Projetando e Avaliando Sistemas Baseados em *Entry Consistency* e *Lock Acquirer Prediction* *

Cristiana B. Seidel[†‡], R. Bianchini[†] e Claudio L. Amorim[†]

[†]COPPE Sistemas e Computação
Univ. Federal do Rio de Janeiro
Rio de Janeiro, Brasil

[‡]Depto. de Eng. de Sistemas
Univ. do Estado do Rio de Janeiro
Rio de Janeiro, Brasil

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

Resumo

Esse trabalho apresenta os resultados da implementação real dos sistemas *software DSM* AEC e AEC-light, desenvolvidos a partir da técnica LAP, que prevê dinamicamente a ordem de transferência dos *locks* numa aplicação paralela. Ambos os sistemas são baseados no modelo de consistência *Entry Consistency*, mas utilizam modelos de programação distintos. AEC utiliza modelo de programação com associação implícita de dado compartilhado com variável de sincronização, enquanto que o modelo utilizado por AEC-light requer que essa associação seja feita de forma explícita. De forma a avaliar os sistemas, realizamos experimentos com um conjunto representativo de aplicações executando no sistema SP-2 com 8 nós de processamento. Nossos resultados mostram que o uso de protocolo de atualização aliado à técnica LAP em AEC permite reduções de até 48% no tempo de execução em relação ao sistema TreadMarks. Nossos resultados demonstram ainda que, através do uso de modelo de programação mais elaborado, AEC-light pode alcançar desempenho até 17% superior ao do sistema AEC. Entretanto, observamos que, quando a aplicação apresenta granularidade fina de acesso ao(s) dado(s) protegido(s) por cada *lock*, AEC e AEC-light apresentam desempenho inferior a TreadMarks. Baseados nesses resultados preliminares, concluímos que AEC e AEC-light podem alcançar bom desempenho, mas ainda precisam ser ajustados para tratar acessos a dados com granularidade fina.

1 Introdução

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

Sistemas *software DSM* 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. Entretanto, esses sistemas ainda apresentam alto *overhead*, principalmente porque ao modificar um dado compartilhado um processador não sabe de antemão qual o próximo processador a acessar esse dado. Para atacar este problema, desenvolvemos uma técnica chamada *Lock Acquirer Prediction* (LAP) [10], a qual prevê dinamicamente a ordem de transferência de *locks*. LAP permite enviar antecipadamente as alterações realizadas nos dados compartilhados protegidos por uma seção crítica ao próximo processador a "adquirir" a seção, eliminando assim o tempo de espera por dados sem o custo adicional de sobrecarregar a rede com grande quantidade de mensagens.

As altas taxas de sucesso com que LAP conseguiu realizar suas previsões num estudo preliminar [10] nos levaram à criação de dois novos protocolos para *software DSM*, *Affinity Entry Consistency* (AEC) [12] e AEC-light, os quais exploram as vantagens oferecidas por LAP. Conforme seus nomes

*Este trabalho recebeu apoio financeiro da FINEP e do CNPq.

já dizem, os protocolos são baseados no modelo de consistência de memória *Entry Consistency*(EC) [2]. O modelo EC é um dos mais relaxados dentre os modelos de consistência de memória e se adequa perfeitamente às otimizações propostas por LAP. EC explora a relação entre cada variável de sincronização e os dados compartilhados protegidos por ela, e garante que os dados estarão coerentes num processador, quando este adquirir a seção crítica associada a variável de sincronização que o protege.

A associação de variável de sincronização com dado compartilhado, imposta pelo modelo EC, é feita em AEC de forma implícita. AEC-light mantém essa associação implícita, mas considera um modelo de programação mais elaborado, no qual todos os acessos a dados compartilhados são realizados em seções críticas. O código de AEC-light é baseado no de AEC, mas por considerar um modelo de programação diferenciado é um protocolo bem mais simples.

Nosso objetivo nesse trabalho é avaliar o impacto de LAP e dos modelos de programação de AEC e AEC-light no desempenho de aplicações paralelas. Assim, estudamos o desempenho de um conjunto representativo de aplicações sob AEC e AEC-light em um multicomputador IBM SP-2 com oito nós de processamento. Nossos resultados mostram que, para aplicações baseadas em *locks*, o uso de LAP permite grande redução no *overhead* de acesso a dados compartilhados, tornando AEC um protocolo bastante eficiente. Nossos resultados demonstram ainda que o uso de um modelo de programação mais elaborado em AEC-light permite reduzir ainda mais esse *overhead*. Entretanto, observamos que, quando a aplicação apresenta granularidade fina de acesso ao(s) dado(s) protegido(s) por cada *lock*, o uso da página como unidade de coerência compromete seriamente o desempenho de AEC e AEC-light. Baseados nos nossos resultados, concluímos que AEC e AEC-light podem alcançar bom desempenho, mas ainda precisam ser ajustados para tratar acessos a dados com granularidade fina.

O restante desse artigo é organizado da seguinte forma. Na próxima seção apresentamos o modelo de programação empregado em AEC e as diferenças entre a implementação corrente do protocolo e nossa proposta inicial [11]. Na seção 3 apresentamos o protocolo AEC-light mostrando as diferenças em relação ao protocolo AEC e seu modelo de programação. Na seção 4 apresentamos a metodologia empregada em nossos experimentos e em seguida, na seção 5, avaliamos os resultados obtidos para os protocolos AEC e AEC-light. A seção 6 apresenta os trabalhos relacionados e na seção 7 concluímos o nosso trabalho.

2 AEC

2.1 Modelo de Programação

AEC procura obter as vantagens de EC sem a necessidade da associação explícita de variável de sincronização com dado compartilhado. A interface de programação é bastante parecida com a utilizada na grande maioria dos sistemas DSM com consistência relaxada. A sincronização entre os processos deve ser feita através de primitivas de sincronização providas pelo sistema, como **locks/unlocks** para delimitar seções críticas e **barreiras** para sincronização global.

Na maioria dos casos, programas que utilizam o modelo de programação com memória compartilhada e sincronização explícita executam corretamente em AEC. Entretanto, o uso do modelo EC implica em restrições adicionais ao modelo de programação [6].

2.2 O Protocolo

Por simplicidade de implementação, AEC utiliza a página como unidade de coerência e, para aliviar o problema de falso compartilhamento, 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 seu *twin*, o resultado da comparação consiste das modificações (*diff*) realizadas na página.

A idéia básica do protocolo é dar tratamento diferenciado para dados acessados dentro e fora de seções críticas. Para os dados compartilhados acessados dentro de seções críticas, LAP viabiliza o uso de protocolo de atualização para manutenção da coerência, porque permite que as atualizações sejam enviadas de forma bastante seletiva. Para dados compartilhados acessados fora de seções críticas, entretanto, AEC matém a coerência através de protocolo de invalidação para não sobrecarregar a rede com grande quantidade de mensagens.

Por falta de espaço, não apresentamos AEC em detalhes nesse artigo. Os mesmos podem ser encontrados em [11]. Nos parágrafos seguintes, apresentamos apenas a descrição detalhada da operação de barreira, a qual foi implementada de forma diferente do apresentado em [11].

Barreiras dividem a execução da aplicação em **fases**. Após a execução de uma barreira, uma nova fase é iniciada e todos os processadores devem ter a mesma visão da memória compartilhada. Assim, numa barreira, AEC tem que garantir a visibilidade de todas as modificações realizadas na fase anterior. Essa visibilidade é garantida pelo processador gerente da barreira. Ao chegar a uma barreira, cada processador envia para o processador gerente a identificação das páginas alteradas fora de seções críticas e de cada seção crítica utilizada. Logo em seguida, o processador pode começar a criar os diffs relativos aos acessos realizados fora de seções críticas. Em geral, essa criação de diffs consegue se sobrepor quase totalmente com o tempo de espera na barreira.

O processador gerente coleta as informações de todos os processadores e determina sobre quais modificações cada processador deve ficar ciente. O gerente informa sobre as modificações realizadas fora de seções críticas, através de avisos de escrita, `write-notices-out`. Um **write-notice-out** indica que uma página foi alterada fora de seção crítica numa determinada fase, mas não contém as alterações realizadas na página. Ele contém apenas a identificação do processador que alterou a página e em qual fase se deu essa alteração. Para as modificações realizadas dentro de seções críticas, o gerente utiliza um outro tipo de aviso de escrita, `write-notice-in`. Um `write-notice-in` indica que a página foi alterada em fase anterior sob determinada seção crítica. Ele contém a identificação da seção, do seu último dono e em qual fase se deu a alteração.

Ao receber um `write-notice-out` e/ou um `write-notice-in`, o processador invalida a página correspondente. Note que para uma mesma página um processador pode receber diversos avisos de escrita. Quando ocorre uma falta de página, então, o processador requisita aos outros processadores os diffs correspondentes às modificações realizadas dentro e fora de seções críticas na página. Considerando que essas modificações podem ter ocorrido em fases distintas, e, portanto, podem sobrepor-se, AEC considera a fase em que a modificação foi realizada quando da aplicação dos *diffs*.

3 AEC-light

3.1 Modelo de Programação

O modelo de programação utilizado pelo protocolo AEC-light apresenta, tal qual o modelo de programação utilizado por AEC, interface com primitivas *lock*/*unlock* e barreira. Entretanto, para uma aplicação executar corretamente em AEC-light, ela deve atender às seguintes restrições: a) para cada estrutura compartilhada deve existir uma variável de sincronização correspondente; b) todos os acessos a dados compartilhados devem ser feitos em seções críticas; e c) operações de barreira devem servir somente para sincronização global, não para coerência.

A primeira restrição impõe que o programador estabeleça uma variável de sincronização para cada dado compartilhado, sendo que um mesmo dado compartilhado deve ser sempre acessado sob a mesma seção crítica. A segunda restrição é a base do modelo de programação utilizado por AEC-light, todos os acessos a dados compartilhados, sejam eles de leitura ou de escrita, devem ser realizados dentro da seção crítica correspondente. Já a terceira restrição é uma consequência direta

da segunda restrição, se todos os acessos são protegidos por *locks*, então não há necessidade de se garantir coerência também na barreira.

Uma aplicação que atende às três restrições acima executa corretamente no protocolo AEC-light. No entanto, para evitar problemas de desempenho causados pelo excesso de sincronização, AEC-light oferece duas primitivas de *lock* que não requerem nenhum tipo de sincronização, `lock-reader/unlock-reader` e `lock-alone/unlock-alone`, além das primitivas `lock/unlock` padrão. As primitivas `lock-reader/unlock-reader` delimitam seções críticas de acesso não exclusivo, devendo ser utilizadas quando, numa determinada fase da computação, os processadores vão realizar apenas acessos de leitura aos dados protegidos pelas seções. As primitivas `lock-alone/unlock-alone` delimitam seções críticas de escrita por um único processador, devendo ser utilizadas quando, numa determinada fase da computação, um único processador escreve no dado compartilhado.

3.2 O Protocolo

Tal qual AEC, AEC-light utiliza a página como unidade de coerência, permite a existência de múltiplos escritores simultâneos para a mesma página e utiliza *twins* e *diffs* para detectar as modificações realizadas numa determinada página.

AEC-light não precisa dar tratamento diferenciado para os dados compartilhados, porque todos são acessados dentro de seções críticas. Dessa forma, utiliza somente protocolo de atualização para manter a coerência dos dados. A técnica LAP é utilizada da mesma forma que em AEC para primitivas `lock/unlock` padrao apenas.

Sincronização Local. Conforme mostramos no modelo de programação de AEC-light, o programador pode utilizar três tipos diferentes de primitivas `lock/unlock`. As primitivas `lock/unlock` padrão são implementadas em AEC-light exatamente da mesma forma que em AEC. Portanto, vamos nos ater aqui a descrição das outras duas primitivas de *lock*.

As primitivas `lock-reader` e `lock-alone` são implementadas de forma bem parecida. Quando um processador executa um `lock-reader` ou um `lock-alone` numa variável de sincronização s, ele deve solicitar ao último dono todos os *diffs* associados a s, caso ele não tenha sido previsto como provável próximo *acquirer* de s. Ao receber os *diffs*, o processador os aplica em suas respectivas páginas. A grande diferença na implementação dessas duas primitivas está nessa aplicação de diffs. A primitiva lock-reader simplesmente os aplica para permitir a leitura correta dos dados. Já na primitiva lock-alone, como haverá escrita no dado, antes de aplicar os *diffs*, são criados *twins* para suas páginas. A primitiva `unlock-reader` não realiza nenhuma operação; ela pode ser utilizada opcionalmente pelo programador para efeito de delimitação da seção crítica. Na primitiva `unlock-alone`, o processador cria os *diffs* das páginas escritas e os associa a s.

Sincronização Global. Em AEC-light a barreira funciona somente como sincronização global, não há necessidade de se garantir coerência de dados. Portanto, não há necessidade de se utilizar avisos de escrita (*write-notices-out* e *write-notices-in*) e nem de se invalidar páginas.

Junto à tarefa de sincronização, a barreira em AEC-light é utilizada também para trocar algumas informações úteis às primitivas lock-reader e lock-alone. Essas informações são: a identificação do último dono de cada seção crítica e o conjunto de páginas alteradas sob cada lock-alone. A troca dessas informações na operação de barreira evita que num pedido de lock-reader ou lock-alone em s, o processador tenha que requisitar ao gerente de s a identificação do último dono de s, e também evita que a cada unlock-alone em s o processador tenha que informar ao gerente sobre o conjunto m_s de páginas alteradas sob s.

A troca dessas informações é feita junto com as mensagens de sincronização da barreira. Ao chegar a uma barreira, o processador envia uma mensagem ao gerente avisando que já está pronto para a sincronização global. Nessa mensagem ele envia também a identificação de cada variável s utilizada num lock ou lock-alone na fase anterior, juntamente com o conjunto m_s. Enquanto espera pelo término da barreira, o processador pode aplicar quaisquer *diffs* recebidos por antecipação.

Depois que o gerente tiver recebido mensagens de todos os processadores, ele determina quais processadores são os últimos donos de cada *lock* e envia mensagem de volta para todos os processadores liberando a barreira. Nessa mensagem o gerente envia também a identificação do último dono de cada *lock* s e para o processador gerente de s, ele envia o conjunto m_s.

4 Metodologia

4.1 Ambiente

Para avaliar o desempenho da implementação de AEC e AEC-light utilizamos um multicomputador IBM SP2 do Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro, com oito nós de processamento, em modo exclusivo.

O sistema base utilizado em nossas avaliações foi o TreadMarks da Universidade de Rice [8], o qual é o mais difundido protocolo na classe de sistemas *software DSM*. TreadMarks utiliza a página como unidade de coerência e protocolo de invalidação para manutenção da coerência dos dados compartilhados. O modelo de consistência empregado é o *Lazy Release Consistency* [7]. Para determinar quais modificações devem estar visíveis num processador quando ele 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 gerente de s, que avança esse pedido para o último dono da seção crítica. Este compara seu vetor de *timestamp* com o de p e envia para p os respectivos avisos de escrita (*write-notices*) das páginas. O processador p invalida as páginas para as quais recebeu *write-notices*. Quando ocorre uma falta de página, o processador consulta sua lista de *write-notices* e requisita as modificações feitas na página para os devidos processadores.

4.2 Aplicações

Utilizamos três aplicações em nossos experimentos: IS, Water e MigDepth. IS e Water fazem parte do pacote de distribuição de TreadMarks e MigDepth é uma versão de migração sísmica 2D da Petrobrás [4]. IS classifica um vetor de N inteiros, utilizando chaves no intervalo [0, B_{max}], através da técnica *bucket sort*. As chaves são igualmente divididas pelos processadores e cada iteração consiste de três fases. Na primeira fase, o vetor global é inicializado pelo processador 0. Na segunda, processadores armazenam seus resultados locais no vetor global. Este armazenamento é feito dentro de uma única seção crítica. Na última fase, os processadores lêem o vetor global para classificar suas chaves locais. A entrada foi $N = 2^{23}$, $B_{max} = 2^{15}$ e 10 iterações.

MigDepth faz migração 2D pós-estaqueamento usando o método $\omega - x$. A paralelização é obtida através de particionamento por profundidade. Cada processador extrapola a seção sísmica para um determinado conjunto de profundidades. Como a migração de cada profundidade depende da anterior, a computação é feita em modo *pipeline*. Este *pipeline* é controlado com uso de *locks*. A entrada utilizada foi de 128 × 128.

Water calcula as forças e potenciais de um sistema de moléculas de água no estado líquido. As moléculas são distribuídas igualmente entre os processadores. Em cada iteração várias fases são computadas. Numa fase inicial o processador 0 inicializa algumas somas globais. Na fase seguinte, chamada fase intramolécula, cada processador computa valores de deslocamento para o seu conjunto de moléculas. A terceira fase, utiliza os valores de deslocamento gerados na fase anterior para computar as forças entre as moléculas, essa fase é chamada de intermolécula. Cada processador computa a interação entre todas as moléculas em sua partição e mais $n/2$ outras moléculas. Nessa fase, a atualização das forças das moléculas é realizada dentro de seção crítica, há uma seção crítica por molécula. A quarta fase utiliza valores das forças calculados na fase anterior e calcula os valores corretos para as moléculas, condições de fronteira e a energia cinética do sistema.

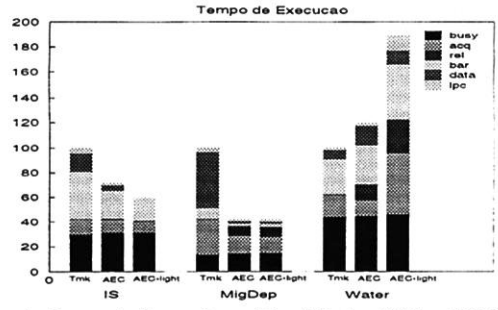

Figure 1: Tempo de Execução em TreadMarks, AEC e AEC-light.

Nessa fase cada processador realiza as computações no seu conjunto de moléculas. Utilizamos como entrada um conjunto de 512 moléculas com 10 iterações.

Note que optamos por estudar um conjunto reduzido, mas representativo de aplicações baseadas em *locks*. IS é um exemplo de aplicação em que os dados compartilhados são alterados dentro e fora de seções críticas e ilustra a utilidade de diferentes modelos de programacao. MigDepth é um exemplo de aplicação em que todas as modificações em dados compartilhados são realizadas sob seções críticas e, portanto, executa com alterações mínimas nos dois modelos de programação. Já Water ilustra o caso em que utilizar página como unidade de coerência numa aplicação com granularidade fina de acesso pode levar a sérios problemas de desempenho.

5 Avaliação de Desempenho

A figura 1 apresenta os tempos de execução paralela das nossas aplicações executando sob os diversos protocolos no SP2. Todos os tempos são normalizados para os resultados de TreadMarks. Na figura, o tempo de execução é dividido em tempo de computação (`busy`) que inclui também tempo de falha na cache e na TLB; tempo de sincronização dividido em *acquire* (`acq`), *release* (`rel`) e barreira (`bar`); *overhead* de falta de página (`data`); e overhead de *ipc* (`ipc`) que representa o tempo que um processador gasta servindo pedidos remotos. A instrumentação necessária para coletar esses tempos afeta o desempenho dos protocolos de forma negligível.

5.1 IS

Conforme podemos observar na figura, AEC obteve em IS uma redução de 29% no tempo total de execução em relação a TreadMarks, em grande parte através de sensíveis reduções nos *overheads* de barreira e acessos a dados. Mais especificamente, AEC obteve 69% de redução no *overhead* de falta de página; 41% de redução no tempo de barreiras e 18% de redução no tempo de *acquire*.

A maior redução obtida por AEC foi no *overhead* de falta de página. Para analisar esse *overhead* com mais detalhes, vamos separá-lo em *overhead* de falta de página ocorrida `dentro` de seção crítica e *overhead* de falta de página ocorrida `fora` de seção crítica. Para as faltas de páginas ocorridas `dentro` de seções críticas, AEC obteve redução de 66% em relação ao sistema TreadMarks. Toda essa redução foi obtida na segunda fase da computação, onde o uso de protocolo de atualização e da técnica LAP consegue evitar 51% das faltas geradas sob o *lock* que protege o vetor compartilhado.

Entretanto, de forma um tanto quanto surpreendente, AEC obteve redução de 71% no tempo de falta de página ocorrida `fora` de seção crítica. Essas faltas ocorrem na terceira fase da computação, onde os processadores lêem o vetor compartilhado para classificar suas chaves locais. Embora AEC e TreadMarks gerem o mesmo número de faltas de páginas fora de seção crítica, o custo dessas faltas

é muito maior em TreadMarks. Isso se explica pelo fato de que em TreadMarks, um processador p, ao tomar uma falta fora de seção crítica numa página k, recebe do último dono da seção crítica que protege o vetor compartilhado em média 5 *diffs* diferentes, relativos aos intervalos que ele ainda não recebeu. Em AEC como os *diffs* gerados dentro do *lock* que protege o vetor compartilhado são armazenados de forma combinada, nessa mesma requisição p recebe somente 1 *diff* do último dono da seção crítica.

A redução de 18% obtida por AEC no tempo de *acquire* pode ser explicada pela redução no número de faltas ocorridas dentro de seção crítica. Em IS, como há apenas um *lock* para proteger todo o vetor compartilhado, a contenção por esse *lock* é alta. Assim, qualquer redução no tempo gasto dentro da seção crítica leva a uma redução no tempo de espera pela seção crítica. Note que a redução obtida no tempo de *acquire* não é diretamente proporcional à redução no *overhead* de falta de página ocorrida dentro de seção crítica. Isso porque o custo do *acquire* em AEC é maior do que em TreadMarks quando o processador já está esperando pelo *lock*. Em AEC, na espera por um *lock*, um processador p deve aguardar por duas trocas de mensagens, do *releaser* para o gerente e do gerente para p. Em TreadMarks, p aguarda somente a mensagem do *releaser*. O tempo de *release*, embora pouco significante no tempo total, aumentou bastante em AEC, devido à criação de *diffs* ocorrida numa operação `unlock`.

O efeito da redução no tempo de seção crítica teve impacto também no tempo de espera da barreira, porque a segunda fase da computação de IS apresenta espera serial pelo *lock* seguida de uma operação de barreira. Assim, quanto menos tempo o processador gasta no *lock*, mais cedo chega à barreira.

O uso de um modelo de programação mais complicado em AEC-light levou a uma redução de 17% no tempo de execução de IS em relação ao protocolo AEC. Comparando os *overheads* gerados pelos dois protocolos, vemos que em AEC-light quase não há *overhead* de falta de página, e os tempos de barreira e *acquire* foram reduzidos respectivamente em 19% e 17%.

A grande redução obtida por AEC-light no *overhead* de falta de página ocorreu por dois motivos principais. Primeiro, porque no modelo de programação empregado por AEC-light não há falta de página `fora` de seção crítica. Essa busca de dados ocorre em operações `lock-reader` e `lock-alone`. Segundo, porque o custo da falta de página ocorrida dentro de seção crítica é menor em AEC-light do que em AEC.

As faltas geradas dentro de seções críticas ocorrem todas na segunda fase da computação de IS, onde os processadores escrevem no vetor compartilhado. Sendo que esse vetor é inteiramente inicializado pelo processador 0 na fase anterior. No modelo de programação de AEC-light, a inicialização do vetor é realizada utilizando-se as primitivas `lock-alone/unlock-alone`. Dessa forma, na segunda fase de computação em AEC-light, somente o primeiro processador a escrever no vetor requisita as modificações realizadas pelo processador 0 na fase anterior. Em AEC, entretanto, a inicialização do vetor compartilhado é realizada fora de seção crítica. Assim, na segunda fase da computação, todos os processadores requisitam as modificações realizadas pelo processador 0 na fase anterior, aumentando o custo da falta de página.

5.2 MigDepth

Na figura 1 observamos que AEC obteve redução de 48% no tempo de execução de MigDepth em relação a TreadMarks, desta vez como resultado de reduções nos *overheads* de *acquire* e acessos a dados. Mais especificamente, AEC obteve 73% de redução no *overhead* de falta de página e 57% de redução no tempo de *acquire*.

Em MigDepth a maior redução obtida por AEC foi, tal qual em IS, no *overhead* de falta página. Sendo que em MigDepth este *overhead* compreende somente faltas de páginas ocorridas `dentro` de seções críticas, dado que todos os acessos a dados compartilhados são realizados sob *locks*. Toda a redução no *overhead* de falta de página em MigDepth foi obtida pela eliminação de 81% das faltas de páginas ocorridas dentro de seções críticas pelo uso de protocolo de atualização com a técnica

LAP. Nessa aplicação, como a computação dos dados compartilhados procede como um *pipeline*, a ordem com que os processadores pegam o *lock* é sempre a mesma, e, portanto, LAP consegue em 95% das vezes acertar suas previsões.

A redução obtida no tempo de *acquire* se explica não só pela redução no tempo de espera pela seção crítica, causada pela eliminação de faltas, como também pelo fato de que, quando não há espera pelo *lock*, o custo do *acquire* é menor em AEC do que em TreadMarks. Em AEC, numa operação de *acquire* em uma seção crítica disponível, um processador p troca mensagens apenas com o gerente do *lock*. Em TreadMarks, p deve enviar uma mensagem para o gerente, que envia para o último dono do *lock*, que então envia a liberação de volta para p. Em MigDepth há pouca contenção pelos *locks*.

AEC e AEC-light apresentam praticamente o mesmo tempo de execução para MigDepth. Isso se deve a que a única alteração realizada em MigDepth para executar corretamente no modelo de programação de AEC-light ocorreu na fase de inicialização, a qual não é computada no tempo de execução paralela.

5.3 Water

Water apresentou comportamento diferente das outras aplicações. Como se observa na figura 1, o tempo de execução aumentou em 17% para o sistema AEC. Analisando os *overheads* mais importantes, vemos que AEC dobrou o *overhead* de falta de página, aumentou em 13% o tempo de barreira e reduziu em 32% o tempo de *acquire*.

O aumento considerável no *overhead* de falta de página se deve basicamente ao *overhead* relativo às faltas de páginas geradas `dentro` de seções críticas. Apesar da utilização de LAP e de protocolo de atualização, esse *overhead* aumentou 4 vezes. Isto se explica por uma característica especial dessa aplicação. A cada processador é associado um conjunto de moléculas que estão alocadas contiguamente nas páginas compartilhadas; cada página contém aproximadamente 8 moléculas. Na fase de computação intermolécula, onde se concentra todo o processamento com *locks*, um processador começa acessando as suas moléculas e em seguida acessa as moléculas de outros processadores. Em TreadMarks, quando um processador p começa a acessar um conjunto de moléculas da página k, ele gera somente uma falta de página ao acessar a primeira molécula de k (dado que p acessa todas as outras moléculas de k antes dos outros processadores). Em AEC, entretanto, as modificações realizadas em seções críticas diferentes são tratadas de forma independente. Assim, para acessar as moléculas de k, p gera 8 faltas quando LAP não acerta sua previsão. Na verdade, mesmo quando LAP acerta a previsão, p deve criar novo *twin* para k a cada operação *acquire*.

A redução obtida no tempo de *acquire* se deve ao fato de não haver contenção pelos *locks*, e, conforme explicado na subseção 5.2, o custo do *acquire* é menor em AEC do que em TreadMarks quando não há espera pelo *lock*. O aumento ocorrido no tempo de *release* se explica pela necessidade de criação de um *diff* a cada *lock*, e está ainda exacerbado pela grande quantidade de operações `unlock` dessa aplicação.

O modelo de programação diferenciado no protocolo AEC-light comprometeu seriamente o desempenho de Water. Para esta aplicação, AEC-light aumentou o tempo de execução em 58% em relação a AEC. O tempo gasto com a primitiva `lock` padrão é o mesmo para os AEC e AEC-light protocolos. Entretanto, comparando *overheads* de busca de dados observamos que: o custo de falta de página dentro de seção crítica é equivalente para os dois protocolos, enquanto que o tempo gasto com busca de dados pelas primitivas `lock-reader` e `lock-alone` de AEC-light é o dobro do *overhead* de falta de página ocorrida fora de seção crítica em AEC. Esta diferença se deve a que lock-readers buscam dados com granularidade muito menor que uma página. Em AEC, numa falta na página k fora de seção crítica, um processador p requisita as modificações realizadas nas moléculas de k na fase anterior. Se um mesmo processador foi o último dono do *lock* que protege todas as moléculas de k, em uma única mensagem ele envia todas as modificações para p e a página está coerente para os acessos seguintes. Em AEC-light, essa mesma operação é realizada com uma

chamada a `lock-reader` para cada molécula. Assim, tendo 8 moléculas em uma página, p vai requisitar as 8 moléculas, uma de cada vez, mesmo que todas as requisições sejam para o mesmo processador.

Em AEC-light, o tempo de *release* aumentou consideravelmente por causa do custo da operação `unlock-alone`, a qual envolve criação de *diffs*. Em AEC, a criação desses mesmos *diffs* é realizada durante a espera por uma barreira.

5.4 Sumário dos Resultados

Os resultados apresentados mostram que, tanto para AEC como para AEC-light, o uso de protocolo de atualização juntamente com a técnica LAP permite sensível redução no *overhead* de busca de dados compartilhados acessados dentro de uma seção crítica e tem efeito direto no tempo de espera pela seção crítica e no *overhead* de barreiras.

Quando a unidade de acesso da aplicação é maior que uma página e o padrão de acesso é migratório, o armazenamento `combinado` das modificações realizadas dentro de seções críticas em AEC e AEC-light reduz a quantidade de dados transferida.

Já quando a unidade de acesso é muito menor que uma página, é potencialmente ineficiente manter a coerência na unidade de página, conforme mostram os resultados de Water para AEC e AEC-light. Entretanto, dependendo do padrão de acesso da aplicação, o uso do modelo LRC permite que as ações tomadas pelo protocolo de coerência, num acesso a um determinado dado compartilhado, adiantem ações de coerência para os outros dados contidos na mesma página.

O fato de que em AEC-light todos os acessos a dados compartilhados são protegidos por *locks* permite outro tipo de redução no *overhead* de busca de dados. Com uma única requisição ao último dono do *lock* que protege o dado, é possível manter coerente um dado que foi escrito em diferentes fases anteriores. Em contraste, num acesso fora de seção crítica a um dado escrito em várias fases diferentes, AEC e TreadMarks têm que requisitar todas as modificações realizadas nas fases anteriores.

6 Trabalhos Relacionados

6.1 Modelos de Programação

Sistemas baseados em RC ou LRC empregam um modelo de programação simples, que exige somente que os programas sejam propriamente sincronizados (*properly labeled* [5]). Já os sistemas Scope Consistency (o qual, na nossa visão, tem uma variante de EC como modelo de consistência) e Brazos, aproveitam as vantagens de um modelo de consistência mais relaxado, sem a necessidade de associação explícita de variável de sincronização com dado compartilhado. Dessa forma, utilizam, na maioria das vezes, modelo de programação tão simples quanto o utilizado por RC e LRC. Os sistemas Midway e AEC-light, na tentativa de diminuir o *overhead* gerado pelo protocolo, utilizam modelo de programação mais elaborado, com todos os acessos a dados compartilhados feitos dentro de *locks*.

Pelo o que sabemos, o único trabalho que trata da comparação entre diferentes implementações de EC com LRC é o trabalho de Adve *et al* [1]. Este trabalho, porém, não faz uma análise sobre como o modelo de programação pode ter efeito em certos *overheads* do protocolo e não considera técnicas como LAP.

6.2 Protocolos de Coerência e Eliminação de *Overheads*

A maior parte dos protocolos de coerência são baseados em invalidações. TreadMarks é um exemplo desse tipo de sistema. AEC, em contraste, utiliza uma combinação de invalidações e atualizações. As atualizações de fato permitem a eliminação de *overheads* de falhas de acesso. Vários outros

sistemas também utilizam esse tipo de combinação. O protocolo Lazy Hybrid [3], por exemplo, envia as atualizações na mensagem de liberação da seção crítica quando o último dono da seção tem uma versão atualizada e sabe que o *acquirer* possui o dado. AEC envia as atualizações de forma ainda mais agressiva, utilizando LAP.

ADSM [9] se adapta dinamicamente entre diferentes protocolos segundo o padrão de compartilhamento exibido pelas páginas da aplicação. AEC não apresenta nenhum tipo de adaptação às características da aplicação, entretanto, o algoritmo utilizado para categorização das páginas de ADSM é aplicável a AEC.

ScC [6] e Brazos [13] utilizam EC com associação implícita, tal qual AEC. Entretanto, AEC difere desses dois sistemas já que ataca *overheads* através de sobreposição com outros *overheads* ou com tempo de computação e não utiliza suporte de hardware específico. Midway utiliza um modelo de programação similar ao de AEC-light, entretanto, a unidade de coerência empregada em Midway é de um objeto, ao contrário de AEC-light que utiliza a página. Embora ambos os sistemas utilizem protocolo de atualização para manutenção da coerência, AEC-light faz as atualizações com antecedência devido ao uso de LAP.

7 Conclusões

Esse trabalho mostrou os resultados da implementação real dos protocolos AEC e AEC-light, ambos baseados na técnica LAP. Nosso estudo preliminar de um conjunto representativo de aplicações mostrou que AEC e AEC-light obtêm grande redução no *overhead* de acesso a dados compartilhados em relação ao sistema TreadMarks. Esta redução levou também à redução de outros *overheads*, como os tempos de *acquire* e barreira. Entretanto, observamos que, quando a aplicação apresenta granularidade fina de acesso na estrutura protegida pelo *lock*, AEC e AEC-light apresentam desempenho inferior a TreadMarks. Concluímos que AEC e AEC-light ainda precisam ser ajustados para tratar acessos a dados com granularidade fina. No futuro próximo, pretendemos tratar o problema de granularidade, além de estudar o impacto de novas variações no modelo de programação empregado.

Agradecimentos

Gostaríamos de agradecer ao Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro pelo uso do sistema SP2 e em particular a Sérgio Guedes pela ajuda na utilização do sistema. Gostaríamos de agradecer também a Maria Clícia Stelling de Castro pela ajuda com o gráfico apresentado nesse artigo.

References

[1] S. Adve, A. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proc. of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 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] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, May 1993.

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

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

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

[7] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[8] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the 1994 Winter Usenix Conference*, January 1994.

[9] L. R. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proc. of the 4nd IEEE Symposium on High-Performance Computer Architecture*, February 1998.

[10] C. B. Seidel, R. Bianchini, and C.L. Amorim. Tecnicas para Previsao Dinamica do Proximo Acquirer em Software DSM. In *Anais do Simposio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, 1996.

[11] C. B. Seidel, R. Bianchini, and C.L. Amorim. Avaliando a Ténica de Previsão Dinâmica da Passagem de Locks em Sistemas DSM. In *Anais do Simposio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, 1997.

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

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