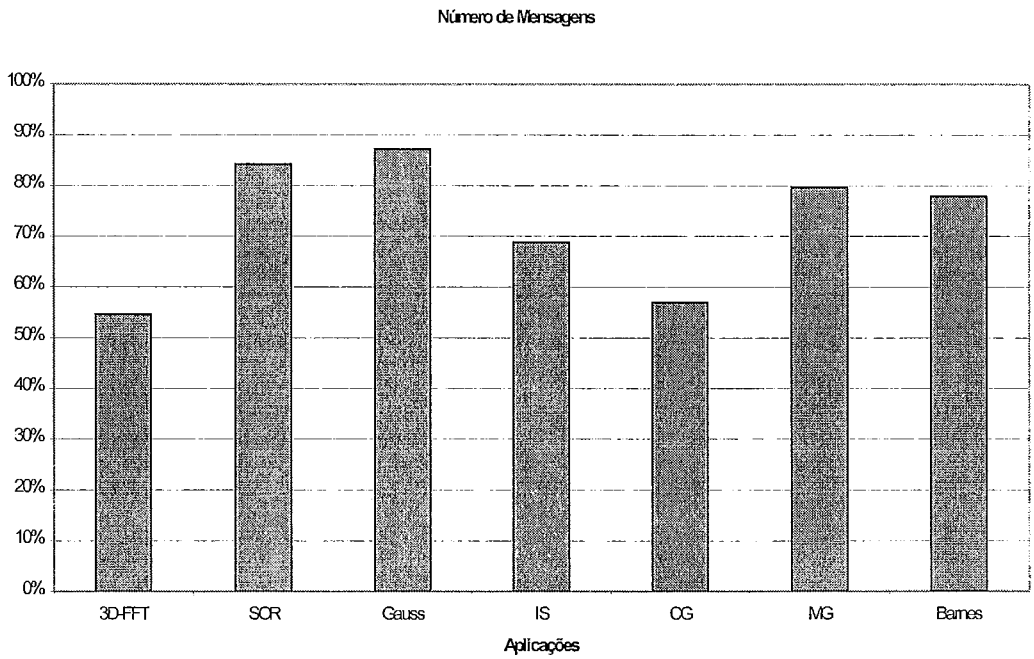


Figura 5.5 Proporção do número de mensagens em relação a *TreadMarks*

Para a aplicação SOR a redução do número de mensagens de pedidos de *diffs* foi ofuscada por um aumento no número de *diffs* enviados. Ainda assim, ocorreu uma redução do número de mensagens transmitidas em relação à versão original do protocolo de aproximadamente 15%.

Na aplicação *Gauss* a baixa redução no número de mensagens se deve a ineficiência da estratégia de previsão nesta aplicação. A redução no número de mensagens atingiu pouco mais de 10% em relação ao protocolo original. Grande

parte das mensagens desta aplicação são pedidos de página (e não de *diffs*) realizados no início da computação.

A aplicação CG também sofreu uma redução significativa no número de mensagens, transmitindo menos de 60% das mensagens transmitidas pelo protocolo original. Esta redução, como no caso da aplicação 3D-FFT, também de deve a ótimas taxas de acerto, e taxas de erros insignificantes exibidas por esta aplicação.

As aplicações IS, MG e *Barnes-Hut* também apresentaram boas reduções no número de mensagens, atingindo uma faixa de 60% a 70% do total de mensagens transmitida originalmente.

Falhas de Leitura

A Figura 5.6 apresenta a proporção do número de falhas de página de leitura do protocolo modificado em relação à versão original de *TreadMarks*.

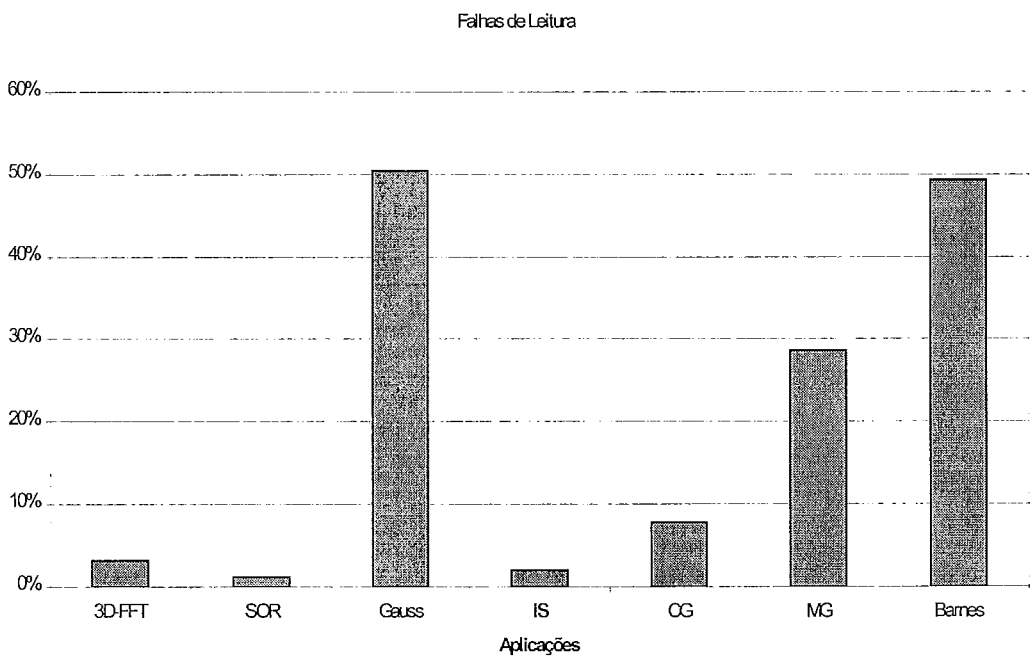


Figura 5.6 Proporção de falhas de leitura em relação a *TreadMarks*

Pela Figura 5.6, podemos observar que a quantidade total de falhas de leitura foi bastante reduzida em todas as aplicações testadas. Além disso, atingiram nos piores casos, uma redução de 50% do total de falhas de leitura das aplicações.

Para a aplicação 3D-FFT, a Figura 5.6 mostra que as falhas de página foram praticamente extintas pela técnica de atualizações. Mais uma vez, esse resultado se deve às ótimas taxas de acerto provenientes da técnica de estados de compartilhamento, aliada a baixíssimas taxas de erro. Esse mesmo comportamento se verifica nas aplicações SOR, IS e CG.

A aplicação MG apresentou um número de falhas de leitura de 30% em relação ao protocolo original. As aplicações *Gauss* e *Barnes-Hut* foram as que menos apresentaram reduções na quantidade total de falhas de leitura atingindo 50% do número de falhas de leitura da versão original de *TreadMarks*.

Diffs Pedidos e *Diffs* Atualizados

A Figura 5.7 apresenta o percentual de *diffs* pedidos e *diffs* atualizados do protocolo modificado, em relação à versão original do protocolo. Através destes resultados é possível analisar a proporção de *diffs* enviados em excesso pelo protocolo de atualizações seletivas em relação ao protocolo origina. Estes resultados também mostram a proporção de *diffs* atualizados em relação aos *diffs* pedidos.

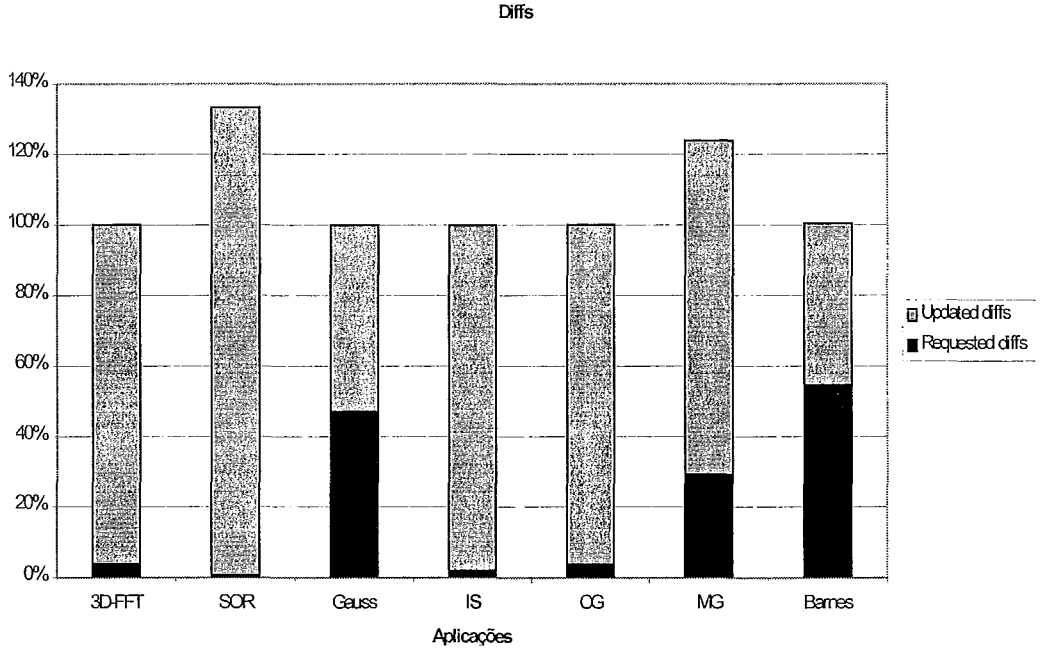


Figura 5.7 Relação entre o número de *diffs* pedidos e o número de *diffs* atualizados

Pela figura 5.7 observamos que, para a maioria das aplicações, não houve envio de *diffs* em excesso. As aplicações SOR e MG foram exceções e apresentaram um acréscimo no total de *diffs* enviados de aproximadamente 30% e

20%, respectivamente. Em relação à quantidade de *diffs* pedidos, vemos que apenas as aplicações *Gauss*, MG e *Barnes-Hut*, apresentaram uma quantidade significativa de pedidos.

As aplicações 3D-FFT, IS e CG apresentaram um excesso na quantidade de *diffs* enviados insignificante. Estas aplicações também apresentaram ótimas taxas de *diffs* atualizados. Esses resultados eram esperados e refletem as ótimas taxas de acertos e de erros de acesso a dados destas aplicações.

Nas aplicações *Gauss* e *Barnes-Hut,* podemos observar que apenas metade dos *diffs* foram atualizados. Isso também confirma os resultados das taxas de previsão de acesso a dados.

Na aplicação SOR, podemos observar claramente o efeito da eliminação da criação de *diffs* preguiçosa na quantidade de *diffs* enviados. Vemos neste caso que o protocolo com atualizações aumentou a quantidade de *diffs* enviados em aproximadamente 30% em relação a versão original de *TreadMarks*.

Para a aplicação MG observamos que o número de *diffs* em excesso não correspondia aos resultados obtidos nas medições de taxas de previsão (Figura 4.15). Eles eram de aproximadamente de 100% em relação ao tot al de falhas ocorridas. Pudemos constatar que isso ocorria pois algumas atualizações, embora consideradas como erros, eram na verdade aproveitadas em fases subseqüentes da aplicação. Isto porque o critério de acerto ou erro é baseado em um acesso preciso, não considerando esse tipo de incorreções. Sendo assim, a aplicação se comportou de forma melhor que o esperado e o aumento na quantidade total de *diffs* pedidos e atualizados chegou a aproximadamente 20% do protocolo original, ao invés de 100% como indicavam os resultados de previsão de acesso.

Capítulo 6

Trabalhos Relacionados

Os sistemas DSM são alvos de uma grande quantidade de trabalhos de pesquisa. Neste capítulo, apresentamos alguns dos trabalhos sobre sistemas DSM que mais se aproximam com o nosso.

O primeiro sistema *software* DSM foi o sistema Ivy[24]. Este sistema implementa um modelo de consistência seqüencial, utiliza a página como unidade de coerência, não possui suporte para páginas com múltiplos -escritores e utiliza invalidações para manter a coerência. Este sistema foi precursor e mostrou que poderia ser possível a implementação de um sistema de computação paralela utilizando um hardware de prateleira, com uma interface de programação similar aos multiprocessadores, e com um menor custo. Ainda assim, e ste sistema apresentava um desempenho pouco satisfatório, por implementar um modelo de consistência muito restritivo e por não tratar o problema do falso compartilhamento.

O sistema *Munin*[12] foi o primeiro sistema software DSM a relaxar o modelo de consistência do sistema para melhorar seu desempenho. Neste sistema foi implementado o modelo de consistência *release consistency*, o que propiciou uma redução significativa na quantidade de comunicação do sistema. *Munin* também foi o primeiro sistema a empregar o mecanismo de *diffs* e *twins*, que propicia um suporte a múltiplos escritores concorrentes em uma mesma página. Além disso, é possível que o programador especifique explicitamente o padrão de compartilhamento de certas estruturas de dados, dentro de um conjunto pré-estabelecido de padrões. Com isso, o sistema pode fazer uma melhor escolha dos protocolos a serem utilizados para cada uma destas estruturas. Em contraste com essa abordagem, nosso trabalho detecta padrões genéricos de compartilhamento, transparente ao programador, e em tempo de execução.

Amza *et al.*[4] implementaram uma adaptação no protocolo de *TreadMarks* que dinamicamente detecta a presença ou ausência de falso compartilhamento nas páginas compartilhadas da aplicação. Com isso, o sistema pode eliminar os mecanismos de suporte a múltiplos escritores (*diff*s e *twins*) no caso da ausência de falso compartilhamento, diminuindo assim o *overhead* do protocolo. O sistema também é capaz de habilitar estes mesmos mecanismos, para as páginas onde o falso compartilhamento aconteça. Para realizar esta adaptação, o sistema utiliza mensagens extras de posse de páginas. Esse tipo de adaptação não foi implementado em nosso sistema.

O sistema ADSM[25] também é um sistema adaptativo baseado em *TreadMarks*. ADSM implementa um algoritmo de categorização que visa detectar, dinamicamente, padrões de compartilhamento nas páginas da aplicação. Para isso, ADSM utiliza uma máquina de estados conhecida como SPC, que é capaz de categorizar as páginas segundo os padrões: migratório , produtor-consumidor e falsamente compartilhada. As páginas categorizadas como migratórias ou produtor-comsumidor são tratadas em modo único-escritor, enquanto que as páginas falsamente compartilhadas obedecem aos mecanismos de *twins* e *diffs*. A técnica de adaptação único-escritor/múltiplos-escritores de ADSM não necessita de mensagens de posse extra. Além disto, para as páginas tratadas como único escritor, a coerência é mantida através de atualização. Nas demais páginas, a coerência é mantida através de invalidações. Diferente de ADSM, em nosso trabalho optamos por estratégias de previsão de padrões de compartilhamento genéricas, que buscam detectar padrões quaisquer, não conhecidos *a priori*. Além disto, nenhuma adaptação único-escritor/múltiplos-escritores foi implementada neste trabalho. Outra diferença consiste no fato de nossa técnica de atualização ser síncrona e baseada em *diffs*, ao invés de ser assíncrona e baseada em páginas, como em ADSM. Isso permite que páginas com múltiplos escritores possam se beneficiar das atualizações. Por enquanto, nossa atualização se restringe as sincronizações por barreiras.

Hill e Mukherjee deram o primeiro passo no sentido de utilizar mecanismos de predição genérica para acelerar protocolos de coerência, desenvolvendo o preditor de mensagens de coerência denominado *Cosmos*[27]. Diferente dos demais sistemas citados neste capítulo, o sistema *Cosmos* foi desenvolvido para predição de mensagens de coerência de sistema multiprocessadores com memória compartilhada. Eles utilizam protocolos de coerência de *cache* baseados em

diretório. Cosmos é capaz de prever a origem e o tipo da próxima mensagens de coerência utilizando uma lógica que é uma extensão do preditor de desvios de dois níveis desenvolvido por Yeh e Patt[31]. Em nosso trabalho, também demos o enfoque no desenvolvimento de mecanismos genéricos de previsão, e adaptamos o predtior de dois níveis ao problema de previsão de acesso a dados em sistemas software DSM.

A máquina de estados FIESTA foi proposta no trabalho de M. C. S. de Castro[14]. Neste mesmo trabalho, foi proposto um algoritmo de categorização de páginas, denominado RITMO. Assim como nossa estratégia de previsão de acesso a dados baseada em estados de compartilhamento, RITMO se baseia em informações disponibilizadas por FIESTA para categorizar as páginas. O objetivo principal de RITMO é o de categorizar mais precisamente as páginas da aplicação. Com isso, é possível realizar otimizações mais agressivas no protocolo do que alguns algoritmos de categorização mais conservadores. Com base nas categorizações, foram implementadas técnicas de *forwarding*, *prefetching* e a adaptação único-escritor/múltiplos-escritortes com o objetivo de avaliar a integração destas diferentes técnicas. Nosso trabalho visa dar continuidade ao trabalho de Castro, no sentido de avaliar técnicas que se utilizem da disponibilidade de informações detalhadas sobre o compartilhamento das páginas. Neste trabalho, optamos por utilizar técnicas de previsão genéricas, ao invés de categorizar as páginas da aplicação em padrões pré-estabelecidos, como RITMO. Para avaliar a eficiência destes preditores genéricos apenas uma técnica de atualizações seletivas foi implementada.

Capítulo 7

Conclusões e Trabalhos Futuros

Neste trabalho implementamos a máquina FIESTA de estados finitos capaz de capturar detalhadamente o padrão de compartilhamento dos dados da aplicação. FIESTA identifica dinamicamente para cada página da aplicação seus processadores leitores, escritores, consumidores e produtores de modo a definir seu estado de compartilhamento no decorrer da execução da aplicação. Para isso, FIESTA é implementada de forma distribuída, na qual os estados de compartilhamento são propagados globalmente, ou seja, são disponibilizados para todos os processadores no decorrer da execução da aplicação.

Foram propostas e desenvolvidas também duas novas estratégias de previsão de acesso a dados compartilhados. Uma estratégia se baseia nos estados de compartilhamento das páginas enquanto outra se baseia em preditores de desvios condicionais. Em contraste com as estratégias existentes que detectam apenas padrões pré-definidos, as estratégias de previsão que desenvolvemos visam detectar padrões de compartilhamento quaisquer.

Um novo mecanismo de atualizações seletivas foi também proposto e implementado para avaliar os benefícios da estratégia de previsão baseada em estados de compartilhamento na diminuição de *overheads* de acesso a dados remotos.

A implementação da técnica de atualizações seletivas se diferencia das anteriores ao utilizar *diffs* e não páginas, permitindo que páginas com múltiplos escritores concorrentes possam ser atualizadas antecipadamente. Além disso, a informação global disponibilizada por FIESTA permitiu duas otimizações extras. A primeira foi proporcionar a implementação síncrona da técnica, na qual emissor e receptor são sincronizados em operações *send* e *receive* no momento do envio da

mensagem de atualização. Dessa forma foi possível eliminar o uso de interrupções para as mensagens de atualizações. A segunda otimização foi eliminar falhas de leitura.

Com relação aos resultados experimentais, observamos que ambas as estratégias de previsão de acesso a dados obtiveram um bom desempenho para o conjunto de aplicações avaliadas. Para grande parte das aplicações as melhores técnicas de ambas as estratégias produziram resultados muito semelhantes. Em algumas aplicações, a situação foi um pouco diferente. Observamos que, em geral, as técnicas baseadas em preditores de desvio têm tendência de apresentarem menores taxas de acertos, mas também, menores taxas de erro, do que as técnicas baseadas em estados de compartilhamento.

Em relação às técnicas de previsão baseadas em preditores de desvio, as que utilizam autômatos de dois níveis são geralmente superiores àquelas de autômatos de apenas um nível, apresentando maiores taxas de acerto e menores taxas de erro. Essa superioridade tende a aumentar junto com o aumento da irregularidade das aplicações.

No caso de técnicas de previsão baseadas em estados de compartilhamento, os parâmetros de limite positivo do conjunto de previsão e de verificação, contribuíram significativamente para a determinação mais precisa do padrão de compartilhamento das aplicações. Porém, o parâmetro de limite negativo do conjunto de previsão não surtiu efeito algum nas aplicações avaliadas. Esses resultados indicam que algum tipo de filtro na captura de padrões se faz necessário, para que eventuais ruídos não interfiram muito na determinação de padrões.

É importante ressaltar que a previsão de acesso é apenas uma das possíveis funções que podem ter o estado de compartilhamento das páginas. Isso porque, diferente das estratégias de previsão de desvio, ela separa os processadores em quatro conjuntos distintos de leitores, escritores, produtores e consumidores, e provê esta informação em nível global no sistema. O trabalho de Castro[14] por exemplo, utiliza as informações de estado de compartilhamento para implementar adptação único-escritor/múltiplos-escritores e técnicas de *prefetching* e *forwarding*.

Nas duas estratégias de previsão não foi possível avaliar suficientemente a capacidade das técnicas em se recuperar de padrões mal detectados porque o

padrão de compartilhamento das aplicações científicas é, em geral, muito regular e apresenta poucas mudanças.

Em relação à técnica de atualização seletiva síncrona, podemos afirmar que ela apresenta bons resultados para aplicações onde o *overhead* de espera por dados é significativo e o número de páginas compartilhadas é grande o suficiente para compensar o custo da técnica. O fato de a técnica de atualização ser síncrona não acarretou problemas de espera por atualizações. Em geral, o número de páginas utilizadas por cada um dos processadores nas aplicações testadas era bastante balanceado, causando assim o efeito desejado de sobreposição de mensagens de envios com as de recebimentos. Mais ainda, a técnica propiciou o benefício da eliminação de interrupções do lado do processador atualizado.

Nos casos onde o tempo de espera por dados não é significativo constata-se que o tempo de execução das aplicações praticamente não foi afetado mesmo quando as taxas de acertos não são muito boas e/ou a técnica não surtiu o efeito desejado. Isso mostra que nos piores casos, a técnica não melhora, e nem degrada o desempenho da aplicação.

Uma implementação de atualizações síncronas para as estratégias de previsão baseadas em preditores de desvio, é consideravelmente mais complicada e possivelmente menos eficiente uma vez que a estratégia se baseia em informações locais. Além disto, não há solução imediata para o problema de interferência entre as atualizações e as previsões. Isto porque este tipo de previsão não mantém informações sobre o estado global da página, não propaga eventos de leitura e nem seus intervalos, e também não é capaz de discernir entre leitores e escritores (cada escritor mantém informação sobre seus leitores). Uma possível solução neste caso seria não validar a página, possivelmente não aplicando os *diffs* a mesma, para que as falhas de página não fossem eliminadas e o resultados das previsões não fossem corrompidos.

Em comparação com a técnica de atualização por páginas, a técnica de atualização de *diffs*, não reduz o número de falhas de escrita nem de criação de *twins*. Porém, só envia pela rede as modificações feitas na página, mantendo o número de dados trafegados na rede constante. Além disto, podemos efetuar atualizações de qualquer página compartilhada da aplicação, especialmente aquelas com múltiplos escritores concorrentes.

É difícil comparar as estratégias de envio de páginas e envio de *diffs*. Isso porque os resultados são altamente dependentes das entradas utilizadas pelas aplicações. A princípio podemos dizer que nossa técnica pode apresentar melhores resultados em aplicações que exibem uma grande quantidade de falso compartilhamento e/ou fragmentação. Porém, pode apresentar resultados piores em aplicações que possuam páginas essencialmente de um único escritor. Portanto, torna-se atraente avaliar a técnica adaptativa único-escritor/múltiplos-escritores em futuras pesquisas.

Um dos principais problemas da técnica de atualização é o *overhead* causado na rede no momento da disseminação de mensagens de atualizações, que ocorrem de uma só vez em nossa implementação. Em determinado ponto a infra-estrutura de rede, quer seja nos *buffers* das interfaces, nos *buffers* do sistema operacional, ou nos *switches*, começa a não suportar a sobrecarga e a eliminar pacotes. Apesar das perdas, o protocolo consegue operar corretamente, uma vez que estas páginas ou *diffs* serão buscados na falha. Porém, a degradação de desempenho aumenta muito podendo tornar esta solução inviável.

Como trabalhos futuros pretendemos investigar soluções para o problema de sobrecarga na rede causado pelas atualizações. Outra pesquisa importante será estender o mecanismo de previsões para as sincronizações por *locks*. Este mecanismo deverá ser similar ao implementado nas barreiras, mas considerando que nesses casos não se conhece o estado global de compartilhamento da página já que só envolve dois processadores por vez, ao contrário da barreira, que promove uma sincronização global.

A partir dos novos protocolos que incorporem essas soluções, será possível avaliar uma gama maior de aplicações. Principalmente aplicações que apresentam uma maior irregularidade nos padrões de acesso a dados, como é o caso das aplicações que utilizam *locks*.

Referências Bibliográficas

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

[2] S. Adve e K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, páginas 66–76, Dezembro 1996.

[3] A. Agarwal, R. Bianchini, D. Chaiken, K. Jhonson, D. Kranz, J. Kubiawicz, B -H. Lim, K. Mackenzie, e D. Yeung. The MIT Alewife Machine: Arquitecture and Performance. In *Proc. of the 22nd An. Int'l Symp. On Computer Architecture (ISCA'95)*, páginas 2–13, Junho 1995.

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

[5] C. Amza, A. Cox, K. Rajamani, e W. Zwaenpoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proc. of the 6th ACM SIGPLAN Symp. on Priciples and Practice of Parallel Programming (PPOPP'97)*, páginas 90–99, Junho 1997.

[6] J. K. Bennett, J. B. Carter, e W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. of the 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, páginas168–177, Março 1990.

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

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

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

DSMs. In *Proc. of the 7t^{h} Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, páginas 198–209, Outubro 1996.

[10] R. Bianchini, R. Pinto, e C. L. Amorim. Data Prefetching for Software DSMs. In *Proc. of the Int'l Conference on Supercomputing'98*, páginas 385–392, Julho 1998.

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

[12] J. B. Carter, J. K. Bennet, e W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Priciples*, páginas 152–164, Outubro 1991.

[13] J. B. Carter, J. K. Bennett, e W. Zwaenepoel. Techniques for Reducing Consistency-Related Information in Distributed Shared Memory Systems. *ACM Trans. on Computer Systems*, páginas 205–243, Agosto 1995.

[14] M. C. S. Castro e C. L. Amorim. Efficient Categorization of Memory Sharing Patterns in Software DSM Systems. *In Proc. of the 15th Int'l Parallel and Distributed Prrocessing Symp. (IPDPS'01)*, Abril 2001.

[15] S. Dwarkadas, P. Keleher, A. L. Cox, e W. Zwaenepoel. Evaluation of Release Consistency Software Distributed Shared Memory on Emerging Network Technology. In *Proc. of the 20th An. Int'l Symp. on Computer Architecture (ISCA'93)* , páginas 244–255, Maio 1993.

[16] K. Gharachorloo, D. E. Lenoski, J. Laudon, P.Gibbons, A. Gupta, e J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessor. In *Proc. Of the 17th Na. Int'l Symp. on Computer Arquitecture*, páginas15–26, Maio 1990.

[17] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, e W. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techiniques. *In Proc. Of the 18th Annual Int'l Symp. on Computer Arquitecture,* páginas 254–263, Maio 1991.

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

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

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

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

[22] P. Keleher, S. Dwarkadas, A. L. Cox, e W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the 1994 Winter Usenix Conference*, páginas 115–131, Janeiro 1994.

[23] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Computer, páginas 690–691, Setembro 1991.

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

[25] L. R. Monnerat e R. Bianchini. ADSM: A Hybrid DSM Protocol that Efficiently Adapts to Sharing Patterns. *Relatório Técnico ES-425/97, COPPE/UFRJ*, Março 1997.

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

[27] S. S. Mukherjee e M. D. Hill. Using Prediction to Accelerate Coherence Protocols. *Proc. of the 25th Int'l Symp. on Computer Architecture (ISCA'98)*, páginas 179–190, Junho 1998.

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

[29] J. E. Smith. A Study of Branch Prediction Strategics. *In Proc. of the 8th Int'l Symp. on Computer Architecture*, páginas 135–148, Maio 1981.

[30] L. Whately, R. Pinto, M. Ragarjan, L. Iftode, R. Bianchini e C. L. Amorim. Adaptative Techniques for Home-based Software DSMs. *In Proc. of the 13th Symp on Computer Architecture and High-Performance Computing (SBAC-PAD'01)*, páginas 164–171, Setembro 2001.

[31] T-Y Yeh e Y. N. Patt. Alternative Implementation of Two-Level Adaptive Branch Prediction. *In Proc. of the 19th Int'l Symp. on Computer Architecture*, páginas 124 – 134, Maio1992.