Avaliando Técnicas de *Prefetching* para *Software DSMs* *

Raquel Gomes Pinto, Ricardo Bianchini, Claudio Luis de Amorim
COPPE Sistemas e Computação - UFRJ
Rio de Janeiro, Brasil 21945-970
raquel@cos.ufrj.br

Resumo

Sistemas de memória compartilhada distribuída com coerência de dados mantida por *software*, denominados *software DSMs*, oferecem a facilidade de programação do modelo de memória compartilhada com o baixo custo das arquiteturas de memória distribuída. No entanto, são poucas as classes de aplicações que alcançam um bom desempenho nesses sistemas. Um dos maiores problemas com *software DSMs* é o *overhead* da busca de dados remotos, que ocorre no caminho crítico da computação. Neste artigo propomos três técnicas de *prefetching* que antecipam essa busca. Analisamos o impacto das técnicas de *prefetching* no desempenho de TreadMarks, usando simulações detalhadas de aplicações reais. Nossos experimentos mostram que o uso dessas técnicas reduz a latência de acesso a dados, mas aumenta o tempo de sincronização e a interferência da busca dos dados com a computação nos processadores remotos. Nossos resultados mostram também que a obtenção de ganhos de desempenho através de *prefetching* depende de um suporte de *hardware* que diminua a interferência em processadores remotos; duas das aplicações que estudamos obtiveram ganhos de 22-39% com o auxílio desse *hardware* especializado.

Abstract

Software-coherent distributed-shared memory systems (software DSMs) provide the ease of shared-memory programming at the low cost of distributed-memory architectures. However, the class of applications that can achieve acceptable performance under software DSMs is fairly limited. One of the main problems with software DSMs is the overhead of fetching remote data in the critical path of the computation. In this paper we introduce the use of three prefetching techniques to minimize this overhead. We study the impact of our prefetching techniques on the performance of TreadMarks, using detailed simulations of real applications. Our results demonstrate that, even though prefetching reduces the latency of data accesses, it increases the synchronization latency and the interference between data fetches and computation on remote processors. Our results also show that performance improvements due to prefetching can only be achieved with hardware support for decreasing the amount of interference between processors; two of our applications exhibited performance gains of 22-39% with this customized hardware support.

*Esta pesquisa foi financiada por FINEP, FAPERJ e CNPq.

1 Introdução

Sistemas de memória compartilhada distribuída com coerência de dados mantida por *software*, denominados *software DSMs*, oferecem a facilidade de programação do modelo de memória compartilhada com o baixo custo das arquiteturas distribuídas. No entanto, são poucas as classes de aplicações que alcançam um bom desempenho nesses sistemas, devido à alta taxa de comunicação e ao *overhead* gerado pela manutenção da coerência dos dados. Um dos maiores problemas de *software DSMs* é o *overhead* da busca de dados remotos, que ocorre no caminho crítico da computação.

Nesse artigo propomos três técnicas de *prefetching* tendo como objetivo minimizar o tempo de espera por dados remotos, através da antecipação da busca dos dados invalidados por outros processadores. Como *software DSMs* modernos utilizam um modelo de consistência de memória relaxada, os dados alterados por processadores remotos só serão invalidados nos pontos de sincronização. As três técnicas de *prefetching* consideram que os dados invalidados em um ponto de sincronização provavelmente serão acessados na seção crítica associada a essa sincronização. Baseado nisso, a primeira técnica realiza o *prefetching* de dados logo após sua invalidação, contanto que tais dados tenham sido efetivamente referenciados. A segunda técnica leva em consideração a existência de aplicações cujos acessos a determinados dados não seguem o padrão de acesso assumido. Criamos então, uma condição que determina quando parar de realizar *prefetching* desses dados. A terceira técnica é uma variação da segunda com o acréscimo da divisão da execução em fases, onde cada fase é delimitada por uma sincronização de barreira.

Analisamos o impacto dessas três técnicas de *prefetching* no desempenho de TreadMarks [KCZ92], usando simulações detalhadas de aplicações reais. Consideramos também o efeito dessas técnicas quando TreadMarks é implementado sobre o computador paralelo NCP_2 [BKP+96, ABS+96], em desenvolvimento na COPPE/UFRJ.

Nossos experimentos mostram que o uso dessas técnicas reduz a latência de acesso a dados, mas aumenta o tempo de sincronização e a interferência da busca dos dados na computação dos processadores remotos. Além disso, mostramos que a obtenção de ganhos de desempenho através de *prefetching* depende do suporte de *hardware* oferecido pelo NCP_2, o qual diminui a interferência em processadores remotos. Com esse *hardware* especializado, duas das aplicações que estudamos, Em3d e Ocean, obtiveram ganhos de desempenho de 22-39%.

O restante desse artigo está organizado da seguinte forma. A seção 2 sumariza as principais características de *software DSMs* e seus *overheads*. A seção 3 descreve as técnicas de *prefetching* estudadas. A seção 4 apresenta a metodologia e as aplicações utilizadas nas simulações. Os resultados são mostrados na seção 5.–Finalizando, a seção 6 conclui o artigo.

2 *Overheads* Envolvidos em *Software DSMs*

Tendo como objetivo apresentar os *overheads* envolvidos em *software DSMs*, descrevemos inicialmente as principais características desses sistemas, tomando como exemplo

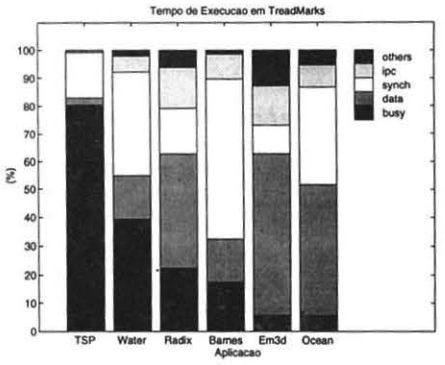

Figura 1: Desempenho de Aplicações sob TreadMarks.

o sistema TreadMarks.

TreadMarks usa *lazy release consistency* [KCZ92] como modelo de consistência de memória, permitindo que alterações a dados compartilhados se tornem visíveis a outros processadores apenas em determinados pontos de sincronização. Nos pontos de aquisição de *locks* ocorrem as invalidações das páginas alteradas por outros processadores. Entretanto, as modificações dessas páginas só são requeridas na ocorrência de um *page fault*. Neste momento são enviados pedidos das modificações (diffs) aos processadores que escreveram na página. As alterações que precisam ser coletadas pelo processador que adquiriu o *lock* são determinadas através da divisão da execução em intervalos, onde cada intervalo é associado a um vetor de *timestamps*. Este vetor descreve uma ordem parcial entre intervalos de processadores distintos. O último dono do *lock* é o responsável pela comparação do seu vetor de *timestamps* com o do novo dono, e pelo envio de *write notices* que indicam a alteração de uma página por um processador em um determinado intervalo. Quando ocorre um *page fault*, o processador percorre sua lista de *write notices* para determinar quais os diffs necessários à atualização da página. Em seguida, ele solicita os diffs correspondentes e espera que sejam gerados e enviados, e neste ponto o processador os aplica na cópia da página que está desatualizada. Uma descrição mais detalhada de TreadMarks pode ser encontrada em [KDCZ94].

Os principais *overheads* em *software DSMs* estão relacionados à latência de comunicação e manutenção de coerência, cujo impacto é ampliado pelo fato de que processadores remotos são envolvidos em todas as transações. A figura 1 apresenta uma visão detalhada do desempenho de aplicações sob TreadMarks em 16 processadores. As barras representam tempos de execução normalizados divididos em *busy time* (tempo de computação útil), latência de busca de dados remotos, tempo de sincronização, *IPC overhead* (tempo dispendido executando pedidos externos), e outros *overheads*. A figura mostra que TreadMarks apresenta significativos *overheads* de busca de dados remotos e sincronização. As técnicas de *prefetching* apresentadas nas

próximas seções têm como objetivo diminuir o *overhead* de espera por dados remotos, sem aumentar significativamente os tempos de sincronização e IPC.

3 Técnicas de *Prefetching* em *Software DSMs*

No caso de TreadMarks, o primeiro acesso a uma página gera um *page fault* e, por conseguinte, a busca da página em um nó remoto. Nos *page faults* seguintes, a página não pode ser inteiramente transferida devido às possíveis escritas feitas por múltiplos processadores simultaneamente. É necessário coletar os diffs de vários processadores e juntá-los. Portanto, realizamos *prefetching* de diffs, enfatizando que isto envolve, além do pedido e envio de diffs, a computação antecipada dessas modificações.

Nossas técnicas de *prefetching* assumem que uma página alterada por um processador e posteriormente invalidada devido a escritas feitas por outros processadores, provavelmente será reutilizada. Dessa forma realizamos *prefetching* para uma página logo após sua invalidação. As seções seguintes descrevem as técnicas propostas.

3.1 Técnica Agressiva de *Prefetching*

A técnica agressiva (**P1**) faz *prefetching* para todas as páginas invalidadas, com a condição de que tenham sido acessadas desde sua última invalidação. Essa técnica é dita agressiva pois não levamos em consideração o número de *prefetches* realizados que não foram úteis à computação.

3.2 Técnica de *Prefetching* Considerando sua Utilização

Considerando-se a existência de aplicações cujos acessos a determinados dados não seguem o padrão de acesso assumido, criamos uma técnica de *prefetching* (**P2**) que é uma variação da anterior, na tentativa de determinar quais os dados que não devem ser trazidos por *prefetching*.

O ponto onde realizamos *prefetching* é o mesmo que na primeira técnica, sendo que levamos em consideração quantas vezes realizamos o *prefetching* dos diffs da página e se ela foi acessada antes de ser invalidada novamente. Para tanto utilizamos dois contadores por página, *useful* e *useless*, indicando quantas vezes a página buscada por *prefetching* foi usada antes de ser invalidada e quantas vezes ela não foi acessada antes da sua invalidação, respectivamente. De posse desses dois contadores, sempre que $useless = 2 \times useful$, paramos de executar *prefetching* dos diffs da página em questão.

3.3 Técnica de *Prefetching* por Fases

Um problema em potencial com a segunda técnica de *prefetch* é que uma página nunca mais será buscada antecipadamente depois de detectada a condição de não realizar *prefetching*. Analisando as aplicações escritas no modelo de memória compartilhada, verificamos que usualmente suas diferentes fases de execução são delimitadas por sincronizações de barreira e que o padrão de acesso aos dados pode variar entre

Constante do Sistema	Valor *Default*
Número de processadores	16
Tamanho da TLB	128 entradas
Tempo de preencher a TLB	100 ciclos
Qualquer interrupção	400 ciclos
Tamanho da página	4K bytes
Tamanho da cache	128K bytes
Tamanho do *write buffer*	4 entradas
Tamanho da linha de cache	32 bytes
Tempo de *setup* da memória	10 ciclos
Tempo de acesso à memória (depois do *setup*)	3 ciclos/palavra
Largura do caminho de dados da rede	8 bits (bidirecional)
Overhead de envio de mensagem	200 ciclos
Latência de chaveamento	4 ciclos
Wire Latency	2 ciclos
Processamento de listas	6 ciclos/elemento
Geração de *twin*	5 ciclos/palavra + acessos à memória
Aplicação e criação de diff	7 ciclos/palavra + acessos à memória

Tabela 1: Valores *Default* dos Parâmetros do Sistema. 1 ciclo = 10 ns.

tais fases. Criamos então uma técnica (**P3**) que recomeça a análise da utilidade de *prefetching* apresentada na técnica anterior a cada nova fase.

4 Metodologia Experimental

4.1 Simulação

Nosso simulador consiste de duas partes: *front end,* Mint [VF94], que simula a execução dos processadores, e *back end,* que simula o sistema de memória em detalhes. O *front end* chama o *back end* em todas as referências a dados (assumimos que busca de instrução sempre resulta em *hit* na cache). O *back end* decide quais processadores ficam bloqueados esperando por memória (ou outros eventos) e quais continuam sua execução.

Simulamos em detalhe uma rede de estações de trabalho com 16 nós. Cada nó consiste de um processador de computação, um *write buffer,* uma cache de dados de primeiro nível diretamente mapeada (assumimos que todas as instruções são executadas em um ciclo), memória local e um roteador de rede em malha (utilizando o roteamento *wormhole*). A tabela 1 resume os parâmetros *default* usados nas simulações. Todos os tempos são dados em ciclos de processador de 10-ns.

4.2 Aplicações Utilizadas

Apresentamos resultados para seis programas: TSP, Barnes, Radix, Water, Ocean e Em3d. TSP é da *Rice University* e faz parte do pacote do TreadMarks. As qua-

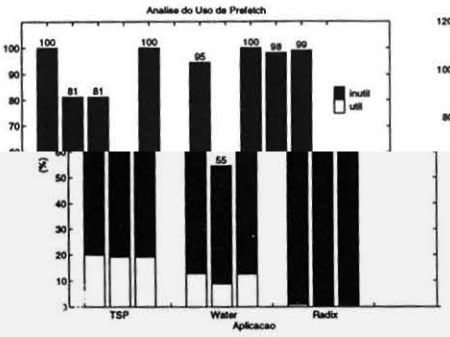

Figura 2: Comparação das técnicas de *prefetching* em relação ao seu uso.

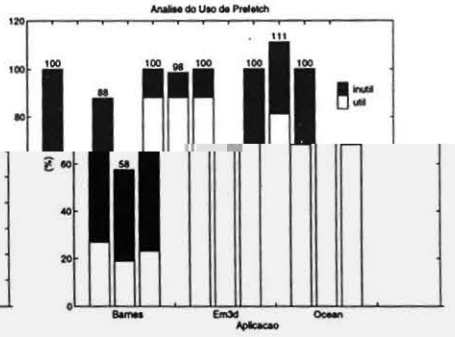

Figura 3: Comparação das técnicas de *prefetching* em relação ao seu uso.

tro aplicações seguintes são do Splash-2 suite [WOT+95]. A última aplicação foi desenvolvida em *University of California at Berkeley.*

TSP usa um algoritmo *branch-and-bound* para descobrir o custo mínimo de se percorrer 18 cidades. Barnes simula a interação de um sistema de 4K corpos sob a influência de forças gravitacionais para 4 passos, usando o método *Barnes-Hut hierarchical N-body.* Radix é um *kernel* que ordena números inteiros. O algoritmo é iterativo, executando uma iteração por dígito das 1M chaves. Water é uma simulação dinâmica de moléculas, que calcula forças enter- e intra-moleculares em um conjunto de 512 moléculas de água. Utiliza-se um algoritmo $\mathcal{O}(n^2)$ para a computação das interações. Ocean estuda movimentos em larga escala de oceanos baseado nas suas correntezas. Simulamos uma grade oceânica de dimensão 258 × 258. Em3d [C*et al.*93] simula a propagação de ondas eletromagnéticas em objetos 3D. Simulamos 40064 objetos elétricos e magnéticos conectados aleatoriamente, com uma probabilidade de 10% de que objetos vizinhos residam em nós distintos. As interações entre objetos são simuladas durante 6 iterações.

5 Resultados Experimentais

Nesta seção avaliamos o desempenho de TreadMarks utilizando as três técnicas de *prefetching* apresentadas na seção 3. Inicialmente analisamos o ganho referente ao número de *prefetches* que foram úteis. E em seguida analisamos as simulações a nível de tempo de execução.

5.1 Análise do Uso das Páginas Trazidas Por *Prefetching*

As figuras 2 e 3 comparam as três técnicas de *prefetching* em cada aplicação. Os gráficos mostram, da esquerda para direita, o número de *prefetches* gerados pelas técnicas: agressiva (**P1**), considerando a utilidade (**P2**), e em fases (**P3**). Cada barra

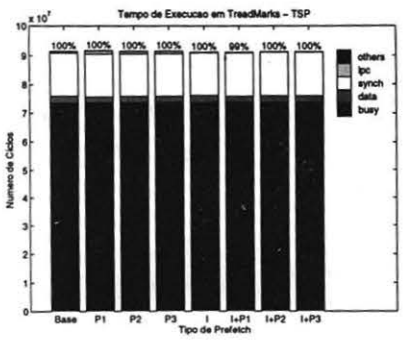

Figura 4: Desempenho de TSP sob TreadMarks com *prefetch*.

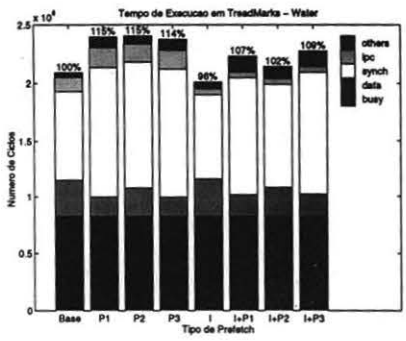

Figura 5: Desempenho de Water sob TreadMarks com *prefetch*.

é dividida em percentual de *prefetches* úteis e inúteis. Um *prefetch* é denominado útil quando a página correspondente é acessada antes de ser novamente invalidada.

Observando as figuras verificamos que Em3d e Ocean são as aplicações que mais se beneficiam com *prefetching*, alcançando 80% de utilização. O comportamento de Em3d nas três técnicas é similar. Em Ocean **P2** aumenta a taxa de utilização de *prefetching*. No entanto, o número total de *prefetches* também aumenta, o que não era esperado visto que esta técnica visa determinar quando se deve parar de realizar *prefetching*. Isto se deve à mudança de temporização, afetando o conjunto de páginas invalidadas em certas operações de sincronização.

Verificamos que TSP, Water e Barnes não se beneficiam com *prefetching* devido aos seus padrões de acesso a dados, que diferem do padrão assumido (páginas invalidadas na sincronização provavelmente serão acessadas na seção crítica em questão). Radix apresenta uma anomalia onde quase todos os *prefetches* não são usados. Essa particularidade se deve a total ausência de localidade de acessos a dados da aplicação.

5.2 Análise do Tempo de Execução

As figuras 4 a 9 mostram o desempenho das aplicações sendo executadas em 16 nós sob TreadMarks com as técnicas de *prefetching*.

Inicialmente analisaremos as quatro barras mais a esquerda que representam, da esquerda para a direita, a versão sem *prefetching* do protocolo (**Base**), a técnica agressiva de *prefetching* (**P1**), *prefetching* considerando sua utilização (**P2**), e a técnica de *prefetching* em fases (**P3**). Novamente, as barras das figuras mostram o tempo de execução dividido em *busy time*, latência de busca de dados remotos, tempo de sincronização, *IPC overhead*, e outros *overheads*.

Observando as figuras verificamos que Ocean é a única aplicação que obteve ganho no desempenho em relação a **Base**. Apesar das técnicas de *prefetching* reduzirem as latências de acesso a páginas em todas as aplicações, invariavelmente há um acréscimo significativo na latência de sincronização, podendo ocorrer também um aumento nos

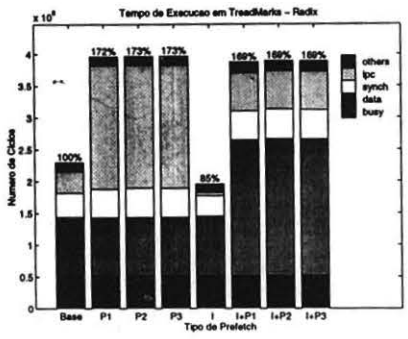

Figura 6: Desempenho de Radix sob TreadMarks com *prefetch*.

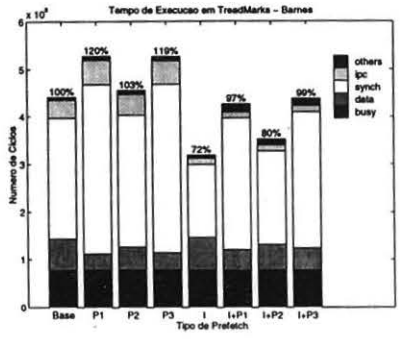

Figura 7: Desempenho de Barnes sob TreadMarks com *prefetch*.

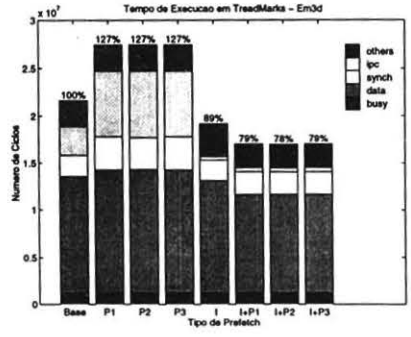

Figura 8: Desempenho de Em3d sob TreadMarks com *prefetch*.

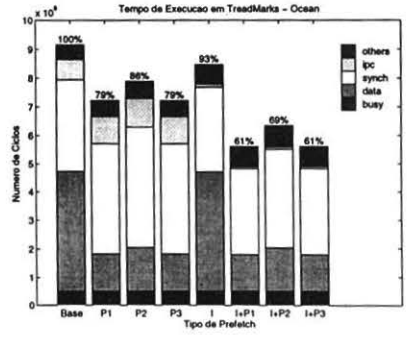

Figura 9: Desempenho de Ocean sob TreadMarks com *prefetch*.

IPC overheads. O aumento do tempo de sincronização é devido ao efeito de *prefetching* sobre seções críticas curtas, tornando-as muito custosas. Esse é o caso de Barnes e Water. O tempo de IPC aumenta quando nós de processamento prevêem incorretamente o seu padrão de acesso futuro, causando a busca de diffs de páginas que não serão realmente usadas. Cada diff buscado desnecessariamente provoca interferência no nó remoto (através de acréscimo ao tempo de IPC) que não ocorre na versão **Base**. De acordo com a subseção anterior, Barnes, Water e Radix apresentam um alto percentual de *prefetches* não usados, explicando o baixo desempenho alcançado por essas aplicações. Temos ainda um outro problema relacionado a transferência de diffs. As técnicas de *prefetching* concentram essas transferências em determinados intervalos de tempo gerando degradação de desempenho da rede e do barramento de memória. Barnes e Em3d são as aplicações que mais sofrem desse tipo de degradação. Verificamos também que não existe grande diferença nos tempos de execução entre

as três técnicas de *prefetching*.

As quatro barras mais a direita representam o tempo de execução das aplicações sob TreadMarks no NCP_2, que utiliza um controlador de protocolos responsável por reduzir a carga de protocolo do processador de computação. A barra **I** representa a versão onde o processador de computação só é interrompido para tratar de operações mais complexas, como percorrer listas e manipular *write notices*. As operações mais simples são tratadas pelo controlador de protocolo. As outras barras (**I+P1, I+P2, I+P3**) representam as execuções com as respectivas técnicas de *prefetching*.

Em3d e Ocean são as aplicações que obtiveram uma melhora do desempenho com o uso de *prefetching*. Ocean alcançou um ganho de até 39%. TSP não obteve nenhum ganho como era esperado, visto que o tempo de espera por dados remotos é mínimo. Water e Barnes apresentam um aumento do tempo de sincronização devido ao efeito de *prefetching* sobre seções críticas curtas. Radix, devido a ausência de localidade, não obteve melhora na latência de página.

Analisando as figuras verificamos que em todas as aplicações a utilização de *prefetching* com um *hardware* especializado (**I+P**) alcança melhor desempenho do que *prefetching* isolado, como resultado da eliminação virtual do IPC e redução do tempo de sincronização. A diminuição desses *overheads* se deve principalmente ao fato de que o processador ao ser interrompido por um pedido remoto de diff, não necessita dispender o seu tempo de computação com geração de diffs.

6 Conclusão

Neste artigo avaliamos a utilização de técnicas de *prefetching* em *software DSMs*, na tentativa de minimizar o *overhead* de busca de dados em nós remotos. Os únicos trabalhos que utilizam *prefetching* em *software DSMs* são [KCDZ95, Kel95, SL94, DCZ95], sendo que o impacto dessa técnica no desempenho dos protocolos não é analisado ou o uso de um compilador que insira diretivas para *prefetching* é necessário.

As técnicas de *prefetching* foram baseadas na suposição de que páginas invalidadas em pontos de sincronização provavelmente serão acessadas na seção crítica referente àquela sincronização. Verificamos que esta suposição não é válida na maioria das aplicações estudadas aqui, pois o número de páginas trazidas com antecedência e não acessadas é significativo. Entretanto, as aplicações que seguem este padrão de acesso, como Ocean e Em3d, alcançam bons resultados. A segunda técnica apresentada (**P2**) demonstrou ser a melhor escolha, considerando sua simplicidade e seu desempenho.

Ressaltamos que o objetivo de diminuir a latência de busca de dados remotos foi alcançado, entretanto não esperávamos o aumento excessivo do tempo de sincronização e *IPC overheads*. Utilizando um *hardware* especializado praticamente eliminamos essas latências, visto que o processador não dispende tempo de computação na geração de diffs durante o atendimento a *prefetches* que podem não ser usados. Logo é essencial o suporte de *hardware*, como no NCP_2 para se obter um bom desempenho. Observamos que possivelmente a ordem de *prefetching* das páginas influencie o desempenho das aplicações, explicando porque o tempo de espera por páginas em Em3d não diminuiu com as técnicas de *prefetching*. Essa ordem é importante no caso em

que a primeira página acessada na seção crítica for uma das últimas que se realizou *prefetch*, obrigando o processador a esperar pelo recebimento de vários diffs antes dos referentes ao *page fault* corrente. *Software prefetching* pode ser uma outra alternativa para se alcançar um maior ganho, mas esta solução envolve o desenvolvimento de compiladores complexos.

Referências

[ABS+96] C. Amorim, R. Bianchini, G. Silva, R. Pinto, M. Hor-Meyll, M. De Maria, L. Whately, and J. Barros Jr. A segunda geração de computadores de alto desempenho da COPPE/UFRJ. *Anais do VIII SBAC-PAD*, 1996.

[BKP+96] R. Bianchini, L. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. Amorim. Hiding communication latency and coherence overhead in software DSMs. *To appear in Proc. of the 7th ASPLOS*, October 1996.

[C*et al.*93] D. Culler *et al.* Parallel programming in split-c. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[DCZ95] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Compiler-directed selective update mechanisms for software and distributed shared memory. Technical Report 95-253, Rice University, 1995.

[KCDZ95] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *The Journal of Parallel and Distributed Computing*, 29, September 1995.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th ISCA*, pages 13–21, May 1992.

[KDCZ94] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter '94 Technical Conference*, pages 17–21, Jan 1994.

[Kel95] P. Keleher. Sparks: Coherence as an abstract type. Technical Report CS-TR-3543, University of Maryland, 1995.

[SL94] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proc. of the 1st OSDI*, 1994.

[VF94] J. E. Veenstra and R. J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proc. of the 2nd MASCOTS*, 1994.

[WOT+95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd ISCA*, pages 24–36, May 1995.