Implementação e Avaliação de Entry Consistency

Maria Carolina Regino Carneiro,
Ricardo Bianchini e Cláudio Luis de Amorim

COPPE Sistemas
Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brasil

carol@cos.ufrj.br

Resumo

Este artigo descreve a implementação e o desempenho de um protocolo de coerência de dados baseado no modelo *Entry Consistency* de consistência de memória. Nossa implementação foi escrita em C e usa a biblioteca PVM para passagem de mensagens. Nossos resultados foram obtidos em um *cluster* de processadores da máquina SP2 e uma rede de *workstations*. Tais resultados foram contrastados com o desempenho de PVM utilizado explicitamente em ambas as plataformas. Essa comparação mostra que nossa implementação atual apresenta escalabilidade deficiente, devido a limitações no tratamento de sinais do próprio PVM. Concluimos que é necessária uma implementação alternativa do nosso protocolo, a qual deve obter desempenho semelhante a PVM aplicado explicitamente para uma grande classe de aplicações.

Abstract

This paper evaluates the performance of a coherence protocol based on the Entry Consistency relaxed memory consistency model. Our protocol implementation was written in C and uses the PVM library for message passing. Performance results obtained on an SP2 and on a network of workstations have been compared against explicit message passing under PVM. This comparison shows that our current implementation of the protocol exhibits limited scalability, due to the poor performance of signal operations in PVM. We conclude that an alternative implementation of our protocol is required. This implementation should then achieve comparable performance to explicit message passing for a large number of applications.

1 Introdução

O modelo mais simples de programação paralela é aquele que assume a existência de um espaço de endereçamento único. Esse modelo de programação é tipico dos multiprocessadores de memória centralizada, tais como o Onyx da Silicon Graphics, e o Balance da Sequent. Entretanto, arquiteturas desse tipo não são escaláveis.

A implementação do modelo de memória compartilhada em sistemas com memória fisicamente distribuída tem se mostrado uma alternativa bastante promissora. Sistemas deste tipo são chamados de sistemas com memória compartilhada distribuída (DSM) e oferecem um modelo de programação simples em um hardware escalável.

O uso de memórias privativas, porém, requer solução para o problema de coerência de memórias. A solução desse problema tem grande impacto no desempenho do sistema como um todo. Algumas soluções são implementadas em software e se baseiam em ações tomadas pelo programador, pelo compilador ou pelo sistema operacional, e outras são fortemente suportados pelo hardware da máquina. Soluções de hardware são mais transparentes para o programador ou compilador, enquanto que soluções de software são mais baratas e adaptáveis.

Para garantir a coerência das memórias eficientemente é necessário utilizar modelos relaxados de consistência de memória [7, 11], já que esses modelos permitem reduzir boa parte da latência dos acessos de escrita a dados compartilhados.

Nesse trabalho, avaliamos o comportamento de um modelo relaxado de consistência de memória, o modelo *Entry Consistency* (EC). O modelo EC determina que apenas os dados a serem acessados dentro de uma seção crítica precisam se tornar coerentes dentro dela. Nossa implementação desse modelo foi denominada ECP (*Entry Consistency Protocol*) e se baseou nos principais aspectos de implementação do sistema Midway, descrito em [3]. Estamos interessados em comparar o custo de se utilizar uma versão compartilhada (que assume espaço de endereçamento global) e uma versão implementada com passagem de mensagens explícita, da mesma aplicação. Com isso, investigaremos o *overhead* gerado pelo protocolo de coerência de memória e o seu custo no tempo total de execução de uma aplicação paralela.

O protocolo descrito nesse trabalho, ECP, apresenta uma versão mais completa e otimizada da implementação do modelo EC descrita em [4]. Naquele artigo, foi feita uma comparação entre protocolos relativamente ineficientes baseados nos modelos *Release Consistency* (RC) [7] e EC, no que diz respeito ao número e ao tamanho das mensagens transferidas nos sistemas. Apesar de importantes, tais métricas não permitem a avaliação completa dos protocolos sugeridos. Assim, nesse artigo, consideramos o tempo total de execução de ECP rodando aplicações paralelas no computador SP2 e em uma rede de *workstations* Sun.

O restante desse artigo está organizado da seguinte forma: na seção 2, descrevemos o modelo *Entry Consistency*. Na seção 3, descrevemos os aspectos mais importantes da implementação do ECP, mostrando como são implementadas operações de sincronização. Na seção 4, discutimos o desempenho do protocolo para uma aplicação de multiplicação de matrizes, comparando seus resultados da versão compartilhada com a versão distribuída da mesma. Finalmente, na seção 5, apresentamos nossas conclusões e possíveis trabalhos futuros.

2 O Modelo *Entry Consistency*

O modelo *Entry Consistency* (EC) de consistência de memória foi desenvolvido por Bershad *et al* em [2]. Protocolos baseados nesse modelo em geral associam cada dado compartilhado a uma variável de sincronização. Essa variável é chamada de **guarda** do dado compartilhado e é ela que controla o acesso à seção crítica. Um dado compartilhado estará consistente quando for realizada uma operação *acquire* na variável de sincronização que o guarda. Sendo s uma variável de sincronização que guarda o dado D_s, a condição formal para que um sistema esteja consistente, segundo o modelo EC, é que uma operação *acquire* a s só pode ser observado em um processador p quando todas as atualizações a D_s já foram observadas em p.

Protocolos de coerência baseados no modelo EC [2, 1] podem se aproveitar da relação entre variáveis de sincronização e os dados compartilhados que elas protegem para diminuir a quantidade de mensagens e dados de coerência enviados.

As operações de sincronização no modelo EC são categorizadas como operações *acquire* e *release*. *Acquires* e *releases* correspondem, de uma forma geral, a operações *lock* e *unlock*, utilizadas para proteger uma seção crítica. Esse modelo diferencia ainda operações *acquire* como sendo de modo **exclusivo** ou de modo **não-exclusivo**. Um *acquire* no modo não-exclusivo permite que vários processadores estejam simultaneamente executando a mesma seção crítica, o que deve ser usado quando são realizadas somente leituras no dado. A atualização de um dado compartilhado deve ser feita no modo exclusivo.

A associação de variáveis de sincronização com dados compartilhados e a diferenciação entre *acquires* exclusivos e não-exclusivos são características específicas dos protocolos baseados no modelo EC. Entretanto, elas requerem algumas alterações no código das aplicações escritas para outros modelos de consistência. Muitas vezes, essas alterações não são triviais.

3 Implementação

O protocolo ECP, baseado no modelo EC, foi implementado conforme o sistema Midway [3]. Foi desenvolvido em linguagem C, utilizando-se Parallel Virtual Machine (PVM) [6]. PVM é um sistema que fornece primitivas de programação para trocas de mensagens em redes de *workstations* e computadores paralelos.

O ECP propaga as atualizações realizadas em um determinado dado somente na execução do proximo *acquire* no mesmo dado. É utilizado um esquema de **atualização** na propagação das modificações nos dados compartilhados, mas também ocorrem **invalidações** nesses dados. O esquema de **invalidação** é utilizado para evitar que *locks* não-exclusivos gerem trocas de mensagens desnecessárias.

Quando um processador p executa um *acquire* numa variável s, se p não possui cópia válida do dado D_s associado a s, ele busca D_s no último processador a executar um *acquire* exclusivo em s (i.e, no último processador a modificar D_s). Se o *acquire* é exclusivo, significa que p vai escrever em D_s e, por esse motivo, p envia mensagens de invalidação para os processadores que possuem cópia válida de D_s. Num *acquire*

não-exclusivo, p só teria necessidade de buscar D_s se ele estivesse inválido.

O ECP consiste de dois componentes principais. O primeiro deles é uma biblioteca que implementa todas as funções utilizadas pela aplicação, para a realização das operações de *lock*, *unlock*, barreiras e outras, que serão descritas ainda nessa seção. Essas funções geram mensagens que deverão ser enviadas a outros nós de processamento. O segundo componente é responsável por monitorar a chegada de uma mensagem em um processador e tratá-la de acordo com seu tipo.

Antes de disparar os processos paralelos, o processo deve realizar a associação das variáveis de sincronização com os dados compartilhados. A função *ec-bind* realiza essa associação. É no momento dessa associação que os dados passam a ser vistos como compartilhados. Variáveis de sincronização são representadas por números inteiros e declaradas somente no momento das chamadas a *ec-bind*. A função *ec-fork* dispara os processos paralelos que vão executar nos outros processadores.

3.1 Implementação das Operações de Lock/Unlock

O ECP fornece ao usuário dois tipos de operações de sincronização: *lock/unlock* e as barreiras, sendo que *locks* podem ser adquiridos em modo exclusivo (pela função *ec-lock-ex*) para leitura e escrita, e em modo não-exclusivo (pela função *ec-lock-nex*) somente para leitura. A função *ec-release* implementa a operação de *unlock* e a função *ec-barrier* faz a realização de uma barreira.

Os *locks* exclusivos e não-exclusivos foram implementados usando-se o conceito de *owner* de uma variável de sincronização. Dado que s é a variável de sincronização que guarda o dado D_s, *owner* é o identificador do processador que possui, ou que possuiu por último, o acesso de *lock* exclusivo a s e, que, portanto, está modificando ou modificou D_s. E chamaremos de *requerente* o processador que está fazendo o pedido de *lock*.

Quando um processador *requerente* precisa de uma cópia de um dado compartilhado, ele tem que fazer a busca desse dado na rede. O algoritmo utilizado nessa busca se chama *best-guess*. Nesse algoritmo, cada processador tem, para cada variável de sincronização, a informação do processador que adquiriu a *ownership* da variável depois dele. Esse identificador é guardado em uma variável chamada *best-guess* e a operação de busca ao dado compartilhado começa por essa informação. O *requerente* envia ao seu *best-guess* um pedido de acesso a variável de sincronização. Se ele é realmente o *owner* da variável, o dado foi encontrado. Se não for, isso significa que ele já transferiu a *ownership* da variável para outro processador. Então, o *best-guess* que recebeu o pedido o avança para o seu *best-guess*. O pedido do *requerente*, então, trafega pela rede até que se encontre o dado compartilhado.

A seguir, faremos a descrição da implementação dos *locks* exclusivos e não-exclusivos, juntamente com a análise de todas as possíveis trocas de mensagens entre os processadores. É através dessa análise que se pode iniciar uma discussão a respeito do *overhead* causado pelo protocolo.

O pedido de *lock* exclusivo em uma variável s é realizado da seguinte forma, quando o processador requerente é o *owner* de s:

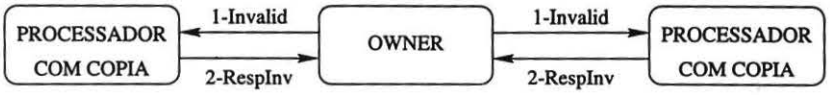

Figura 2: Protocolo de Invalidação

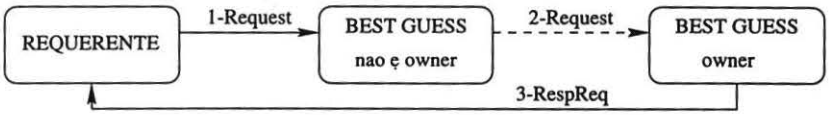

Figura 3: Operação de *lock* exclusivo quando o requerente não é o *owner*

- caso nao hajam cópias de D_s fornecidas para leitura, o *lock* é adquirido imediatamente, sem a troca de mensagens.
- no caso de existirem cópias de D_s, essas são invalidadas antes do *lock* ser adquirido. Mensagens de invalidação são enviadas a todos os processadores que possuem o *lock* não-exclusivo de s. Assim que os processadores completam a invalidação, eles enviam ao *owner* uma mensagem RESP-INV de resposta. Quando chegarem ao *owner* todas as respostas, o processador requerente já pode adquirir o *lock* exclusivo de s e modificar D_s. A figura 2 mostra a troca de mensagens necessária para que seja feita essa invalidação.

Quando o processador requerente não é o *owner* de s:

- um pedido é enviado ao *best-guess* de s, e se ele não for o *owner*, o pedido vai ser passado adiante até que chegue ao *owner*. O *owner* então, vai tomar o mesmo procedimento de invalidação das cópias de D_s, descrito acima, caso hajam cópias de leitura. Depois, ele envia ao processador requerente uma mensagem contendo o dado D_s em sua versão mais recentemente modificada, transformando o processador requerente no novo *owner* de s. Essa operação é mostrada na figura 3 e a mensagem 2 representa o *forward* do pedido até se achar o *owner*.

O pedido de *lock* não-exclusivo em uma variável s é realizado da seguinte forma:

- se o processador requerente é *owner* de s, o *lock* é adquirido, lembrando que, mesmo sendo *owner*, o processo só poderá fazer leituras a D_s, sem troca de mensagens.
- se não for *owner*, o processador requerente envia uma mensagem ao processador *best-guess* e esse verifica se tem uma cópia válida do dado pedido. Se não tiver, ele passa adiante o pedido até que se chegue a um processador que tenha uma cópia de D_s. Tal processador pode então retornar D_s ao requerente. Note, portanto, não é necessário que se encontre o *owner* de s, tornando esta operação mais eficiente. A figura 4 representa essa operação.

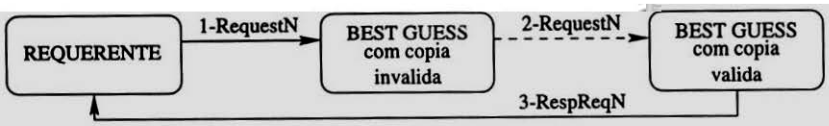

Figura 4: Operação de *lock* não-exclusivo quando o requerente não possui uma cópia válida de D_s

Assim que leituras e/ou modificações ao dado D_s terminam, o processo que adquiriu o *lock* já pode fazer o *unlock* (i.e., o *release*) da variável s, para que ela fique disponível aos demais processadores. Durante o tratamento de um *lock* exclusivo a s, os outros pedidos de *lock* exclusivo e não-exclusivo a s que chegam ao *owner* são enfileirados. Durante o tratamento de um *lock* não-exclusivo a s, os pedidos de *lock* exclusivo a s tambem são enfileirados, para que possam ser atendidos quando da liberação da variável s (no *unlock*). Essa operação não requer troca de mensagens.

3.2 Implementação da Barreira

Barreiras são utilizadas somente para sincronizar um conjunto de processadores. Quando um processador alcança uma barreira, ele permanece paralisado até que os demais processadores envolvidos na operação alcancem também a barreira.

O conceito de **gerente** foi usado na implementação das barreiras. O gerente de uma dada barreira, cuja variável de sincronização correspondente é b, é o processador que controla a variável b e quantos processos estão envolvidos nessa operação. Os processos envolvidos formam o conjunto P. A barreira é, então, implementada da seguinte forma:

- um processo do conjunto P, assim que alcança a barreira b, envia uma mensagem ao gerente, indicando que alcançou a barreira.
- o gerente vai contabilizando essas mensagens, a medida em que as vai recebendo e, assim que as mensagens de todos os processos do conjunto P são recebidas, o gerente libera esses processos para darem prosseguimento ao seu processamento.
- os processos que fazem parte de P recebem a mensagem e já podem prosseguir no seu processamento.

Note que as barreiras não lidam com operações de coerência de dados. Elas trabalham somente no sentido de sincronizar a execução da aplicação. Todo tráfego de dados gerado durante uma execução fica a cargo das variáveis de sincronização do tipo *lock*.

4 Avaliação de Desempenho

Nesta seção, observamos o comportamento de um programa paralelo simples, escrito sob o modelo de programação imposto pelo ECP, implementando *Entry Consistency*.

Todas as medidas feitas foram em função dos tempos da aplicação. Descrevemos, também nessa seção, a plataforma de hardware e software usada para implementar e executar programas escritos para o ECP.

4.1 Ambiente Computacional

O ECP foi implementado em linguagem C e utilizamos o PVM versão 3.0 como ferramenta para passagem de mensagem. PVM foi desenvolvido no Oak Ridge National Laboratory. Ele permite que uma rede de computadores Unix, paralelos ou sequenciais, pareça um único recurso computacional paralelo, ao que chamamos de máquina virtual. PVM fornece funções para se iniciar processos na máquina virtual e permite que esses processos se comuniquem uns com os outros através de mensagens.

Nossa implementação do ECP é executada em dois ambientes diferentes. Um deles é uma rede de *workstations* Sun, modelos Sparc 4 e Sparc 5. O outro é o computador paralelo IBM-SP2, com um *cluster* de 16 nós de processamento. No SP2, utilizamos 10 nós de processamento para execução exclusiva, que é o máximo cedido pelo sistema a que temos acesso. A rede de interconexão dos nós é uma Ethernet nos dois ambientes.

4.2 Aplicações

Inicialmente, escolhemos uma aplicação simples, multiplicação de matrizes, para a validação da implementação do ECP e para que fosse possível comparar as duas versões, compartilhada e com passagem de mensagem.

Na versão compartilhada, as matrizes A e C serão acessadas por todos os processadores, para leitura e escrita dos dados. A matriz B é acessada pelos processadores da seguinte forma: cada processador é o *owner* de um conjunto de m/n colunas diferentes de B e realiza somente leituras sobre essas colunas. A distribuição inicial das colunas da matriz B foi feita de forma a otimizar a execução da aplicação, evitando tráfego inútil de dados pela rede. Cada processador calcula os elementos de m/n colunas da matriz C. A matriz C resultante foi distribuída de maneira idêntica a matriz B.

Na versão com passagem de mensagem, inicialmente é realizada a distribuição dos dados, enviando a matriz A para os n processadores e dividindo a matriz B em colunas de forma a enviar m/n colunas para cada processador. Os processos, ao receberem esses dados, calculam suas respectivas colunas da matriz C e as enviam para o processador que realizou a distribuição dos dados.

O uso de *benchmarks* padrões para memória compartilhada, como o SPLASH-2 [10], será considerado em um segundo momento, pois essas aplicações não possuem versão com passagem de mensagem conhecida.

4.3 Resultados

Nas Tabelas 1 e 2 apresentamos os resultados obtidos para a aplicação de multiplicação de matrizes, programada nas duas versões citadas anteriormente. Para cada

uma das execuções mostramos o tempo total de execução da aplicação em segundos e o respectivo *speedup*. Mostramos também o *overhead* gerado pelo sistema ECP (no caso da versão compartilhada) e o *overhead* gerado pelo PVM (no caso da versão com passagem de mensagem explícita). Esse *overhead* corresponde à porcentagem do tempo de execução dispendida com operações do sistema em questão.

		Usando ECP			Usando passagem de msg		
		Execução		Overhead	Execução		Overhead
Matriz	Procs	Tempo	Speedup		Tempo	Speedup	
512x512	4	32	2.84	37%	32	3.4	21%
	8	32	2.84	68%	23	4.73	47%
	10	35	2.6	77%	22	4.95	59%
1024x1024	4	231	3.04	29%	234	3.64	12%
	8	179	3.93	53%	145	5.87	29%
	10	185	3.8	64%	127	6.7	40%

Tabela 1: Multiplicação de Matrizes executada no SP2.

		Usando ECP			Usando passagem de msg		
		Execução		Overhead	Execução		Overhead
Matriz	Procs	Tempo	Speedup		Tempo	Speedup	
512x512	4	49	3.06	31%	52	3.3	23%
	8	40	3.75	57%	49	3.51	61%
	10	52	2.88	69%	39	4.41	61%
1024x1024	4	470	2.97	39%	403	3.57	16%
	8	330	4.23	58%	301	4.78	42%
	10	550	2.54	74%	377	3.81	63%

Tabela 2: Multiplicação de Matrizes executada na rede de *workstations*.

A rede de *workstations* utilizada não estava totalmente dedicada à aplicação, o que significa que havia multiprogramação nos processadores, causando aumento do tempo total de execução da aplicação. O sistema SP2 foi utilizado com exclusividade.

Pelas tabelas podemos notar que os *overheads* obtidos na execução da aplicação no SP2 são menores do que os *overheads* obtidos na execução em uma rede de *workstations*, devido ao alto custo de troca de mensagens desta arquitetura.

Em ambos as plataformas, comparando o *overhead* obtido para o ECP e a passagem de mensagens, observamos que PVM é responsável pela maior parte do *overhead* gerado pelo ECP. O restante desse *overhead* se refere a trocas de mensagens extras por motivos de sincronização e coerência de memória. No caso do sistema ECP, por exemplo, o algoritmo *best-guess* gera diversas mensagens somente para encontrar o *owner* de uma variável de sincronização.

Comparando o tempo de execução da versão compartilhada com a versão com passagem de mensagens explícita, em ambas as plataformas, notamos que a versão com passagem de mensagens obteve melhor desempenho. Em nossos experimentos obtivemos para o sistema ECP desempenho médio 25% inferior ao sistema PVM, o que consideramos bastante aceitável para um protocolo de consistência implementado totalmente em software, e que está de acordo com resultados descritos em trabalhos anteriores [5, 9].

Os *speedups* do sistema ECP obtidos para 8 e 10 processadores foram muito baixos. A explicação para esse fato está no alto custo do sistema PVM. Esse sistema executa um processo *deamon* em cada processador, compartilhando tempo da CPU com a aplicação, e precisa ainda chamar o sistema operacional para trocar mensagens. Esse mesmo custo influenciou negativamente o desempenho das versões com troca de mensagens conforme mostram resultados de *speedup* obtidos.

Além disso, no caso dessa aplicação, quanto maior o número de processadores, mais pedidos de leitura de um dado compartilhado chegam a um único processador. Como o atendimento desses pedidos é feito sequencialmente e depende do tratamento de sinais em PVM, o tempo de atendimento acaba aumentando muito. No momento, estamos modificando o código de ECP para reduzir esse *overhead.*

5 Conclusões

Apresentamos nesse trabalho a implementação e o desempenho de um protocolo de coerência de dados baseado no modelo *Entry Consistency.* Executamos no nosso protocolo a aplicação de multiplicação de matrizes e comparamos com o desempenho da mesma aplicação executando com passagem de mensagem explícita utilizando PVM. Obtivemos *speedup* baixo, mas não tão distante do *speedup* obtido para o sistema PVM. O que indica que o alto custo do PVM tem bastante influência no tempo total de execução da aplicação.

Nossa implementação atual apresenta escalabilidade deficiente, devido a limitações no tratamento de sinais do próprio PVM. Concluimos que é necessária uma implementação alternativa do nosso protocolo, a qual deve obter desempenho semelhante a PVM aplicado explicitamente para uma grande classe de aplicações.

Como trabalhos futuros, pretendemos executar outras aplicações no ECP e utilizar uma rede de interconexão mais eficiente. Para isso será necessário a utilização do sistema PVMe que possibilita a execução em redes mais velozes como redes ATM.

Agradecimentos

Gostaríamos de agradecer ao Laboratório Nacional de Computação Científica, que nos forneceu acesso ao IBM-SP2. Agradecemos também a todo pessoal do Laboratório de Computação Paralela da COPPE-Sistemas, especialmente a Cristiana Seidel, Gabriel Silva, Lauro Whately, Paula Maciel e Raquel Pinto, por todas as suas contribuições.

Referências

[1] Amorim,C.L., Seidel,C.B., Bianchini,R., "The Afinity Entry Consistency Protocol",*Technical Report ES-399/96, COPPE Systems Engeneering, Federal University of Rio de Janeiro*, May 1996.

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

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

[4] Carneiro, M.C.R, Pinto, R.C.G., Seidel, C.B., Silva, A.B.R., Silva, M.G. e Amorim, C.L., "Desempenho Simulado de Modelos Fracos de Consistência de Memória", *Anais do VII SBAC - PAD*, Gramado, RS, 1995.

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

[6] Geist, A., Beguelim, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.,*PVM: Parallel Virtual Machine - a User's Guide and Tutorial for Networked Parallel Computing*, The MIT Press, 1994.

[7] Gharachorloo, K., Gupta A. and Henessy, J.L. "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors",*Proc of the 4th Int. Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp 245-257.

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

[9] Lu, H., Dwarkadas, S., Cox, A., and Zwaenepoel, "Message Passing Versus Distributed Shared Memory on Networks of Workstation", *Proceedings of Supercomputing' 95*, December 1995.

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

[11] Zucker, R.N. and Baer, J-L, " A Performance Study of Memory Consistency Models", *Proc of the 19th Int. Symp. on Computer Architecture*, May 1992, pp 2-12.