Um Framework para Definição e Análise de Modelos de Consistência de Memória

Hammurabi C. Mendes, Diego F. Aranha, Alba C. M. A. Melo, Mário A.R. Dantas
Department of Computer Science
University of Brasilia, Brazil
{hmendes,dfaranha,albamm,mario}@cic.unb.br

Resumo

A Memória Distribuída Compartilhada (DSM) foi proposta para permitir que o paradigma de programação por memória compartilhada seja utilizado em arquiteturas distribuídas. O comportamento dos sistemas DSM é determinado pelo modelo de consistência da memória. De modo a prover melhor compreensão sobre a semântica dos modelos de consistência da memória, vários pesquisadores propuseram formalismos para defini-los. Mesmo com definições formais, ainda é difícil dizer que tipos de históricos de execução podem ser produzidos em um determinado modelo de consistência. Neste artigo, é proposto um framework que serve como base para a implementação da análise de históricos de execução em diversos modelos de consistência, provendo os algoritmos e estruturas de dados convenientes para tal. O nosso framework é organizado em módulos e permite que o usuário defina novos modelos de consistência de uma maneira simples. Um protótipo foi implementado em C, incluindo também os programas que lidam com históricos em PipelinedRAM e Release Consistency. Quando comparados com suas implementações originais, o uso do framework permitiu uma redução de em torno de 70% no tamanho do código para os dois modelos.

1. Introdução

A Memória Distribuída Compartilhada (Distributed Shared Memory - DSM) é uma abstração que permite que o paradigma de programação por memória compartilhada seja utilizado em arquiteturas paralelas ou distribuídas onde não existe memória fisicamente compartilhada entre os processadores. O Modelo de Consistência da Memória (MCM) é uma parte essencial de um sistema DSM pois define a ordem na qual os acessos à abstração de memória criada pela DSM serão vistos pelo programador. Os primeiros sistemas DSM tentaram fazer com que a memória compartilhada distribuída se comportasse exatamente como a memória física do monoprocessador. Para oferecer um modelo de memória tão forte, é necessário um alto tráfego na rede, que reduz o desempenho das aplicações DSM e freqüentemente leva o sistema como um todo a um estado de saturação. Para tratar deste problema, os pesquisadores propuseram relaxar algumas condições de consistência, criando assim novos comportamentos da memória que são diferentes do comportamento da memória do monoprocessador.

Muitos Modelos de Consistência têm sido propostos na literatura. Originalmente, os requisitos de consistência não foram formulados de maneira formal. Em alguns casos, isso levou a diferentes interpretações sobre o comportamento do mesmo Modelo de Consistência de Memória. Para sanar este tipo de problema, os pesquisadores propuseram modelos de sistema formais onde diversos Modelos de Consistência podem ser definidos [1] [4] [6]. Mesmo com definições formais, fica ainda difícil dizer se ordenamentos indesejáveis de operações poderão se produzir em determinado modelo, já que a maioria dos modelos de memória mais difundidos é extremamente complexa.

Após termos investigado diversos modelos de consistência, achamos que é de fundamental importância determinar, em primeiro lugar, que características são intrínsecas a todos os modelos de consistência e que características são particulares e, em segundo lugar, se as características particulares de cada modelo podem ser enquadradas no mesmo arcabouço. Destes questionamentos, surgiu a proposta de um framework único onde diversos modelos de consistência podem ser especificados e analisados.

O objetivo deste artigo é descrever o projeto e implementação deste framework, que permite que o usuário analise diversos modelos de consistência definidos segundo a mesma base formal e também possa, caso desejar, definir e analisar novos modelos de consistência. O framework aqui proposto utiliza o modelo de sistema definido em [6] como base sobre a qual os modelos serão construídos. Um programa que tenha sido implementado sobre o framework recebe como entrada

um histórico de execução e aplica as restrições impostas pelo Modelo de Consistência. Dependendo do MCM, é considerada uma visão global ou uma visão parcial das operações de acesso à memória. Em cima da visão, são definidas as restrições de ordem. A partir daí, são geradas automaticamente todas as seqüências que respeitam as restrições constantes no modelo. Para analisar um histórico de execução baseado em um novo modelo, basta que se defina as restrições impostas por ele, através do uso das funcionalidades providas pelo framework.

Para mostrar a funcionalidade do framework, o mesmo foi utilizado para a implementação de um modelo relaxado uniforme (PipelinedRAM) e de um modelo híbrido (Release Consistency), sem problemas. Conseguimos com isso uma redução de em torno de 70% em linhas de código para ambos os modelos. Aliada a esta redução no esforço de programação, há uma independência entre a implementação das estruturas necessárias ao processo de obtenção das ordens válidas e ao processo em si.

A análise de um mesmo histórico de execução em diversos modelos de consistência no nosso framework mostra que o mesmo histórico pode ser válido em um modelo e inválido em outro. Ainda, alguns históricos que o programador achava que não seriam produzidos em determinado modelo são mostrados como válidos, o que é de grande valia, pois indica que o programa deve ser reestruturado ou que um modelo de consistência mais forte deve ser utilizado.

O restante deste artigo está organizado como se segue. Na seção 2, os modelos de consistência de memória são apresentados. A seção 3 descreve como obter as possíveis ordens que poderiam ter levado à produção de um determinado histórico de execução. O framework para análise de modelos de consistência é apresentado na seção 4. Alguns resultados experimentais são apresentados na seção 5. Finalmente, a seção 6 conclui o artigo e apresenta trabalhos futuros.

2. Modelos de Consistência de Memória

Intuitivamente, o programador assume que as instruções que compõem o seu programa são executadas uma após a outra (de uma maneira seqüencial) e que as operações de acesso à memória são atômicas. Este modelo informal é utilizado pelo programador para raciocinar sobre os resultados que podem ser produzidos por seu programa. Um Modelo de Consistência de Memória formaliza este conceito pois define a ordem na qual as operações de acesso à memória devem ser percebidas pelo programador [1]. Como ele define a ordem aparente da execução de operações de acesso à memória e não sua ordem real, muitas otimizações puderam ser feitas em monoprocessadores, ainda respeitando o modelo intuitivo. No entanto, quando as mesmas otimizações são aplicadas a um ambiente paralelo ou distribuído, o modelo intuitivo é violado e a programação torna-se mais complexa, já que o programador torna-se consciente da natureza distribuída da memória física.

Históricos de execução são utilizados freqüentemente para estudar modelos de consistência de memória [9] e podem ser vistos como um "trace" de uma execução de um programa paralelo. A figura 1 apresenta um histórico de execução composto por 3 processadores, P1, P2 e P3. O tempo corresponde ao eixo horizontal e cresce da esquerda para a direita. Nesta representação, cada processador *Pi* possui sua memória *Mi,* que é uma cache completa de todo o espaço de endereçamento, ou seja, a memória é totalmente replicada [6]. No início da execução, todas as posições de memória são inicializadas com zero (0). As operações de acesso à memória não são atômicas e a notação *w(x)v* representa o instante onde a operação de escrita do valor *v* na posição de memória *x* se iniciou e *r(y)t* representa o instante onde a operação de leitura do valor *t* em *y* terminou. Para modelos híbridos, são consideradas as operações de sincronização *acquire* e *release*. As notações *A(l)v* e *R(l)v* representam os instantes onde o *acquire* no *lock l* e o *release* no *lock l* foram terminados.

O modelo do sistema considerado neste artigo [6] só admite seqüências legais, ou seja, o valor lido em uma posição de memória *x* deve ser o último valor escrito em *x* na seqüência.

2.1 Consistência Sequencial (SC)

A Consistência Sequencial (SC) foi proposta por Lamport [10] como um critério de corretude para multiprocessadores com memória compartilhada. Na Consistência Sequencial, o resultado de qualquer execução deve ser equivalente a alguma execução sequencial das operações de todos os processadores, onde a ordem do programa entre as operações é mantida [10]. Em outras palavras, SC impõe uma ordem global sobre os acessos à memória compartilhada, onde a ordem de cada programa e a seqüência legal devem ser mantidas.

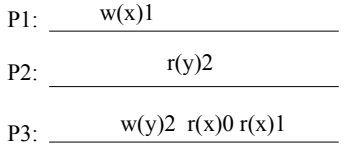

Figura 1. Um Histórico Válido em SC [9]

O histórico de execução apresentado na figura 1 é válido na Consistência seqüencial pois é possível obter pelo menos uma seqüência que contenha todas as operações de acesso à memória que respeite a ordem do programa de todos os processadores. A seguinte ordem de execução é válida em SC: $w_{p3}(y)2$ $r_{p2}(y)2$ $r_{p3}(x)0$ $w_{p1}(x)1$ $r_{p3}(x)1$. Note, no entanto, que esta não é a única seqüência que

pode ser obtida. A seqüência: $w_{p3}(y)2$ $r_{p3}(x)0$ $w_{p1}(x)1$ $r_{p3}(x)1$ $r_{p2}(y)2$ também é válida em SC.

2.2 Consistência PipelinedRAM

A Consistência PipelinedRAM foi definida por Lipton and Sandberg [11] e relaxa algumas restrições de ordem impostas pela SC, requerendo que somente as escritas feitas pelo mesmo processador sejam observadas na mesma ordem por todos os processadores. Em outras palavras, somente as operações de escrita dos outros processadores devem obedecer à ordem do programa [6].

Como a consistência PipelinedRAM é um modelo de consistência relaxado, cada processador só necessita enxergar as suas operações e todas as operações de escrita dos outros processadores (histórico H_{pi+w}) [6]. Para que uma execução seja válida na consistência PipelinedRAM, deve existir, para cada processador, um histórico H_{pi+w} onde a ordem do programa e a seqüência legal sejam respeitadas.

P1: w(x)1

P2: r(x)1 w(x)2 r(x)2

P3: r(x)2 r(x)1

Figura 2. Um histórico válido em PipelinedRAM

Considerando o histórico apresentado na figura 2, os seguintes H_{pi+w} podem ser derivados:

H_{p1+w}: $w_{p1}(x)1$ $w_{p2}(x)2$

H_{p2+w}: $w_{p1}(x)1$ $r_{p2}(x)1$ $w_{p2}(x)2$ $r_{p2}(x)2$

H_{p3+w}: $w_{p2}(x)2$ $r_{p3}(x)2$ $w_{p1}(x)1$ $r_{p3}(x)1$

Como foi possível derivar H_{pi+w} válidos para todo p_i, este histórico de execução é válido na consistência PipelinedRAM. Note, no entanto, que o mesmo histórico não é válido em SC, pois é impossível derivar uma ordem total que inclua todas as operações de P1, P2 e P3 e respeite a ordem do programa.

2.3 Consistência do Release (RC)

A Consistência do Release (RC) foi definida por [3] e é um dos modelos híbridos mais populares. Na Consistência do Release, os acessos conflitantes são chamados acessos especiais. Os acessos especiais são divididos em operações de sincronização e operações de não-sincronização. Existem dois tipos de operações de sincronização: *acquire* e *release*. As operações de leitura e escrita em memória são chamadas acessos ordinários.

Informalmente, em sistemas que obedecem à consistência do Release, deve ser garantido que: *antes que um acesso ordinário termine, todos os acessos acquire anteriores devem ter sido terminados; e antes que um release termine, todos os acessos ordinários anteriores devem ter sido terminados* [3]. Existe também uma terceira condição que requer que os acessos especiais respeitem a consistência do processador (RC_{pc}) ou que respeitem à consistência sequencial (RC_{sc}).

Um histórico de execução válido na Consistência do Release é apresentado na figura 3.

P1: $\text{acquire}_{p1}(l)1$ $w_{p1}(x)1$ $w_{p1}(y)2$ $\text{release}_{p1}(l)2$ $w_{p1}(z)1$

P2: $r_{p2}(z)1$ $r_{p2}(x)0$ $\text{acquire}_{p2}(u)3$ $r_{p2}(x)1$ $r_{p2}(y)2$ $\text{release}_{p2}(u)4$

Figure 3. Um histórico de execução válido em RC

Como RC também é um modelo relaxado, cada processador possui uma visão parcial dos acessos à memória, que engloba suas próprias operações e todas as operações de escrita e release dos outros processadores (histórico $H_{pi+w+release}$) [6].

Algumas ordens válidas em RC para o histórico apresentado na figura 3 são:

$H_{p1+w+release}$: $\text{acquire}_{p1}(l)1$ $w_{p1}(x)1$ $w_{p1}(y)2$ $\text{release}_{p1}(l)2$ $w_{p1}(z)1$ $\text{release}_{p2}(u)4$

$H_{p2+w+release}$: $w_{p1}(z)1$ $r_{p2}(z)1$ $r_{p2}(x)0$ $w_{p1}(x)1$ $w_{p1}(y)2$ $\text{release}_{p1}(l)2$ $\text{acquire}_{p2}(u)3$ $r_{p2}(x)1$ $r_{p2}(y)2$ $\text{release}_{p2}(u)4$

Em um sistema que obedece à consistência do Release, um acesso *release* deve ser ordenado após todos os acessos ordinários anteriores. No entanto, nenhuma ordem é imposta em relação ao *release* para os acessos posteriores ao mesmo. Por esta razão, é válido que a operação $w_{p1}(z)1$ apareça antes do $\text{release}_{p1}(l)2$ em $H_{p2+w+release}$.

2.4 Formalização de Modelos de Consistência

Basicamente, a definição formal de um Modelo de Consistência qualquer deve responder a duas perguntas: (1) quais operações devem ser propagadas; e (2) quais processadores devem receber as modificações. Para os modelos híbridos, uma terceira questão deve ser respondida: (3) qual evento de sincronização deve garantir que as modificações que o precedem estão certamente terminadas.

A Consistência Sequencial, a Consistência PipelinedRAM e a Consistência do Release respondem a estas questões de maneira diferente e, por esta razão, são modelos de consistência distintos.

A SC especifica que (1) todas as leituras e escritas devem ser propagadas imediatamente para (2) todos os processadores.

A Consistência PipelinedRAM, por sua vez, exige que (1) todas as escritas sejam propagadas imediatamente para (2) todos os processadores.

A Consistência do Release especifica que (1) todas as escritas anteriores devem ser propagadas para (2) todos os processadores (3) no momento que um acesso release é executado.

Estas definições mostram-se bastante próximas em termos semânticos, refletindo particularidades de cada modelo a respeito de tópicos presentes a todos. Isto foi um fator que se apresentou como um indício de que uma estrutura única seria adequada para o estudo de variados modelos, inclusive outros que poderiam eventualmente via a ser definidos.

3. Análise de Históricos de Execução em Múltiplos Modelos de Consistência

O objetivo da nossa análise é obter todos os ordenamentos possíveis de operações para um dado histórico de execução em um modelo de consistência de memória particular. A computação de ordenamentos para históricos de execução foi extensamente estudada por [7], onde se mostra que este problema é co-NP-difícil.

A título de ilustração, considere o histórico de execução bastante simples apresentado na figura 4. Neste caso, nenhum modelo de consistência de memória é considerado.

P1: w(x)1

P2: r(x)1 w(y)2

(1) wp1(x)1 -> rp2(x)1 -> wp2(y)2
(2) wp1(x)1 -> wp2(y)2 -> rp2(x)1
(3) rp2(x)1 -> wp1(x)1 -> wp2(y)2
(4) rp2(x)1 -> wp2(y)2 -> wp1(x)1
(5) wp2(y)2 -> rp2(x)1 -> wp1(x)1
(6) wp2(y)2 -> wp1(x)1 -> rp2(x)1

Figura 4. Um histórico de execução e os ordenamentos associados

Na figura 4, existem 6 caminhos de execução possíveis. Cada caminho de execução é uma seqüência ordenada de operações. Como pode ser facilmente visto, existem *p!* caminhos de execução lineares que podem ser obtidos, onde *p* é o número de operações consideradas.

A estratégia escolhida para gerar todos os caminhos de execução possíveis de um histórico de execução particular sem a necessidade de se explorar um espaço de busca exponencial foi modelar as restrições impostas pelo modelo de consistência como um conjunto parcialmente ordenado (poset). Um par (D, π) é um poset se e somente se D é um conjunto e π é uma relação irreflexiva transitiva em D x D. Os posets são frequentemente utilizados para modelar restrições semânticas de programas paralelos [2].

Sendo assim, a tarefa de analisar um determinado histórico em um dado modelo de consistência é decomposta em 3 passos [14]. Inicialmente, um modelo de consistência é definido e um histórico de execução é fornecido. As restrições de ordem são extraídas automaticamente e convertidas em um poset segundo as restrições de ordem do modelo de consistência em questão. Em segundo lugar, para os modelos híbridos, todas as possíveis ordens de sincronização são produzidas. E, finalmente, para cada possível ordem de sincronização, todas as extensões lineares que respeitam o poset são geradas. Para os modelos uniformes, neste último passo, as ordens de sincronização não são consideradas na geração das extensões lineares.

O primeiro passo é baseado na definição formal de cada modelo de consistência de memória [14]. Por exemplo, para o PipelinedRAM, que é um modelo de consistência relaxado, cada processador possui sua visão das operações de acesso à memória. Além disso, para o PipelinedRAM, a ordem a ser considerada é a ordem do programa (po) e a ordem das escritas.

A ordem do programa simplesmente exige que cada processador respeite a ordem na qual as operações aparecem no código do programa local. Por exemplo, na figura 3, a ordem do programa exige que $A_{p1}(l)1 \quad w_{p1}(x)1 \quad w_{p1}(y)2 \quad R_{p1}(l)2 \; w_{p1}(z)1$ seja respeitado na ordem do programa P1 e $r_{p2}(z)1 \quad r_{p2}(x)0 \quad A_{p2}(u)3 \quad r_{p2}(x)1 \quad r_{p2}(y)2 \; R_{p2}(u)4$ seja respeitada na visão do processador P2. Como o PipelinedRAM é um modelo uniforme, desconsideramos as operações de sincronização para efeito de consistência, obtendo assim as seguintes ordens do programa: $w_{p1}(x)1 \quad w_{p1}(y)2 \; w_{p1}(z)1$ e $r_{p2}(z)1 \quad r_{p2}(x)0 \; r_{p2}(x)1 \quad r_{p2}(y)2$, para P1 e P2, respectivamente.

A ordem das escritas exige que as escritas de cada processador sejam vistas na ordem do programa por todos os processadores que compõem o sistema. Ora, esta ordem nada mais é do que a ordem do programa definida para o histórico H_{pi+w}. Sendo assim, para o histórico da figura 3, todo processador deve respeitar a seguinte ordem: $w_{p1}(x)1 \quad w_{p1}(y)2 \; w_{p1}(z)1$.

Além disso, consideramos somente as seqüências legais, onde uma operação de leitura deve ler sempre o valor escrito pela operação de escrita mais recente para o mesmo endereço de memória. Na figura 3, a restrição da seqüência legal impõe que $w_{p1}(z)1 \rightarrow r_{p2}(z)1$, $r_{p2}(x)0 \rightarrow w_{p1}(x)1 \rightarrow r_{p2}(x)1$ e $w_{p1}(y)2 \rightarrow r_{p2}(y)2$.

A Consistência do Release determina que as seguintes ordens devem ser respeitadas: ordem do programa (po), ordem do acquire (acqo) e ordem do release (relo), descritas em detalhe em [14].

Para PipelinedRAM, as ordens de sincronização não são consideradas para efeito de consistência, logo, o passo 2 não é executado.

A Consistência do Release (RC_{sc}) exige que todos os processadores vejam exatamente a mesma ordem de sincronização, ou seja, as operações de sincronização devem obedecer à consistência seqüencial. Sendo assim, para o exemplo da figura 3, temos 6 possíveis ordens de sincronização: $A_{p1}(l)1 \rightarrow R_{p1}(l)2 \rightarrow A_{p2}(u)3 \rightarrow R_{p2}(u)4$; $A_{p1}(l)1 \rightarrow A_{p2}(u)3 \rightarrow R_{p1}(l)2 \rightarrow R_{p2}(u)4$; $A_{p1}(l)1 \rightarrow A_{p2}(u)3 \rightarrow R_{p2}(u)4 \rightarrow R_{p1}(l)2$; $A_{p2}(u)3 \rightarrow R_{p2}(u)4 \rightarrow A_{p1}(l)1 \rightarrow R_{p1}(l)2$; $A_{p2}(u)3 \rightarrow A_{p1}(l)1 \rightarrow R_{p2}(u)4 \rightarrow R_{p1}(l)2$; $A_{p2}(u)3 \rightarrow A_{p1}(l)1 \rightarrow R_{p1}(l)2 \rightarrow R_{p2}(u)4$. Para gerar automaticamente estas ordens, as operações de sincronização são convertidas em um poset a parte.

A última condição imposta por todos os modelos estudados até então é que todas as operações contidas na visão de cada processador apareçam exatamente uma vez. Assim, devem ser geradas todas as permutações possíveis que respeitam as ordens descritas anteriormente. Para os modelos híbridos, esta última parte é executada para cada ordem de sincronização gerada no passo 2.

No passo 3 e no passo 2 para os modelos híbridos, usamos o algoritmo desenvolvido por [8], que gera extensões lineares de um poset P em tempo constante amortizado , ou seja, com complexidade de tempo O(e(P)) onde e(P) é o número de extensões lineares válidas de P [8]. Este algoritmo é bastante eficiente porém impõe uma restrição muito séria na construção dos posets. Para construir um poset, todas as operações devem ser numeradas de 1 to *n*, onde *n* é o número total de operações. O algoritmo impõe que a ordem lexicográfica deve ser respeitada, ou seja, não é possível definir a restrição *3 -> 1*, por exemplo. Por esta razão, o processo de geração de extensões lineares foi dividido em 2 partes. Na primeira parte, todas as extensões lineares que respeitam as restrições na ordem lexicográfica são geradas. Na segunda parte, as extensões lineares são examinadas uma a uma e somente são mantidas se as restrições remanescentes forem respeitados.

Após executarmos este passo, vemos que o histórico da figura 3 não é válido na Consistência PipelinedRAM pois não existem ordenamentos válidos na visão de P2.

A estrutura desta estratégia para análise de históricos está resumida na figura 5.

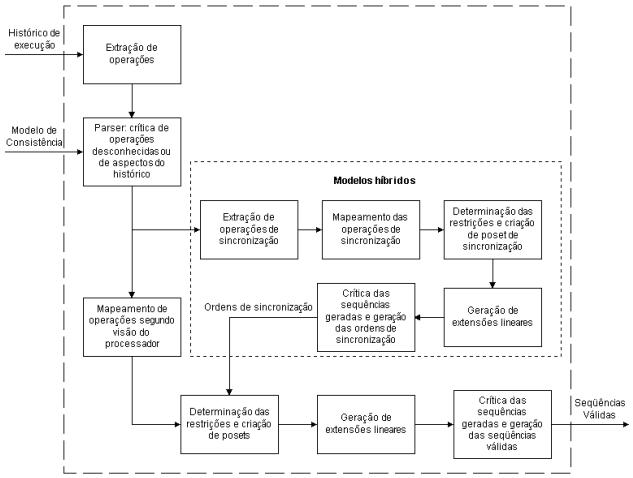

Figura 5. Estrutura usada para a análise dos históricos

Descrição do Framework

O framework proposto provê para o programador uma série de rotinas separadas em módulos por funcionalidade que permitem a criação e processamento de posets de restrições de ordem para históricos de execução. Não há qualquer limitação no número de processadores e operações envolvidas ou de generalidade nas operações presentes nos históricos, já que as fases de filtro de operações desconhecidas e de estabelecimento de restrições entre as operações são de responsabilidade completa do modelo de consistência implementado.

4.1. Módulos

O framework possui quatro módulos principais, listados a seguir:

a) *structures.c/structures.h*: O módulo *structures* disponibiliza as estruturas básicas para análise dos históricos de execução. É composto por vetores que representam as operações do histórico separadas por processador. Ainda o formam constantes úteis como número de processadores do histórico em análise, número de operações envolvidas e número de operações por classe (read, write, acquire, release). O módulo ainda provê as estruturas adequadas para permitir o mapeamento de operações, tanto ordinárias como de sincronização.

b) *translation.c/translation.h:* Este módulo apresenta uma interface que, em primeiro lugar, permite a criação das estruturas de mapeamento.

O mapeamento de operações é necessário para o estabelecimento das visões de cada processador e para a definição das ordens de sincronização. Para fazê-lo, usa-se a sub-rotina build_map_information,

onde é indicado se estas estruturas representarão a visão de um processador específico, uma visão geral ou apenas um mapa que será necessário para que se permute as operações de sincronização. Analogamente, dispõe-se da função clear_map_information, que destrói as estruturas criadas pela sub-rotina anterior. Os mapeamentos em si são inseridos por insert_mapping, e podem ser removidos por remove_mapping_from ou remove_mapping_to, de acordo com o tipo de dado disponível ao programador.

Para realizar o mapeamento de um código real de operação para um traduzido, ou vice-versa, usam-se as funções map e demap, respectivamente. A figura 6 ilustra a numeração original das operações (em negrito) do histórico ilustrado na figura 2 e o mapeamento das mesmas (em itálico) para a visão de P3.

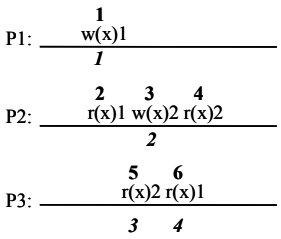

Figura 6. Mapeamento de Operações

c) *poset.c/poset.h:* Este módulo permite a definição de uma estrutura em memória que representa um poset, que pode se referir a um conjunto de operações de um processador específico, a todos eles simultaneamente ou se tratar de um poset que se relaciona com operações de sincronização. Este atributo, assim como o número de operações, devem ser informados à função create_poset.

O número de operações consideradas pode ser modificado ou incrementado posteriormente com o uso das funções set_number_operations e increment_number_operations, respectivamente. Para inserir as restrições no poset, faz-se o uso da sub-rotina insert_restriction, que recebe o poset no qual as restrições serão inseridas, as duas operações relacionadas (não mapeadas: o mapeamento é feito automaticamente com base no que foi definido no módulo descrito em (b)) e uma indicação descrevendo o modo de atualização do campo que indica o número de operações no poset. Se for indicado que a atualização deve ser automática, o campo citado acima sempre se apresentará com o maior valor traduzido de operação inserido até então. Caso contrário, este valor não é automaticamente atualizado, cabendo ao programador fazer tal atualização, a fim de ter maior controle sobre quantas operações estão envolvidas. Há ainda uma sub-rotina que faz uma junção de dois posets, chamada de merge_posets. Como última funcionalidade provida, existe uma função que limpa as estruturas na memória que descrevem o poset chamada de destroy_poset.

d) *linear_extensions.c/linear_extensions.h:* Este módulo é responsável pela geração das extensões lineares dado um poset. O módulo define estruturas para armazenar as extensões lineares que podem ser criadas a partir da função create_linear_extensions.

Existem, ainda, no módulo, procedimentos para reverter o mapeamento utilizado nas restrições do poset de entrada (demap_linear_extensions), liberação da memória alocada para o armazenamento das extensões lineares (destroy_linear_extensions) e para percorrimento das extensões geradas a fim de pós-processamento ou simples impressão (gen_next_sequence).

Um aspecto importante à geração das extensões lineares é o tratamento de extensões em relação às restrições que quebram a ordem lexicográfica (no qual o número da operação precedente é maior que o da operação subsequente). Restrições desse tipo não são processadas pelo algoritmo utilizado na geração das extensões lineares. O programador, então, possui o poder de indicar ao framework se deseja que o tratamento seja realizado pelas sub-rotinas internas de forma transparente ou se deve este papel deve ser deixado para o programador.

4.2 Inclusão do PipelinedRAM no Framework

O primeiro passo na implementação do modelo PipelinedRAM no framework é garantir que a entrada seja válida. Após esta checagem, para cada processador no sistema cria-se um mapeamento representando a visão deste em relação às operações globais. Este mapeamento pode ser criado visando desempenho, ou qualquer outro critério que o programador queira atingir. Com o mapeamento pronto, segue-se a definição de um poset no qual estão incluídas as restrições de ordem referentes a disposição das operações locais, das operações de escrita remotas e das relações lógicas de precedência das operações de escrita às de leitura dos valores correspondentes. Com base neste poset, são criadas as sequências lineares relativas às restrições deste. Estas sequências são então filtradas por uma sub-rotina externa ao framework, que faz as validações finais, garantindo que toda operação de leitura esteja refletindo o último valor escrito na mesma variável. As sequências válidas estão prontas para serem apresentadas ao usuário. Os pseudo-código do PipelinedRAM é apresentado a seguir:

```
PRAM_consistency() {
    se(parse_operations_está_ok()) {
        de(1 até número_de_processadores) {
            fazer_mapeamento(parcial);
            result = permutar_operações();
            imprimir_extensões_lineares(result);
            destruir_extensões_lineares(result);
        }
    }
}
permutar_operações() {
  p = criar_poset();
  inserir_restrições_ordem_operações_locais(p);
  inserir restrições_ordem_escritas_remotas(p);
  inserir_restrições_ordem_relações_lógicas(p);
  el = criar_extensões_lineares(p);
  el_final= filtrar_extensões_lineares(el);
  retorne (el_final);
}
```

4.3 Inclusão do RC no framework

A implementação do modelo de consistência do Release é mais complexa que a implementação do modelo PRAM, pois inclui as restrições que dizem respeito às operações de sincronização. Após as validações, para cada ordem de sincronização possível, tenta-se obter, para a visão de cada processador, extensões lineares que respeitem a ordem do programa, a ordem do acquire, a ordem do release e a seqüência legal.

O pseudo-código do RC é apresentado abaixo:

```
Release_consistency() {
  se(parse_operations_está_ok()) {
    fazer_mapeamento(sinc);
    el_sinc = permutar_operações_sinc();
    enquanto(seq_sinc =
      pegar_próxima_sequência(el_sinc)) {
        imprimir_sequência(seq_sinc);
        de(1 até num_processadores) {
          fazer_mapeamento(parcial);
          construir_restrições_adicionais();
         result= permutar_operações(seq_sinc);
         imprimir_estensões_lineares(result);
         destruir_estensões_lineares(result);
         }
       }
      destruir_estensões_lineares(el_sinc);
  }
}
construir_restrições_adicionais() {
 construir_restrições_ordem_relações_lógicas(p);
  construir_restrições_ordem_acquire(p);
  construir_restrições_ordem_release(p);
}
permutar_operações() {
  p = criar_poset();
  inserir_restrições_ordem_operações_locais(p);
  inserir_restrições_ordem_sincronização(p);
  inserir_restrições_adicionais(p);
  el = criar_extensões_lineares(p);
  el_final= filtrar_extensões_lineares(el);
  retorne (el_final);
}
```

A rotina construir_restrições_adicionais é apenas um recurso de otimização, já que restrições obtidas através de processamentos não-triviais são agrupadas nesta entidade e, através de controle de flags, é possível evitar que tais cálculos sejam efetuados cada vez que houver necessidade de obtê-los na construção do poset relativo a um processador específico.

Com base na facilidade tanto da implementação do modelo PRAM quanto do Release, é possível observar um comportamento adequado do Framework, se adaptando bem às restrições e definições de ambos modelos.

5. Resultados Experimentais

A figura 7 apresenta um mesmo histórico de execução sendo analisado em PipelinedRAM e em RC.

P1: w(x)1 r(y)0

P2: w(y) 1 r(x)0

Analysis on PipelinedRAM
Valid orderings for P1:
wp1(x)1 -> rp1(y)0 -> wp2(y)1
Valid orderings for P2:
wp2(y)1 -> rp2(x)0 -> wp1(x)1
This history is valid on PipelinedRAM

(a) Análise do histórico em PipelinedRAM

P1: A(l)1 w(x)1 r(y)0 R(l)2

P2: A(l)3 w(y) 1 r(x)0 R(l)4

Analysis on Release Consistency
Case Ap1(l)1->Ap2(l)3->Rp1(l)2->Rp2(l)4
Valid orderings for P1:
Ap1(l)1->wp1(x)1 -> rp1(y)0 -> Ap2(l)3->wp2(y)1->Rp1(l)2->Rp2(l)4
Valid orderings for P2:
Ap1(l)1->Ap2(l)3->wp2(y)1 -> rp2(x)0 -> wp1(x)1->Rp1(l)2->Rp2(l)4
Case Ap1(l)1->Ap2(l)3->Rp2(l)4->Rp1(l)2
Valid orderings for P1:
Ap1(l)1->wp1(x)1 -> rp1(y)0 -> Ap2(l)3->wp2(y)1->Rp2(l)4->Rp1(l)2
Valid orderings for P2:
Ap1(l)1->Ap2(l)3->wp2(y)1 -> rp2(x)0 -> wp1(x)1->Rp2(l)4->Rp1(l)2
Case Ap2(l)3->Ap1(l)1->Rp2(l)4->Rp1(l)2
Valid orderings for P1:
Ap2(l)3->Ap1(l)1->wp1(x)1 -> rp1(y)0 ->wp2(y)1->Rp2(l)4->Rp1(l)2
Valid orderings for P2:
Ap2(l)3->wp2(y)1 -> rp2(x)0 ->Ap1(l)1->wp1(x)1->Rp2(l)4->Rp1(l)2
Case Ap2(l)3->Ap1(l)1->Rp1(l)2->Rp2(l)4
Valid orderings for P1:
Ap2(l)3->Ap1(l)1->wp1(x)1 -> rp1(y)0 ->wp2(y)1->Rp1(l)2->Rp2(l)4
Valid orderings for P2:
Ap2(l)3->wp2(y)1 -> rp2(x)0 ->Ap1(l)1->wp1(x)1->Rp1(l)2->Rp2(l)4
This history is valid on Release Consistency

(b) Análise de (a) com 4 operações de sincronização em RC

P1: A(l)1 w(x)1 R(l)2 r(y)0 R(l)3

P2: A(l)3 w(y)1 R(l) 4 r(x)0 R(l)5

Analysis on Release Consistency
This history is not valid on Release Consistency

(c) Análise de (a) com 6 operações de sincronização em RC

Figura 7. Análise de histórico em PipelinedRAM e RC

Como pode ser visto, o histórico, apesar de produzir valores finais para x diferentes na visão de P1 e P2, é válido na consistência PipelinedRAM, ou seja, pode ser produzido neste modelo (figura 7.a). Como normalmente este é um resultado que o programador não deseja, analisamos o mesmo histórico na consistência do Release, colocando algumas operações de sincronização. O resultado, como pode ser visto na figura 7.b, ainda pode ser produzido na consistência do Release, pois RC não impõe que os locks sejam exclusivos [3], o que torna as seqüências do tipo A(l)1 -> A(l)3 -> R(l)2 -> R(l)4 válidas.

Sendo assim, incluímos mais sincronização entre as operações (figura 7.c) e, neste caso, a análise nos mostra que o resultado não pode ser produzido, ou seja, que os valores finais das cópias, com esta ordem de operações, não será diferente para P1 e P2, o que significa dizer que eliminamos a violação da ordem do programa.

6. Conclusões e Trabalhos Futuros

A DSM é uma abstração que simula uma memória compartilhada entre diversos processadores. Por razões de desempenho, a DSM não se comporta exatamente como a memória física e, sendo assim, pode produzir resultados que seriam impossíveis em um sistema com memória física única. Por esta razão, a programação de sistemas DSM é muitas vezes considerada complexa. A nosso ver, a análise dos resultados que podem ser produzidos em um sistema DSM que usa determinado modelo de consistência é crucial para que a programação de tais sistemas se torne mais simples.

Neste artigo, apresentamos um framework que pode ser utilizado para definição e análise de múltiplos modelos de consistência de memória. Os históricos de execução são analisados e são mostrados os caminhos válidos de execução que poderiam ter gerado estes históricos.

O framework proposto neste artigo foi implementado em C e os modelos de consistência PipelinedRAM e RC foram incorporados a ele. Nesta incorporação, notamos uma redução drástica no número de linhas de código utilizadas e um aumento grande na estruturação do código, o que facilitou muito sua depuração.

Conforme o mostrado na seção 5, a análise de históricos em diversos modelos contribui para uma melhor compreensão da programação em sistemas DSM, já que apresenta o que poderia ou não ser produzido em determinado modelo.

Como trabalhos futuros, vamos incorporar pelo menos os modelos de consistência seqüencial (SC) [10] e do escopo [5] (ScC) ao framework. Em paralelo, será definida uma interface gráfica para entrada do histórico de execução e apresentação dos resultados.

7. References

[1] S. V. Adve, "Designing Multiple Memory Consistency Models for Shared-Memory Multiprocessors", PhD dissertation, University of Wisconsin-Madison, 1993.

[2] E. Best, “Semantics of Sequential and Parallel Programs”, Prentice Hall Int. Series in Computer Science, 1996, 352p.

[3] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Proc. Int. Symp. On Computer Architecture*, May, 1990, p15-24.

[4] A. Heddaya and H. Sinha, ″An Implementation of Mermera: a Shared Memory System that Mixes Coherence with Non-Coherence", Tech Report BU-CS-93-006, Boston University, 1993.

[5] Iftode, L., Singh, J. P., and Li, K. "Scope Consistency: A Bridge between Release Consistency and Entry Consistency" . *Proc 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96),* pages 277-287, June 1996.

[6] A. Melo, “Defining Uniform and Hybrid Memory Consistency Models on a Unified Framework”, Proceedings of the 32nd Hawaiian International Conference on System Sciences, Vol. VIII, Software Technology, Maui, 1999.

[7] R. Netzer, B. Miller, “On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions”, Technical Report TR-908, University of Wisconsin-Madison, (1990).

[8] Yaakov L. Varol, Doron Rotem: An Algorithm to Generate all Topological Sorting Arrangements. The Computer Journal 24(1): 83-84, 1981.

[9] D. Mosberger, “Memory Consistency Models”, Operating Systems Reviews, January, 1993.

[10]Lamport L., *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, 1979, 690-691.

[11] R. Lipton, J. Sandberg, “PRAM: A Scalable Shared Memory, Technical Report 180-88, Princeton University, 1988.

[12]W. Hu., W. Shi., Z. Tang.: JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In Proc. of HPCN'99, LNCS 1593, pp. 463-472, Springer-Verlag, April, 1999.

[13]E. Speight, J. Bennet, “Brazos: a Third Generation DSM System”, *Proc. Of the USENIX/WindowsNT Workshop,* p.95-106, August, 1997.

[14]A. Melo, N. Silva, “Visualizing Execution Histories on Release Consistency and Scope Consistency Memory Models”, Proc. Of the 2nd Int. Workshop on Software Distributed Shared Memory, Santa Fe, USA, May, 2000.