Avaliando a Técnica de Previsão Dinâmica da Passagem de *Locks* em Sistemas DSM

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

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

[‡]Depto de Engenharia de Sistemas
UERJ
Rio de Janeiro

{seidel,ricardo,amorim}@cos.ufrj.br

Resumo

Em sistemas DSM (*Distributed Shared Memory*) o *overhead* gerado pelo uso de seções críticas está diretamente relacionado ao tempo que um processador fica paralisado dentro de uma seção crítica, esperando por dados provenientes de outros processadores. Para evitar o tempo de espera por dados dentro de uma seção crítica, sem o custo adicional de sobrecarregar a rede com grande quantidade de mensagens, desenvolvemos em um trabalho anterior uma técnica, chamada LAP (*Lock Acquirer Prediction*), para prever a ordem de aquisição de um determinado *lock*. Com essa técnica podemos enviar seletivamente as atualizações dos dados compartilhados realizadas dentro da seção crítica para o próximo *acquirer* do *lock*.

Nesse trabalho mostramos uma avaliação do uso de LAP em sistemas DSM implementados em hardware e em software. Desenvolvemos dois protocolos, *Affinity Release Consistency* (ARC) para *hardware DSM* e *Affinity Entry Consistency* (AEC) para *software DSM*, e avaliamos o efeito de LAP no desempenho de um conjunto de três aplicações. No protocolo AEC, o uso de LAP obteve reduções de até 28% no tempo total de execução das aplicações testadas. Para o protocolo ARC, desenvolvemos um modelo analítico que provê um limite superior para o ganho de LAP. Mostramos que, em ARC, LAP pode reduzir o tempo de execução das aplicações em até 36%. Concluímos que, para ambos os sistemas, LAP mostrou-se bastante efetiva na redução do tempo gasto dentro da seção crítica obtendo ganhos significativos no tempo total de execução das aplicações.

Abstract

Critical sections are often a source of significant overhead in distributed shared-memory (DSM) systems, as they represent serialization points in parallel applications. In a previous paper we proposed and evaluated a technique, called Lock Acquirer Prediction *(LAP), intended to optimize critical sections by predicting the next acquirer of a lock at the time of the release, so that the acquirer can be updated even before requesting ownership of the lock.*

This work presents an evaluation of the LAP technique when applied to hardware and software-based DSM systems. We propose two protocols that apply LAP: the Affinity Release Consistency (ARC) protocol for hardware-based DSMs and the Affinity Entry Consistency (AEC) protocol for software-based DSMs. Our evaluation assesses the performance of LAP for these protocols running three different applications. Using simulation, we find that the use of LAP improves the performance of the AEC protocol by as much as 28%. Using an analytical model of the performance improvements achievable by LAP for the ARC protocol, we find that LAP can potentially reduce execution time by as much as 36%. We conclude that LAP is a useful technique for reducing critical section overheads and that it can significantly improve the performance of applications.

1 Introdução

O modelo de programação com memória compartilhada oferece uma forma bastante simples de comunicação entre processos: o uso de estruturas de dados compartilhadas. O acesso a essas estruturas, entretanto, deve ser feito de forma coordenada entre os processos, i.e., o programador deve incluir sincronização na aplicação para evitar que acessos conflitantes ocorram ao mesmo tempo. Um dos meios mais comuns de sincronização de processos paralelos é o uso de seções críticas para garantir a exclusão mútua de acesso a determinado conjunto de dados. Seções críticas são delimitadas por primitivas *lock/unlock* e representam uma grande fonte de *overhead*, devido à serialização imposta na sua execução. Quanto mais longa a seção crítica, maior a serialização imposta.

Sistemas DSM (*Distributed Shared Memory*) permitem o emprego do modelo de programação com memória compartilhada em um hardware escalável. Nesses sistemas, a coerência dos dados compartilhados pode ser mantida por hardware ou por software através de protocolo de invalidação ou de atualização. Em sistemas DSM baseados em protocolo de invalidação, o tempo que um processador fica paralisado dentro de uma seção crítica, esperando por dados provenientes de outros processadores, pode agravar ainda mais o problema da serialização. Já em protocolos baseados em atualizações, o processador geralmente não fica parado dentro de uma seção crítica esperando por dados, ele recebe as devidas atualizações antecipadamente. Em compensação, o tráfego desnecessário gerado na rede pode afetar substancialmente o tempo de se adquirir e liberar uma seção crítica.

Para evitar o tempo de espera por dados dentro de uma seção crítica sem o custo adicional de sobrecarregar a rede com grande quantidade de mensagens, desenvolvemos uma técnica, chamada LAP (*Lock Acquirer Prediction*) [10], para prever a ordem de aquisição de um determinado *lock*. Com essa técnica podemos, num sistema DSM, enviar seletivamente as atualizações dos dados compartilhados realizadas dentro da seção crítica para o próximo *acquirer* do *lock*.

Nesse trabalho mostramos uma avaliação do uso de LAP em sistemas DSM implementados em hardware e em software. Sistemas *software DSM* são mais baratos porque utilizam o próprio hardware de memória virtual da máquina para manter a coerência dos dados. Entretanto, têm desempenho bastante limitado devido ao *overhead* gerado pelo protocolo de coerência empregado. Desenvolvemos um protocolo para *software DSM* baseado na técnica de LAP e mostramos que o uso de LAP pode reduzir bastante o *overhead* do sistema, melhorando o desempenho de aplicações em até 28%.

Sistemas *hardware DSM* apresentam *overhead* bem menor, mas mesmo assim o tempo de sincronização imposto por operações *lock/unlock* pode degradar o desempenho de algumas aplicações. Nosso trabalho apresenta um estudo analítico do impacto do *overhead* de espera por dados no desempenho de seções críticas. Através desse estudo podemos mostrar o potencial de ganho da técnica LAP num *hardware DSM*. Apresentamos um protocolo para *hardware DSM* que permite o emprego de LAP de forma simples e barata, e avaliamos o modelo proposto em um conjunto de três aplicações diferentes. Mostramos que LAP pode reduzir o tempo de execução das aplicações em até 36%.

O restante desse artigo é organizado da seguinte forma. Na seção 2 apresentamos

a técnica *Lock Acquirer Prediction* (LAP). Na seção 3 mostramos o protocolo *Affinity Release Consistency* (ARC) para *hardware DSM*, o qual utiliza LAP para reduzir o tempo gasto dentro de seções críticas. Em seguida, apresentamos a modelagem analítica do ganho de LAP em sistemas *hardware DSM* e avaliamos o comportamento de três aplicações diferentes, segundo o modelo desenvolvido. Na seção 4 descrevemos o protocolo *Affinity Entry Consistency* (AEC) para *software DSM* que foi desenvolvido baseado na técnica LAP. O protocolo foi simulado e mostramos o efeito da técnica LAP no desempenho das aplicações. A seção 5 apresenta os trabalhos relacionados e na seção 6 concluímos o nosso trabalho.

2 A Técnica *Lock Acquirer Prediction*

A técnica LAP provê meios para previsão do próximo *acquirer* de um determinado *lock*. A primeira técnica de previsão, chamada **fila de espera**, é a mais simples. Ela se baseia no fato de que a fila FIFO de processadores esperando por um determinado *lock* é uma perfeita descrição da ordem de aquisição do *lock*. Então, quando há contenção pelo *lock*, LAP determina que o próximo *acquirer* é exatamente o primeiro processador da fila de espera.

Quando não há fila de espera, LAP se baseia numa técnica de previsão chamada de **afinidade**. Essa técnica foi desenvolvida a partir da observação de que existem determinados padrões nas passagens de *locks*. Ou melhor, um processador p tende a passar o *lock* para um conjunto bem limitado de processadores, chamado de conjunto de afinidade de p.

A técnica de afinidade tenta descobrir o próximo *acquirer* de um *lock* através de um histórico das passagens de *locks* anteriores. Para cada variável de sincronização s, utilizamos uma matriz A, onde A_{ij} representa o número de vezes que o processador j foi o próximo *acquirer* de s após o processador i. Quando um processador p está liberando um determinado *lock*, a técnica de afinidade prevê que o próximo *acquirer* desse *lock* deve ser o processador i tal que A_{pi} tenha o maior valor.

3 O Protocolo *Affinity Release Consistency*

Nessa seção apresentamos o protocolo *Affinity Release Consistency* (ARC) para *hardware DSM*. ARC consiste de uma versão modificada (para suportar a técnica LAP) do protocolo utilizado no sistema DASH [8] que é baseado no modelo de consistência *Release Consistency* [4]. Em seguida, vamos analisar o potencial de ganho de desempenho da técnica LAP num sistema *hardware DSM*.

3.1 Descrição do Protocolo

A previsão do próximo *acquirer* de uma seção crítica no protocolo ARC ocorre tal qual a descrição de LAP feita anteriormente. Quando o processador p deixa uma seção crítica s, o próximo *acquirer* de s é o primeiro processador da fila de espera ou, caso a fila esteja vazia, o processador que tem maior afinidade (A_{pi}) com p. LAP foi implementada em *hardware DSM* de forma bem simples, com suporte das rotinas

de sincronização, de modo a exigir hardware adicional mínimo. Sua implementação em software é, entretanto, bem mais complexa, conforme veremos mais tarde.

O protocolo ARC utiliza a linha de cache como unidade de coerência e esquema de diretório para manutenção de informações de coerência. A cada linha de cache está associado um processador *home* que mantém informações de diretório sobre a linha. Somente um único processador pode atualizar uma linha num determinado instante, este processador é chamado *owner* da linha.

Para dados acessados fora de seções críticas, ARC utiliza protocolo de invalidações semelhante ao utilizado no sistema DASH. Nesse protocolo, pedidos de leitura e escrita que não podem ser satisfeitos localmente são enviados ao processador *home*. Quando o pedido é de escrita, o *home* se responsabiliza por enviar invalidações para todos os processadores que têm cópia da linha em suas caches e estes enviam confirmação da chegada da mensagem de invalidação ao processador que fez o pedido de escrita. Quando o pedido é de leitura, se o *home* tem cópia válida da linha, ele mesmo fornece a linha; caso algum processador tenha escrito na linha, o *home* se encarrega de avançar o pedido para o processador *owner* da linha que pode, então, fornecer uma cópia atualizada.

Dados acessados dentro de seções críticas são mantidos coerentes através de protocolo de atualização, onde as atualizações são enviadas seletivamente para o próximo *acquirer* previsto da seção crítica. Para tal, num sistema com n processadores, as rotinas de sincronização devem manter, para cada seção crítica s, as seguintes informações:

- uma fila FIFO contendo a identificação dos processadores que estão esperando pela seção crítica;
- a identificação (r) do último processador a liberar a seção crítica;
- uma matriz $A : n \times n$, onde A_{ij} representa o número de vezes que o processador i passou o *lock* para o processador j;
- para cada linha i da matriz dois inteiros que correspondem ao índice (ind_i) e valor (val_i) de A_{ij}, tal que A_{ij} é máximo;

O valor de r deve ser alterado cada vez que for executada uma operação *unlock*. Quando um processador p executa uma operação *lock* na seção crítica s, se s estiver ocupada, p é inserido na fila de espera. Caso a seção crítica esteja liberada, o valor A_{rp} é incrementado. Se A_{rp} for maior que val_r, então val_r deve ser atualizado com o valor de A_{rp} e ind_r deve ser atualizado com p. A partir dessas estruturas, a aplicação de LAP é imediata; quando p liberar a seção crítica, a identificação do próximo *acquirer* da seção crítica s está na primeira posição da fila de espera, ou (se a fila estiver vazia) está armazenada em ind_p.

Todas as escritas realizadas dentro de seção crítica são enviadas para os processadores *home* das respectivas linhas. Escritas realizadas em palavras diferentes da mesma linha são enviadas de forma agrupada. Junto com os dados escritos o processador envia para o *home* a identificação do próximo *acquirer* previsto da seção crítica. O processador *home* se encarrega de re-enviar as escritas ao próximo *acquirer* e de enviar invalidações para os processadores que têm cópia válida da linha, mas não foram designados como "próximo *acquirer*".

Note que não basta enviar apenas as modificações realizadas no dado para o próximo *acquirer*, já que ele pode ter tido a linha invalidada ou substituída anteriormente e, portanto, não tem como aplicar as modificações recebidas. Uma forma simples de resolver esse problema seria fazer com que o *home* envie a linha de cache inteira para o próximo *acquirer*. Otimizações aqui são possíveis, mas devem ser deixadas para uma versão mais avançada do protocolo.

Em multiprocessadores como FLASH [7], o protocolo ARC não requer nenhum hardware especial; tudo pode ser feito em software através do controlador MAGIC, que é programável e responsável por suportar operações do protocolo de coerência. Em máquinas como DASH há necessidade de alterações no hardware por causa das mudanças na lógica do protocolo e nas entradas do diretório.

3.2 Modelagem de Desempenho para LAP em ARC

Para avaliar o desempenho de LAP em sistemas *hardware DSM*, desenvolvemos um modelo analítico que provê um limite superior para o ganho fornecido por LAP.

Assumindo que o *hardware DSM* utiliza protocolo baseado em invalidações, o ganho total da técnica LAP é modelado a partir do ganho obtido pela redução nas falhas na cache ocorridas dentro de uma seção crítica.

Assim, o tempo gasto dentro de uma seção crítica s (TD_s) para um processador pode ser modelado como a soma dos tempos gastos com: espera para entrar na seção crítica ($wait_s$); computação ($busy_s$); falhas ocorridas em dados privados (pm_s); falhas em dados compartilhados (sm_s); e espera pelo esvaziamento do *write-buffer* (wb_s). Conforme descrito na equação 1:

$$TD_s = wait_s + busy_s + pm_s + sm_s + wb_s; \quad (1)$$

LAP atua na redução das falhas em dados compartilhados ocorridas dentro de uma seção crítica, essa redução tem efeito direto no parâmetro sm e no parâmetro $wait$, porque se o tempo gasto na seção crítica diminui, a espera pelo *lock* também diminui.

Portanto, num sistema que utiliza LAP, o tempo gasto dentro de seção crítica s (TD_s^{lap}) para cada processador é modelado de forma diferente:

$$TD_s^{lap} = wait_s^{lap} + busy_s + pm_s + wb_s + sm_s^{lap}; \quad (2)$$

Onde os parâmetros $wait_s^{lap}$ e sm_s^{lap} representam respectivamente:

$$wait_s^{lap} = qs_s \times (busy_s + pm_s + wb_s + sm_s^{lap});$$

$$sm_s^{lap} = sm_s \times fail_s^{lap};$$

Quando se utiliza LAP, o tempo de espera na fila ($wait_s^{lap}$) é modelado como o número médio de processadores esperando pela seção crítica (qs_s) multiplicado pelo tempo gasto dentro da seção crítica ($busy_s + pm_s + wb_s + sm_s^{lap}$). Nesse caso, estamos assumindo que em ARC o número médio de processadores na fila pelo *lock* é o mesmo utilizando-se LAP ou não. O tempo gasto com falhas dentro da seção crítica s (sm_s^{lap}) é modelado como o tempo de falha num sistema sem LAP

multiplicado pela taxa de erro da técnica LAP ($fail_s^{lap}$), ou melhor só vai haver custo relativo a falhas dentro da seção crítica quando LAP errar a previsão.

O tempo total de execução de uma aplicação (TT) é então modelado como o tempo total gasto fora de seções críticas (TF) somado ao tempo gasto dentro de todas as seções críticas. Considerando uma aplicação com k seções críticas diferentes, para cada processador o tempo total de execução num sitema sem LAP é dado pela equação 3:

$$TT = TF + \sum_{s=1}^{k} TD_s; \tag{3}$$

Num sistema que utiliza LAP o tempo total de execução (TT^{lap}) por processador é dado pela equação 4:

$$TT^{lap} = TF + \sum_{s=1}^{k} TD_s^{lap}; \tag{4}$$

O potencial de ganho de desempenho de LAP é dado, portanto, pela razão TT^{lap}/TT. A porcentagem de redução obtida no tempo total de execução da aplicação (red) é dada pela equação 5:

$$red = 100 - \frac{TT^{lap} \times 100}{TT} \tag{5}$$

Existem, no entanto, determinadas aplicações em que a execução de um conjunto de seções críticas se dá antes da execução de uma barreira. Nessas aplicações, a redução no tempo de espera das seções críticas obtida por LAP, vai afetar também o tempo de espera na barreira. A modelagem do ganho obtido no tempo de espera da barreira é muito simples se a ordem em que os processadores adquirem os *locks* antes da barreira for sequencial. Mais especificamente, considere um grafo direcionado que mapeia a espera por *locks*. Os nós do grafo representam processadores e um arco de p_i para p_j indica que p_i está esperando a liberação do *lock* por p_j. Quando não há ciclos nesse grafo, significa que a espera é serial, e nesse caso o ganho no tempo de espera pelo *lock* é refletido diretamente no tempo de espera na barreira. Quando há ciclos no grafo, a modelagem é bem mais complicada. Para as aplicações estudadas nesse trabalho, a utilização do modelo mais simples foi suficiente.

Considerando uma aplicação em que existem m seções críticas diferentes com espera serial antes de uma barreira, o ganho no tempo de espera por um *lock* s obtido por LAP (dw^{lap^*}) é dado por:

$$dw^{lap^*} = \sum_{s=1}^{m} (wait_s - wait_s^{lap}). \tag{6}$$

O tempo gasto fora de seção crítica (TF^{lap^*}) fica reduzido, então, do fator dw^{lap^*}, conforme descrito na equação 7.

$$TF^{lap^*} = TF - dw^{lap^*}; \tag{7}$$

Portanto, o tempo total de execução (TT^{lap^*}), é dado pela equação 8:

$$TT^{lap^*} = TF^{lap^*} + \sum_{s=1}^{k} TD_i^{lap}; \quad (8)$$

A porcentagem de redução obtida no tempo total de execução da aplicação (red^*) é dada pela equação 9:

$$red^* = 100 - \frac{TT^{lap^*} \times 100}{TT} \quad (9)$$

3.3 Avaliação de LAP em ARC

Através do modelo analítico desenvolvido, avaliamos o potencial de ganho de LAP em um conjunto de três aplicações reais. Utilizamos um simulador baseado no sistema DASH para obtenção dos parâmetros necessários à aplicação do modelo. Esses parâmetros correspondem aos tempos gastos com: computação (*busy*); falhas ocorridas em dados privados (*pm*); espera pelo esvaziamento do *write-buffer* (*wb*); falhas em dados compartilhados (*sm*) e número médio de processadores na fila de espera pelo lock (*qs*).

O simulador é composto de duas partes. A primeira é um *front-end*, Mint [13], que simula a execução das instruções. A segunda é um *back-end* que simula o sistema de memória. A arquitetura simulada é composta de 16 nós de processamento, onde cada nó de processamento contém um único processador, um *write-buffer* de 4 entradas, uma memória cache de dados diretamente mapeada com 32KB e linhas de 64 bytes, uma memória local, um diretório totalmente mapeado e uma interface de rede. Assumimos que as leituras das instruções da aplicação não causam falha na cache e levam 1 ciclo para serem completadas. Uma falha de leitura paralisa o processador até que o pedido de leitura tenha sido satisfeito. As escritas vão para o *write-buffer* e levam 1 ciclo, a não ser que o *write-buffer* esteja cheio, nesse caso o pedido de escrita tem que esperar o *write-buffer* esvaziar. Leituras podem passar a frente de escritas que estejam enfileiradas no *write-buffer*. Um módulo de memória provê a primeira palavra 20 ciclos depois que o pedido foi feito. As outras palavras são entregues numa taxa de 1 ciclo por palavra. A contenção na memória é totalmente modelada. A rede de interconexão é uma malha bi-direcional que utiliza roteamento *wormhole* ordenado por dimensão. A velocidade do *clock* da rede é a mesma do *clock* do processador. Nós chaveadores (*switchs*) introduzem um atraso de 2 ciclos no cabeçalho da mensagem. A largura da rede é de 16 bits. A contenção na rede é modelada somente na saída e na chegada das mensagens.

O protocolo utilizado no simulador mantém as caches coerentes através de invalidações, segundo o modelo de consistência *Release Consistency*. É idêntico ao protocolo utilizado no sistema DASH que está descrito na seção 3.1.

Na avaliação de LAP em sistemas *hardware DSM*, utilizamos as seguintes aplicações: IS, Raytrace e Water-nsquared. Essas aplicações apresentam como característica comum o fato de que os acessos à estrutura de dados central da aplicação são protegidos por *locks*. **IS** (Integer Sort), da Universidade de Rice, realiza a ordenação de um vetor de chaves. Há um *lock* para proteger todos os acessos a um vetor de chaves compartilhado. A entrada utilizada foi de 64K chaves. **Raytrace**, do conjunto Splash-2 [14], reproduz uma cena tridimensional utilizando caminho dos raios. *Locks* protegem os acessos às filas de tarefas. A entrada utilizada foi *teapot*.

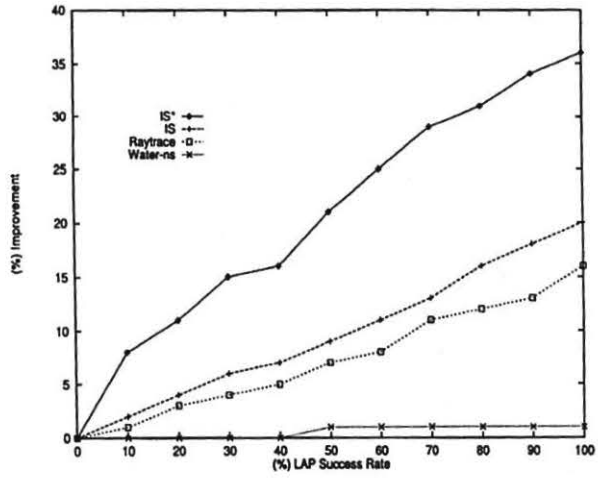

Figura 1: Limite Superior do Ganho de LAP

Water-nsquared, também do conjunto Splash-2, calcula as forças e potenciais num sistema de moléculas de água. Há um *lock* para cada molécula do sistema. A entrada utilizada foi 512 moléculas em 5 passos.

No gráfico da figura 1 apresentamos os resultados do potencial de ganho de desempenho de LAP para as três aplicações. Nas curvas IS, Raytrace e Water-ns apresentamos a redução no tempo de execução obtida por LAP (tal qual descrito na equação 5), em função da taxa de acerto da previsão fornecida por LAP. Na curva IS* a redução obtida no tempo de execução é dada pela equação 9.

Podemos observar que, para as aplicações IS e Raytrace, que apresentaram grande contenção pelos *locks*, o uso de LAP tem efeito considerável no tempo total de execução da aplicação para taxas de acerto maiores que 60%. Resultados obtidos em estudos anteriores [10, 11], mostram que as taxas de acerto para ambas as aplicações é bem alta, em torno de 95%, o que nos leva a crer que o limite superior do ganho de LAP está na faixa de 19% para IS e na faixa de 14% para Raytrace.

Na aplicação Water-nsquared, embora LAP tenha sido responsável por diminuir o tempo gasto dentro de seções críticas em até 80%, essa redução não teve o menor efeito no tempo total de execução. O que é explicado pelo fato de que em Water-nsquared o *overhead* relativo a operações realizadas dentro de seções críticas corresponde a apenas 1% do tempo total de execução da aplicação.

A aplicação IS apresenta também a característica especial, descrita na seção anterior, de espera serial pelo *lock*, seguida de uma barreira. Assim, a maneira mais correta de modelar o ganho de LAP em IS seria utilizar TT^{lap^*} como tempo total de execução. A curva IS* mostra o ganho de LAP em IS, considerando também a redução obtida no tempo de espera da barreira. Comparando com a curva IS, notamos que para essa aplicação a contenção pelo lock afeta bastante o tempo de espera da barreira. O ganho máximo obtido por LAP passou de 20% para 36% do tempo total de execução da aplicação.

4 O Protocolo *Affinity Entry Consistency*

Nessa seção apresentamos o protocolo *Affinity Entry Consistency* (AEC) para *software DSM* [11], que desenvolvemos a partir da técnica de LAP. O protocolo utiliza o modelo de consistência *Entry Consistency* [2] que explora a relação entre variável de sincronização e dado compartilhado. A idéia do modelo é que se um processador está entrando numa seção crítica, naquele momento ele só deve necessitar dos dados associados à seção crítica em questão. O problema é que a implementação de EC no sistema Midway [3] deixa para o programador o encargo da associação da variável de sincronização com o dado compartilhado. Dificultar a tarefa de programação raramente é uma boa estratégia, por isso nosso protocolo procura obter as vantagens de EC sem a necessidade da associação explícita de variável de sincronização com dado compartilhado.

4.1 Visão Geral

A idéia básica do protocolo AEC é fazer com que o processador que está saindo de uma seção crítica s envie antecipadamente para o próximo *acquirer* de s todas as modificações já realizadas dentro da seção crítica. Essas modificações são acumuladas de forma combinada (*merged*) numa mesma estrutura chamada M_s. Como é comum que numa seção crítica se acesse um conjunto bastante limitado de dados, o tamanho de M_s não é um problema para o nosso protocolo.

Dados compartilhados acessados fora de seções críticas, entretanto, não podem se beneficiar da técnica LAP e por isso recebem tratamento diferente em AEC; são mantidos coerentes através de invalidações.

A unidade de coerência empregada é a página e, para aliviar o problema de falso compartilhamento, AEC permite a existência de múltiplos escritores numa mesma página. Cada escritor armazena localmente as modificações realizadas na página e para saber exatamente o que foi modificado dentro de uma página, o processador mantém uma cópia (*twin*) com a versão original da mesma. Quando ele precisa recuperar as modificações realizadas, compara a página alterada com sua cópia, o resultado da comparação consiste de todas as modificações realizadas na página (*diff*).

4.2 LAP em AEC

A técnica LAP em AEC utiliza as duas técnicas citadas anteriormente (fila de espera e afinidade) e mais uma terceira técnica chamada de fila virtual. A técnica de fila virtual tenta antecipar a criação de uma fila de espera por uma determinada seção crítica. Ela é baseada na inserção no código da aplicação, alguns passos antes da operação *lock*, de uma instrução especial, chamada de "aviso-antecipado", que tem a função de informar que o processador vai, alguns passos adiante, executar um *lock*. 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 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 e é formado da seguinte maneira: [1]

1. Se há fila de espera: $U_p = q$ (q é o primeiro da fila de espera de s); Fim.
2. São inseridos em U_p os processadores j, tal que A_{pj} é 60% acima da média das afinidades A_{ij} de todos os processadores;
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.

4.3 Lock/Unlock

Quando um processador p executa um *lock* numa variável de sincronização s, ele envia uma mensagem para o processador gerente de s. O gerente ao receber o pedido de entrada na seção crítica calcula U_p. Se a seção crítica estiver ocupada, o gerente mantém a identificação de p na fila pela seção crítica. Se a seção crítica está liberada o gerente envia resposta a p com U_p, a identificação do último *releaser* de s e a informação se p está ou não no conjunto de atualização do último *releaser*. Se p não está no conjunto de atualização do último *releaser*, a mensagem do gerente inclui também uma lista de páginas a serem invalidadas. Nesse caso, o último *releaser* não acertou a sua previsão e p, ao entrar na seção crítica, deve invalidar todas as páginas presentes em M_s.

Enquanto o gerente realiza suas atribuições, o processador p pode aproveitar para criar *diffs* de páginas acessadas fora de seção crítica e aplicar *diffs* de M_s recebidos antecipadamente. Principalmente quando há contenção pela seção crítica, AEC consegue sobrepor a maior parte desses *overheads*. Note que os *diffs* de páginas acessadas fora de seção crítica só precisam ser efetivamente criados se uma mesma página for acessada dentro e fora da seção crítica, como esse raramente é o caso, os *twins* das páginas acessadas fora de seções críticas são mantidos durante a execução da seção crítica.

Quando um processador p executa um *unlock* numa variável de sincronização s, ele deve criar os *diffs* relativos às modificações realizadas dentro da seção crítica s, combinar com os *diffs* de M_s, recebidos do último *releaser* de s, e depois enviar o novo M_s para os processadores de U_p.

4.4 Barreira

Numa barreira cada processador deve receber informações sobre os dados acessados dentro e fora de seções críticas. AEC implementa barreiras dividindo a execução da aplicação em passos, um novo passo começa quando os processadores saem de uma barreira.

Cada processador p que chega na barreira envia uma mensagem para o processador gerente da barreira contendo três listas: lista de seções críticas que p utilizou;

[1] O tamanho do conjunto de atualização de um processador é pré-determinado pelo usuário.

uma lista de páginas acessadas dentro de seções críticas e uma lista de páginas acessadas fora de seções críticas. Depois de mandar suas listas para o gerente, o processador pode começar a criar os *diffs* relativos aos acessos realizados fora de seções críticas. Dado que normalmente os processadores chegam na barreira em tempos diferentes, essa criação de *diffs* consegue se sobrepor quase totalmente com o tempo de espera na barreira.

Depois que o gerente recebeu todas as listas, ele determina para cada processador p o conjunto de processadores para os quais p deve mandar *diffs* e/ou aviso de escrita (*write-notices*). *Diffs* correspondem às modificações realizadas dentro de seções críticas, *write-notices* indicam as páginas modificadas fora de seções críticas para serem invalidadas.

Os processadores, então, trocam os *diffs* e *write-notices* conforme estabelecido pelo gerente. Processadores que não possuem cópia válida de uma página no passo não recebem nem *diffs*, nem *write-notices* para essa página; no futuro, quando eles precisarem da página, eles precisam requisitá-la ao processador *home* da página. A cada passo o gerente da barreira escolhe um *home* para cada página. O *home* deve ser um processador que possua cópia válida da página.

4.5 Falta de Página

Se a falta de página ocorreu porque o processador não recebeu nem os *diffs* nem os *write-notices* da página, conforme descrito acima, ele deve requisitar uma versão atualizada da página para o processador *home*.

Caso contrário, o processador deve simplesmente requisitar os *diffs* para validar a página. Se a falta de página ocorreu dentro de uma seção crítica s, então, se ele não está no conjunto de atualização do último *releaser* r, ele requisita os *diffs* de M_s para r. Se estiver, ele simplesmente termina de aplicar os diffs de M_s que não conseguiu aplicar no *lock*. Se a falta de página ocorreu fora de uma seção crítica o processador através dos *write-notices* recebidos requisita os *diffs* aos respectivos processadores.

4.6 Avaliação de Desempenho

Para avaliar o desempenho da técnica LAP no protocolo AEC, simulamos o protocolo de duas maneiras diferentes: com a técnica LAP (o próprio AEC) e sem a técnica LAP (AEC com taxa de acerto da previsão de 0%). Tal qual o simulador de DASH, o simulador de AEC é composto de um *front-end* (Mint), que simula a execução das instruções, e de 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 [11].

Avaliamos AEC a partir das mesmas aplicações utilizadas na seção 3.3 para avaliar o potencial de LAP em sistemas *hardware DSM*. As entradas utilizadas também foram as mesmas. O peso do *overhead* de sincronização dessas aplicações, no entanto, é totalmente diferente para sistemas *software DSM*.

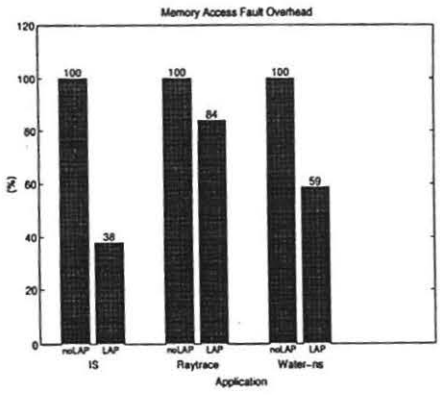

Figura 2: Overhead de Falta de Página em AEC sem LAP e AEC.

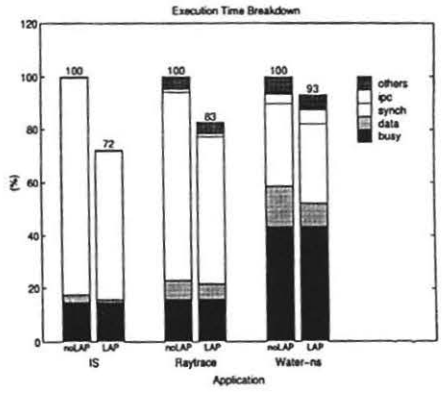

Figura 3: Tempo de Execução de AEC sem LAP e AEC.

Como a técnica LAP foi aplicada para diminuir a quantidade de faltas de páginas dentro da seção crítica, ela tem efeito direto no *overhead* de faltas de páginas do protocolo. No gráfico da figura 2 avaliamos o efeito de LAP em AEC no que diz respeito ao *overhead* de falta de página. Na barra da esquerda temos esse *overhead* para o protocolo AEC sem utilizar a técnica LAP, na barra da direita temos o mesmo *overhead* para o protocolo AEC com a técnica LAP. Essa figura mostra que LAP reduz o *overhead* de falta de página em até 62% para IS. O menor ganho foi em Raytrace (16%) porque nessa aplicação 80% das faltas são devidas a *cold-start*.

O quanto o ganho no *overhead* de falta de página vai afetar o tempo total de execução da aplicação depende da importância desse *overhead* no tempo total de execução da aplicação. Na figura 3, mostramos a execução das três aplicações em AEC sem LAP (esquerda) e AEC com LAP (direita). As barras da figura mostram o tempo de execução dividido em tempo de computação (`busy`), *overhead* de falta de página (`data`), tempo de sincronização (`synch`), overhead de IPC (`ipc`) e outros *overheads* (`others`) como falha na TLB, tempo de tratamento de interrupção, falha na cache e espera para esvaziar o *write-buffer*.

Para esse conjunto de aplicações podemos observar claramente o efeito de LAP no tempo total de execução. Para as aplicações IS e Raytrace, que apresentam alta contenção por *locks*, a redução imposta por LAP teve grande efeito no tempo total de execução. Em IS, particularmente, o efeito da redução no tempo de seção crítica teve impacto também no tempo de espera da barreira. Para a aplicação Water-nsquared, o ganho no *overhead* de falta de página teve pouco efeito no tempo total de execução porque esse *overhead* corresponde a somente 10% do tempo total de execução.

O protocolo AEC foi comparado também com o sistema TreadMarks [1]. Os resultados obtidos foram muito bons tanto para aplicações com sincronização baseada em *locks* (ganhos de até 47%), como para aplicações com sincronização baseada em barreira (ganhos de até 25%). Mais detalhes sobre esses resultados podem ser encontrados em [11].

5 Trabalhos Relacionados

A técnica LAP pode ser utilizada em dois tipos de sistemas bem distintos, *software* e *hardware DSMs*. O custo e o desempenho desses sistemas não são comparáveis, eles constituem, portanto, áreas de pesquisa totalmente independentes.

Sistemas *software DSM* atuais se baseiam em modelos relaxados de consistência de memória, em que a coerência da memória só é mantida em determinados pontos de sincronização. O modelo *Entry Consistency* (EC) explora a relação entre variável de sincronização e dado compartilhado. O protocolo ScC [5] desenvolvido a partir do modelo de consistência *Scope Consistency* e o protocolo ADSM [9] utilizam, tal qual AEC, EC sem associação explícita de variável de sincronização com dado compartilhado. A principal diferença entre AEC e o protocolo ScC é que AEC é implementado totalmente em software, já o protocolo ScC se baseia em suporte de hardware para realizar atualizações automáticas de um processador para outro. ADSM é um protocolo adaptativo baseado em TreadMarks que utiliza protocolo de atualização para dados protegidos por seções críticas, mas somente para dados acessados no padrão *single-writer*. As medidas tomadas para manter a coerência de dados acessados fora de seções críticas são, também, totalmente diferentes em AEC, no protocolo ScC e em ADSM.

A otimização de seções críticas em sistemas *hardware DSM* também foi alvo de estudo de alguns trabalhos [6, 12]. O trabalho de Trancoso e Torrelas [12] propõe duas técnicas para evitar falhas dentro de seções críticas, *prefetching* e *forwarding*. A técnica de *forwarding* é semelhante ao modo fila de espera de LAP, entretanto ela é válida somente para seções críticas com contenção. Este trabalho, entretanto, não faz uma análise do ganho de desempenho da técnica de *forwarding* isoladamente. O trabalho de Koufaty *et al* [6] estuda a possibilidade de se enviar antecipadamente atualizações do produtor direto para o consumidor do dado. No entanto, a análise de qual processador vai consumir o dado é feita unicamente pelo compilador.

6 Conclusões

Nesse trabalho mostramos uma avaliação do uso de LAP em sistemas DSM implementados em hardware e em software. Desenvolvemos dois protocolos, ARC para *hardware DSM* e AEC para *software DSM*, e avaliamos o efeito de LAP no desempenho de um conjunto de três aplicações. Nos dois tipos de sistemas a avaliação de LAP foi feita de forma diferente. Para *software DSM*, simulamos o protocolo AEC. Para *hardware DSM*, desenvolvemos um modelo analítico que provê um limite superior para o ganho de LAP. Em ambos os sistemas a técnica LAP foi bastante efetiva na redução do tempo gasto dentro da seção crítica. Entretanto, o efeito dessa redução no desempenho global da aplicação depende de quanto o *overhead* relativo às operações realizadas dentro de seções críticas representa no tempo de execução da aplicação. Quanto maior a contenção por *locks*, maior a serialização imposta pelo uso de seções críticas. Por isso, em nossos experimentos, as aplicações que apresentaram maior contenção por locks, IS e Raytrace, foram as que mais se beneficiaram da técnica LAP. Para IS obtivemos redução de até 28% no tempo de execução pelo uso de LAP em AEC e um limite superior de 36% de redução no tempo de execução com o uso de LAP em ARC. Para Raytrace a redução no tempo de execução obtida

em AEC foi de 17% e o limite superior de redução em ARC foi de 16%.

Em resumo, nossas contribuições são: uma avaliação completa da técnica LAP em *hardware* e *software DSM*; e a proposta de dois novos protocolos para esses sistemas. Concluímos que LAP é uma técnica geral que pode ser utilizada efetivamente para reduzir o tempo gasto dentro de seções críticas em sistemas DSM.

Agradecimentos

Gostaríamos de agradecer a Leonidas Kontothanassis pelo auxílio na implementação dos simuladores e a Enrique Vinicio Carrera pela ajuda na obtenção dos parâmetros necessários à aplicação do modelo analítico.

Referências

[1] C. Amza, A. Cox, S. Dwarkadas, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2), Feb 1996.

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

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. *Proc. of the IEEE COMPCON'93 Conference*, Feb 1993.

[4] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Proc. of the 17th International Symposium on Computer Architecture*, May 1990.

[5] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, Jun 1996.

[6] D. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. *IEEE Transactions on Parallel as Distributed Systems*, 29(2), Dec 1996.

[7] J. Kuskin *et al.* The Stanford FLASH Multiprocessor. *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April 1994.

[8] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.

[9] L. R. Monnerat and R. Bianchini. ADSM: A Hybrid DSM Protocol that Efficiently Adapts to Sharing Patterns. Tech. Report ES-425/97, COPPE/Sistemas, Universidade Federal do Rio de Janeiro, March 1997.

[10] C. B. Seidel, R. Bianchini, and C.L. Amorim. Técnicas Para Detecção Dinâmica de Padrões na Passagem de *Locks* em Sistemas *Software DSM*. *Anais do Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, Aug 1996.

[11] C. B. Seidel, R. Bianchini, and C.L. Amorim. The Affinity Entry Consistency Protocol. *Proc. of the International Conference on Parallel Processing*, Aug 1997.

[12] P. Trancoso and J. Torrellas. The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding. *Proc. of the 1996 International Conference on Parallel Processing*, Aug 1996.

[13] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. *Proc. of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, 1994.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. of the 22nd Annual International Symposium on Computer Architecture*, May 1995.