Técnicas para Detecção Dinâmica de Padrões na Passagem de *Locks* em *Software DSMs**

Cristiana Bentes Seidel†‡, Claudio Luis de Amorim†, Ricardo Bianchini†

†Programa de Eng. Sistemas e Computação
COPPE/UFRJ
Rio de Janeiro

‡Depto de Engenharia de Sistemas
UERJ
Rio de Janeiro

seidel@cos.ufrj.br

Resumo

Sistemas de memória compartilhada distribuída, com coerência de dados mantida por software, têm se mostrado uma alternativa de baixo custo para programação no modelo de memória compartilhada. Entretanto, tais sistemas, denominados *software DSMs*, apresentam desempenho limitado devido principalmente ao grande *overhead* gerado pelo protocolo de coerência empregado. Nesse trabalho apresentamos técnicas para previsão dinâmica da passagem de *locks* entre os processadores. Essas técnicas podem ser utilizadas por *software DSMs* para sobrepor o *overhead* de comunicação e a computação útil. Mais especificamente, se um processador p pode prever o próximo processador q a adquirir a seção crítica em que ele se encontra, ele pode enviar antecipadamente suas mensagens de coerência a q e assim esconder a latência dessas mensagens com computação útil em q. Realizamos simulações com dois *software DSMs* diferentes e, para um conjunto de seis aplicações reais, mostramos que podemos estabelecer dinamicamente um pequeno conjunto de processadores que são os mais prováveis "próximos *acquirers*" de uma seção crítica. Os resultados obtidos em nossos experimentos indicam que as nossas técnicas alcançam taxas de acerto altas, variando de 60% a 97% do total de eventos de *lock*.

Abstract

Software-coherent distributed-shared memory systems (software DSMs) stand out as a very low cost approach to shared-memory computing. However, several applications running on these systems suffer high communication overheads that limit performance. In this work we introduce techniques that can dynamically predict lock acquiring order and, therefore, can be used to hide communication overheads behind useful computation. More specifically, if a processor p can predict the next acquirer q of the critical section it is computing in, it can send coherence messages to q in advance, thus overlapping communication overhead with useful computation on q. We simulated six real applications on two different software DSMs, showing that it is possible to guess a small subset of processors that will likely include the next acquirer. We obtained a high percentage of correct guesses, ranging from 60 to 97% of the lock events.

* Esse trabalho foi parcialmente financiado por Finep e CNPq.

1 Introdução

Sistemas com memória compartilhada distribuída (*Distributed Shared Memory - DSM*), com coerência de dados mantida por software, têm se mostrado uma alternativa de baixo custo para programação no modelo de memória compartilhada. Entretanto, tais sistemas, denominados *software DSMs*, apresentam desempenho limitado devido principalmente ao grande *overhead* gerado pelo protocolo de coerência empregado.

Software DSMs baseados em modelos de consistência relaxada de memória tentam reduzir esse *overhead* atrasando e/ou restringindo ao máximo a comunicação e as transações de coerência. No sistema Munin [5], por exemplo, as atualizações a dados compartilhados realizadas por um processador p são enviadas a todos os outros processadores numa única mensagem, no momento que p executa uma operação *release*. TreadMarks [8] atrasa as operações de coerência ainda mais, até a próxima operação *acquire*. O sistema Midway [3] também atrasa a comunicação e as operações de coerência até o próximo *acquire*, mas restringe essas operações aos dados associados ao *lock*.

Os protocolos empregados nesses sistemas, entretanto, ainda apresentam overhead alto: em Munin, as atualizações realizadas em dados compartilhados são enviadas para **todos** os processadores que possuem cópia dos dados; em TreadMarks o processador tem que esperar as atualizações chegarem quando ocorre um *page-fault*; já em Midway, uma operação *acquire* só pode completar quando as atualizações do dado associado ao *lock* estiverem localmente consistentes.

Nosso trabalho se baseia na observação que parte desse *overhead* pode ser escondido se o protocolo puder prever dinamicamente a ordem de aquisição dos locks. Estamos propondo algumas técnicas para realizar dinamicamente a previsão da ordem de ocorrência dos eventos de sincronização. Mais especificamente, nossas técnicas determinam qual o próximo processador a executar uma operação *acquire* numa dada seção crítica. Embora a ocorrência dos eventos de sincronização seja totalmente dependente do comportamento dinâmico da aplicação, mostramos que a previsão é possível, com uma taxa alta de acerto, através do histórico dos eventos de sincronização anteriores e da inserção de algumas instruções adicionais no código da aplicação.

Como os sistemas *software DSMs* existentes não possuem qualquer mecanismo que permita esse tipo de previsão, desenvolvemos técnicas que são gerais o suficiente para serem aplicadas em qualquer sistema. Nossas técnicas podem ser utilizadas para esconder overhead de comunicação e de transações de coerência da seguinte forma: se o processador p sabe, no momento que está executando um *acquire* na variável de sincronização s, que o próximo *acquirer* de s é o processador q, ele pode, portanto, enviar **antecipadamente** as mensagens de coerência para q, conforme descrito em [1].

O restante desse artigo é organizado da seguinte forma. Na próxima seção apresentamos as três técnicas que suportam nosso mecanismo de previsão. Na seção 3 descrevemos as aplicações e os *software DSMs* simulados utilizados em nossos experimentos. Na seção 4 apresentamos os resultados das simulações com as taxas de acerto obtidas em cada aplicação. E, finalmente, na seção 5 concluímos o trabalho identificando possíveis trabalhos futuros.

2 Técnicas de Previsão do Próximo *Acquirer*

Desenvolvemos três técnicas diferentes para previsão do próximo *acquirer* de determinada variável de sincronização. Na primeira delas, estamos considerando um sistema em que um processador que executa uma operação *acquire* em uma seção crítica ocupada é bloqueado, numa fila FIFO, até que a seção crítica seja liberada por um outro processador. Quando essa fila existe, nossa técnica de previsão, chamada de **fila de espera**, determina que o próximo *acquirer* é exatamente o primeiro processador da fila.

Aplicações paralelas, entretanto, nem sempre apresentam fila de espera. A segunda técnica de previsão, chamada **fila virtual**, tenta antecipar a criação de uma fila de espera por uma determinada seção crítica. A idéia se baseia em inserir no código da aplicação, alguns passos antes da instrução *acquire* propriamente dita, uma instrução especial, que chamamos de "aviso-antecipado", que tem a função de informar que o processador vai, alguns passos adiante, executar um *acquire*. O processador, obviamente, não é bloqueado, mas sua identificação é inserida numa fila virtual de espera pela seção crítica. Os componentes dessa fila seriam, então, bons candidatos a próximo *acquirer* da seção crítica.

A inserção dessa instrução aviso-antecipado pode ser facilmente realizada pelo compilador. A precisão dessa técnica, entretanto, depende da posição, dentro do código da aplicação, em que essa inserção é realizada. Se o aviso-antecipado ficar muito próximo à operação *acquire*, não haverá tempo suficiente para a formação da fila virtual. Enquanto que, se o aviso-antecipado ficar muito distante da operação *acquire* propriamente dita, alguns eventos como *page-faults* podem ocorrer entre o aviso e o *acquire* e podem fazer com que a ordem dos processadores na fila virtual não reflita a ordem efetiva de pedidos de *acquire*.

Como nem sempre podemos contar com a fila virtual, desenvolvemos uma terceira técnica de previsão chamada de **afinidade**. A técnica de afinidade tenta descobrir o próximo *acquirer* de uma variável de sincronização através de um histórico das passagens de *locks* anteriores. Observamos que, durante a execução de aplicações paralelas, se um processador p executa um *acquire* em s, o próximo *acquirer* de s está em um conjunto bastante limitado dos processadores do sistema, chamado de **conjunto de afinidade** de p. O conjunto de afinidade de p em geral varia muito pouco durante a execução da aplicação, o que significa que ele pode ser facilmente determinado em tempo de execução.

Para determinar o conjunto de afinidade de um processador em relação a uma variável de sincronização, utilizamos uma matriz A, onde A_{ij} representa o número de vezes que o processador j foi o próximo *acquirer* da variável após o processador i. Utilizamos uma heurística simples baseada somente na frequência de passagens de *locks* para analisar o passado, já que ela se mostrou bastante satisfatória em nossas simulações.

O cálculo do conjunto de afinidade de um determinado processador p em relação à variável de sincronização s é realizado da seguinte forma. O conjunto de afinidade de p contém os processadores j, tal que A_{pj} é 60% acima da média das afinidades A_{ij} de todos os processadores.

A previsão do próximo *acquirer* de uma variável s é baseada na determinação do **conjunto de atualização** do processador. O conjunto de atualização de um

processador p em relação à variável s, U_p, contém os processadores que são os mais prováveis "próximos *acquirers*" de s depois de p. O tamanho do conjunto de atualização de um processador é pré-determinado pelo usuário. O algoritmo abaixo mostra como esse conjunto é formado para um processador p em relação a uma variável de sincronização s.

1. Se há fila de espera: $U_p = q$ (q é o primeiro da fila de espera de s);Fim.
2. Os processadores do conjunto de afinidade de p são inseridos em U_p;
3. Se U_p não está completo, incluir os processadores j que constituem a interseção entre os componentes da fila virtual e os processadores com $A_{pj} > 0$;
4. Se U_p ainda não está completo, inserir primeiro os processadores da fila virtual e depois os processadores j tal que $A_{pj} > 0$ até que U_p esteja completo; Fim.

3 Metodologia

Para avaliar as técnicas descritas na seção anterior, utilizamos seis aplicações reais que apresentam um número significativo de eventos *acquire* por variável de sincronização. As aplicações foram executadas em dois sistemas *software DSM* simulados detalhadamente, TreadMarks[8] e Directory. Cada simulador é composto de duas partes. A primeira é um *front-end*, Mint[10], que simula a execução das instruções das aplicações. A segunda é um *back-end* que simula o sistema de memória. A arquitetura simulada é composta de 16 nós de processamento. Cada nó é formado por um processador, um *write buffer*, uma memória cache de dados de primeiro nível diretamente mapeada, uma memória local e um roteador de rede em malha que utiliza roteamento *wormhole*. Os parâmetros utilizados em nossas simulações correspondem aos de um sistema real de uma rede de estações de trabalho e podem ser encontrados em [4].

3.1 Sistema TreadMarks

O sistema TreadMarks, desenvolvido pela Rice University, utiliza a página como unidade de consistência e esquema de invalidação para o recebimento das atualizações nos dados compartilhados. O modelo de consistência de memória empregado é o *Lazy Release Consistency* [7]. Para determinar quais atualizações um processador deve receber quando executa um *acquire*, o sistema divide a execução do programa em intervalos e computa um vetor de *timestamps* para cada intervalo. Esse vetor descreve uma ordem parcial entre os intervalos de diferentes processadores.

Ao executar um *acquire* um processador p envia uma mensagem para o processador *manager* da variável de sincronização. O *manager* avança esse pedido para o último *acquirer* da seção crítica. Este compara seu vetor de *timestamp* com o de p e envia para p os respectivos avisos de atualização das páginas (*write-notices*). O processador p invalida as páginas para as quais recebeu *write-notices*. Quando ocorre um *page-fault* numa certa página, o processador consulta sua lista de *write-notices* e requisita as modificações feitas na página para os devidos processadores.

A instrumentação realizada no simulador do TreadMarks para avaliar nossas técnicas de previsão foi a seguinte: o processador *manager* da variável de sincronização mantém a fila de espera, a fila virtual, e a matriz A de afinidade, i.e., ele é o responsável pelo cálculo do conjunto de atualização de um processador.

3.2 Sistema Directory

O protocolo Directory, adaptado do sistema CASHMERe [9], utiliza o modelo de consistência *Release Consistency* [6]. Nesse protocolo, há um processador *home* para cada página do sistema. Quando um processador sofre um *write-fault* numa página k ele cria uma cópia (*twin*) da página e envia *write-notices* para todos os processadores que possuem cópia de k. Quando o processador executar uma operação *release* na saída de uma seção crítica, as modificações realizadas durante a seção crítica são coletadas, na forma de *diffs*, e enviadas para os processadores *home* das respectivas páginas.

O próximo processador a executar um *acquire* invalida todas as páginas para as quais ele recebeu *write-notices* e, quando ocorrer um *page-fault*, a versão atualizada da página é buscada do processador *home*.

Mantivemos a mesma instrumentação utilizada no Treadmarks para o simulador do Directory, e para tal foi necessária a introdução de um processador *manager* para cada variável de sincronização, o que não causou nenhuma alteração no funcionamento básico do protocolo.

3.3 Aplicações

Utilizamos em nossos experimentos cinco aplicações do conjunto Splash-2 [11] de *benchmarks*, são elas Barnes, Cholesky, Ocean, Water-spatial e Water-nsquared. E uma aplicação do conjunto NAS [2], Appbt. As aplicações são do tipo SPMD, estão escritas na linguagem C e utilizam macros ANL para implementar operações paralelas como criação de processos, alocação de memória compartilhada e operações *lock/unlock* e barreira para sincronização entre processos.

	Barnes	Cholesky	Ocean	Water-nsq	Water-sp	Appbt
Entrada	16K	tk15.O	258x258	512	512	16x16x16
No. variaveis	78	35	4	518	6	100
No. eventos	69125	27915	3328	18704	305	19000
Eventos/var	886	797	832	36	51	190

Tabela 1: Características dos Eventos de Sincronização

Na Tabela 1 temos quantificados, para cada uma das aplicações, o tamanho da entrada, o número de variáveis de sincronização, o número total de eventos *acquire/release* realizados e o número médio de eventos *acquire/release* por variável de sincronização.

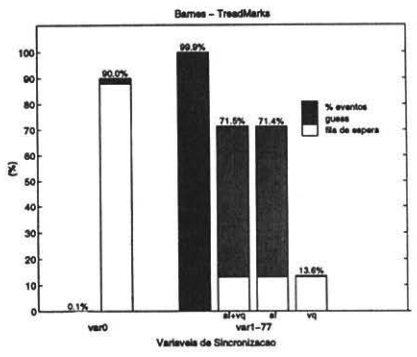

Figura 1: Taxas de Acerto para Barnes, ca=2, TreadMarks

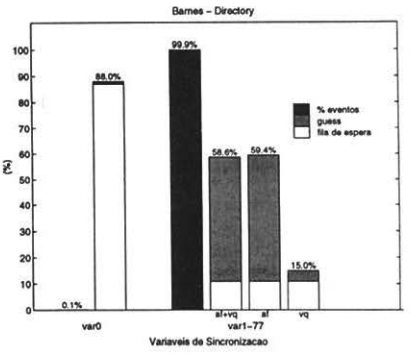

Figura 2: Taxas de Acerto para Barnes, ca=2, Directory

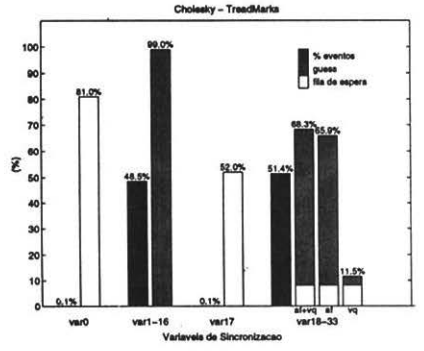

Figura 3: Taxas de Acerto para Cholesky, ca=2, TreadMarks

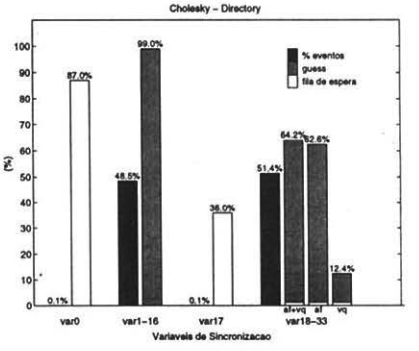

Figura 4: Taxas de Acerto para Cholesky, ca=2, Directory

4 Resultados

Na avaliação de nossas técnicas de afinidade, variamos o tamanho do conjunto de atualização (*ca*) de 1 a 3. Sendo que, por motivos de espaço, vamos apresentar aqui somente os gráficos para $ca = 2$. Consideramos um **acerto** quando o processador que executou o *acquire* estava no conjunto de atualização do último *acquirer* da seção crítica. Para cada variável de sincronização s calculamos a taxa de acerto H como sendo a porcentagem de vezes, no número total de eventos *acquire/release*, que ocorre um acerto.

As taxas de acerto obtidas para cada aplicação estão descritas nos gráficos das Figuras 1 a 12. O eixo horizontal representa as variáveis de sincronização. O eixo vertical representa a porcentagem de acerto para cada variável, sendo que a barra mais escura mostra a porcentagem de eventos *acquire* da variável, no número total de eventos *acquire* da aplicação.

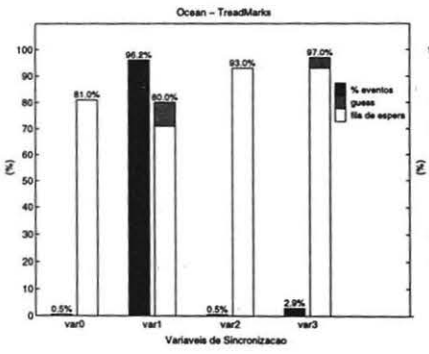

Figura 5: Taxas de Acerto para Ocean, ca=2, TreadMarks

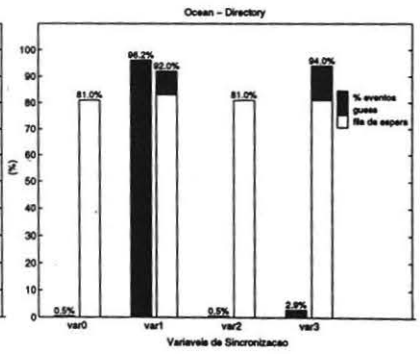

Figura 6: Taxas de Acerto para Ocean, ca=2, Directory

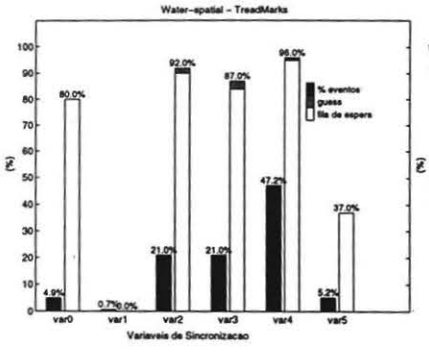

Figura 7: Taxas de Acerto para Water-spatial, ca=2, TreadMarks

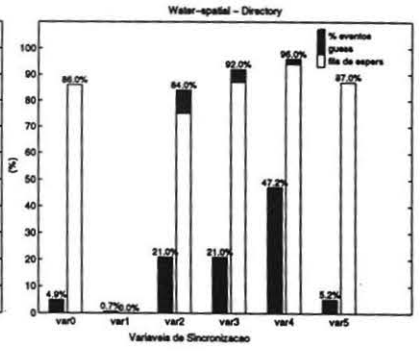

Figura 8: Taxas de Acerto para Water-spatial, ca=2, Directory

Para comparar as técnicas de fila virtual e de afinidade, chamadas **técnicas de *guess***, realizamos três tipos de simulações diferentes. Na primeira delas, chamada **af+vq**, o conjunto de atualização de um processador p, U_p, é obtido segundo o algoritmo descrito na seção 2.1. A segunda simulação, chamada **af**, foi realizada considerando o primeiro passo do algoritmo citado, e um segundo passo em que U_p é formado somente com os processadores j, tal que $A_{pj} > 0$. Na terceira, chamada **vq**, o primeiro passo também se matém, mas no segundo U_p é formado somente com elementos da fila virtual. Apresentamos nos gráficos as taxas de acerto para essas três simulações diferentes (af+vq, af e vq) somente para as variáveis de sincronização que apresentam grande peso na quantidade de eventos total. Nas aplicações Water-spatial e Ocean não houve necessidade de comparar afinidade e fila virtual porque o peso das técnicas de *guess* foi muito pequeno para todas as variáveis.

Para as aplicações que possuem uma grande quantidade de variáveis de sincronização, agrupamos algumas variáveis em um único ponto do gráfico. Foram

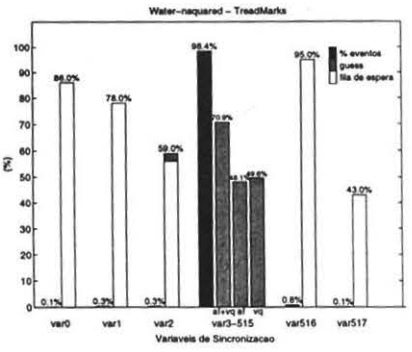

Figura 9: Taxas de Acerto para Water-nsquared, ca=2, TreadMarks

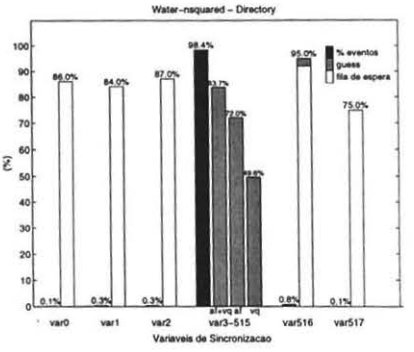

Figura 10: Taxas de Acerto para Water-nsquared, ca=2, Directory

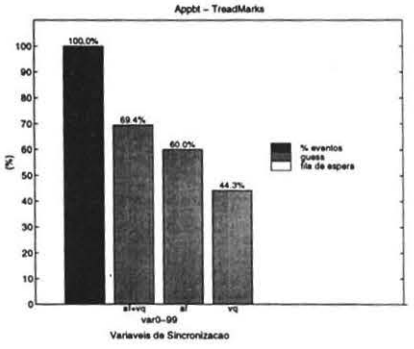

Figura 11: Taxas de Acerto para Appbt, ca=2, TreadMarks

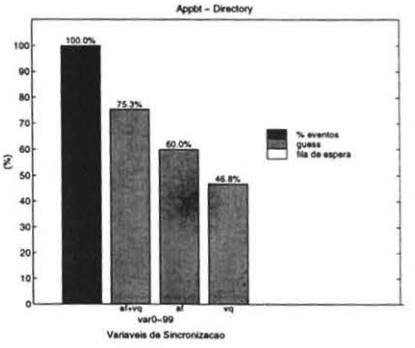

Figura 12: Taxas de Acerto para Appbt, ca=2, Directory

agrupadas num mesmo ponto variáveis de sincronização que têm alguma relação entre si, a taxa de acerto especificada corresponde à média, ponderada pela quantidade de eventos, das taxas de acerto de cada variável.

4.1 Análise dos Resultados

Conforme podemos observar nos gráficos, a eficiência das técnicas de previsão é alta. Nas aplicações Ocean e Water-spatial, a técnica **fila de espera** prevalece e é responsável pelas maiores taxas de acerto (acima de 90%). Nas aplicações Barnes e Cholesky, a técnica de **fila virtual** não teve bons resultados porque nessas aplicações a instrução aviso-antecipado só pode ser inserida algumas poucas instruções antes da operação de *acquire*, dessa forma raramente há formação da fila virtual. Já nas aplicações Water-nsquared e Appbt, a técnica **fila virtual** teve grande impacto na taxa de acerto total. Tanto nessas aplicações como nas aplicações Barnes e Cholesky,

a técnica **afinidade** foi bastante efetiva.

Sobre a aplicação Cholesky, há ainda um comentário importante. Embora tenhamos obtido uma taxa de 99% de acerto para as variáveis que protegem as listas de nós livres, ocorreu que cada processador só teve necessidade de buscar nós livres dentro da sua própria lista. Assim, cada variável desse grupo só é acessada por um único processador do sistema. Um esquema de previsão para esse grupo de variáveis, por mais efetivo que seja, não teria a menor utilidade num protocolo de coerência, uma vez que para essas variáveis não há necessidade de troca de mensagens de coerência.

Variando o tamanho do *ca*, observamos que aumentando de 1 para 2 elementos obtivemos grande diferença nas taxas de acerto. Nesse caso, as taxas de acerto aumentaram de um fator de 1.12 (Cholesky) a 1.25 (Appbt). O aumento do tamanho do *ca* de 2 para 3, entretanto, não apresentou grandes efeitos nas taxas de acerto, elas aumentaram de um fator de 1.01 (Water-nsquared) a 1.10 (Appbt). Portanto, em aplicações onde a técnica de afinidade é dominante um tamanho maior para *ca* é preferível.

Com relação à diferença das taxas obtidas para os dois protocolos, observamos que para a maioria das aplicações, considerando as variáveis com grande peso no total de eventos, essa diferença foi pequena. A diferença apresentada na aplicação Water-nsquared se deve ao fato dessa aplicação gerar poucos eventos por variável, assim qualquer alteração na ordem dos pedidos de *acquire* tem grande efeito no total da taxa de acerto. Na aplicação Barnes, a alteração na ordem dos eventos de sincronização leva à construção de uma árvore totalmente diferente.

5 Conclusões

Este trabalho mostrou técnicas para previsão do próximo *acquirer* de uma variável de sincronização. Testamos as três técnicas desenvolvidas em um conjunto de seis aplicações diferentes e mostramos que, com uma alta precisão, podemos estabelecer dinamicamente um pequeno conjunto de processadores que são os mais prováveis "próximos *acquirers*" de uma seção crítica.

Nossos experimentos mostraram que dependendo da aplicação, uma técnica é mais efetiva que outra, mas em todos os casos a combinação das três técnicas foi a que obteve melhores resultados, alcançando 70% de acerto ou mais para a maioria dos casos.

O tamanho do conjunto de atualização tem grande influência na taxa de acerto das previsões. Entretanto, nossos resultados revelaram que para tamanhos de *ca* maiores que 2 essa diferença diminui muito. O que nos leva a crer que este é um tamanho razoável para o conjunto de atualização.

A utilização de dois protocolos diferentes em nossas simulações mostrou que, embora os protocolos alterem a ordem dos eventos de sincronização e se baseiem em modelos de consistência distintos, as diferenças nas taxas de acerto obtidas pelos dois protocolos foram pequenas. O que mostra que nossas técnicas de previsão são bastantes robustas.

Um protocolo de coerência de memória de um *software DSM* pode, portanto, beneficiar-se das técnicas desenvolvidas nesse trabalho para esconder o *overhead* de troca de mensagens. Um processador poderia enviar com antecedência as atuali-

zações realizadas na sua seção crítica para os processadores do seu conjunto de atualização. Assim, evitaríamos *overheads* causados por espera de páginas ou pelo excesso de mensagens. Acabamos de concluir o protocolo AEC [1] que utiliza essas idéias.

Agradecimentos

Gostaríamos de agradecer a Leonidas Kontothanassis pelo auxílio na implementação dos simuladores e a Raquel G. Pinto pela ajuda durante a instrumentação dos mesmos.

Referências

[1] Amorim, C.L., Seidel, C.B. and Bianchini, R. "The Affinity Entry Consistency Protocol", Technical Report ES-388/96, COPPE/Sistemas, Maio 1996.

[2] Bailey, D. "The NAS Parallel Benchmarks", Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[3] Bershad, B.N., Zekauskas, M.J and Sawdon, W.A. "The Midway Distributed Memory System", *COMPCON* 1993.

[4] Bianchini, R., Kontothanassis, L., Pinto,R., De Maria, M. Abud, M, Amorim, C.L., "Hiding Communication Latency and Coherence Overhead in Software DSMs", *to appear in Proc of the 7th ASPLOS*, October 1996.

[5] Carter, J.B., Bennet, J.K. and Zwaenepoel, W. "Implementation and Performance of Munin", *Proc of the 13th SOSP*, October 1991, pp 152-164.

[6] Gharachorloo, K. Lenoski, D., Laudon, J., Gibbons, P., Gupta A. and Henessy, J.L. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors",*Proc of the 17th ISCA*, May 1990, pp 148-159.

[7] Keleher, P., Cox, A. and Zwaenepoel, W., "Lazy Release Consistency for Software Distributed Shared Memory", Technical Report TR91-168 Computer Science Department, Rice university, November 1991.

[8] Keleher, P., Dwarkadas, S., Cox, A. and Zwaenepoel, W., "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", *Proc of the USENIX Winter 94 Technical Conference*, January 1994, pp 17-21.

[9] Kontothanassis, L.I., and Scott, M.L. "Distributed Shared Memory for New Generation Networks", *Proc of the 2nd HPCA*, February 1996.

[10] Veenstra, J.E. and Fowler, R.J."Mint: A front end for effiecient simulation of shared memory multiprocessors", *Proc of the 2nd MASCOTS*, 1994.

[11] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., and Gupta, A. "The Splash-2 programs: Characterization and methodological considerations",*Proc of the 22nd ISCA*, May 1995, pp 24-36.