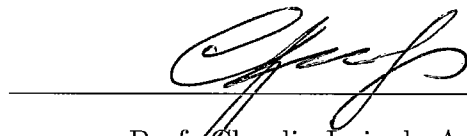
SUPORTE DE COMUNICAÇÃO PARA SISTEMAS DE MEMÓRIA COMPARTILHADA DISTRIBUIDA EM SOFTWARE

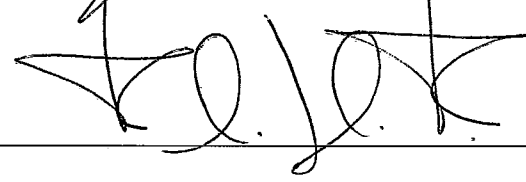
Fabio de Matos Quaresma Gonçalves

DISSERTAÇÃO 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.

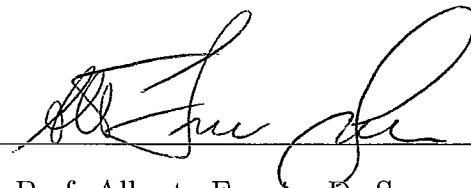
Aprovada por:



Prof. Claudio Luis de Amorim, Ph.D.



Prof. Felipe Maia Galvão França, Ph.D.



Prof. Alberto Ferreira De Souza, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

SETEMBRO DE 2007

GONÇALVES, FABIO DE MATOS QUARESMA

Suporte de Comunicação para Sistemas de Memória Compartilhada Distribuída em *Software* [Rio de Janeiro] 2007

XIV, 79 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2007)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1. Sistemas Operacionais

2. Memória Compartilhada Distribuida

3. Suporte de Hardware para Computação Distribuida

I. COPPE/UFRJ II. Título (série)

À minha familia

Agradecimentos

Ao longo dos longos anos em que foi realizado este trabalho, muitas águas passaram, e vieram muitas pessoas a quem agradecer.

Tudo começou com o convite do Fabio e do Andre, para participar do projeto do Relógio Global. Era uma vez ...

Nessa época eu já tinha o prazer de cantar no Coral da COPPE, com o maestro Sansão. Heidi, cade vc?

A equipe do LCP tem um peso enorme neste trabalho, de diversas formas. No tempo que passei com eles, aprendi muitas coisas que espero levar na minha bagagem. Preciso agradecer eternamente ao Lauro (oráculo) e Leo, pelo suporte moral e ajuda em todos os momentos, ainda mais nos de crise. Do Maltar, sem cujo apoio teria sido impossivel chegar ao fim, entendi a importância da insistência. Do Alberto, que proporcionou momentos memoráveis de aprendizado, entendi o significado da persistência para atingir um objetivo. Do Claudio, quis levar um pouco da objetividade e clareza na escrita, mas o tempo foi curto para isso.

Este trabalho me acompanhou por muitos lugares, nos quase 5 anos que durou. Teve uma passagem breve por Vitória, ali na UFES com o Sotério e o Alberto. Me hospedei na casa da tia Ieda semanas antes do carnaval. Não deu muito tempo, mas consegui umas horinhas pra brincar. Depois voltei pro Rio e brinquei também, já de viagem marcada pra Salvador. Foi a primeira e única vez que passei o mesmo carnaval em 3 cidades.

Já passou pela 2305, onde Raphael, Reinaldo, Heitor, Márcio, Fabiano e Leo animavam a República. Era difícil estarem todos em casa, mas quando acontecia sempre dava em alguma coisa: cada um pegava um instrumento e acabava saindo

uma festa improvisada. Com a pizza do Reinaldo.

Não dá pra esquecer da turma do CTI-Seg do LPS, especialmente o Tadeu. Nas horas negras, estava sempre intercedendo, me fazendo voltar para o lado bom da Força ;). Quase abandonei tudo algumas vezes, mas ele não deixava. E as conversas com o Edson, no meio do corredor ou onde fosse, eram um gás adicional, sempre com milhares de ideas não convencionais para qualquer assunto.

Tenho que agradecer também ao pessoal da UBA (Universidade de Buenos Ayres), cujo *cluster* ajudou a desenvolver parte do protocolo. Ali mesmo, na companhia de Francisco e Christian, sempre trocando *mails* e mensagens com William no LCP. Com a ajuda de Lucia e Pablo Oveja acabei me adaptando, e passei os dias sem dificuldade.

Depois seguiu por um tempo em São José dos Campos. Ali fui recebido pelo pessoal da BCC e Avibrás. Foram 6 meses, mas deu tempo até de me entrosar com o Coro da UNESP, dar uma volta em Rio Claro e virar fã da Sandra. Também, com a torcida do Elias, Takashi, Suzi e Silvio, não fica pra menos.

Depois voltei por um tempo a Buenos Aires, precisava da concentração do Jardim Botânico. Como ia perceber mais tarde, de repente ganhei uma família a mais. Apareceu também Ariel e os sanduiches de lomito. E Sandra e Luli passaram a frequentar a casa. Foram todos me enrolando, quase não me deixaram voltar para o Rio pra terminar com as pendências. Para o azar de Alicia consegui, e ainda trouxe uma jóia rara que levo a tiracolo até hoje.

Quando voltei ao Rio fui pra 4MR, onde conheci o Bernardo, o cara mais prestativo que já pude conhecer.

Também participaram o Prof. Márcio, sempre dando conselhos, mesmo que obrigado por mim. E o Prof. LW, meu eterno futuro orientador, que eu fiz questão de pôr à prova tantas vezes.

Era dificil, mas de vez em quando conseguia dividir uma cerveja ou fazer um churrasco com Diego e Thiago — com limite de piadas. Coitada da Flávia. Depois vieram os sucos de laranja/chopps com Paty e Marlene, que espalham alegria por onde passam.

Já devo tanto a tanta gente que é melhor não continuar ...

Minha família sempre esteve presente, antes mesmo do inicio deste trabalho. Hoje eu fico impressionado com minha mãe, Claudia e Marina com as doses cavalares de paciência e bom humor quando tudo dava errado, mas eu preciso convencer elas de que valeu a pena.

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

SUPORTE DE COMUNICAÇÃO PARA SISTEMAS DE MEMÓRIA COMPARTILHADA DISTRIBUIDA EM SOFTWARE

Fabio de Matos Quaresma Gonçalves

Setembro/2007

Orientador: Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Neste trabalho é descrito o desenvolvimento de uma rede auxiliar de sincronização para *clusters* de computadores, cuja arquitetura favorece a transmissão de pacotes de mensagens da ordem de dezenas de bits. A principal característica da Rede de Sincronização é a disseminação de dados em *broadcast*, permitindo a criação de uma ordenação total para as mensagens transmitidas.

Para avaliar o potencial benefício desta rede, foi implementado o protocolo Sincro, um Sw-DSM que utiliza consistência relaxada e estabelecimento de residências para as páginas. Este protocolo é avaliado em um *cluster* de quatro computadores, interligados com *Fast Ethernet* e utilizando a Rede de Sincronização como rede auxiliar. As características de operação da Rede permitem que cada nó do *cluster* tenha conhecimento sobre as operações de sincronização e coerência efetuadas pelos outros nós, quase que instantaneamente. Isto permite uma remodelagem das estruturas de coerência do protocolo.

Este trabalho realiza experimentos com o protocolo Sincro, de forma a avaliar os benefícios que a utilização da Rede pode trazer para aplicações distribuídas. São realizadas análises das medidas tomadas sobre 5 aplicações DSM, e são feitas comparações com um protocolo de consistência relaxada baseado em residência. As conclusões mostram o impacto do suporte oferecido pela Rede de Sincronização ao protocolo Sincro, permitindo uma redução no tempo de disseminação de dados de coerência em comparação com o protocolo de referência.

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

NETWORK SUPPORT FOR SOFTWARE DISTRIBUTED SHARED MEMORY

Fabio de Matos Quaresma Gonçalves

September/2007

Advisor: Claudio Luis de Amorim

Department: Systems Engineering and Computer Science

In this work we present the development of an auxiliary communication network for computer clusters, with an architecture optimized for small messages up to a few dozens of bits. The main characteristic of the Synchronization Network is the data dissemination via broadcast which allows the establishment of a total order of the messages transmited.

In order to evaluate the potential of such communication network, we conceived Sincro, a Sw-DSM protocol that implements home-based lazy release consistency. This protocol is evaluated in a cluster of computers with four processors interconnected with Fast Ethernet cards; the cluster uses the Syncronization Network as an auxiliary communication device. Each node of the cluster uses this network to learn about synchronization and coerence events ocurring in all nodes, almost instantaneously. This is the base of a new scheme for the exchange and management of coerence data, used in the Sincro protocol.

Experiments were made to evaluate the benefits of the auxiliary network for distributed applications. Analysis were made from the data colected on the run of 5 DSM aplications, and then compared to another home-based lazy release protocol. The conclusions show the impact of the Syncronization Network support used on Sincro, shrinking the time spent on propagation of coerence data when compared to the reference protocol.

Sumário

Lista de Tabelas

Lista de Figuras

Aplicação Paralela	Dito de um programa executado em um *cluster* de computadores.
Controlador de Memória Primária	Presente nas Placas-Nó, é responsável pela criação de uma FIFO de mensagens na SRAM
HLRC	*Lazy Release Consistency*, ou consistência relaxada baseado em residência.
LRC	*Lazy Release Consistency*, ou consistência relaxada.
Memória Primária	Fila de mensagens da Rede de Sincronização localizada na Placa-Nó
Memória Principal	Memória acessada pela CPU, localizada na Placa-Mãe dos nós processadores.
Memória Secundária	Fila de mensagens da Rede de Sincronização, localizada na Memória Principal
Nó-Folha	Placa da Rede de Sincronização, instalada nos nós de computação
Nó-Raiz	Placa da Rede de Sincronização, presente na raiz da árvore de disseminação
Operação Ordinária	Dita das operações realizadas por um programa que se utilize de protocolos DSM, cujas tarefas se limitem à manipulação de variáveis sem o envolvimento de operações de sincronização.
Operação de Sincronização	Dita das operações realizadas por um programa que se utilize de protocolos DSM, utilizadas para criar mecanismos de acesso a recursos compartilhados no sistema [1]. Podendo ser do tipo barreira ou *lock*.
Programador de Aplicação	Agente responsável pelo desenvolvimento da aplicação paralela.
Protocolo de Comunicação	Se refere ao protocolo empregado pelo sistema *software*, no âmbito do Projeto da Rede de Sincronização, para realizar a troca de mensagens segura entre os Nós de Computação.
Protocolo de Transmissão	Protocolo implementado na camada física da Rede de Sincronização.
Sw-DSM	Dita de uma implementação em *software* de protocolos DSM.
Tempo Sequencial	Tempo de execução de uma aplicação empregando apenas um processador.
Twin	Cópia de uma página distribuida, utilizada para localizar as modificações geradas por um processador, de modo a gerar *diffs*.
Thread Servidora	*Thread* encarregada de receber as mensagens UDP.
Thread Principal	*Thread* encarregada de realizar computação, e operações de sincronização.
Zona de Memória	Região da fila de mensagens da Rede de Sincronização.

Capítulo 1

Introdução

Temos visto na última década o desenvolvimento de programas distribuidos para *clusters* de computadores, que são sistemas onde dois ou mais computadores conectados por uma rede de alta velocidade trabalham de maneira conjunta para executar aplicações. Com o atrativo de atingir um grande poder computacional sem utilizar obrigatoriamente elementos especializados, com *clusters* é possível obter uma relação custo/processamento melhor do que as dos supercomputadores. Ferramentas específicas de *hardware* e *sofware* são escolhidas sob medida para os problemas de interesse, dada a grande diversidade de soluções disponibilizadas comercialmente, como MPI, OpenMP, Myrinet ou Infiniband, para citar alguns.

1.1 Trabalhos anteriores

A partir da década de 70, com o surgimento da tecnologia de multiprocessadores, foram desenvolvidas teorias para o correto desenvolvimento de programas nestas novas plataformas. Para estes ambientes, pela ausência de relógios precisos para determinar a ordem dos eventos, em 1978 Lamport criou o conceito de Relógio Lógico [23]. Posteriormente, com os trabalhos independentes e simultâneos de Fidge [15] e Mattern [24], para estabelecer formalmente as relações de causa e efeito entre eventos originados em diferentes processadores, o Relógio inventado por Lamport evoluiu, sendo criado o conceito de Relógio Vetorial. Propriedades e premissas

também foram formalizadas para a aplicação dos Relógios Vetoriais em problemas de processamento distribuido [10, 30], e outras formas de Relógios Lógicos foram criadas, como o Relógio Matricial [16], e o Relógio de Barreiras e Locks[3].

Na programação dos multicomputadores, uma série de modelos de consistência foi criada, visando oferecer a ilusão de um endereçamento único de memória para todos os processadores de um mesmo sistema. Surgiam os sistemas de memória compartilhada (DSM — *Distributed Shared Memory*) [1]. Com a larga aceitação dos *clusters* de computadores, rapidamente este nicho ganhou suas próprias versões, com o advento de protocolos Sw-DSM.

Porém o desempenho de Sw-DSM é muito sujeito às caracteristicas de latência da rede de interconecção empregada, gerando altos *overheads* nos pontos onde há muita comunicação inter-processos. Os protocolos Sw-DSM foram, com o passar do tempo, evoluindo para os modelos que adiavam a troca de mensagens, como Threadmarks[2], e reduziam o tráfego de dados pela rede, como HLRC [26]. Estes modelos acabam de certa maneira criando métricas importantes do projeto de sistemas Sw-DSM.

Ao mesmo tempo, foram criados mecanismos que visaram minimizar ou esconder estes problemas de modo que o impacto sobre a eficiência do *software* fosse minimizado. Surgiram as redes SAN — *System Area Network* — como Myrinet [8], Infiniband [22] e Memory Channel[18]. De modo geral, inovações visaram eliminar os pontos fracos das redes de comunicação, evitando chamadas de *kernel* — que obrigam a troca de contexto — e utilizando os protocolos tipo *zero copy* — onde mensagens não são copiadas sucessivamente na memória, no caminho entre a aplicação e o *hardware*. Alguns protocolos, como o utilizado no sistema Paragon [29], e VIA (*Virtual Interface Architecture*)[11], criam mecanismos de RDMA (*Remote Direct Memory Access*), que permitem que a memória de processadores remotos seja acessada sem que o processador recipiente da mensagem interfira.

Propostas de redes de comunicação para o problema de sincronização em barreiras surgem, em soluções customizadas com tecnologias TTL [13], FPGA [21], ou com soluções de interconexão já estabelecidas [22, 8].

Recentemente, foi desenvolvida uma rede auxiliar de comunicação em *hardware* para a criação de primitivas de barreira e *locks* na infraestrutura da própria rede [12].

1.2 Motivação

Apesar da relativa abundância de sistemas de sincronização para *clusters*, sua utilização em protocolos Sw-DSM teria pouco impacto na diminuição de seus *overheads*. Isto ocorre porque, mesmo otimizando as primitivas de sincronização, nenhum suporte é oferecido às tarefas secundárias das operações de sincronização. Nestas operações, além da informação de eventos de sincronização, existe um grande *overhead* associado à propagação de eventos ocorridos entre os pontos de sincronização, como *Write Notices* (WNs). Segundo as regras criadas para adaptar o modelo de memória de *clusters* ao modelo de consistência relaxada, sem estas informações, os processadores não podem prosseguir além dos pontos de sincronização. Cria-se uma situação onde nivela-se o tempo por cima, quando uma sincronização extremamente eficiente é insignificante perto do elevado tempo de propagação de dados que se segue, restringindo o potencial de otimização.

1.3 Objetivos

Neste trabalho foi criada a Rede de Sincronização é um *hardware* especializado para a transmissão em *broadcast* de mensagens pequenas com baixa latência e baixo custo computacional. Agregado à Rede de Sincronização encontramos também o Relógio Global [28], capaz de fornecer uma base de tempo única de alta precisão para todos os processadores de um *cluster*.

Ela torna-se uma aliada de protocolos Sw-DSM, onde existe espaço para a otimização das operações de sincronização, que consomem proporcionalmente pouca banda e muito tempo dos processadores. Ainda, com a propriedade de serialização inerente à sua topologia, aliado ao baixo custo de *broadcast* de mensagens, é possível

mesmo reavaliar alguns aspectos de protocolos Sw-DSM, remodelando e adaptando seus modelos de disseminação de mensagem.

Tendo em vista as propriedades da Rede de Sincronização, são analisadas as necessidades de protocolos DSM de consistência relaxada com residência. É proposto um protocolo com este mesmo modelo que, ao propagar as mensagens por esta rede de comunicação, propicia aos seus processadores participantes o conhecimento quase imediato de todos os eventos de consistência e coerência realizados pelo sistema. Com isso, montam-se estruturas de dados com visibilidade global, gerando redução tanto na contenção dos pontos de sincronização do protocolo, quanto no tráfego de dados entre os processadores.

1.4 Contribuições

No decorrer deste trabalho foi implementada a Rede de Sincronização, uma rede de comunicação em *hardware* otimizado para a transmissão de pequenas mensagens com baixa latência, da ordem de poucos microssegundos. Utilizando o *hardware* desenvolvido durante o projeto do Relógio Global [28], é proposto e implementado em *firmware* e *software* um protocolo de comunicação confiável, com garantia de entrega e de integridade das mensagens. A Rede de Sincronização foi feita levando em conta as necessidades de sincronização e coerência de protocolos Sw-DSM, porém seu uso é irrestrito para outras aplicações que necessitem difundir mensagens pequenas em tempo reduzido.

Este trabalho também propõe e implementa o protocolo Sincro, um Sw-DSM de consistência relaxada baseado em residência, que utiliza a Rede de Sincronização como rede auxiliar, tanto para a disseminação das mensagens de sincronização, quanto para outras mensagens diretamente relacionadas a eventos de coerência. Este protocolo é avaliado com aplicações DSM, e são realizadas análises que mostram o impacto da rede auxiliar na redução dos tempos de disseminação de dados de coerência do protocolo.

1.5 Organização

Esta dissertação prossegue com a revisão de sistemas DSM, no Capítulo 2. Em seguida, o Capítulo 3 descreve os componentes de *firmware* e *software* desenvolvidos. Prossegue no Capítulo 4 com experimentos e avaliações da Rede de Sincronização e do protocolo Sincro. Finalmente, no Capítulo 5 são apresentadas as conclusões e trabalhos futuros.

Capítulo 2

Sistemas de Memória Compartilhada Distribuída em *Software*

Este capítulo apresenta os conceitos básicos de sistemas Sw-DSM. Ao longo do capítulo, progressivamente são explicados os modelos e técnicas envolvidos na construção destes sistemas. É mostrado o funcionamento interno de uma implementação, que utiliza o modelo de comunicação em nível de usuário. Em seguida, termina com uma discussão sobre os *overheads* encontrados nesta implementação.

2.1 Sistemas *Software* DSM

De maneira a prover a abstração de memória compartilhada em ambiente de memórias distribuídas, vários Sw-DSMs fazem uso dos mecanismos de proteção de páginas de memória virtual disponíveis na maioria dos sistemas operacionais modernos. Páginas que contém dados que não são válidos localmente são protegidas contra leitura. Isto permite que o sistema intercepte acessos a posições da memória compartilhada inválidas localmente (falha de acesso às páginas) para assim emitir as mensagens necessárias para buscar estes dados em nós remotos (acesso remoto), de maneira a torná-los válidos. Buscando melhorar o desempenho através da minimização do número de acessos remotos, Sw-DSMs replicam, quando conveniente, dados compartilhados nas memórias privativas de diversos processadores.

Esta replicação traz no entanto o problema de coerência de memória, já que há a necessidade de se manter a coerência das diversas cópias do mesmo dado. A manutenção desta coerência deve ser feita segundo um modelo de consistência de memória. Sw-DSMs que usam modelos de consistência relaxada podem amenizar problemas de desempenho atrasando e/ou restringindo as tarefas de comunicação e coerência. Trabalhos como o HLRC [26], que representam o estado da arte, mostram que quanto mais relaxado for o modelo de consistência melhor é o desempenho obtido. A implementação de Sw-DSMs exige ainda a escolha de um protocolo para manutenção da coerência dos dados compartilhados, que pode ser de atualização ou de invalidação. Buscando minimizar os problemas de falso compartilhamento, pode-se também optar pelo emprego de suporte a múltiplos escritores. Neste capítulo serão analisados estes protocolos.

Permeando o modelo de consistência e o protocolo de coerência, se encontram as operações de sincronização, baseadas nas primitivas de sincronização. Estas operações criam e mantém *timestamps*, implementados como relógios vetoriais [24, 15], cuja função é o estabelecimento de relações de ordenação parcial entre eventos distribuidos [10, 30], permitindo que o protocolo garanta a validade dos dados no momento do acesso às páginas.

2.2 Protocolos de Atualização e Invalidação

Para manutenção da coerência dos dados é necessário que escritas a posições da memória compartilhada feitas por um processador sejam propagadas para os demais processadores do sistema. Esta propagação de escritas pode ser feita por protocolos de invalidação ou de atualização. No mecanismo de invalidação, uma modificação em um dado compartilhado feita localmente é vista em um processador remoto através de uma mensagem de invalidação. Ao receber a mensagem, o processador remoto invalida o dado modificado, de maneira que um acesso subsequente a este dado gerará uma falha de acesso, para somente então a versão atual do dado ser buscada. Já no protocolo de atualização os dados não são invalidados, uma vez

que a mensagem que informa que um determinado dado foi modificado já carrega a sua nova versão. Desta maneira, em um próximo acesso a este dado não ocorrerá uma falha de acesso.

Se por um lado o protocolo de atualização minimiza o número de falhas de acesso, por outro ele acarreta uma maior carga de comunicação em relação a protocolos de invalidação. Isto ocorre porque muitas vezes são feitas repetidas atualizações na mesma região de memória, sem que o processador tenha visto todas elas.

2.3 Suporte a Múltiplos Escritores

Devido ao uso comum de unidades de coerência grandes em Sw-DSMs, em geral páginas de memória virtual, estes protocolos podem apresentar problemas de desempenho devido ao falso compartilhamento de páginas. Nesta situação, o protocolo deve permitir que cada página seja acessada por apenas um processador por vez, gerando contenção de tais recursos. Este tipo de problema pode ser minimizado através do uso de protocolos com suporte a múltiplos escritores (em inglês, *multiple writers* — MW), os quais permitem escritas concorrentes à mesma página, postergando a consolidação das atualizações para uma posterior sincronização entre os processadores.

A técnica mais utilizada para implementação de protocolos com múltiplos escritores é o mecanismo de *twinning* e *diffing* [9], que não exige suporte de compiladores. Inicialmente todas as páginas compartilhadas são protegidas contra escrita. Assim, quando um processador tenta atualizar uma página, é gerada uma falha de acesso. O Sw-DSM intercepta esta falha de acesso, faz uma cópia da página (*twin*) e a libera para escrita. Quando se torna necessária a propagação das modificações feitas localmente nesta página, o Sw-DSM faz uma comparação entre o *twin* gerado e a versão modificada da página. As regiões de memória modificadas são utilizados para a criação de um *diff*. De maneira a consolidar modificações feitas por diferentes processadores à mesma página, basta aplicar os diferentes *diffs* à versão local da página.

É importante notar que atualizações feitas por dois processadores concorrentemente à mesma página devem ser em posições de memória diferentes, evitando condições de corrida. Ainda, a ordenação imposta pelas operações de sincronização deve ser preservada quando da aplicação dos *diffs* à página, para evitar que uma atualização recente seja sobreposta por alguma outra atualização mais antiga à mesma posição de memória.

Protocolos Adaptativos

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

2.4 Protocolo *LRC*

A partir de protocolos do tipo *Release Consistency* (RC)[17], originalmente criado para implementação de DSMs em *hardware*, foram derivados protocolos *Lazy Release Consistency* (LRC), mais adaptados às características de Sw-DSM. Os modelos de programação de RC e LRC exigem que programas utilizem explicitamente pontos de sincronização, como as operações de adquirir *lock* (*acquire*), liberar *lock* (*release*) e barreira (modelada como um *release* seguido de *acquire*), para proteger seções críticas. RC garante a consistência de memória de todos os processadores após cada *release*.

Em Sw-DSMs, a implementação denominada LRC atrasa a propagação de mensagens de coerência até o próximo *release*. A consistência da memória é garantida de acordo com a realização de *acquire* pelo processador. Para garantir a coerência de memória, com o atraso das mensagens, LRC usa vetores de *timestamps* para manter a ordem parcial, e estabelecer relações de causalidade entre os eventos de sincronização. Um *timestamp* é um contador lógico dos intervalos locais delimitados pelos eventos de sincronização. Um vetor de *timestamp* é um relógio lógico vetorial, que possui em cada posição o valor do último intervalo que o nó local conhece do respectivo nó remoto.

Conceitualmente, um Relógio Matricial [16] está implementado através do sistema. Ele é formado pela agregação dos N relógios vetoriais de cada um dos N nós do sistema. O acesso ao Relógio Matricial ofereceria aos nós locais informação sobre a troca de dados de sincronização (e consequentemente, de coerência) entre quaisquer 2 nós do sistema.

A Figura 2.1 ilustra as operações de coerência realizadas em LRC, assumindo um protocolo de invalidações. Quando um nó quer adquirir um *lock*, o nó envia seu vetor de *timestamp* junto com o pedido de *lock*. O dono do *lock* usa o vetor de *timestamp* para determinar o conjunto de intervalos ainda não conhecidos pelo nó que está pedindo o *lock*. Junto com o envio do *lock*, o dono envia as notificações de escrita para todas as páginas que foram modificadas em seu passado. Ao receber o *lock*, o nó deve invalidar as páginas de acordo com as notificações recebidas. Uma notificação de escrita, ou WN (*Write Notice*), é uma identificação da página modificada que contém também um *timestamp* associado ao seu respectivo intervalo. O nó precisa invalidar a página para poder gerar, futuramente, uma falha de acesso, que por sua vez permite buscar e ver modificações realizadas pelo nó remoto no intervalo. Após invalidar as páginas, o nó deve atualizar o seu vetor de *timestamp* para indicar quais intervalos do nó remoto já são conhecidos. Desta maneira, o LRC implementa a ordem parcial entre os eventos realizados antes de liberar o *lock*, satisfazendo o modelo de consistência de memória LRC.

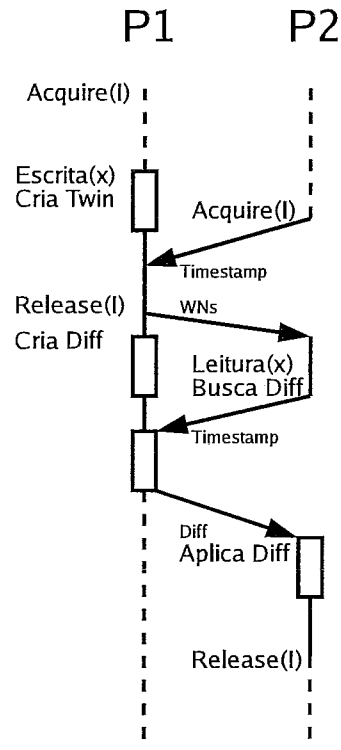


Figura 2.1: A coerência em protocolos LRC.

2.5 Protocolo LRC Baseado em Residência

O modelo LRC permite implementações eficientes de Sw-DSMs. Protocolos LRC com suporte a múltiplo escritor tradicionais demonstraram bons desempenhos para certas classes de aplicações em sistemas com um número pequeno ou médio de processadores[2], mas o custo extra do protocolo se torna substancial em sistemas com um número grande de processadores. O LRC múltiplo escritor tradicional não mantém bom desempenho aumentando o número de processadores porque a necessidade de manter os *diffs* distribuidos por vários nós induz custos extras de comunicação e memória. Um protocolo baseado em residência de página, onde cada página compartilhada possui um processador residência para o qual todos os *diffs* são enviados e de onde todas as cópias atuais são recebidas, pode diminuir os custos e permitir que o protocolo LRC múltiplos escritores continue a ter bom desempenho com o aumento do número de processadores.

O protocolo LRC múltiplos escritores baseado em residência [26] (também chamado *Home-based* LRC, ou simplesmente HLRC), implementa um esquema de múltiplos escritores através da escolha de um processador residência fixo para cada página compartilhada. A idéa basica é descobrir as modificações apenas no final de cada intervalo através das operações de *diffing*, e transferir os *diffs* para a residência da página. Assim, a cópia da página na sua residência será constantemente atualizada e poderá ser utilizada para atualizar as outras cópias nos nós remotos quando necessário. Diferente de LRC, onde os *diffs* de um intervalo são descobertos sob demanda, armazenados localmente no nó e usados possivelmente muitas vezes para atualizar cópias inválidas, em HLRC o tempo de vida dos *diffs* é extremamente curto, tanto nos escritores quanto na residência. Isto alivia o consumo de memória nestes sistemas, não permitindo que cresça com o tempo ou tamanho da entrada da aplicação paralela. Este protocolo permite que o desempenho acompanhe o crescimento do número de processadores: o número de mensagens necessárias para atualizar todas as cópias é linear com o número de processadores e o consumo de memória é constante. Existem versões de HLRC com o uso de mecanismos de troca de mensagens em nível do usuário, e também com interfaces de rede que implementam mapeamento em memória [11, 8, 14]. Isto possibilita que as páginas atualizadas possam ser buscadas nas residências sem precisar fazer cópias extras e *diffs* possam ser aplicados diretamente nos endereços da página em sua residência [26].

A Figura 2.2 mostra como HLRC funciona. O *diff* contendo o novo valor de x é enviado para a residência na liberação do *lock*. O *timestamp* da página contendo x é incrementado e enviado junto com o *diff*. Na residência, o *diff* é aplicado na cópia local da página e o vetor de *timestamp* correspondente é atualizado. Numa falha de página, o nó 1 busca a página atualizada na sua residência. Para obter as modificações correspondentes aos intervalos ainda não vistos, o nó envia junto ao pedido o vetor de *timestamp*. Ao ser comparado ao vetor de *timestamp* da página na sua residência, é certificado que a página já foi atualizada.

Resumindo, as vantagens do HLRC podem ser descritas como: i) diminuir o número de mensagens, pois a falha de página pode ser satisfeita por apenas um

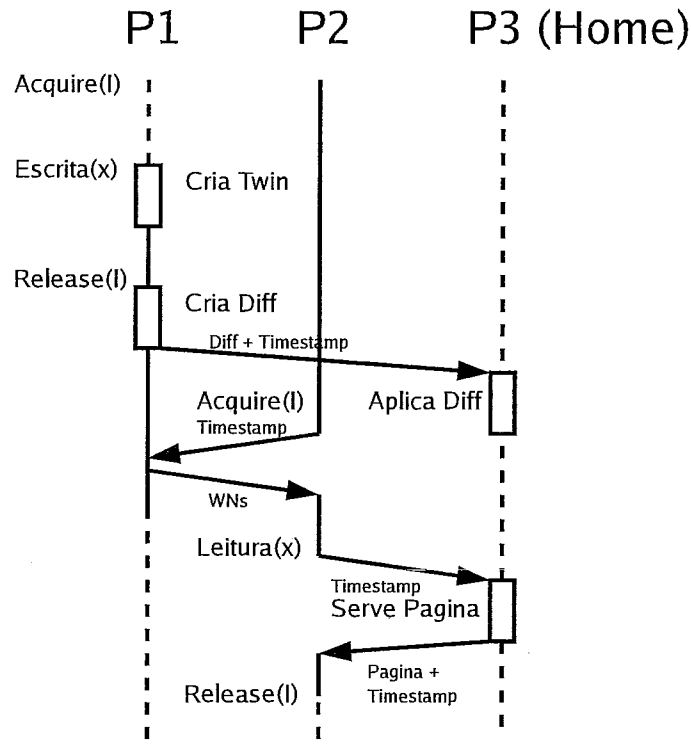


Figura 2.2: A coerência em HLRC.

pedido à residência da página; ii) os *diffs* são aplicados apenas uma vez na residência e não precisam ser armazenados; iii) não é preciso gerar *diffs* para as modificações feitas na residência da página. Note que última característica pode ter um grande efeito no desempenho de aplicações que tenham perfil de único escritor.

As potenciais desvantagens do HLRC são: i) o nó residência precisa ser bem escolhido, e a aplicação não deve mudar o padrão de acesso dinamicamente; ii) comunicação de dados entre dois nós que não são residências requer que o dado seja enviado através do nó residência; iii) potencialmente mais *diffs* são criados, pois ao final do intervalo os *diffs* devem ser gerados e enviados para a residência da página, ao invés de serem criados sob demanda como no protocolo LRC.

2.6 HLRC sobre UDP

O protocolo HLRC de Princeton [26], em sua versão de UDP sobre Ethernet [27], implementa o modelo *multi-threading* de programação em memória compartilhada executando um processo em cada estação de trabalho. Cada processo possui duas *threads*: a Principal, para executar a aplicação e partes do protocolo, e a Servidora, dedicada ao protocolo. As atividades do protocolo ocorrem em vários pontos na execução da aplicação. Os pontos de entrada podem ser divididos em síncronos e assíncronos. Os pontos síncronos são aqueles em que a aplicação chama o protocolo para executar alguma ação, como adquirir e liberar *locks*, atingir barreiras, e falhas de acesso a páginas. Nos pontos síncronos são geradas mensagens de pedidos que devem ser servidos por um processador remoto. HLRC provê pontos de entrada assíncronos para receber estes pedidos, com a *Thread* Servidora. Todas as mensagens de pedido do protocolo chegam em uma porta UDP, onde esta *thread* se encontra bloqueada aguardando eventos.

2.6.1 Modelo de Comunicação

HLRC pode gerar pedidos remotos para dados e sincronização. Estes pedidos, que requerem resposta, são enviados usando o modelo Transmissão-Recepção. Como cada nó executa apenas uma *thread* de computação, pode existir apenas um pedido pendente para aquele nó, e uma resposta correspondente. Daí, cada nó espera apenas $N - 1$ pedidos remotos simultâneos, onde N é o número de nós do sistema.

As mensagens que não necessitam de respostas, como de chegada em barreira, *diffs*, e as próprias mensagens de resposta, são enviadas utilizando um modelo de comunicação RDMA. Ao chegar na *Thread* Servidora, o pacote UDP é processado e os dados são copiados para a área de memória da aplicação, conforme as informações de *offset* e tamanho de pacote, que geram a máscara de endereços onde a mensagem é copiada.

2.6.2 Operações de Sincronização

Locks

Quando um processador entra em uma seção crítica, ele deve adquirir um *lock*. Segundo o protocolo de coerência, a aquisição de um *lock* implica que ele deve incorporar informações de seu passado. Por outro lado, a saída de uma seção crítica causa a liberação de um *lock*, implicando a disseminação de seu passado a alguém que esteja no futuro.

Inspirado nas operações de *lock*, o protocolo de coerência criou os conceitos de *acquire* e *release*, significando respectivamente disseminação e incorporação de informações de coerência.

Para a implementação de *locks*, é criado o conceito do gerente de *locks* G_L. Quando um processador P_2 está por entrar em uma seção crítica e necessita de um *lock*, ele transmite uma mensagem com seu *timestamp* a G_L. Este mantém um apontador para o último processador P_1 que pediu o *lock*, e encaminha a ele o pedido. Se o recurso ainda não estiver disponível, o pedido é enfileirado para ser servido futuramente.

Quando o *lock* está disponível, P_1, antigo detentor do recurso, envia para o novo dono P_2 uma mensagem contendo listas de WNs que ele ainda não recebeu. Estas listas são geradas em P_1 utilizando o *timestamp* fornecido por P_2. Finalmente, quando a mensagem de *lock* chega em P_2, este incorpora os WNs recebidos, invalidando as respectivas páginas. Este comportamente é ilustrado na Figura 2.3.

Para cada *lock*, o gerente é estabelecido estaticamente em um esquema *round robin*.

Barreiras

A operação de barreira é constituida por duas primitivas de barreira, para informar os pontos de chegada e saida da operação, e deve realizar a disseminação de WNs. Segundo o protocolo de coerência, ela é modelada como um *release* seguido de *acquire*, pois deve disseminar e depois receber informações de WNs.

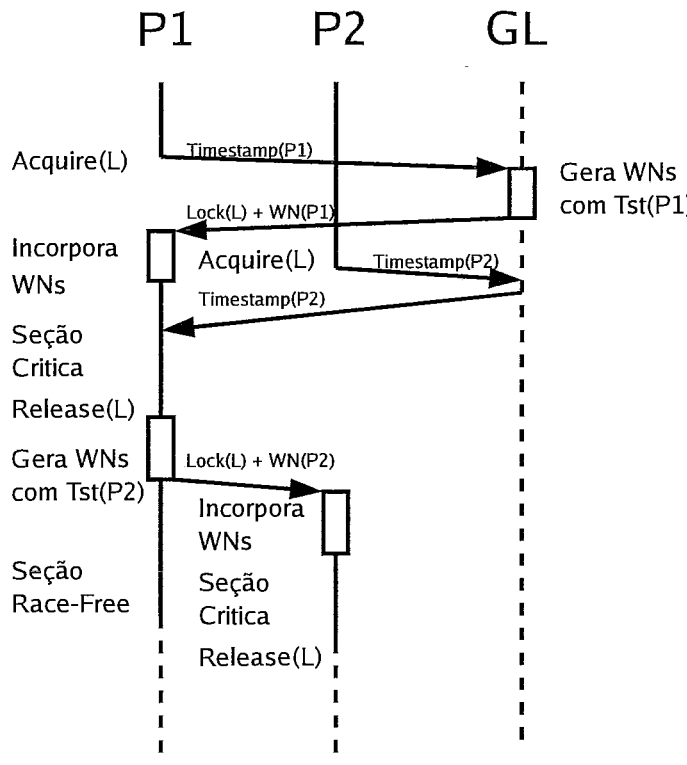


Figura 2.3: Algoritmo de *locks* com gerente.

Em uma sincronização por barreiras, cada processador deve tomar conhecimento de todas as modificações feitas pelos demais processadores. Ao chegar a uma barreira, cada processador envia a todos os outros uma mensagem de chegada em barreira. Em seguida uma mensagem de coerência é disseminada contendo seu *timestamp* e todos os WNs gerados localmente que não foram vistos pelos outros nós.

A cada mensagem de coerência recebida, o processador incorpora os WNs nela contidos, invalidando as respectivas páginas, até receber as mensagens de todos os processadores. Em seguida, o processador transmite a todos os outros uma mensagem de fim de barreira, para informar o fim da operação de barreira. Quando o processador recebe esta segunda mensagem de todos os processadores, ele pode sair da operação de barreira, voltando a realizar o código da aplicação.

2.7 *Overheads*

Como visto anteriormente em protocolos LRC, os pontos de sincronização do programa são os momentos onde os processadores realizam trocas de informações de coerência e dados, relativas às operações de escrita realizadas no seu último intervalo. Assim, as operações de sincronização passam a utilizar, além das primitivas de sincronização, outras tarefas indiretamente associadas à sincronização de programa.

Como os WNs podem ser gerados até o momento da sincronização, elas são agregadas em listas, que podem crescer até o momento da sincronização. Sendo transmitidas através de uma rede otimizada para banda, existe uma grande vantagem em realizar a transmissão em apenas um pacote por processador de destino. Por isso, a disseminação de WNs é realizada dentro das operações de *release*.

Em protocolos baseados em residência, este é também o momento em que são calculados e transmitidos os *diffs* das páginas.

A única tarefa que intrinsecamente faz parte das operações de *acquire* é a incorporação dos WNs gerados pelos outros nós. Porém, para obter esta informação, é necessário aguardar a conclusão da operação de *release* em outros processadores, para que então ela seja transmitida através da rede, passando pelas várias camadas de *kernel* em ambos processadores, chegando finalmente ao processo.

Em relação à primitiva de *locks* com algoritmo de gerente, mesmo que o *lock* esteja disponível existe um custo mínimo de 2 mensagens, para contactar o gerente e o último detentor, gerando um tempo de espera mínimo mesmo em aplicações onde não há contenção. O mesmo sintoma ocorre com as operações de barreira, quando são utilizados 2 ciclos de *broadcast* para uma única operação de sincronização.

Assim podemos perceber que o tempo de barreira e *locks*, que inicialmente era intrinsecamente uma simples tarefa de sincronização, se transforma em um local de crescente custo computacional. Desbalanceamentos causados pelas tarefas adicionais são sentidos pelos outros processadores envolvidos na sincronização, que devem aguardar o tempo necessário para que chegue a informação de coerência, sem a qual não podem entrar na seção crítica.

Capítulo 3

Trabalho Desenvolvido

Neste capítulo serão demonstrados os componentes de *hardware*, *firmware* e *software* desenvolvidos ao longo deste trabalho. Inicialmente é descrita a arquitetura da Rede de Sincronização. O capítulo prossegue com os mecanismos encontrados no protocolo Sincro, um Sw-DSM de consistência relaxada baseado em residência. No final do capítulo, são descritos alguns trabalhos de suporte de *hardware* para sincronização e para Sw-DSM, relacionados com o trabalho desenvolvido.

Para estudar o sistema como um todo, é proposta uma divisão interna no formato de uma pilha de protocolos, estruturada hierarquicamente conforme são expostas as primitivas. Como observamos na Figura 3.1, em primeiro lugar vem a camada de aplicação, onde o programador da aplicação paralela manipula variáveis e ponteiros, realizando operações lógicas e matemáticas para a geração de resultados. Ele também deve utilizar as operações de sincronização, para criar seções críticas e distribuir resultados através dos nós do *cluster*, conforme as regras de *Lazy Release Consistency* (vide Seção 2.4).

A seguir vem a camada de memória distribuida compartilhada (DSM — *Distributed Shared Memory*), que dá ao programador a ilusão de estar trabalhando com uma área de memória unificada. A partir daí o protocolo se divide em dois ramos, sendo o primeiro responsável pelas operações de sincronização, e o segundo pelas chamadas operações ordinárias [1]. A camada de DSM emprega as primitivas oferecidas por ambos os ramos para garantir a consistência da memória de acordo

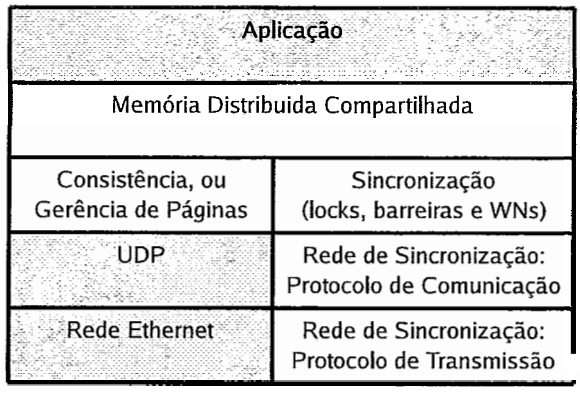


Figura 3.1: Pilha de Protocolos do Sw-DSM.

com as regras de LRC, como esperado pelo programador de aplicação.

Seguindo o ramo de operações ordinárias, encontramos a camada de consistência, onde são implementadas as tarefas relacionadas com a gerência de memória, tais como serviço, atualização e invalidação de páginas. No caso do protocolo DSM proposto, esta camada se apóia no protocolo UDP/IP para troca de mensagens entre processadores, que por sua vez utiliza placas de rede comerciais *Fast Ethernet*.

Continuando pela camada de operações de sincronização, encontraremos o Protocolo de Sincronização, proposto especialmente para a Rede de Sincronização. Esta camada implementa e oferece à camada DSM as primitivas de barreiras e *locks*. Ainda, como suporte à camada de consistência, e para otimizar e explorar melhor as caracteristicas oferecidas pela Rede, ela é utilizada também para a propagação de WNs. Adicionalmente, a sinalização de residência de páginas é propagada através desta camada.

Seguindo por este ramo da pilha entramos na camada de *software* da Rede de Sincronização, que implementa o Protocolo de Comunicação. Este é responsável pelas camadas de *Link* de Dados, Rede, e Transporte (camadas 2, 3 e 4, segundo o modelo OSI). Finalmente chegamos ao *firmware* e *hardware*, representando a camada física da Rede de Sincronização.

A Figura 3.1 ilustra em destaque as camadas desenvolvidas no escopo deste trabalho: os Protocolos de Transmissão, Comunicação, Sincronização, Consistência

e DSM.

3.1 Rede de Sincronização

A infraestrutura da rede é implementada em uma topologia em forma de árvore, com 1 Placa-Folha em cada nó, e 1 Placa-Raiz adicional à estrutura, que atua como roteador da Rede de Sincronização. Cada Placa-Folha conta com um Relógio Global de alta resolução [28], e também com a lógica necessária para a transmissão e recepção de mensagens. Nesta hierarquia, as mensagens são produzidas por quaisquer folhas, e sempre são transmitidas por difusão para todas as folhas. Estas são as produtoras e consumidoras finais de toda e qualquer mensagem que trafega pela Rede. Os pacotes são disseminados sincronamente, o que garante que o tempo de chegada das mensagens seja associado a um tempo global idêntico em todos os nós.

Através da Placa-Raiz, que centraliza e ordena todas as mensagens no momento em que são transmitidas, ocorre a serialização de todas as mensagens transmitidas pela Rede. Com isso, é criada uma ordem total, e as mensagens são recebidas pelas Placas-Folha em uma fila única, com garantia de que todos os processadores percebem a mesma sequência de mensagens. Esta propriedade pode ser utilizada pelo protocolo DSM para criar mecanismos determinísticos de tomada de decisões distribuida, mesmo sob condições de corrida.

O cabeçalho das mensagens oferece um campo de 4 bits, utilizado como identificador de Tipo de Mensagem, e um campo de 48 bits, que registra o valor do Relógio Global no momento em que a mensagem chega nos nós. Contém também campos dedicados para identificador de placa de origem, e um contador de mensagens, que é incrementado para cada mensagem transmitida no nó. O tamanho da carga de dados (*payload*) é fixo para cada Tipo, e varia entre 0, 8, 16, 32 e 48 bits. Para garantir a integridade dos dados, é empregado cálculo de CRC de 16 bits.

Através de um *device driver*, a placa é mapeada pelo sistema operacional na memória, que é liberada para uso em contexto de usuário. Assim, operações de leitura e escrita na placa são realizadas sem a troca de contextos. Também é possível

realizar a trasmissão do tipo *zero-copy*, onde não é necessário realizar cópias extras dos dados em *buffers* intermediários.

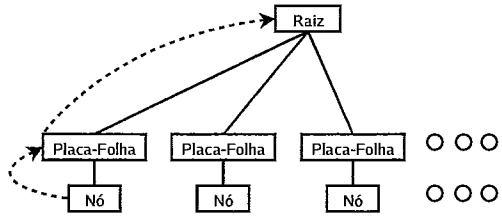
3.1.1 Protocolo de Transmissão

A rede é interconectada com cabos de rede CAT5 de 4 vias blindados, utilizando a sinalização LVDS. A Placa-Raiz emprega uma base de tempo de freqüência programável (PLL — *Phase Locked Loop*), gerando o sinal do relógio global, que é distribuido para as Placas-Folha através de um par trançado dedicado. Nas Placas-Folha, são implementados contadores, que são incrementados a cada flanco positivo do relógio global [28].

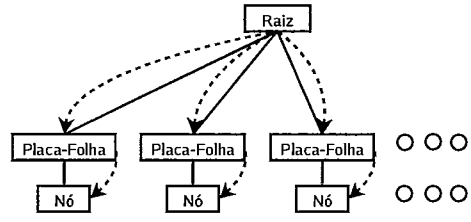
Para transmitir mensagens, são utilizados os 3 pares trançados remanescentes. Estes assumem as funções de *Uplink* — comandado pelas Placas-Folha — *Downlink* e Controle de Fluxo — comandados pela Placa-Raiz. Finalmente, para estabelecer um nível de referência elétrica entre os circuitos, o fio terra do cabo CAT5 é conectado apropriadamente nas placas transceptoras.

Transmissão na Placa-Folha

Na Figura 3.2 é ilustrada a transmissão pela Rede de Sincronização. Ela começa com a CPU copiando os dados a serem transmitidos desde a memória principal do nó de computação, para um registrador específico da Placa-Folha. Ali o pacote é montado, obtendo o Tipo de Mensagem a partir da decodificação do endereço de escrita, e utilizando o valor escrito como carga de dados (*payload*). O identificador de origem é copiado de um registrador contido na placa, o valor do contador de mensagens é incrementado e copiado para o pacote, e o CRC de 16 bits do pacote é calculado e agregado. Então a máquina de estados responsável pela transmissão de pacotes coloca o *Uplink* em valor '1'. Ela examina o sinal de Controle de Fluxo, comandado pela Placa-Raiz, e quando este assume o valor '1', a máquina de estados gera os *start bits*, iniciando a propagação do pacote para a Placa-Raiz.



(a) Propagação até a Placa-Raiz



(b) Propagação até as Placas-Folha

Figura 3.2: Propagação de mensagens: atingindo a Raiz.

Transcepção na Placa-Raiz

O papel da Placa-Raiz na recepção e transmissão de dados é ilustrado na Figura 3.3. Como se observa, a Placa-Raiz verifica o barramento de *Uplinks*, provenientes das Placas-Folha, e ao perceber a transição de '0' para '1' em algum bit, seu controlador passa do estado ocioso para o estado de recepção. Neste estado, o bit que realizou a transição é utilizado para identificar qual das Placas-Folha está apta a transmitir pacotes.

O controlador realiza a transição do barramento de Controle de Fluxo no bit correspondente à Placa-Folha selecionada, o que libera a sua máquina de estados de transmissão. Este bit alimenta também um multiplexador, selecionando o bit de *Uplink* da placa correspondente. Este é copiado para o barramento de *Downlink*, realizando a transmissão de pacotes em *wormhole*. A transcepção do pacote é monitorada por um bloco receptor, que indica ao controlador o fim da transmissão de cada pacote, fazendo com que este retorne ao estado ocioso.

Para garantir ausência de *starvation*, quando ocorrem transições em outros

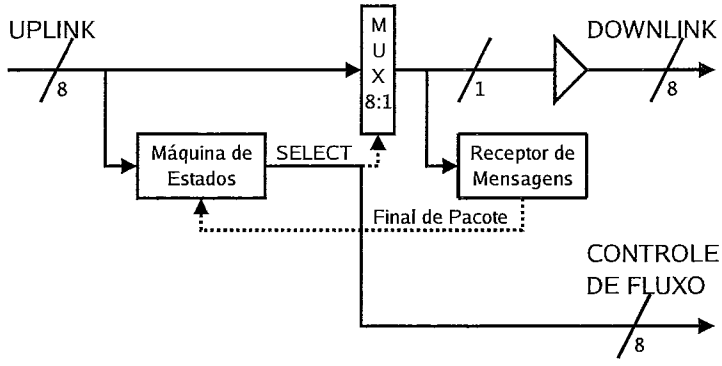


Figura 3.3: Propagação de mensagens na Raiz.

bits do barramento de *Uplink* durante a recepção, estes são processados e armazenados pelo controlador. Sempre que está no estado ocioso, uma consulta é realizada, afim de localizar placas que estejam prontas para transmitir pacotes, desencadeando o mecanismo descrito acima.

O mecanismo de prioridade, que deve garantir ausência de *starvation* na rede, é constituido pelo detector de flanco positivo D1, pelos registradores R1 e R2, um decodificador de prioridade D2, um Mux M1 e um receptor de mensagens M2. Na Figura 3.3 são ilustrados tais componentes, e o caminho de dados é representado por linhas cheias, enquanto os sinais de controle são indicados por linhas tracejadas. Quando é detectado um flanco positivo em quaisquer dos pinos de *uplink*, é gravado seu bit correspondente no registrador R1. A cada ciclo de relógio, é avaliado o valor de R2, e se for igual a zero, então o valor de R1 é copiado para R2, e em seguida R1 é zerado. Então D2 escolhe o bit ativo mais significativo de R2, ativando o pino de sinalização que libera a transmissão da folha correspondente, e ativando também a entrada de M1. A partir dai é iniciada a transmissão de mensagem na folha escolhida, recebida por M2. Quando este detecta o fim da mensagem, transmite um pulso para D2, que zera em R2 o bit escolhido.

Recepção na Placa-Folha

Nas Placas-Folha, o bloco receptor acompanha os sinais que chegam pelo *Downlink*, montando assim os pacotes. Após esta etapa, a mensagem muda de formato. No cabeçalho, os bits de CRC são transformados em um *flag*, que indica falha ou sucesso, que será processado pelo Protocolo de Comunicação. O campo de tamanho de mensagem é dispensado, e é adicionado um campo de 48 bits com a contagem do Relógio Global no instante do recebimento. Finalmente, as mensagens são alinhadas para um tamanho de 128 bits, capaz de conter cabeçalho e dados da maior mensagem que a rede é capaz de transmitir.

Após esta reformatação, a mensagem é copiada para uma fila temporária no *firmware*. Então, as mensagens são copiadas e armazenadas na SRAM presente na Placa-Folha. Ali, elas são organizadas de modo a formar uma única fila de mensagens para todos os pacotes, independente do nó de origem, numa área denominada como Memória Primária.

3.1.2 Protocolo de Comunicação

O Protocolo de Comunicação é a camada de *Software* encarregada de copiar as mensagens que chegam na Memória Primária, para a memória principal do computador, organizando uma fila de mensagens denominada Memória Secundária.

Mecanismo

No momento em que a aplicação paralela se utiliza de uma operação de sincronização, tal como aquisição de *lock* ou chegada em barreira, o Protocolo de Sincronização necessita coletar e processar mensagens de sincronização. É iniciada uma reação em cadeia envolvendo várias camadas da Pilha de Protocolos, como pode ser visualizada na Figura 3.4. A camada de aplicação necessita que chegue a liberação de sincronização, e ativa a camada de sincronização. Esta se comunica com a camada de comunicação, processando uma a uma as mensagens presentes na Memória Secundária que ainda não foram lidas. Caso a Memória Secundária esteja

vazia, a camada de comunicação é ativada e realiza a cópia de mensagens entre a Memória Primária e Secundária.

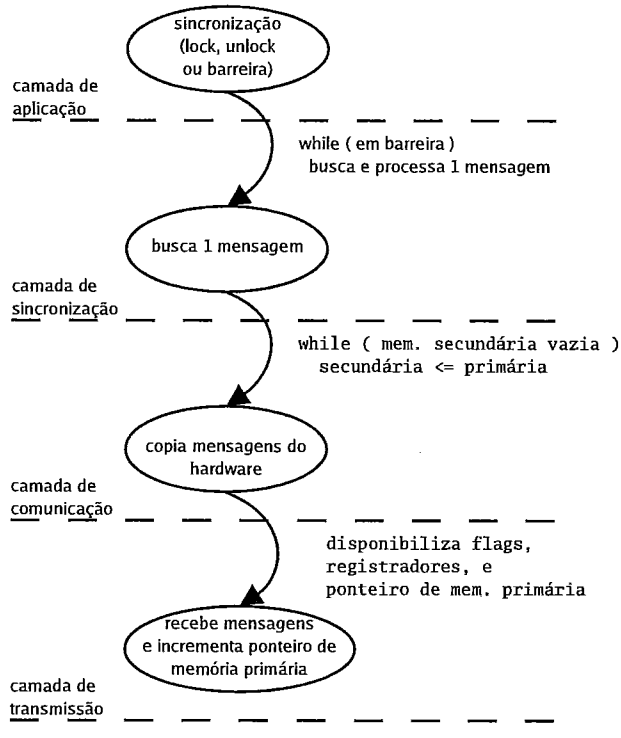


Figura 3.4: Reação em cadeia de busca de mensagens.

Tratamento de Erro

Depois disso é realizada a conferência de integridade das mensagens copiadas. Vale lembrar que esta conferência é uma medida complementar às tomadas pelo *firmware*, descritas na Seção 3.1.1. Cabe notar que sempre que possível as conferências e decisões são tomadas pelo Protocolo de Comunicação, que por ser implementado em *software*, permite algoritmos mais complexos sem onerar o *firmware* em prejuizo de suas funções elementares. Vários procedimentos são empregados, sendo os mais importantes a observação do bit de CRC, o tempo global (palavra de 48 bits), que deve ser sempre maior que a mensagem anterior, e os contadores de mensagens, que são incrementados individualmente de acordo com o processador que transmite a mensagem. As duas primeiras informações são gravadas pelo *hard-*

ware no momento em que chegam à placa, e a terceira é agregada ao cabeçalho da mensagem no momento em que é iniciada a transmissão na placa.

Como todas as mensagens são transmitidas em *broadcast*, o Nó que transmitiu a mensagem compara a mensagem recebida com uma cópia realizada no momento da transmissão.

Ainda, uma mensagem de segurança adicional é transmitida periodicamente, cujo *payload* é um CRC de 16 bits, inicializado no inicio do programa e que acumula o *payload* de todas as mensagens transmitidas pelo Nó. A rotina de recepção calcula também o CRC acumulado das mensagens que recebe, e na chegada de uma mensagem de segurança, ambos os valores são comparados.

Quando qualquer dos mecanismos mencionados detecta discordência entre os valores observado e esperado, o processo pára e transmite pela Rede de Sincronização uma mensagem de ABORT, o que termina com todos os processos do programa, gerando mensagens e arquivos para sua depuração.

Transmissão de Mensagens

Para que uma mensagem seja transmitida por um Nó, o Protocolo de Comunicação deve cumprir uma série de tarefas, para colocar a Placa-Folha em um estado próprio para a transmissão segura da mensagem desejada. Resumidamente, os passos são: verificação de que a fila de saida está vazia, controle de fluxo para gerenciamento de uso das zonas de memória (visto na próxima subseção), verificação de chegada da última mensagem transmitida, e finalmente, transmissão da mensagem.

Como a fila de saida possui espaço para apenas 1 mensagem, e a transmissão é assincrona, é necessário verificar se a última transmissão pedida já terminou. O Nó lê o registrador de *status* do *firmware* e verifica se o bloco transmissor ou o *loopback* interno da placa se encontram em estado ocupado. Se isto ocorrer, o Protocolo de Comunicação executa *polling* nestes *flags*. Quando todas as condições forem favoráveis, o Nó estará livre para a próxima etapa.

O segundo passo da transmissão de mensagens de sincronização envolve o

controle de fluxo com respeito à utilização das memórias Primária e Secundária. Este é necessário para evitar que as mensagens presentes na Memória Primária do sistema sejam sobrescritas antes de serem lidas, e que as mensagens da Memória Secundária sejam sobrescritas antes de serem processadas. Como a arquitetura da Rede de Sincronização implementa mensagens em *broadcast*, significa que a qualquer momento, as Memórias Primárias de todas as placas possuem conteúdos idênticos (salvo quando as FIFOs de saida contém mensagens). Por isso, é necessário implementar um controle de fluxo global, que tenha visibilidade tanto dos ponteiros de escrita, quanto dos de leitura, para todos os nós simultaneamente. Este mecanismo está descrito na próxima subseção.

Gerenciamento de Filas

De forma a suportar um número ilimitado de mensagens com o modelo de coleta de mensagens empregado, e contando com um tamanho finito de memória na Placa-Folha, esta camada implementa um esquema de sinalização entre placas capaz de gerenciar conjuntamente as memórias Primária e Secundária.

Considere a denominação de 3 ponteiros de memória: i) escrita, referente à última posição de Memória Primária que contém uma mensagem válida (p_e); ii) leitura, referente à ultima posição de Memória Secundária que contém uma mensagem válida (p_l); e iii) processamento, referente à última posição de memória que contém uma mensagem processada pelo Protocolo de Sincronização (p_p). Pela restrição de tamanho de memória SRAM disponivel nas Placas-Folha, cada ponteiro tem um contador auxiliar de *wraps*, para que se acompanhe quantas vezes ele chegou até o final da memória e retornou à primeira posição. A associação dos ponteiros e *wraps* dá origem às variáveis P_e,P_l e P_p.

Por simplificação, o tamanho de Memória Secundária é igual ao de Memória Primária, facilitando a aritmética com ponteiros. Assim, em qualquer momento $P_e \geq P_l \geq P_p$, e particularmente $P_e = P_l = P_p$ quando todas as mensagens disponíveis foram processadas.

Como a arquitetura da Rede de Sincronização implementa mensagens em

broadcast, significa que, a qualquer momento, as Memórias Primárias de todas as placas possuem conteúdos idênticos (salvo quando a fila de *firmware* contém mensagens). Por isso, é necessário implementar um controle de fluxo global, que tenha visibilidade tanto dos ponteiros de escrita, quanto dos de leitura, para todos os nós simultaneamente.

A solução encontrada é a divisão da memória em 8 zonas de igual tamanho. Seja M o tamanho da memória, P_e o número de mensagens recebidas, e P_l o número de mensagens copiadas para a Memória Secundária. Assim calculamos as projeções na memória $p_e = resto(P_e/M)$, e $p_l = resto(P_l/M)$. Considere também o operador Z que calcula a zona de memória que um ponteiro ocupa.

A idea geral é proibir que $P_e > P_l + M$, a todo momento para qualquer nó, e a estratégia é inibir a transmissão quando são conjugados dois eventos: tais ponteiros ocupem a mesma zona de memória e a escrita (p_e) esteja por alcançar a leitura (p_l). Formalmente, $Z(p_e) = Z(p_l) | p_e < p_l$. Então é criado o conceito de zonas livres e ocupadas, por nó e por programa. Um nó tem uma zona livre se esta zona tem mensagens desde sua primeira até sua última posição, e a zona foi inteiramente copiada para a memória secundária. Um programa tem uma zona livre se todos os nós tem esta zona livre. Como regra geral, para o caso $p_e < p_l$ fica estabelecido que não podem ser realizadas transmissões se houver risco de que uma nova mensagem transmitida seja gravada em $Z(p_l)$.

Um nó, antes mesmo de transmitir uma nova mensagem, tem uma boa estimativa de onde esta estará gravada na Memória Primária após a transmissão. Para isto é necessário considerar os fatores controlados, que são: o atual valor de p_e, obtido diretamente da placa, o número de nós participantes do programa, e uma eventual taxa de retransmissão. Considerando que a raiz da Rede de Sincronização implementa uma arbitragem de prioridade com algum grau de *fairness*, se um dado nó deseja transmitir uma mensagem, sabe-se que cada nó pode transmitir no máximo 1 mensagem antes dele próprio, sujeito a um percentual de retransmissão $r \ll 1$. Ainda deve ser reservado um espaço para a transmissão de mensagens de controle do próprio algoritmo, como será visto a seguir. Desta maneira mostra-se

necessária a criação de uma área de segurança, decorrente da imprecisão da posição em que estará gravada uma nova mensagem. Por simplicidade, o tamanho da área de segurança é definido como tendo o mesmo tamanho de uma zona, que se mostra 3 ordens de grandeza maior que o necessário para esta implementação.

Assim, a regra para controle de fluxo passa a ser descrita da seguinte forma:

$$\text{Mensagens são inibidas quando } Z(p_e) + 1 = Z(p_l).$$

Esta condição é garantida pela verificação de zonas de programa liberadas, durante a transmissão de mensagens. Esta informação é construida através de um esquema para a disseminação das mensagens de liberação de zonas de memória. Estas mensagens são transmitidas pela rotina de cópia de mensagens entre as memórias Principal e Secundária, quando o Protocolo de Comunicação detecta que o ponteiro de leitura trocou de zona, o que significa que o processador liberou tal zona.

3.2 Protocolo de Sincronização

Uma característica especial da Rede de Sincronização vem de sua topologia em árvore. Pela necessidade de serialização de mensagens, é oferecida como vantagem a criação de uma ordem total de todas as mensagens transmitidas pela Rede. Esta propriedade é explorada para a implementação de primitivas que, mesmo em condições de corrida, possibilita o cômputo do mesmo resultado para todos os participantes da Rede, através de um critério visto globalmente: a ordem de pedido dos recursos.

Uma outra vantagem da Rede de Sincronização é a disseminação de informações em *broadcast*. Com o modelo de comunicação presente na Rede, são construídas primitivas de sincronização que permitem que qualquer nó tenha conhecimento das ações tomadas por todos os nós, a qualquer instante, sem necessidade de buscar informações remotas.

Desta forma, pela adoção de um suporte à disseminação em *broadcast*, é possivel antecipar a troca de informações de sincronização necessárias na abertura de novos intervalos. Com isto, é reduzido o tempo de bloqueio do protocolo.

A seguir são descritos os mecanismos básicos que envolvem as mensagens que trafegam pela Rede de Sincronização.

3.2.1 Primitivas de Sincronização

Locks

A primitiva de *locks* foi implementado utilizando algoritmo de *tickets* [25], explorando a caracteristica de serialização de mensagens oferecida pela Rede de Sincronização para substituir as operações atômicas. Estas são primitivas comumente empregadas em multiprocessadores para a criação de mecanismos de sincronização.

Inicialmente, o nó N que deseja obter um *lock* transmite uma mensagem de *ticket lock* que viaja pela rede atingindo todas as folhas. Neste momento, as folhas incrementam o contador de *locks* pedidos, e o nó N anota o número atual, que passa a ser seu número de *ticket*. Este número é comparado com um outro contador, de liberação de *lock*. Então o algoritmo realiza um laço, alternando entre a comparação de valores e o processamento de mensagens da Memória Secundária (vide Figura 3.4). Quando finalmente o detentor do *lock* o libera, emite uma mensagem, que é recebida e processada por todos os nós, incrementando o contador apropriado. Percebendo o novo valor, o nó N adquire o *lock* e causa a saida do laço de processamento de mensagens.

O nó N prossegue normalmente, entrando na seção crítica. Quando deseja sair, ele transmite uma mensagem de *release*, que faz com que o nó detentor do próximo *ticket* passe a possuir o *lock*.

Como pode ser facilmente demonstrado, o esquema de *ticket locks* oferece garantia de *fairness*, na medida em que concede o *lock* segundo a ordem em que são feitos os pedidos. Para implementar este algoritmo para um conjunto C de locks, basta implementar vetores de contadores de *ticket* e de liberação de tamanho C.

Este algoritmo é vantajoso em relação ao implementado comumente em protocolos Sw-DSM como o HLRC, eliminando a necessidade de um gerente de *locks*. Com isso, um *lock* que esteja disponível pode ser obtido mais rapidamente, sem a

necessidade de um mecanismo de pergunta e resposta.

Como se vê, a viabilidade de implementação deste algoritmo está baseada na garantia de ordem de chegada de mensagens idêntica para todos os participantes da rede. Uma vez que esta caracteristica está garantida pela Rede de Sincronização, o algoritmo é então totalmente implementado em *software*.

Barreiras

A implementação de barreiras utilizando a Rede de Sincronização emprega uma variável que conta o número de barreiras liberadas, e um vetor que registra a chegada individual dos nós [25].

No inicio do algoritmo, o nó anota a variável de barreiras liberadas, e em seguida transmite pela rede a sua mensagem de chegada à barreira. Depois disso, é realizado um laço que aguarda o incremento da variável de barreiras liberadas. Caso o valor não seja o esperado, é chamada a rotina que processa mensagens da Memória Secundária. Este *loop* ocorre até que o contador de barreiras liberadas atinja o valor esperado, como acontece com o algoritmo de *locks*.

O Protocolo de Sincronização, implementado em *software*, processa todas as mensagens da Memória Secundária, preenchendo o vetor de barreira conforme são encontradas as mensagens de barreira dos processadores. Quando percebe o vetor cheio, o Protocolo de Sincronização incrementa o contador de barreiras liberadas, e limpa o vetor de barreira.

3.3 Sw-DSM

Para avaliar o impacto do suporte de *hardware* no cenário de Sw-DSM, é proposta a implementação do protocolo Sincro. Ele emprega o modelo de consistência relaxada com residência (*Home-Based Lazy Release Consistency*), e utiliza um protocolo de invalidação, com transmissão de páginas inteiras e geração de *diffs* para atualização das páginas. Para controle de versão de páginas os *timestamps* utilizados são Relógios Vetoriais.

A camada de consistência é a responsável pela administração das páginas distribuidas, e utiliza protocolos de rede comerciais, especificamente UDP/IP em uma infraestrutura de *Fast Ethernet*. A Rede de Sincronização é empregada como uma rede auxiliar, utilizada para implementar primitivas de sincronização e oferecer suporte às operações de coerência, conforme descrito a seguir.

3.3.1 Relógios Lógicos no DSM

Para a criação de *timestamps* com o uso de Relógios Lógicos Vetoriais, protocolos DSM interceptam as mensagens de sincronização. Eventos gerados internamente incrementam diretamente o vetor, enquanto que eventos gerados em outros nós devem carregar consigo vetores de *timestamp*, que devem ser incorporados ao *timestamp* local.

Com a utilização da Rede de Sincronização, e a consequente disponibilização de todas as mensagens de sincronização para todos os nós, é possível interceptar qualquer mensagem de sincronização e com isso rastrear os eventos de qualquer processador. A conseqüência direta disso é que um determinado nó passa a poder computar o tempo lógico de qualquer outro, a qualquer instante. Sem qualquer custo adicional, podem ser montados Relógios Lógicos Matriciais, constituidos pelo agrupamento dos Relógios Vetoriais de todos os nós. A conseqüência indireta é que as mensagens de sincronização não precisam mais carregar consigo os vetores de *timestamp*, pois este passa ser suprido pelo Relógio Matricial. Isto é fundamental para que estas mensagens possam ser transmitidas pela Rede de Sincronização, pois acaba por reduzir o tamanho de suas mensagens para um tamanho fixo, que carrega apenas informação de sincronização e processador. Em contraste, em outros sistemas as mensagens assumem complexidade de tamanho $O(p)$, linear com o número p de processadores participantes.

3.3.2 Coerência

Uma caracteristica dos protocolos Lazy DSM é a divisão do tempo em intervalos, onde cada intervalo contém WNs e é delimitado por operações de sincronização (*locks* e barreiras). As operações de sincronização passam a estar atreladas à disseminação de WNs, surgindo um gargalo associado à latência de rede.

Aproveitando a disseminação de mensagens pela Rede de Sincronização, as mensagens de falha de página também são transmitidas por ela a todos os processadores, que têm acesso a esta informação momentos após o evento que as produz. Estas informações são então agrupadas nos seus respectivos intervalos. Quando um nó abre um novo intervalo, ele precisa "incorporar" os intervalos anteriores ao novo. Para isso, são necessárias as informações dos WN produzidos pelos nós que fazem parte de seu passado transitivo, disponíveis localmente para o processador. Pelo fato de tais informações já serem acessíveis, é reduzido um tempo de espera importante no protocolo DSM, que é o momento em que uma operação de sincronização deve aguardar a chegada do vetor de WNs.

Revisão

Para implementar este algoritmo, primeiro imaginemos um vetor de *Page Faults* (PF) de tamanho infinito, onde são acumuladas as falhas de página geradas pelo processador p. São criados também os ponteiros p_{new}, que indica a posição onde deve ser escrito o número de cada página distribuida capturada pelo *Page Handler* para modo de escrita, e p_{int}, que indica a posição onde começa o WN associado ao intervalo atual. Estas informações são suficientes para estabelecer todas as listas de falhas de página do passado de um processador com respeito a ele mesmo. Quando ocorre a criação de um novo intervalo, faz-se $p_{int} = p_{new}$.

Este é o modelo utilizado tradicionalmente quando, no fechamento de intervalos, o protocolo utiliza ponteiros para montar listas de WNs transmitidas para os nós que criam novos intervalos e que, portanto, necessitam de informações sobre o seu passado. O vetor PF e os ponteiros p_{int} e p_{new} são apresentados na Figura 3.5, onde são ilustrados os conceitos de passado e futuro.

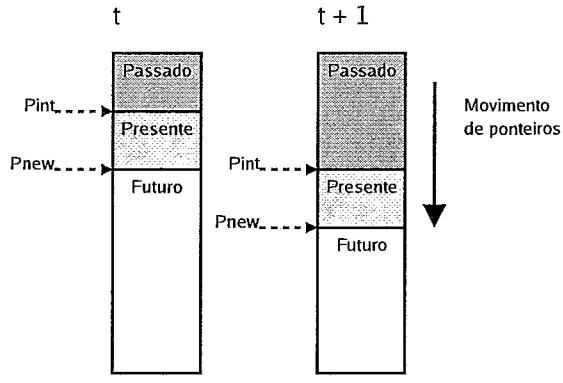


Figura 3.5: Estruturas de coerência com ponteiros escalares.

Agregação de Vetores: estruturas com visibilidade global

Agora considere que temos não um vetor PF, mas p vetores, um para cada nó, que passa a ser denominado $[PF]$. O ponteiro p_{new} passa também a ser o vetor de ponteiros $[p_{new}]$, também de dimensão p. Então, se utilizamos a Rede de Sincronização para disseminar informações de falha de página em modo de escrita, e com isso atualizar $[PF]$ e $[p_{new}]$, podemos facilmente mostrar através das propriedades da Rede que ambos vetores devem ter o mesmo valor em qualquer nó, a qualquer instante. As estruturas $[PF]$ e $[p_{new}]$ podem ser vistas na Figura 3.6.

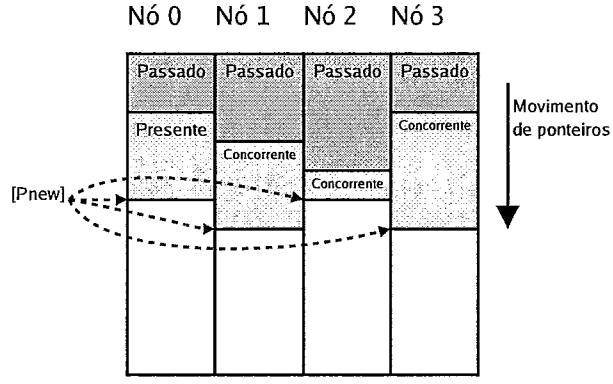


Figura 3.6: Estruturas de coerência com ponteiros vetoriais.

Porém, é imperativo notar que, apesar de disponíveis quase que simultaneamente com sua geração, as informações de *WN* de abrangência global não podem ser utilizadas para invalidar páginas antecipadamente, antes da ocorrência das ope-

rações de sincronização. Como o DSM proposto emprega um modelo LRC, esta decisão violaria o quesito de relaxamento. Ainda que fosse abandonado este modelo em busca de outros benefícios, efeitos prejudiciais poderiam acontecer na ocorrência de *false-sharing* em um cenário de múltiplos escritores, onde uma mesma página seria invalidada alternadamente pelos mesmos nós, dentro de um mesmo intervalo — o chamado efeito *ping-pong*.

Montando listas de *WNs*

Afim de adaptar a nova organização dos dados, agora com visão global, com as necessidades das regras de DSM adotadas, é necessário novamente associar a noção de consistência com a de passado transitivo, sob a ótica das novas possibilidades oferecidas. A pergunta a seguir é como localizar *WNs* de passados transitivos para realizar a incorporação de intervalos.

Traduzindo esta idéia para a organização de dados globais, oferecida pela estrutura $[PF]$, criamos o conceito de $[p_{int}]$, que captura a visibilidade local de um dado nó N sobre as falhas de página geradas por seus vizinhos, contidas nos intervalos do passado transitivo de N. Se considerarmos as listas obtidas sobre $[PF]$ na região $\{[0] \Rightarrow [p_{int}]\}$, obtemos a lista de páginas que o nó já viu terem sido alteradas, por herança de intervalos de seu passado. E na região $\{[p_{int}] \Rightarrow [p_{new}]\}$ se encontram as falhas de página de intervalos concorrentes, isto é, que não têm nenhuma relação de causalidade com seu intervalo atual.

Veja na Figura 3.7 a visão de 2 nós sobre a estrutura $[PF]$. Apesar do conteúdo de $[PF]$ e os valores de $[p_{new}]$ serem idênticos para todos os nós, $[p_{int}]$ reflete o conhecimento individual de WNs incorporado por cada um dos nós, durante as operações de coerência de seu histórico.

Quando surge uma operação de sincronização em N no tempo $t+1$, os intervalos a serem incorporados trariam consigo um novo valor $[p_{int}(t)]'$, gerado nos outros nós, que agregaria a visão dos processadores no passado imediato de N. Na operação de incorporação, da mesma forma como acontece com os *timestamps*, $[p_{int}(t+1)]$ é criado pela majoração dos termos de $[p_{int}(t)]$ e $[p_{int}(t)]'$, e estabelece a transitivi-

dade de passado para os próximos intervalos. Assim, as páginas contidas em $[PF]$ na região $\{[p_{int}(t)] \Rightarrow [p_{int}(t+1)]\}$ devem ser invalidadas com o carimbo de versão $t+1$.

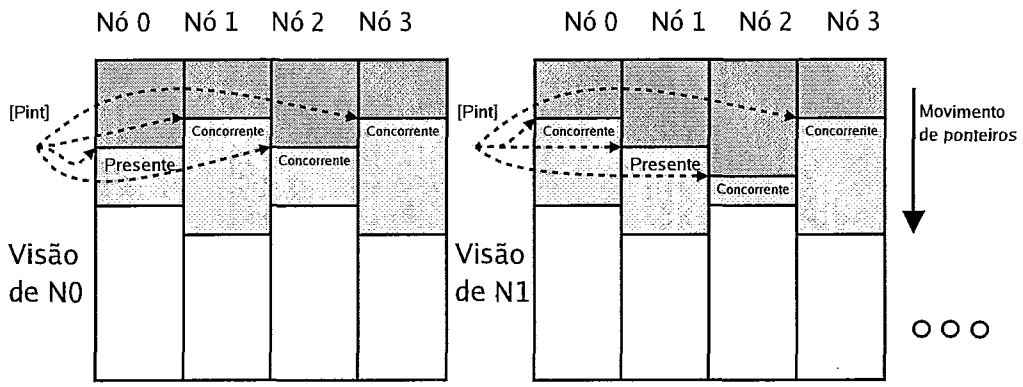


Figura 3.7: Coerência: visão de processadores sobre estruturas globais

Estruturas de Coerência Globais

Veja a Figura 3.8. Nela, se observa a agregação virtual de $[PF]$ através dos nós de computação, desnecessária, já que ela é idêntica para todos os nós. Percebemos que, como qualquer nó tem conhecimento sobre as operações de coerência de todos os nós, todas as variáveis envolvidas no mecanismo descrito anteriormente estão disponíveis a qualquer momento. Porém, no raciocínio ainda é utilizado o termo $[p_{int}]$, que reflete a visibilidade local de um nó sobre seu passado. Novamente recorremos à agregação de vetores, e criamos agora a matriz $[[p_{int}]]$, de dimensão $p \times p$, formada pelo conjunto dos vetores $[p_{int}]$, agora sim com visibilidade global. Igualmente como acontece para as variáveis $[PF]$ e $[p_{new}]$ agregadas anteriormente, esta matriz possui a propriedade de ter o mesmo valor em qualquer nó, a qualquer instante.

3.3.3 Consistência

Os algoritmos de interceptação de operações de sincronização passam também a ter carater global, capturando e processando todas mensagens da Rede de

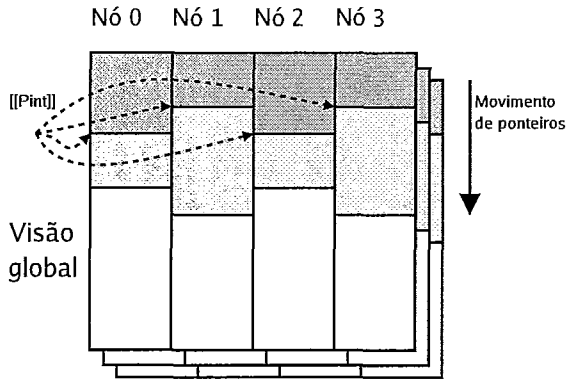


Figura 3.8: Estruturas de coerência com ponteiros matriciais.

Sincronização, independente de origem ou destino. As atualizações de $[PF]$ e $[p_{new}]$ são realizadas na chegada de mensagens de falhas de página. E as mensagens de sincronização atualizam tanto o Relógio Matricial quanto a matriz $[[p_{int}]]$, que passam a realizar operações de incorporação entre suas próprias colunas. Com isso, as operações de sincronização deixam de carregar tais colunas em suas mensagens, fazendo com que tenham um tamanho fixo, independente do número de nós, e sejam ainda mais apropriadas para a transmissão pela Rede de Sincronização, por seu tamanho extremamente reduzido. Com esta proposta, e pelo protocolo da Rede de Sincronização, assumindo um número máximo de 256 *locks*, uma mensagem de *lock* passa a ter o tamanho de 4 bits de identificação de mensagem + 8 bits de identificação de *lock* = 12 bits. Por sua vez, a mensagem de barreira é reduzida tão somente aos 4 bits de identificação de mensagem, já que não necessita transmitir nenhuma informação adicional.

Para o algoritmo de coerência, ainda são obtidos outros benefícios. A existência de várias estruturas de dados em sua forma de visibilidade global, proporcionam ao protocolo conhecimento de todas as operações relevantes ocorridas no histórico acumulado de qualquer nó. Por isso, nos pontos de sincronização, que criam gargalos no protocolo, a única informação pendente para a montagem completa do passado transitivo passa a estar contida no intervalo $I(t-1)$ imediatamente anterior. E mesmo estas informações chegam gradualmente, conforme são produzidas as falhas

de página de tal intervalo. Sendo assim, no instante em que chega a mensagem de sincronização, que encerra $I(t-1)$ e cria $I(t)$, todos os dados já estarão disponíveis localmente, bastando apenas a realização das incorporações das matrizes de tempo e $[[p_{int}]]$, operações executadas rapidamente pela CPU local.

A Figura 3.9 mostra o mecanismo de consistência do protocolo Sincro. Quando uma página é escrita por P1, é gerada sua *twin*, e um WN é disseminado através da Rede de Sincronização. Ao realizar o *release*, P1 calcula os *diffs*, e os transmite junto com seu *timestamp* à residência da página, o nó P3. Quando um novo nó P2 deseja acessar a página escrita por P1, ele processa $[PF]$ e $[[p_{int}]]$, invalidando a página atualizada no intervalo anterior. Assim, um acesso a esta página gera uma falha de acesso, quando P2 gera um pedido de página à residência, anexando seu *timestamp*. P3 responde com a página desejada, anexando na mensagem a versão de página que é transmitida.

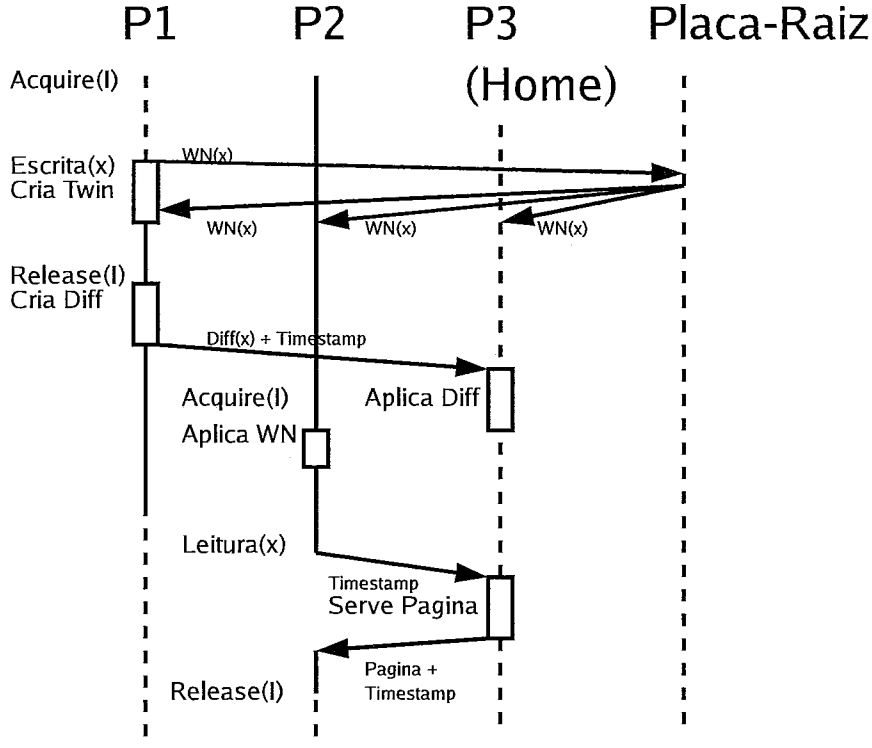


Figura 3.9: A Consistência no Protocolo Sincro.

Como demonstrado, no protocolo Sincro as mensagens de sincronização e de

WNs são transmitidas através da Rede de Sincronização. A Figura 3.10 apresenta o diagrama das operações de *lock*. Conforme visto na seção 3.2, a primitiva de *lock* transmite mensagens de *ticket* através da Rede de Sincronização. A operação de coerência, ao criar um novo intervalo, precisa tão somente processar as informações de WNs acessíveis localmente, que são disseminadas conforme são criadas. Quando deseja sair da seção crítica, o protocolo transmite apenas uma mensagem de *release*, sem necessidade de disseminar *timestamps* ou WNs junto com a mensagem de sincronização.

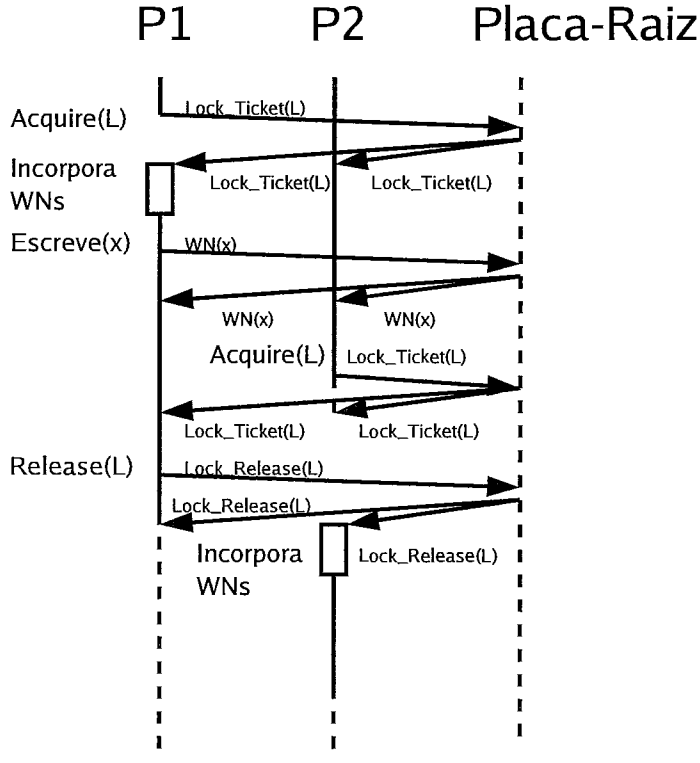


Figura 3.10: As operações de *lock* no Protocolo Sincro.

3.3.4 Estabelecimento de Residências

Pelas características de serialização da Rede de Sincronização, é possível implementar um algoritmo em que, mesmo em situações de *multiple writer*, a residência de uma dada página é conhecida instantaneamente em todos os nós. Esta

facilidade elimina a necessidade de esquemas mais complexos de busca, disseminação e *forwarding* de informações. Esta característica poderia ser explorada também para estabelecer politicas de migração de baixo custo e alta escalabilidade.

Um nó, ao desconhecer a residência de uma página, transmite pela rede uma mensagem se candidatando à residência de uma página. Como todos os nós vêem as mesmas mensagens de candidatura, a ordem das mensagens é um critério suficiente para eliminar condições de corrida. O nó então processa as mensagens da Memória Secundária, até encontrar a primeira mensagem de candidatura à residência, que servirá para estabelecer o nó responsável por tal página.

3.4 Trabalhos Relacionados

3.4.1 Sincronização

Gerações anteriores de sistemas multicomputadores, como o Cray T3E e CM-5, implementavam *hardware* especializado para a realização de operações coletivas. Com a adoção gradual de *clusters* de computadores, estas operações passaram a ser realizadas freqüentemente através de redes ponto a ponto, em *software*. Nestes sistemas, são empregadas redes de comunicação convencionais, que costumam vir com pilhas de *software* de comunicação muito custosas. Para eliminar ou reduzir o *overhead* associado ao *software* de comunicação, surgiram trabalhos propondo o suporte de primitivas de sincronização.

TTL PAPERS [13] é uma implementação em lógica TTL do *Purdue's Adapter for Parallel Execution and Rapid Synchronization* (PAPERS). Este sistema faz uso das portas paralelas dos nós processadores para a implementação de barreiras, utilizando uma árvore de portas *AND*.

Outro trabalho que tem correlação é o sistema nomeado *Synchronization/-Communication Controller* (SCC) [19]. Esse trabalho apresenta uma solução para implementação, em clusters, de barreiras baseada em *hardware* dedicado, composto de placas específicas que são conectadas ao barramento PCI dos nós processadores. A interconecção ocorre através de cabos paralelos de 40 vias, numa topologia tipo

barramento, onde cada placa possui dois conectores, possibilitando o encadeamento. Assim como no TTL PAPERS, descrito anteriormente, é utilizado o conceito de árvore de portas AND para verificar se uma barreira foi atingida. Mensagens com tamanho fixo de 64 bytes também podem ser transmitidas através desta rede.

Em um trabalho realizado por pesquisadores do Departamento de Informática da UFES em parceria com pesquisadores da COPPE/UFRJ, e utilizando a mesma infra-estrutura do Relógio Global que a empregada neste trabalho, foi desenvolvida uma rede de comunicação capaz de implementar primitivas de sincronização [12]. São disponibilizadas distintas barreiras, cada uma com seu próprio grupo de processadores, e distintos *locks*. Através de placas conectadas ao barramento PCI, este sistema implementa os algoritmos de sincronização em *firmware* dedicado, através da troca de mensagens. Oferecendo ao programador uma biblioteca do tipo *Message-Passing*, para conhecer a condição da primitiva o processador realiza *polling* na memória da placa. Neste trabalho, as mensagens são recebidas por estruturas de *hardware* distintas, de acordo com o tipo de mensagem (barreira, *lock*, WN). Aqui, não existe maneira de que os eventos de *lock* ocorridos em processadores remotos sejam conhecidos por todos os processadores. Esta caracteristica elimina a possibilidade de reconstrução de intervalos passados, o que faz que seja impossivel utilizar o suporte de *locks* para a construção de DSMs. Para a utilização do suporte de barreiras, a única maneira de associar intervalos a WNs seria a comparação dos *timestamps* das barreiras e dos WNs. A caracteristica empregada seria a de que apenas o *timestamp* da última barreira é guardado. Este problema é similar ao de barreiras em *software*, descrito por Mellor-Crummey [25]. A solução é utilização de 2 primitivas de barreira por operação de barreira: a primeira serviria para habilitar a coleta de WNs e montagem de intervalos, e a segunda serviria para inibir a escrita do *timestamp* da primeira barreira até que ela fosse lida.

Também são encontrados trabalhos desenvolvidos sobre redes SAN. Um trabalho utiliza o protocolo de comunicação VIA sobre Infiniband [22], onde são propostos 3 algoritmos de barreira, baseadas em escrita remota e *multicast*, para uma implementação de biblioteca MPI. Em outro trabalho de barreiras visando a uti-

lização em bibliotecas do tipo MPI [8], desta vez empregando Myrinet/GM, o co-processador da placa de rede é utilizado para processar mensagens de sincronização.

Em GeNIMA [5], é utilizado o esquema NIL — *Network Interface Locks*. Os *locks* são implementados na placa de comunicação, empregando VMMC sobre Myrinet. Assim como realizado no escopo desta dissertação, e diferentemente dos esquemas acima, este é também um trabalho em Sw-DSM. Baseado em HLRC-SMP, o RDMA é utilizado para suporte de coerência, sendo realizado com escritas remotas. *Locks* são implementados totalmente na interface de rede, eliminando o papel do gerente e eximindo o protocolo de qualquer custo. Com isto, é relaxado o acoplamento entre sincronização e coerência, que se reduz a uma sinalização simplificada. A interface de rede associa a cada *lock* um *timestamp*, sem porém realizar interpretação ou operações sobre estas estruturas. O protocolo, por sua vez, sabe invalidar páginas de acordo com o *timestamp* do *lock*. É sugerido que a interface de rede ofereça suporte apenas a operações atômicas remotas, levando a complexidade do *firmware* de interconexão para o *software* do protocolo [6]. Esta é a abordagem realizada pelo binômio Rede de Sincronização / Protocolo Sincro, que utiliza a propriedade de serialização da Rede para substituir as operações atômicas.

3.4.2 Sistemas DSM

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

Nos sistemas DSM baseados em *software* com suporte de *hardware*, a manutenção da coerência continua a ser feita primordialmente por *software*, mas o *hardware* adicional permite a obtenção de melhores niveis de desempenho. Este tipo de aproximação é empregada nos sistemas SHRIMP [7], CASHMERe [14] e NCP 2 [4]. O multicomputador SHRIMP é composto por PCs Pentium conectados através de uma rede similar à usada no Intel Paragon [9]. SHRIMP provê uma facilidade de mapeamento remoto de memória, segundo o qual uma página de memória local

pode ser associada à uma página da memória de um outro nó. Escritas feitas a esta página local são automaticamente propagadas para a página remota pelo *hardware*.

Aproveitando-se desta facilidade foi desenvolvido o protocolo AURC [20], baseado no modelo LRC utilizando o conceito de residência de páginas. Este protocolo atinge desempenho superior ao de TreadMarks [2]. CASHMERe usa suporte de *hardware* similar ao de SHRIMP através da rede Memory Channel [18]. Apesar do uso de suporte de *hardware* especial, CASHMERe apresentou desempenho similar ao de TreadMarks, devido à maior carga de comunicação gerada por seu protocolo. O multicomputador NCP 2 [4] provê modelo de programação de memória compartilhada em uma rede de estações com processadores PowerPC, conectadas via a rede de interconexão Myrinet. É utilizado um controlador de protocolo de maneira a reduzir ou esconder vários atrasos envolvidos na comunicação entre processadores e na manutenção da coerência dos dados. Mais especificamente, este controlador é capaz de gerar *diffs* automaticamente sem uso de *twins*, permite afastar as tarefas básicas de comunicação e coerência do processador, podendo ainda responder a pedidos remotos sem interromper o processador local.

Capítulo 4

Metodologia Experimental

Para a coleta de resultados, é empregado um *cluster* constituido de 4 máquinas Pentium III de 550 MHz, com 256 MB de memória e placas Ethernet 100 Mb/s. São também utilizadas placas contendo o *firmware* da Rede de Sincronização, configuradas para uma banda nominal de 32 Mb/s, conectadas ao sistema por um barramento PCI de 32 bits e 33 MHz. O sistema operacional utilizado é Linux 2.4.26.

Este capítulo começa apresentando resultados dos experimentos de medida de latência para pacotes transmitidos pela Rede de Sincronização. Em seguida são apresentados os programas selecionados para experimentos com o protocolo Sincro, e os métodos de instrumentação empregados no protocolo. Por fim, são apresentados os resultados e análises do protocolo Sincro, e são realizadas comparações com o protocol HLRC/UDP.

4.1 Medidas de Latência da Rede

Nesta seção é mostrado o método empregado para determinar as componentes de latência envolvidas na disseminação de mensagens através da Rede de Sincronização. O experimento consiste em variar a velocidade do *clock*, mudando por conseqüência a taxa nominal de transmissão da Rede. São empregados valores entre 14 e 32 MHz, que constitui a faixa de trabalho do sistema. É utilizada uma

sinalização de 1 bit por período, e os tempos medidos consideram a latência para a transmissão de 1 pacote de 1 byte de dados.

Para medir o tempo de transmissão, é implementado um programa que examina a fila de saida da Placa-Folha, e realiza *polling* até que ela esteja livre. Em seguida, o tempo é coletado a partir do Relógio Global, e é copiado 1 byte para o registrador de transmissão da Placa-Folha, o que desencadeia a transmissão. Após esta etapa, o programa realiza *polling* até que seja atualizado o ponteiro de escrita da Memória Primária, o que indica que chegou uma nova mensagem. Então o programa lê a mensagem, e lê novamente o tempo do Relógio Global. Então os *timestamps* obtidos antes e depois da transmissão são comparados. A diferença fornece a (latência de transmissão) $+2\times$(latência de leitura do Relógio Global). Para calcular a latência de transmissão, basta conhecer a latência de leitura do Relógio Global. Esta pode ser calculada a partir da diferença de tempo entre sucessivas leituras do registrador de tempo global.

Os resultados são apresentados na Figura 4.1. Para melhor visualizar os dados e parâmetros, o gráfico mostra a latência no eixo vertical e o período do *clock* da Rede no eixo horizontal. Com esta conformação, apresentamos os dados experimentais com "x". Conforme se pode observar, o sistema apresentou um comportamento idealmente linear. Traçamos então uma linha tracejada que representa o comportamento do sistema para valores arbitrários de período.

Trabalhando com a linha tracejada, podemos obter informações importantes a respeito do sistema. Considerando uma velocidade de *clock* infinito, quando o valor de período tende a zero, obtemos o tempo de latência envolvendo o processador de 1,51 μs. Segundo o algoritmo empregado, são necessários 3 acessos ao PCI, para i) copiar os dados à placa de comunicação, ii) observar o ponteiro de fim de fila, e iii) copiar os dados desde a placa até a memória principal. Como cada acesso custa quase 500 ns, vê-se que o PCI é o principal componente do custo computacional do protocolo.

Pela derivada da reta, se pode inferir que o *hardware/firmware* de comunicação gasta 174 ciclos para a transmissão de cada pacote.

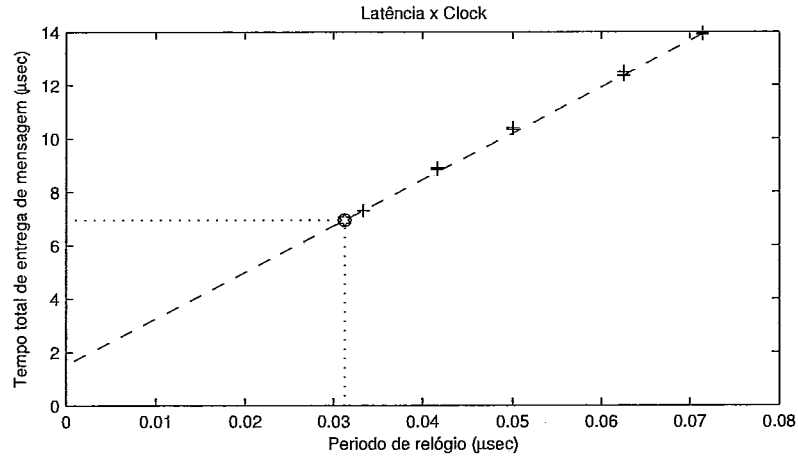


Figura 4.1: Latência da Rede de Sincronização.

De acordo com testes de estabilidade, a maior freqüência que atende a requisitos de taxa de erro adequados é de 32 MHz, que passa a ser a freqüência de trabalho oficial do sistema. Para determinar a latência para esta taxa de transmissão, projetamos sobre a linha tracejada uma linha pontilhada representando o período equivalente a 32 MHz. A partir dai, projetamos o ponto encontrado sobre o eixo vertical com uma segunda linha pontilhada. Desta forma, o valor encontrado analiticamente para a transmissão de 1 pacote na freqüência de trabalho é de 6,97 μs.

Cabe notar que o tempo de processador envolve as tarefas de transmissão e recepção de pacotes. Porém vale fazer a ressalva de que, para o recebimento de vários pacotes, o tempo de recepção de processador pode ser amortizado, transferindo para a memória principal várias mensagens.

As primitivas de comunicação para envio e recebimento de mensagens e seus respectivos tempos são apresentados na Tabela 4.1.

Tabela 4.1: Parâmetros da Rede de Sincronização versus Rede Ethernet

Rede de Sincronização	Tempo de processador	1,51 μs
	Tempo de propagação	174 ciclos
	Latência total de pacote	6,97 μs †

†Transmissão de 1 pacote, utilizando um *clock* de 32 MHz.

4.2 Experimentos com Protocolo DSM

4.2.1 Instrumentação

A instrumentação do protocolo Sincro foi realizada utilizando 2 primitivas de medida de tempo: 1) a instrução RDTSC, que lê o valor do registrador de contador de ciclos da CPU (TSC — *Time Stamp Counter*), e 2) o Relógio Global, através da leitura do registrador de relógio global presente na placa-folha. A instrução RDTSC permite uma leitura acurada do tempo com um custo praticamente nulo, e por isso é usada na maioria dos pontos de instrumentação. Porém, ele representa uma base de tempo local, e por isso não pode ser usado para medir o tempo decorrido entre eventos que ocorrem em processadores distintos. Para isto é utilizado o Relógio Global, que é empregado para realizar a decomposição do tempo de falha de acesso às páginas. Uma descrição destes tempos componentes é encontrada na Seção 4.2.3.

4.2.2 Aplicações

Foram utilizadas 5 aplicações para avaliar o protocolo DSM proposto. FFT, LU e Radix são aproveitadas da suite SPLASH2. SOR foi desenvolvido pela Universidade de Rochester, e IS faz parte do pacote de aplicações de Threadmarks. Estas aplicações foram utilizadas em diversas avaliações de Sw-DSM, servindo como uma boa base de comparação para o protocolo Sincro.

Na Tabela 4.2 é possível ver os tamanhos das entradas, tempo de execução sequencial, e perfil de sincronização utilizada por cada aplicação.

Tabela 4.2: Entradas de dados dos programas

Programa	Dados	Algoritmo	Tempo Seq.	Sincronização
FFT	2^{20}	1D, complexos	9,66 seg.	barreiras
IS	2^{23}	10 iterações	22,2 seg.	*locks* e barreiras
LU	1024×1024	Bloco=16 el.	16,1 seg.	barreiras
Radix	5×10^6 chaves	R=65535	5,36 seg.	*locks* e barreiras
Sor	2096×2096	8 iterações	4,84 seg.	barreiras

Em seguida são descritas as aplicações empregadas para avaliar o protocolo

DSM proposto.

FFT

FFT realiza a operação *Fast Fourier Transform* em N pontos complexos. A comunicação existente é encontrada essencialmente na transposição de uma matriz de $\sqrt{N} \times \sqrt{N}$. FFT distribui a matriz de forma que cada processador receba um conjunto contínuo de N/P linhas. As matrizes fonte e destino são trocadas em cada transposição. Nesta versão de FFT, cada processador, na operação de transposição, lê uma submatriz de $\sqrt{N}/P \times \sqrt{N}/P$ de outro processador e escreve no seu conjunto de linhas locais. Consequentemente, a granularidade de acessos de escrita é larga, enquanto a granulação do acesso de leitura depende de N e P. Para problemas de tamanhos típicos, normalmente são encontrados fragmentação e falso compartilhamento. Assim, o desempenho de FFT é afetado pela alta taxa de comunicação/computação e pela comunicação extra induzida pela fragmentação. Normalmente, FFT apresentará o padrão de acesso às páginas de 1 produtor e múltiplos consumidores, mas para problemas muito grandes, este padrão pode passar para 1 produtor e 1 consumidor, evitando o falso compartilhamento e fragmentação.

IS

IS, ou *Integer Sort*, ordena um vetor de N inteiros usando chaves no intervalo $[0, Bmax]$ através da técnica *Bucket Sort*. As chaves são divididas pelos processadores e cada iteração consiste de três fases separadas por barreiras. Na primeira fase, o processador zero inicializa o vetor global de *buckets*. Na segunda fase, cada processador conta suas próprias chaves armazenando resultados parciais em um vetor local para, imediatamente antes de chegar à barreira, adicionar os resultados locais ao vetor global de *buckets*. A serialização dos acessos a este vetor global é feita usando-se uma variável de *lock*: cada processador adquire o *lock*, adiciona os valores locais ao vetor global atualizando-o totalmente, e libera o *lock*. Nesta fase, o padrão de compartilhamento é caracterizado por páginas migratórias dentro da seção crítica. Nesta última fase, todos os processadores lêem o vetor global para

classificar suas chaves locais, não havendo escritas a dados compartilhados.

LU

LU fatora uma matriz densa em um produto de duas metades triangulares da mesma matriz. Barreiras são usadas para assegurar a sincronização entre o processador que está produzindo a linha pivô e os processadores que irão consumir esta linha. O padrão induzido pelos acessos de leitura é de 1 escritor e múltiplos consumidores. Este *kernel* exibe acessos de grão grosso e pouca sincronização para a freqüência de computação, mas a computação é inerentemente desbalanceada.

Radix

Radix ordena uma série de chaves inteiras em ordem crescente. A fase dominante é a permutação das chaves. Em Radix, um processador lê seu conjunto de chaves locais de um vetor fonte e escreve-os em um vetor destino de uma maneira bastante irregular e espalhada. Este padrão de escrita induz substancialmente falso compartilhamento no acesso às páginas.

SOR

SOR resolve equações diferenciais parciais usando uma estratégia do tipo "vermelho-preto" para realizar as relaxações sucessivas. Cada iteração é composta por duas fases separadas por barreiras. Os valores da matriz vermelha são calculadas na primeira fase e os da matriz preta na segunda. Cada elemento da matriz é calculado como sendo a média de seus vizinhos na matriz original. Como a matriz é dividida em fatias entre os processadores, o compartilhamento de dados ocorre nas bordas de cada fatia e segue o padrão 1 produtor e múltiplos consumidores. A primeira iteração não é computada nas estatísticas para eliminar os efeitos da inicializaçao.

A Tabela 4.3 resume os padrões de acesso das aplicações.

Tabela 4.3: Perfil dos programas

Programa	Perfil
FFT	Granularidade de escrita grossa. Fragmentação e falso compartilhamento.
IS	Páginas migratórias nos *locks*. Fase final sem escritas compartilhadas.
LU	1 escritor e múltiplos consumidores. Computação desbalanceada, granularidade grossa.
Radix	Padrão de escrita irregular e espalhado, gerando falso compartilhamento. Granularidade fina.
Sor	Múltiplos escritores e múltiplos consumidores. Grão grosso, com compartilhamento de dados nas bordas das fatias das matrizes.

4.2.3 Componentes de Desempenho

A análise de desempenho do protocolo Sincro é dividida em 5 tempos básicos, sendo: Barreira, *Locks*, Falha de Página, Computação, e Outros. A instrumentação ainda dividide estes tempos em outras operações mais elementares, oferecendo uma análise mais detalhada do comportamento do protocolo. Inicialmente, as análises apresentarão dados baseados nos tempos básicos e, conforme necessário, serão mostrados os tempos detalhados mais relevantes.

- Relacionados à Barreira
  - Processamento em Barreira: responde pela segunda e última fase, representando o tempo de processamento associado às atividades de consistência, incluindo as tarefas de geração de *diffs* e invalidação de páginas;
  - Espera em Barreira: responde pela primeira fase da barreira, quando o processo deve esperar a notificação de chegada de todos os outros processos na barreira, e representa o desbalanceamento de trabalho entre os processadores;
- Relacionados a *Locks*
  - Aquisição: é o tempo dispendido desde que o processador transmite um pedido de *lock*, até o momento em que ele é realmente adquirido;

- Liberação: tempo envolvido na disseminação do *release*, e do fechamento do intervalo;

- Relacionados a Falhas de Página

  - Pedido: propagação do pedido de página desde a montagem da mensagem no nó requerente até o momento em que o nó *home* começa a tratar a mensagem, sua componente dominante está relacionada à latência da rede *Ethernet*;

  - Resposta: propagação da resposta de pedido de página, desde o momento em que a mensagem de resposta está montada até o momento em que chega na *Thread* Servidora, sua componente dominante está relacionada à latência da rede Ethernet;

  - Alocação e cópia de *Twin*: no código do *Page Handler*, cobre as partes referentes à técnica de *twinning*;

  - Outros: o tempo remanescente não contabilizado pelas componentes anteriores é acumulado nesta componente.

- Relacionados a *Diffs*

  - Cálculo: é o tempo decorrido durante a geração de pacotes de *diffs*;

  - Transmissão: tempo dispendido pelo processador para transmitir os pacotes de *diffs* para o nó de residência da página.[1]

- Relacionados a WNs

  - Transmissão: é o tempo dispendido pelo processador para transmitir as mensagens de falha de página; no protocolo proposto, faz parte do código do *Page Handler*, enquanto que na implementação de HLRC sobre UDP pertence às operações de sincronização;

[1] Sujeito ao controle de fluxo UDP, presente no canal *Ethernet*.

– Recepção: é o *overhead* que o protocolo impõe às operações de sincronização; no protocolo proposto este tempo é definido como nulo, enquanto que no HLRC sobre UDP é o tempo em que o protocolo está ocioso aguardando a chegada destas mensagens pela rede *Ethernet*. [2]

4.2.4 Análise de Desempenho de Protocolos DSMs

Nesta seção são apresentados os resultados dos experimentos envolvendo as aplicações selecionadas. Inicialmente são mostradas análises do Protocolo Sincro, apresentando os componentes envolvidos no *overhead* do protocolo. Em seguida, é realizada uma análise comparativa com o protocolo HLRC de Princeton [26], em sua implementação UDP sobre *Ethernet* [27]. Ambas implementações foram medidas em um *cluster* de 4 processadores. Para avaliação, cada aplicação é executada 5 vezes, e então é calculado o tempo médio.

4.2.5 Análise do Protocolo Sincro

Na Figura 4.2 é possível visualizar a aceleração do Protocolo Sincro para as aplicações selecionadas. Conforme vemos, o desempenho de IS se comporta próximo do linear, vindo em seguida LU e SOR com boas acelerações. FFT e Radix apresentam *speedup* ligeiro para 4 processadores.

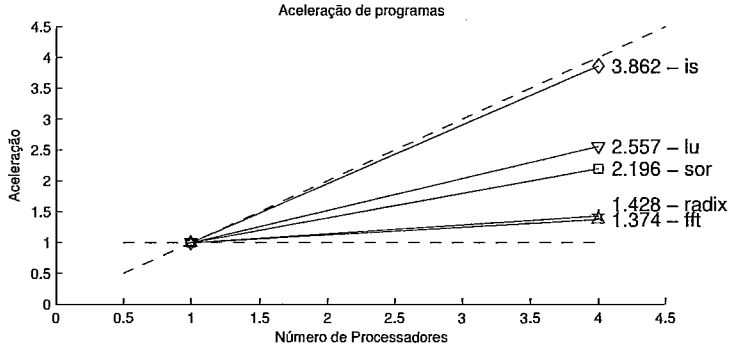
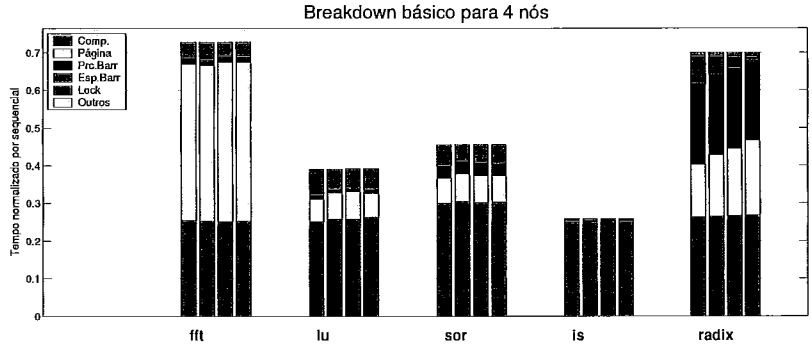


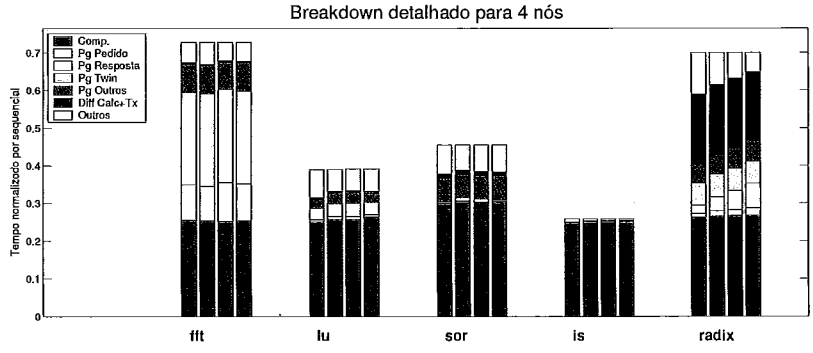
Figura 4.2: Aceleração das aplicações.

[2] O tempo de recepção de WNs é amortizado junto com o tempo de recepção de mensagens de sincronização, tornando a instrumentação desta medida imprecisa.

Na Figura 4.3 podemos observar 2 gráficos de *breakdown* para as aplicações selecionadas. O gráfico (a) apresenta os componentes básicos de *breakdown*, e já apresenta os tempos de barreira detalhados em Espera e Processamento. O gráfico (b) permite a visualização dos componentes detalhados. Os tempos são normalizados usando o tempo sequencial como referência, de tal forma que a barra mostra no eixo à esquerda o valor da aceleração de cada programa. Ambos os gráficos apresentam um conjunto de 4 barras para cada aplicação, cada uma representando 1 processador do *cluster*. Maiores detalhes sobre o significado dos tempos podem ser encontrados na Seção 4.2.3.



(a) Componentes básicos do *overhead*



(b) Componentes detalhados do *overhead*

Figura 4.3: Componentes envolvidos em *Overhead.*

FFT

A aplicação FFT consome um tempo considerável, mais de 1/2 de seu tempo de *overhead*, aguardando páginas. O *overhead* restante se concentra em tempo de espera em barreira, conforme se observa na Figura 4.3(a).

Analisando a Figura 4.3(b), se percebe a concentração dos tempos de página principalmente nos componentes de Resposta e Pedido, significando que a aplicação poderia ser beneficiada com uma rede de banda mais larga e de menor latência, respectivamente.

Nenhum *diff* é transmitido durante a fase deste programa onde a instrumentação está ativa.

LU

Conforme se observa no gráfico de componentes básicos, o *overhead* de LU se divide entre barreira e falha de página. No primeiro caso, domina o tempo de espera em barreira, e no segundo, se divide em tempo de resposta de página e outros tempos de falha de página.

Esta aplicação não transmite nenhum *diff* enquanto a instrumentação está ativa.

SOR

Na aplicação SOR, os *overheads* se dividem quase que igualmente em barreira e página. Cerca de 2/3 do tempo de barreira é de espera. Quanto ao *overhead* de página, este se concentra em outros tempos.

Esta aplicação não transmite nenhum *diff* enquanto a instrumentação está ativa.

IS

Esta aplicação é extremamente paralela, e alcança aceleração próxima do linear. Os tempos de *overhead* são extremamente reduzidos, se dividindo basicamente

em barreira, e falha de páginas, que predominam, e *locks*. Nos tempos de barreira, a maior subcomponente é de processamento. O tempo dos *locks* se concentra na aquisição.

Radix

A aplicação Radix apresenta grandes componentes de barreira e de página, surgindo também um tempo pequeno mas significativo de *locks*.

Na sincronização, os tempos de barreira se concentram em processamento, e os de *lock* se dividem quase que igualmente em liberação e aquisição, sem muita variação entre os processadores. De acordo com o *breakdown* detalhado, estes tempos são quebrados, onde predomina o cálculo de e transmissão de *diffs*. A disseminação de WNs aparece com uma contribuição modesta, mas de menor peso.

Os tempos de falha de página são ligeiramente desbalanceados, gerando tempo de espera em barreira que os complementam. A origem deste fenômeno é de difícil localização. O estabelecimento de residência de páginas é bem balanceado, com residências por processador em torno de (2521 x 2525 x 2525 x 2526). As 5 execuções desta aplicação também apresentam um desvio padrão extremamente reduzido para as medidas instrumentadas. Apesar destes fatos, surge uma diferença no tempo de Resposta de Página entre os processadores, causado aparentemente pelo desbalanceamento na quantidade de *miss* de falhas de página remotas (180 x 300 x 430 x 540).

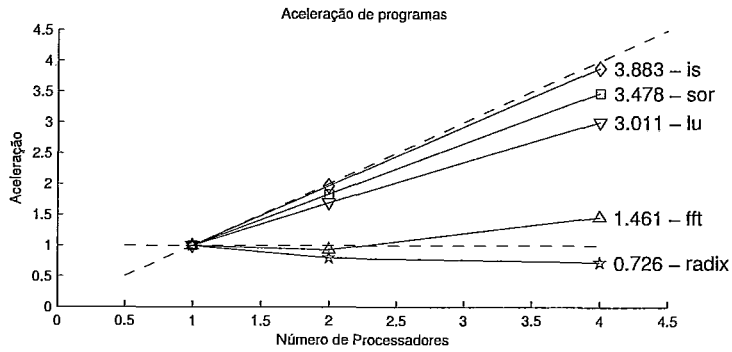
O tempo restante de falha de página se concentra em *Twin*, surgindo uma parcela significativa diluida em tarefas secundárias. Isto é dado pelo fato de que a maioria das falhas de escrita são do tipo *hit* remoto, o que também explica a contribuição elevada dos *overheads* relacionados a *diff*.

4.2.6 Análise comparativa do Protocolo Sincro

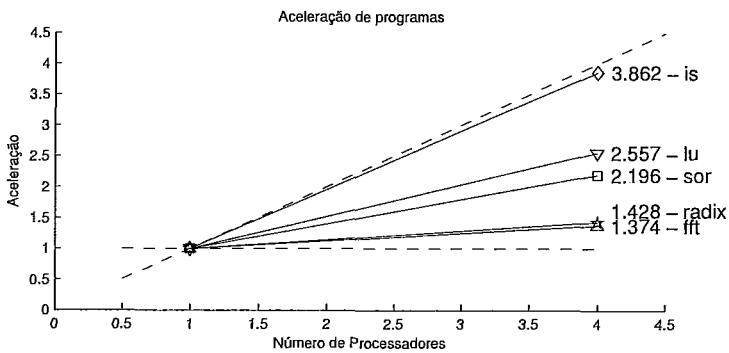
Nas Figuras 4.4(a), e 4.4(b), podemos ver as acelerações das aplicações para os protocolos HLRC e Sincro. Com esta figura é possível observar o comportamento geral de ambos protocolos para todas as aplicações.

O protocolo Sincro apresenta em geral um comportamento inferior ao do HLRC. IS apresenta aceleração próxima do linear, seguido de LU e SOR, com boas acelerações. Depois, praticamente juntos, vêm Radix e FFT, com acelerações moderadas.

As aplicações IS e FFT obtiveram resultados bem idênticos entre os protocolos HLRC e Sincro.



(a) HLRC



(b) Protocolo Sincro

Figura 4.4: *Speed Ups* das aplicações.

Em seguida apresentamos a Figura 4.5, que mostra a divisão dos tempos das aplicações em seus tempos básicos, quando executados em 4 processadores, para as 2 implementações medidas.

FFT

Na Figura 4.6 são mostrados os gráficos de *breakdown* básico e detalhado para a aplicação FFT, para os protocolos HLRC e Sincro. A aceleração passa de 1,46 no HLRC para 1,37 na versão pós-toque. Com o uso do Relógio Global, é possivel instrumentar os tempos de falha de página de pedido e resposta separadamente. O maior *overhead* se concentra em tarefas de Pedido e Resposta de páginas. O tempo total de falha de página, assim como o tempo total de barreira, é ligeiramente maior ao medidos com HLRC.

LU

Na Figura 4.7 se observa os gráficos de *breakdown* básico e detalhado da aplicação LU. Se vê no protocolo HLRC o tempo de falha de páginas e de barreiras dominando quase que igualmente o tempo de *overhead*, sendo ambos ligeiramente maiores no protocolo Sincro que no HLRC. Isto é sentido com uma diminuição no *speedup*, que vai de 3,01 no HLRC para 2,56 no Sincro.

SOR

SOR é a aplicação que gera proporcionalmente mais falhas de acesso de escrita por barreira (cerca de 1500 WNs por barreira). Por isso, se observa para 2 nós que o tempo de disseminação de WNs é menor no HLRC que para o protocolo proposto. Isto se dá pelo fato de que HLRC utiliza a rede Ethernet para a transmissão de tal informação, agrupadas em um único pacote. Pelo comportamento da Rede de Sincronização de tempo linear com o tamanho de dados a transmitir, existe um limiar tal que ela deixa de ser vantajosa. Quando se passa a 4 nós, novamente ela volta a ser competitiva, consumindo 1,8 % do tempo sequencial, frente a 2,1 % do HLRC.

Os gráficos de *breakdown* básico e detalhado da aplicação SOR são aprentados na Figura 4.8. Em comparação com o HLRC, podemos perceber um aumento dos tempos de barreira e de falha de página para o protocolo proposto. Curiosamente, se observa também um ligeiro aumento no tempo de computação.

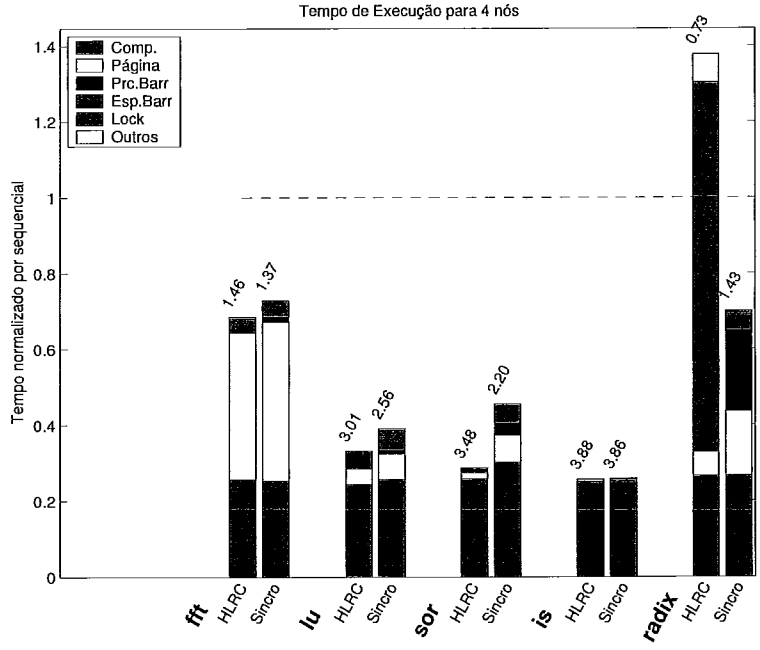


Figura 4.5: Tempos normalizados de Aplicações.

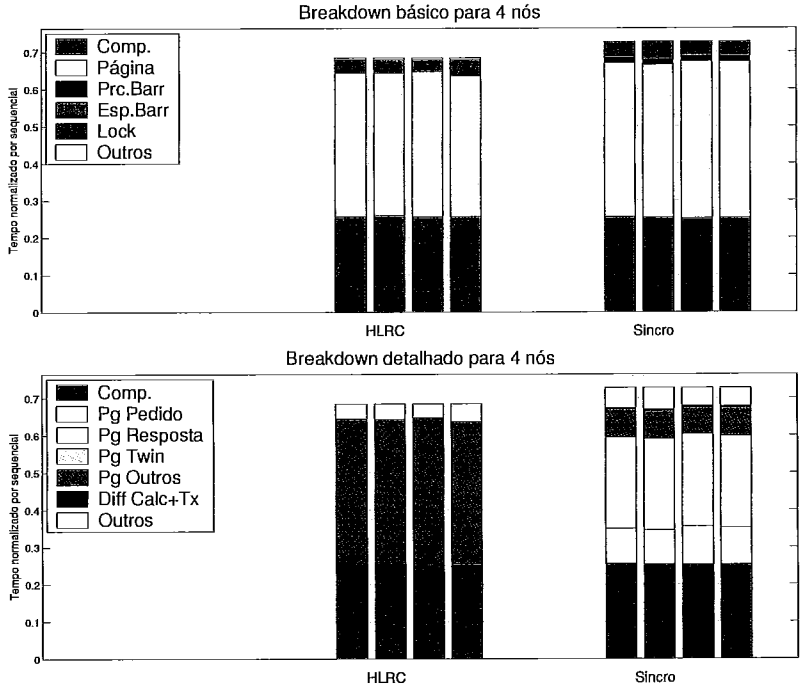


Figura 4.6: Tempos de Aplicação: FFT

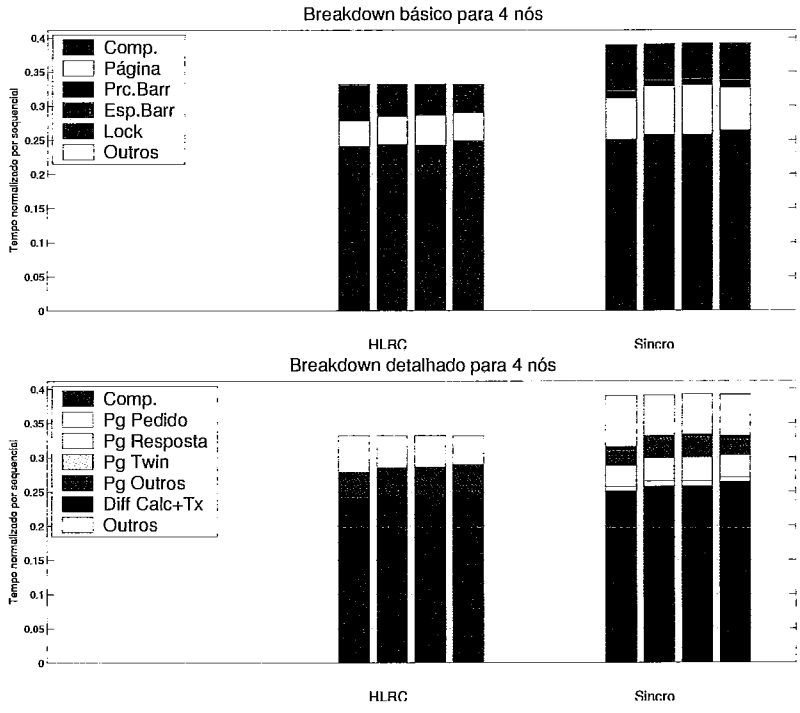


Figura 4.7: Tempos de Aplicação: LU

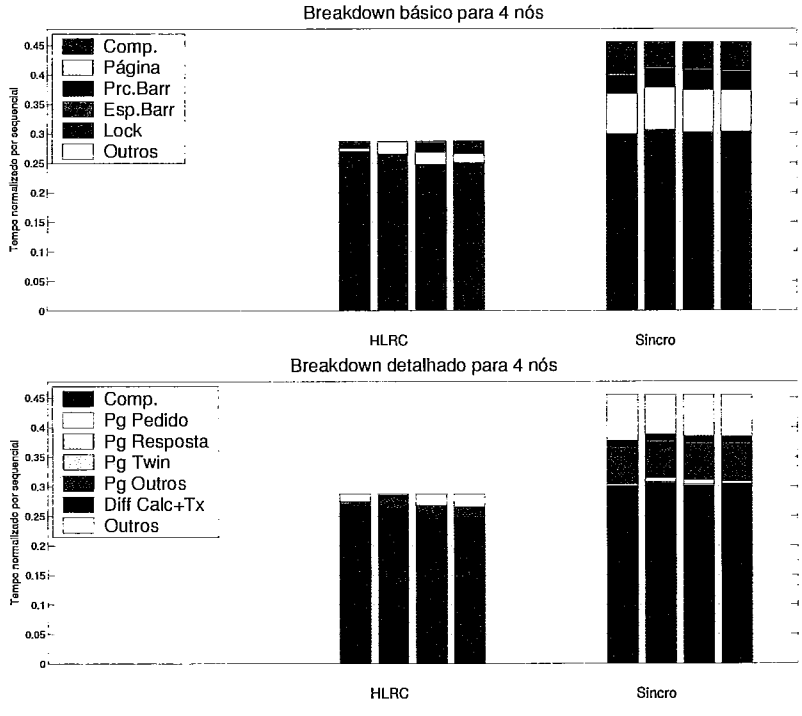


Figura 4.8: Tempos de Aplicação: SOR

Contribuindo com cerca de 1/6 do tempo de *overhead*, equivalente a 1/3 do tempo de falha de página, encontramos as chamadas de *mprotect*. No caso deste programa, como a maioria das falhas de página é do tipo *hit* local, as chamadas de *mprotect* em questão ficam concentradas no momento de fechamento de intevalos. Isto significa que o processamento de listas de WNs é o principal responsável. Apesar de não implementada a invalidação de múltiplas páginas com um único *system call*, a taxa indicativa de páginas consecutivas, de valor 0,9994, mostra que poderiam ser economizadas muitas destas chamadas.

Com cerca de 1/5 do *overhead* total, encontramos as atividades envolvendo transferencia de dados da Rede de Sincronização para a CPU, dos quais a quase totalidade recai sobre operações de coerência dentro da barreira. Embora este seja um tempo significativo, não há garantias de que a aceleração deste fator contribua para a melhoria dos tempos de programa, uma vez que ainda é percebido um tempo de espera nas barreiras por aguardar a chegada de outros processadores.

IS

Na Figura 4.9 são mostrados os gráficos comparativos para a aplicação IS. Se observa no protocolo HLRC que o tempo de disseminação de WNs domina o tempo de *overhead* desta aplicação, que apresenta aceleração de 3,88, bem próxima do linear. Esta componente é reduzida no protocolo Sincro, que apresenta aumento nos tempos de barreira e *lock*, com uma diminuição mínima da aceleração, para 3,86.

Radix

O *breakdown* comparativo para a aplicação Radix é apresentado na Figura 4.10. Se observa no protocolo HLRC o tempo de barreira como dominando o tempo de programa. O tempo de página contribui também, sendo de menor impacto no tempo de *slowdown* desta aplicação. Note que a instrumentação deste protocolo não permite distinguir os subcomponentes de espera e processamento em barreira, nem tampouco nenhum dos detalhamentos de tempo de página.

No protocolo Sincro, se observa um aumento do tempo de páginas, e uma

enorme diminuição do tempo de barreiras, que faz com que o programa passe de um forte *slowdown* a uma moderada aceleração. Observando o gráfico detalhado, é possível determinar a causa do enorme tempo de barreira, em sua maioria atribuído à transmissão de *diffs*.

Esta grande diferença pode ser atribuido a 2 fatores: o método de disseminação de *diffs* do HLRC, e o padrão de escrita da aplicação Radix. Explica-se da seguinte forma: no protocolo HLRC, cada região de página que é alterada gera um pacote UDP, enquanto que no Sincro é utilizada uma estrutura de pacote que gera no máximo 1 pacote de *diff* por página. Isto associado ao padrão de escrita do Radix, de alta granularidade, tem o efeito de criar um enorme *overhead* na camada de comunicação UDP, que é obrigada a enviar vários pacotes quando seria suficiente apenas 1.

É curioso perceber um desbalanceamento similar nos tempos de página entre os dois protocolos instrumentados. Explicações para este fenômeno são encontradas na Seção 4.2.4.

Análise de disseminação de *WN* nas aplicações

Visualizamos na Figura 4.11 a sobrecarga atacada diretamente pelo protocolo, a disseminação de WNs. O gráfico apresenta os tempos percentuais gastos nesta tarefa, normalizados de acordo com o tempo do programa sequencial, de forma a oferecer uma medida do impacto na aceleração dos programas. A Figura apresenta tempos médios para 4 processadores, onde podemos ver a redução drástica deste componente em relação à referência, o protocolo HLRC.

No protocolo de referência, o tempo de transmissão é efetivamente desprezivel, e o *overhead* se concentra na recepção. No protocolo proposto, a instrumentação impede que sejam feitas medidas do tempo de recepção, que é amortizado com outras mensagens recebidas, que são mensuradas no tempo de coleta de mensagens da Rede de Sincronização[3]. A aplicação mais notável é o Radix, onde o HLRC gasta

[3]Ocorre ainda um efeito curioso na amortização: se o processador chega muito cedo na barreira, os WNs serão colhidos em várias etapas, fazendo com que aumente o tempo de recepção de WNs. Todavia, este tempo estaria mascarando o tempo de espera, já que o justo seria não coletar WNs e

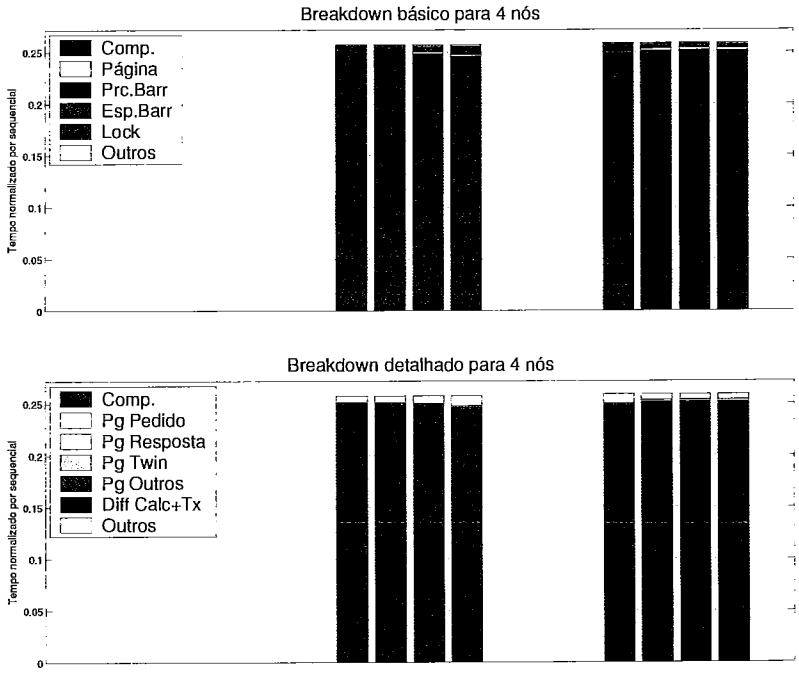


Figura 4.9: Tempos de Aplicação: IS

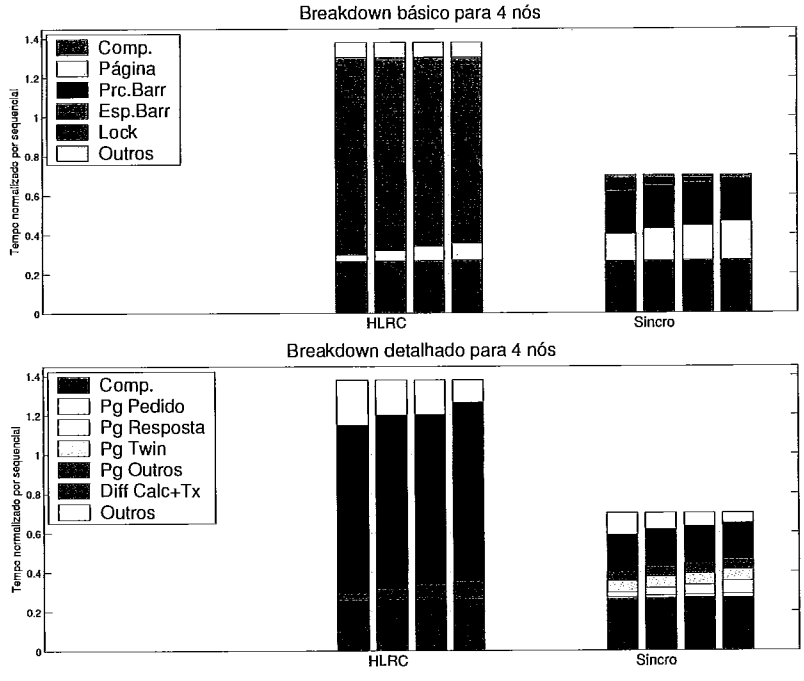


Figura 4.10: Tempos de Aplicação: Radix

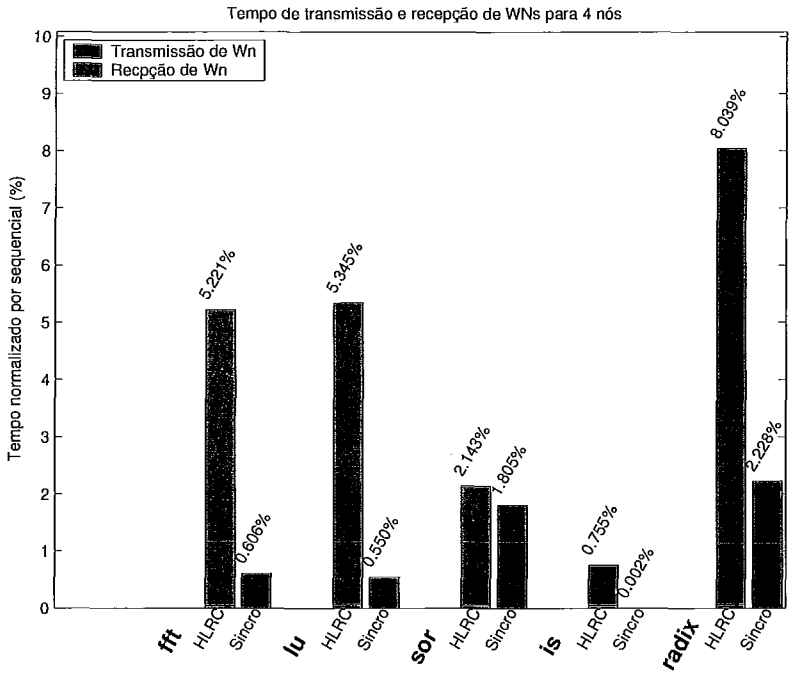


Figura 4.11: Tempos de propagação de WNs.

mais de 8% do tempo sequencial com a disseminação de WNs. Enquanto isso, o protocolo proposto oferece uma aceleração de quase 4x, dispendendo pouco mais de 2% do tempo sequencial com tais tarefas. A maior aceleração medida foi para a aplicação IS, de mais de 300x. Porém, como o protocolo de referência dispende menos de 1% nesta atividade, pouco ou nenhum impacto positivo é esperado.

4.3 Resumo geral de resultados

As medidas e análises encontradas neste capitulo não envolvem a avaliação do impacto do suporte de *hardware* sobre os tempos de *locks*. Apesar de serem utilizadas 2 aplicações que empregam esta primitiva de sincronização, e o protocolo contemplar tais primitivas, o *overhead* de protocolo atribuido a *locks* é muito reduzido em proporção ao tempo total de programa. Com isso, torna-se dificil decidir

sim contabilizar o tempo de espera, realizando uma única recepção em *burst* de mensagens da Rede de Sincronização. Isto permitiria amortizar os acessos ao *hardware* e permitiria uma comparação justa com os processadores que chegam por último nos pontos de sincronização.

conclusivamente quais os fatores influentes nas medidas, dificultando análises mais precisas.

De forma geral, o protocolo Sincro foi eficiente na meta de auxiliar na disseminação de WNs através dos processadores. Porém, as acelerações obtidas foram inferiores ao HLRC, sendo fortemente influenciadas por tempos relacionados a espera em barreira e pedido de páginas. Destaque é dado ao protocolo Sincro para a aplicação Radix, que passa de uma situação de *slowdown* no HLRC a uma aceleração moderada.

Capítulo 5

Conclusão

5.1 Avaliação

Neste trabalho foi desenvolvida a Rede de Sincronização, uma rede de comunicação auxiliar para a disseminação de mensagens pequenas com baixa latência.

Este rede foi empregada como rede auxiliar de comunicação do protocolo Sincro, um Sw-DSM de consistência relaxada baseado em residência. Ela foi utilizada para implementar um esquema de disseminação de mensagens de sincronização e coerência em modo de *broadcast*. Com isto, foi reduzido o *overhead* associado à disseminação de informações de WN, que surgem nos pontos de sincronização das aplicações. Este modelo de transmissão de mensagens, aliado às características inerentes à arquitetura da Rede, levou à remodelagem de estruturas de coerência, que adquirem visibilidade global e permitem redução do tráfego de dados.

Avaliações do protocolo Sincro mostram a redução do tempo de disseminação de WNs, apesar de seu desempenho em geral ser inferior ao protocolo de referência, o HLRC de Princeton. Porém, a comparação entre os protocolos mostra que Sincro oferece aceleração para todas as aplicações medidas, ao contrário do protocolo pré-existente.

5.2 Trabalhos Futuros

Novas funcionalidades

Uma ressalva deve ser feita quanto ao fenômeno ocorrido com as medidas para 2 nós da aplicação SOR, onde o protocolo Sincro gasta mais tempo com a disseminação de WNs do que o HLRC. Deve ser mencionado que esta situação é facilmente revertida com a utilização de compressão de WNs. Em casos onde isto não seja possível, pode ser realizada a atualização das tecnologias empregadas na camada física da Rede de Sincronização, não sendo necessariamente preciso alteração de sua arquitetura.

O conhecimento global sobre informações de coerência poderia ser empregado pelo Sincro para realizar migrações de páginas sem o custo adicional de disseminação da informação de migração, desde que empregado um critério determinístico. Em particular, é potencialmente interessante realizar a migração no cenário onde uma dada página foi acessada apenas por um processador no último intervalo, evitando o cálculo e disseminação de *diffs* na nova residência, e aplicação de *diffs* na antiga. Espera-se que no caso médio também signifique uma falha de acesso local para a nova residência no novo intervalo.

Atualmente, as mensagens de requisição de páginas, tanto para modo escrita quanto leitura, são realizadas também pela rede Ethernet, e sujeitas ao escalonamento da *Thread* Servidora. Na sua atual versão, a Rede de Sincronização transmite mensagens de falha de página em modo escrita, mas somente para fins de coerência, não oferecendo suporte ao serviço de páginas. Futuramente, seria possível modificar o seu *firmware*, para gerar uma interrupção e tratar a mensagem de falha de página. A instrumentação das aplicações utilizadas durante este trabalho demonstra que este é um aspecto potencialmente interessante para o protocolo Sincro. Projeções dos tempos de aplicações que empreguem estes métodos podem ser encontradas na Seção ??, junto com outros resultados.

Investigação de Desempenho

O Protocolo Sincro obteve desempenho similar ao HLRC para algumas aplicações. Embora empregue mecanismos semelhantes, em alguns casos o desempenho do Sincro foi inferior ao HLRC, que possui um código mais amadurecido e portanto, mais otimizado. Ainda, o código do Sincro possui alguns mecanismos de proteção e conferência de integridade de mensagens, que poderiam ser "desligados" de modo a se melhorar a velocidade das tarefas executadas.

Aumento de Robustez

Existe espaço para a melhoria do Protocolo de Sincronização, quanto a robustez. Pode ocorrer a situação em que a Memória Primária chegue próxima de um *overflow*, e que o processador não desencadeie a cópia de mensagens entre a Memória Primária e Secundária. A principio esta operação ocorre somente nos pontos de sincronização — barreiras e *locks* — porém o mecanismo pode ser melhorado basicamente de 3 maneiras, cada uma com seu *trade-off*. Vale notar que todos os métodos envolvem a cópia entre memórias e o processamento das mensagens contidas na fila de entrada.

O método empregado atualmente envolve a alteração da rotina de transmissão de mensagens, que examina as Memórias Primária e Secundária, e dispara a rotina de cópia e processamento quando apropriado. Considerando que a cópia e processamento de mensagens é um evento extremamente raro, sua desvantagem é um aumento no tempo de transmissão de mensagens, que deve ler registradores da placa-folha para saber do *status* da Memória Primária. Tem ainda a desvantagem de não oferecer garantia de tempo máximo para a ocorrência do evento: este fica à mercê de que o processador realize uma falha de página de escrita, para que seja então transmitida uma mensagem de WN, desencadeando o mecanismo.

O método mais robusto e com melhor desempenho seria a geração de interrupções quando as Memórias Primária e Secundária estivessem próximas de um *overflow*. Sua desvantagem seria o uso de recursos de *hardware* para detectar esta situação, e a criação de código adicional, para tratamento de interrupção, *bottom*

half, e uso de semáforos. Estes seriam utilizados para criar seções criticas e proteger os recursos relacionados com a Memória Secundária, que hoje não são *thread-safe*.

O segundo método seria a criação de uma *thread* de manutenção, que à maneira de um *watchdog*, é chamada periodicamente para realizar a varredura e processamento das mensagens. Apesar de ser de implementação simples, esta *thread* estaria disputando o processador com a *thread* principal, que executa o programa. Ainda, esta alternativa implica na utilização de semáforos e seções criticas, como o primeiro método, que podem aumentar o caminho crítico mesmo quando não existirem eventos a serem tratados.

Apêndice A

Projeção de Otimizações

Nesta seção, são apresentadas medidas complementares, com o objetivo de analisar o impacto da tecnologia disponivel nos tempos de programa. Para isso, o código é instrumentado de forma a medir o tempo gasto em atividades do protocolo. Em seguida, é proposto um modelo analítico de custo para estas mesmas atividades, baseado em valores estimados para outras tecnologias. Calculando a diferença entre estes valores, estima-se então o tempo de programa final, caso os mesmos algoritmos fossem empregados numa implementação otimizada do protocolo. Com isto, espera-se uma projeção realista das melhorias esperadas ao se atacar os diferentes aspectos do protocolo, e assim direcionar trabalhos futuros.

Primeiramente são descritas as técnicas sugeridas, acompanhadas da Tabela A.1, que apresenta os valores dos parâmetros empregados no cálculo das projeções. Em seguida é apresentado o gráfico de projeção com as aplicações selecionadas, na Figura A.1, prosseguindo com a análise do impacto das técnicas sugeridas em tais aplicações.

A.1 Técnicas de Otimização

Overhead de *Hardware*

Para o cálculo deste fator, é medido o tempo gasto pelo processador para a transmissão e recepção de mensagens. Influenciam neste tempo o barramento de

acesso à placa, e *overheads* associados à controle de fluxo e recebimento confiável de mensagens.

O modelo estima a utilização do mesmo barramento PCI utilizado pelo experimento. Porém, as tarefas de transmissão assumem a existência de uma fila de saida dimensionada adequadamente, levando a um tempo ideal de transmissão a t_{send} por mensagem.

Pedido de Páginas

O tempo de pedido de páginas é sempre sujeito ao escalonamento da *Thread* servidora, devido ao modelo de pedido/resposta. Para reduzir este tempo, é proposto um modelo de serviço de páginas que utilize a Rede de Sincronização. Com o desenvolvimento de um suporte de *hardware* apropriado, a mensagem geraria uma interrupção ao chegar na Placa-Folha do processador que possui a residência de uma página. A página seria servida então sem a necessidade de uma *thread*, evitando efeitos de co-escalonamento no tempo de programa. O tempo de pedido de página assume o valor t_{ask_new}.

Em trabalhos futuros, seria até mesmo posssível propor um esquema sem o envolvimento da CPU, onde a Placa-Folha poderia ser empregada para transmitir páginas, em um modelo RDMA ou conversando diretamente com a placa Ethernet. Esta proposta teria potencial bem maior do que o oferecido pelo modelo anterior, já que poderia eliminar outros componentes do tempo de Falha de Página.

Cópia de Twin

Este tempo poderia ser eliminado, se empregado um esquema de otimização do código de consistência, de tal forma que a *Thread* Servidora forneça ao protocolo o *buffer* onde é recebida a resposta de página. Esta seria reutilizada como a *twin* da própria página, evitando a cópia da mesma.

Geração de *diffs*

É proposto um esquema onde a Placa-Folha seria responsável pelo cálculo de *diffs*, aliviando o processador para a realização de outras tarefas.

O modelo pressupõe a cópia da página e de sua *twin* para a placa através de DMA ($2 \times t_{pag}$). A geração do *diff* seria realizado concomitantemente com a cópia da *twin*, e despejado na memória principal em um tempo proporcional ao tamanho do *diff* (tam_{diff}/DMA_{SPEED}). O cálculo leva em conta somente páginas com falha de acesso para escrita remota do tipo *hit*. Assim é possivel projetar o impacto do esquema de migração pelo critério descrito nos trabalhos anteriores.

Compressão de WNs

Este componente mede a otimização possibilitada pela codificação compacta de WNs consecutivos para a disseminação de WNs. Computado sobre o tempo de disseminação de WNs e o fator de compressão de WNs, coletados da execução das aplicações. É descontado o tempo de *overhead* de transmissão sobre a Rede de Sincronização, por já ter sido utilizado em métrica anterior.

Código de Paranóia

Código de segurança adicional é inserido no Protocolo Sincro, para verificar o acesso a regiões válidas de memória, conferência extra de ordenamento de *timestamps*, entre outras coisas. Alguns destes tempos são instrumentados, e agregados em um contador, que representa possível melhorias de tempo do código.

A.2 Análises de projeções

A Figura A.1 mostra as projeções calculadas para as aplicações utilizadas no Capítulo 4. Novamente, os tempos são normalizados pela medida do tempo sequencial das aplicações. São apresentados os tempos médios de 5 execuções de cada aplicação, e são geradas barras individuais para cada um dos processadores. Acima das barras, é apresentado o valor de aceleração com o protocolo Sincro, e entre

parênteses o valor da aceleração estimada, caso as técnicas de otimização descritas neste apêndice fossem empregadas.

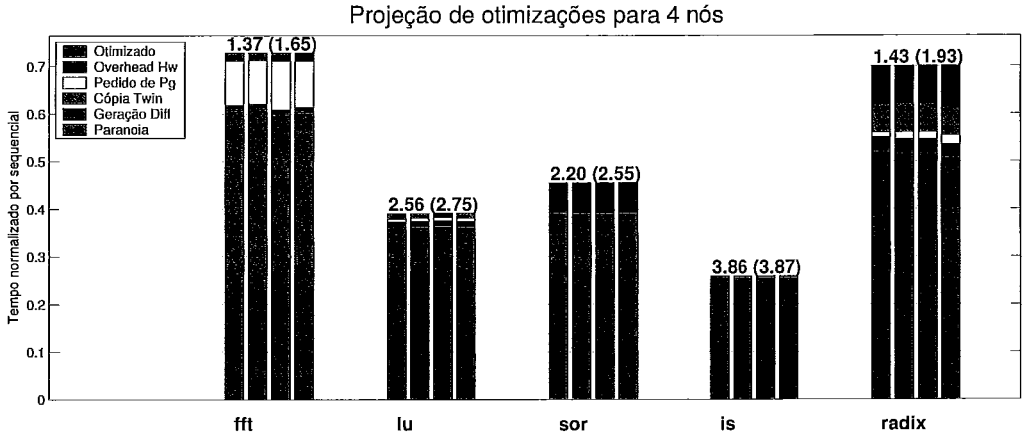


Figura A.1: Projeção de otimizações no protocolo

De acordo com a figura, podemos observar que a aplicação FFT se beneficiaria bastante de um esquema de suporte de pedido de páginas. SOR ganharia com o "desligamento" dos códigos de segunrança, e com a melhoria da camada de *firmware* da Rede de Sincronização, que atualmente obriga o Protocolo de Comunicação, implementado em *software*, a realizar muitas operações desnecessárias. O tempo de IS seria pouco afetado por quaisquer das melhorias analisadas. Por fim, Radix ganharia principalmente com a melhoria na alocação de *twins*, suporte a cálculo de *diffs*, e código de segurança, mostrando menores mas significativas influências na otimização do *firmware* da Rede e no suporte a pedido de páginas.

Símbolo	Significado	Valor
t_{clock}	Velocidade do relógio dos tranmissores e receptores	100 MHz
t_{send}	Tempo de processador para transmitir uma mensagem	0,5 μs
$t_{ask\ \ new}$	Novo tempo de requisição de página	$t_{int} + t_{lat} = 3{,}27\ \mu$s
t_{int}	Tempo de Serviço de Interrupção	1 μs
t_{lat}	Latência da Rede de Sincronização	2,27 μs
t_{pag}	Tempo de Cópia de páginas para a Placa-Folha por DMA	31 μs †
tam_{diff}	Tamanho médio dos *diffs* para a aplicação	de 0 a 4kB
DMA_{SPEED}	Velocidade de transferência de dados entre a Rede de Sincronização e a memória principal	133 MB/s †

†Assumindo velocidade de transferencia do barramento PCI de 133 MB/s.

Tabela A.1: Simbolos e Valores utilizados nas equações da Seção A.

Apêndice B

Programação DSM

Este apêndice contém o pseudo-código do programa Jacobi, escrito para a biblioteca ThreadMarks. Este código foi extraido de Amza [2].

Código Fonte B.1: Pseudo-código do programa Jacobi

```
#define M 1024
#define N 1024

float **grid;              /* vetor compartilhado */
float scratch[M][N];       /* vetor privado */
main()
{
  Tmk_Startup();  /* Dispara processos e inicializa ambiente DSM */

  if ( Tmk_proc_id == 0 ) {
    grid = Tmk_malloc(M*N*sizeof(float)); /* variavel global */

    initialize_grid();
  }

  Tmk_barrier(0);
```

```
  length = M/Tmk_nprocs;
  begin = lenght * Tmk_proc_id;
  end   = lenght * (Tmk_proc_id + 1);

  for ( number of iterations ) {
    for ( i = begin ; i < end ; i++ )
      for ( j = 0 ; j < N ; j++ )
        scratch[i][j] = (grid[i-1][j] + grid[i+1][j] + \
          grid[i][j-1] + grid[i][j+1])/4;

    Tmk_barrier(1);

    for ( i = begin ; i < end ; i++ )
      for ( j = 0 ; j < N ; j++ )
        grid[i][j] = scratch[i][j];

    Tmk_barrier(2);
  }

}
```

Referências Bibliográficas

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Peter J. Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Threadmarks: Shared memory computing on networks of workstations. In *IEEE Computer*, pages 18–28, 1996.

[3] L. Arantes, B. Folliot, and P. Sens. A customized logical clock for timestamp-based relaxed consistency DSM systems. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, 1999.

[4] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding communication latency and coherence overhead in software DSMs. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 198–209, 1996.

[5] A. Bilas, C. Liao, and J. Singh. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proc. of 26th ISCA*. 1999.

[6] Angelos Bilas, Dongming Jiang, and Jaswinder Pal Singh. Accelerating shared virtual memory via general-purpose network interface support. *ACM Trans. Comput. Syst.*, 19(1):1–35, 2001.

[7] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li. Protected, user-level DMA for the SHRIMP network interface. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, 1996.

[8] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-level Barrier over Myrinet/GM. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[9] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, 1995.

[10] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communications. In *Distributed Computing*, 1995.

[11] Compaq Corporation, Intel Corporation, and Microsoft Corporation. Virtual Interface Architecture specification version 1.0. In *http://www.viaarch.org*, 1997.

[12] Sotério Ferreira de Souza. Mecanismos de sincronização distribuída para Clusters implementados em hardware. *Tese de Mestrado, UFES, 2005.*

[13] H.G. Dietz, R. Hoare, and T. Mattox. A fine-grain parallel architecture based on barrier synchronization. *icpp*, 01:0247, 1996.

[14] Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *Proceedings, 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 153–159, Los Alamitos, Calif., April 1999. IEEE Computer Society.

[15] C. J. Fidge. A limitation of vector timestamps for reconstructing distributed computations. *Information Processing Letters*, 68(2):87–91, 1998.

[16] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *PODS '82: Proceedings of*

the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems, pages 70–75, New York, NY, USA, 1982. ACM Press.

[17] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

[18] R.B. Gillet. Memory Channel Network for PCI. In *IEEE Micro*, number 16(1):12-18, February 1996.

[19] Kiyoshi Hayakawa and Satoshi Sekiguchi. Design and implementation of a synchronization and communication controller for cluster computing systems. *hpc*, 01:76, 2000.

[20] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, 1996.

[21] T.A. Johnson and R.R. Hoare. Cyclical cascade chains: a dynamic barrier synchronization mechanism for multiprocessor systems. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, 2001.

[22] S. Kini, J. Liu, J. Wu, P. Wyckoff, and D. Panda. Fast and scalable barrier using RDMA and multicast mechanisms for infiniband-based clusters. In *In Euro PVM/MPI Conference, Venice, Italy*, September 2003.

[23] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[24] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. 1998.

[25] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[26] Muralidharan Rangarajan and Liviu Iftode. Software distributed shared memory over virtual interface architecture: Implementation and performance. In *Proceedings of the 4th conference on 4th Annual Linux Showcase and Conference, Atlanta - Volume 4*, pages 341–352, 2000.

[27] R.M. da Silva, L. Whately, M. Lobosco, and C.L. Amorim. Memória compartilhada distribuída para redes UDP/IP: Implementação e avaliação. In *Proceedings of the VIV Worshop on High Performance Computer Systems, WSCAD*, pages 9–16, 2003.

[28] A.F. De Souza and C.L. de Amorim. Relógio global distribuído para Clusters de computadores. *Patente USTPD 7240230.*

[29] R. Traylor and D. Dunning. Routing chip set for the Intel Paragon parallel supercomputer. In *Proc. of Hot Chips'92 Symposium*, 1992.

[30] Igor A. Zhuklinets and Denis A. Khotimsky. Logical time in distributed software systems. In *Programmirovanie, Vol. 28, No. 3, 2002. Reproduced in ACM Programming and Computer Software, 2002: 174-184.*