Analyzing Execution Orderings on Hybrid Memory Consistency Models

Alba Cristina Melo, Nilo Sergio Barros Silva
Computer Science Department, University of Brasília (UnB)
Campus Universitário - Asa Norte - CEP -70910-900 Brasília - Brazil
{albamm, nilo}@cic.unb.br

Abstract—
The behavior of Distributed Shared Memory Systems is dictated by the memory consistency model. To provide a better understanding on the semantics of the memory consistency models, researchers have proposed formalisms to define them. Even with formal definitions, it is still difficult to say what kind of execution histories can be produced on a particular memory model. In this paper, we propose a visualization tool that generates the operations orderings that could have led to user-defined execution histories on different memory models. We also describe a prototype of our tool that analyses execution histories for two memory consistency models: Release Consistency and Scope Consistency. Some preliminary results are also presented.

Keywords— **Memory Consistency Models, Distributed Shared Memory**

I. INTRODUCTION

In order to make shared memory programming possible in distributed or parallel architectures where no physical shared memory exists, we must create a shared memory abstraction that parallel processes can access. This abstraction is called Distributed Shared Memory (DSM).

The behavior of the DSM is dictated by the Memory Consistency Model. The first DSM systems tried to make the Distributed Shared Memory behave exactly as if it was a physical uniprocessor memory. Providing such a strong Memory Consistency Model created a huge coherence overhead, slowing down the parallel application and frequently bringing the system into a thrashing state. To alleviate this problem, researchers have proposed to relax some consistency conditions, thus creating new memory behaviors that are different from the traditional uniprocessor one.

Many Memory Consistency Models have been proposed in the literature. Originally, the consistency requirements have not been defined in a formal way. In some cases, this led to different interpretations of the same Memory Consistency Model. Memory Models proposed as Processor Consistency have this kind of problem. For instance, Processor Consistency definitions proposed by [Goo89] and [Gh+90] describe distinct memory models. To avoid this kind of problem, researchers proposed formal frameworks where Memory Consistency Models can be defined [ADV93] [HED93] [MEL99a]. Unfortunately, the most widely used Memory Consistency Models are so complex that, even with formal definitions, it is still difficult to say if some undesirable operations interleavings can be produced.

In this article, we describe a tool that assists DSM designers and programmers in the task of analyzing a memory consistency model. This new tool analyses a particular execution history on a formally defined memory consistency model and generates the possible operation interleavings that could have led to the execution history. In our prototype, Release Consistency (RC) and Scope Consistency (ScC) Memory Models were implemented.

As expected, the results produced by our tool showed that both RC and ScC are very effective to augment performance by allowing a great overlapping of basic memory operations. On the other hand, some results were quite surprising. The most interesting one shows that the same execution history can be valid in RC but not in ScC. Formally speaking, this result shows that, contrary to our intuitive notion, ScC is not more relaxed than RC since there are results produced in RC which cannot be reproduced in ScC.

The rest of this paper is organized as follows. Section II shortly introduces the memory consistency models. Section III describes the formalism used to define memory consistency models. Section IV presents the approach used to decide the validity of an execution history on a chosen memory model. The implementation of a prototype of our tool and some preliminary results are presented in section V. Related work is discussed in section VI. Finally, conclusions are presented in section VII.

II. MEMORY CONSISTENCY MODELS

Intuitively, the programmer assumes that the instructions that compose his or her program are executed one after the other (in a serial way) and memory operations are executed atomically. This informal model is used to reason about the results a program can produce. A Memory Consistency Model formalizes this concept by defining the order in which the memory operations must be perceived by the programmer. Since the Memory Consistency Model defines the apparent order and not the real order of memory operations' execution, many optimizations can be made in a uniprocessor machine while still respecting the intuitive model. Unfortunately, when we try to apply the same optimizations to a distributed or parallel environment, the intuitive model is violated and programming becomes more complex since the programmer must be conscious of the distributed nature of the physical memory.

Hybrid Memory Consistency Models guarantee that processors only have a consistent view of the shared memory at synchronization time. These models are quite successful in the sense that they permit a great overlapping of basic memory operations while still providing a reasonable programming model. Release Consistency and Scope Consistency are the most popular hybrid models.

Release Consistency (RC) was defined by [GHA90]. In Release Consistency, competing accesses are called special accesses. Special accesses are divided into synchronization operations and non-synchronization operations. There are two subtypes of synchronization operations: *acquire* accesses and *release* accesses. Read and write memory operations are called ordinary accesses.

Informally, in Release Consistent systems, it must be guaranteed that: *before an ordinary access performs, all previous acquire accesses must be performed; and before a release performs with respect to any other processor, all previous ordinary accesses must be performed* [GHA90]. There is also a third condition that requires special accesses to be processor consistent (RC_{pc}) or sequentially consistent (RC_{sc}).

The goal of Scope Consistency (ScC) is to take advantage of the association between synchronization variables and ordinary shared variables they protect. It was proposed by [IFT96]. In Scope Consistency, executions are divided into consistency scopes that are defined in a per lock basis. Scope Consistency orders only synchronization and data accesses that are related to the same synchronization variable. The association between shared data and the synchronization variable that guards them is implicit and depends on program order. Informally, a system is scope consistent if *(1) before a new section of a consistency scope is allowed to open at process P, any write previously performed with respect to that consistency scope must be performed with respect to P; and (2) A memory access is allowed to perform with respect to a process P only after all consistency scope sessions previously entered by P (in program order) have been successfully opened* [IFT96].

III. Formalization of Memory Models

To describe memory models formally, we use a history-based system model that was already described in [MEL99a]. In table 1, we only review some definitions.

At the beginning of the execution, it is assumed that all memory positions are initialized to 0.

In our definitions, we use the notion of *linear sequences*. If Q is a history, a *linear sequence* of Q contains all operations in Q exactly once. A linear sequence is *legal* if all read operations $r(x)v$ return the value written by the most recent write operation on the same address in the sequence.

In execution histories, there are some operation orderings that are allowed and some orderings that are forbidden. The decision of which orderings are valid is made by the *memory consistency model*. One execution history is valid on a memory consistency model if it respects the order relation defined by the model.

All formal definitions of memory consistency models presented in this paper will use the template shown in figure 1.

System	A finite set of processors
Processor p_i	Executes operations on the Shared Global Memory M
Shared Global Memory M	Contains all memory addresses.
Local memory m_i	Caches all memory addresses of M.
$O_{pi}(x)v$	Operation executed by processor p_i on address x with value v.
Types of operations on M	read®, write (w) and synchronization (sync)
Subtypes of sync operations on M	acquire(A);release® or user-defined operations
$O_{pi}(x)v$ is issued	Processor p_i executes the instruction o(x)v.
$r_{pi}(x)v$ is performed	A write operation on x cannot modify the value returned to p_i
$w_{pi}(x)v$	$C = \sum_{a=0}^{n-1} w_{pa}(x)v$, where n is the number of processors
$w_{pi}(x)v$ perfomed with respect to p_i	Value v is written to the address x on the local memory m_i of p_i
$w_{pi}(x)v$ performed	$w_{pi}(x)v$ is performed with respect to all processors.
Local execution history H_{pi}	A sequence of memory operations issued by p_i
Execution history H	$\cup H_{pi}$
Memory Consistency Model	Defines an order relation on a set of shared memory accesses

Table 1. System Model

<X> Consistency: A history H is *x-consistent* if there is a legal linear sequence of <history> that respects the order $\overset{x}{\rightarrow}$ which is defined as follows:

i) requirement_1 and
ii) requirement_2 and
....
iii) requirement_n.

Figure 1. Template used to define memory models

In the rest of this section, the formal definitions of Release Consistency and Scope Consistency are shortly presented. More details about these definitions can be found in [MEL99a].

Definition 1. Release Consistency [Mel99a]: A history H is *release consistent* if there is a legal linear sequence of $H_{pi+w+release}$ that respects the order $\overset{RC}{\rightarrow}$ which is defined for each processor p_i as follows:

i) $\forall$ o_1, o_2, o_3: if $o_1 \overset{so}{\rightarrow} o_2 \overset{cb}{\rightarrow} o_3$ on H and subtype(o_1) = *release* and subtype(o_2) = *acquire* and type(o_3) $\in$ {r,w} then $o_1 \overset{RC}{\rightarrow} o_3$ and

ii) $\forall$ o_1, o_2: if $o_1 \overset{cb}{\rightarrow} o_2$ on $H_{pi+w+release}$ and type(o_1) $\in$ {r,w} and subtype(o_2) = *release* then $o_1 \overset{RC}{\rightarrow} o_2$ and

iii) $\forall$ o_1, o_2: if processor(o_1)=processor(o_2)=p_i and $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{RC}{\rightarrow} o_2$.

As RC is a relaxed memory model, processors must only see their own operations and all write and release operations issued by the other processors (history $H_{pi+w+release}$) [MEL99a].

In short, the definition of $\overset{RC}{\rightarrow}$ imposes that all processors must agree on the order of all synchronization operations, i.e, synchronization order $\overset{so}{\rightarrow}$ must be preserved. In addition, all basic memory operations that follow the acquire must be ordered after the acquire (i) and all basic memory operations must be performed before the release is issued (ii). Condition (iii) simply states that program order ($\overset{po}{\rightarrow}$)of p_i must be preserved in p_i's view.

An execution history is valid on a Memory Consistency Model if there is at least one valid interleaving of memory operations for each processor that compose the system.

A history that is valid on Release Consistency is presented in figure 2. In this representation, there are 2 processors, P1 and P2. Time corresponds to the horizontal axis and flows from left to right.

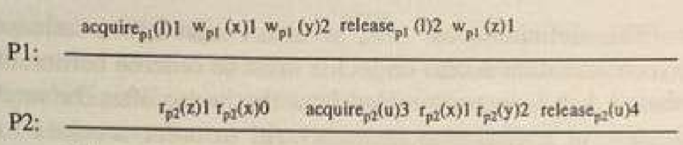

P1:
acquire$_{p1}$(l)1 w$_{p1}$(x)1 w$_{p1}$(y)2 release$_{p1}$(l)2 w$_{p1}$(z)1
P2:
r$_{p2}$(z)1 r$_{p2}$(x)0 acquire$_{p2}$(u)3 r$_{p2}$(x)1 r$_{p2}$(y)2 release$_{p2}$(u)4

Figure 2. A Release Consistent Execution History

Some possible valid orderings in RC for the history presented in figure 2 are:

$\mathbf{H_{p1+w+release}}$: $acquire_{p1}(l)1 \overset{RC}{\rightarrow} w_{p1}(x)1 \overset{RC}{\rightarrow} w_{p1}(y)2 \overset{RC}{\rightarrow} release_{p1}(l)2 \overset{RC}{\rightarrow} w_{p1}(z)1 \overset{RC}{\rightarrow} release_{p2}(u)4$

$\mathbf{H_{p2+w+release}}$: $w_{p1}(z)1 \overset{RC}{\rightarrow} r_{p2}(z)1 \overset{RC}{\rightarrow} r_{p2}(x)0 \overset{RC}{\rightarrow} w_{p1}(x)1 \overset{RC}{\rightarrow} w_{p1}(y)2 \overset{RC}{\rightarrow} release_{p1}(l)2 \overset{RC}{\rightarrow} acquire_{p2}(u)3 \overset{RC}{\rightarrow} r_{p2}(x)1 \overset{RC}{\rightarrow} r_{p2}(y)2 \overset{RC}{\rightarrow} release_{p2}(u)4$

In a Release Consistent system, a release access cannot be issued until all previous ordinary accesses are performed. However, once the release is issued, accesses that succeed this operation do not need to wait until it is performed. That's why it is valid for operation $w_{p1}(z)1$ to appear before $release_{p1}(l)2$ in $H_{p2+w+release}$.

For the formal definition of Scope Consistency, we must define two new orders: *synchronization-orderl (sol)* and *scope-comes-before (scop)* [MEL99a]. *Sol* only orders synchronization operations that access the same memory address whereas *scop* relates only synchronization operations to shared data operations that belong to the same consistency scope, i.e., operations that occur after the acquire operation on variable x and before the release operation on the same variable in the program code.

Definition 2. Scope Consistency [MEL99a]: A history H is *scope consistent* if there is a legal linear sequence of $H_{pi+w+release}$ that respects the order $\overset{SCOP}{\rightarrow}$ which is defined for each processor p_i as follows:

i) $\forall$ o_1, o_2, o_3: if $o_1 \overset{sol}{\rightarrow} o_2 \overset{scopcb}{\rightarrow} o_3$ in H and subtype(o_1) = *release* and subtype(o_2) = *acquire* and type(o_3) $\in$ {r,w} then $o_1 \overset{SCOP}{\rightarrow} o_3$ and

ii) $\forall$ o_1, o_2: if $o_1 \overset{scopcb}{\rightarrow} o_2$ on $H_{pi+w+release}$ and type(o_1) $\in$ {r,w} and subtype(o_2) = *release* then $o_1 \overset{SCOP}{\rightarrow} o_2$ and

iii) $\forall$ o_1, o_2, o_3: if $o_1 \overset{SCOP}{\rightarrow} o_2$ and $o_2 \overset{SCOP}{\rightarrow} o_3$ then $o_1 \overset{SCOP}{\rightarrow} o_3$ and

iv) $\forall$ o_1, o_2: if processor(o_1)=processor(o_2)=p_i and $o_1 \overset{po}{\rightarrow} o_2$ then $o_1 \overset{SCOP}{\rightarrow} o_2$.

The definition of $\xrightarrow{SCOP}$ states that (i) a release synchronization access on lock *x* must be ordered before all shared data accesses guarded by *x* that occur after the next acquire to *x*. It is also necessary (ii) to order accesses to shared data guarded by a lock *x* before the release on lock *x*. This relation and transitivity of SCOP (iii) guarantee that all previous accesses to shared data guarded by *x* will be performed before the next acquire on *x* is performed. Also, local program order must be preserved (iv).

P1: $acquire_{p1}(l)1 \quad w_{p1}(x)1 \quad w_{p1}(y)1 \quad release_{p1}(l)2$

P2: $acquire_{p2}(v)1 \quad r_{p2}(x)0 \quad release_{p2}(v)2 \quad acquire_{p2}(l)3 \quad r_{p2}(x)1 \quad release_{p2}(l)4$

Figure 3. A Scope Consistent Execution History

Figure 3 presents a Scope Consistent Execution History. Some possible orderings that respect Scope Consistency for the history in figure 3 are:

$H_{p1+w+release}$: $acquire_{p1}(l)1 \xrightarrow{SCOP} w_{p1}(x)1 \xrightarrow{SCOP} w_{p1}(y)1 \xrightarrow{SCOP} release_{p1}(l)2 \xrightarrow{SCOP} release_{p2}(v)2 \xrightarrow{SCOP} release_{p2}(l)4$

$H_{p2+w+release}$: $acquire_{p2}(v)1 \xrightarrow{SCOP} r_{p2}(x)0 \xrightarrow{SCOP} release_{p2}(v)1 \xrightarrow{SCOP} w_{p1}(x)1 \xrightarrow{SCOP} w_{p1}(y)1 \xrightarrow{SCOP} release_{p1}(l)2 \xrightarrow{SCOP} acquire_{p2}(l)3 \xrightarrow{SCOP} r_{p2}(x)1 \xrightarrow{SCOP} release_{p2}(l)4$

Note that, in Scope Consistency, it is no longer necessary for processors to agree on the order of all synchronization accesses. That's why it is possible for P1 and P2 to observe the operations *release $_{p1}(l)2$* and *release $_{p2}(v)1$* on different orders.

IV. VALIDATING EXECUTION HISTORIES ON HYBRID MODELS

The goal of our tool is to automatically generate the possible execution orderings for a given execution history on a particular Memory Consistency Model. Computing orderings for execution histories is a problem that has been extensively studied in [NET90] and this problem is shown to be co-NP-hard.

To illustrate this problem, consider the very simple execution history presented in figure 4. Note that, in this case, no memory consistency model is considered.

P1: w(x)1

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

wp1(x)1 -> rp2(x)1 -> wp2(y)2
wp1(x)1 -> wp2(y)2 -> rp2(x)1
rp2(x)1 -> wp1(x)1 -> wp2(y)2
rp2(x)1 -> wp2(y)2 -> wp1(x)1
wp2(y)2 -> rp2(x)1 -> wp1(x)1
wp2(y)2 -> wp1(x)1 -> rp2(x)1

Figure 4. An execution history and its execution orderings

In figure 4, there are six possible execution paths. Each path is an ordered sequence of operations. As it can be easily seen, there are *p!* linear execution paths that can be derived, where *p* is the number of memory operations to be considered. In general, the memory consistency model will reduce the number of possible execution paths, as it imposes ordering restrictions that must always be guaranteed.

The strategy chosen to generate the possible execution paths of a particular execution history without having to exploit an exponential search space is to model the constraints imposed by the memory model as partially ordered sets (posets). A pair (D, <) is called a poset iff D is a set and < is an irreflexive transitive relation on D x D. Posets are often used to model some semantic restrictions of parallel programs [BES96].

Figure 5 shows the main modules of our visualization tool. The task of validating a given execution history is decomposed in three main steps. First, a memory model is chosen and some of its constraints are extracted. The constraints are then converted into partially ordered sets (posets). Second, all possible synchronization orderings are produced. Third, for each possible synchronization ordering, possible linear sequences that respect the posets are generated.

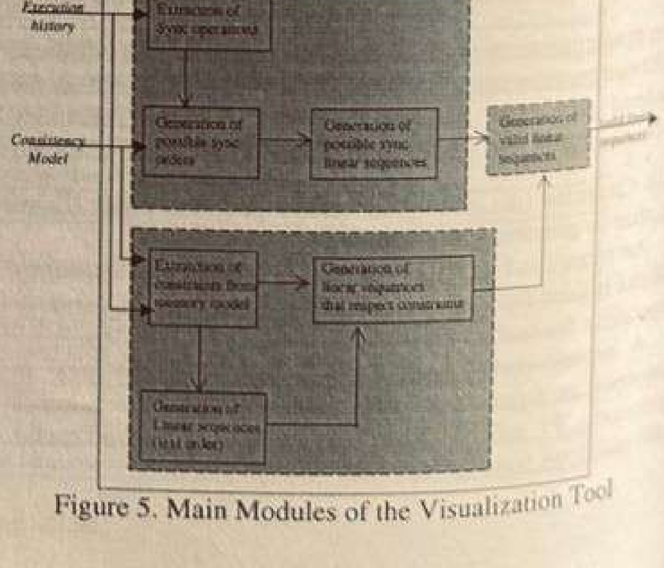

Execution history
Extraction of sync operations
Consistency Model
Generation of possible sync orders
Generation of possible sync linear sequences
Generation of valid linear sequences
Extraction of constraints from memory model
Generation of linear sequences that respect constraints
Generation of Linear sequences

Figure 5. Main Modules of the Visualization Tool

A. Extracting Constraints from the Memory Model.

This first step is based on the formal definition of each Memory Consistency Model (section III). For both Release Consistency and Scope Consistency, there are four orderings to be first considered: program-order, acquire-order, release-order and legal sequence.

Program-order simply states that each processor must respect its local program order. There is a program order for each processor in the history (po_1, po_2,..., po_n). Acquire-order (acqo) is defined for RC and states that all read and write operations that follow the acquire must be ordered after the acquire. For ScC, a slightly different order is considered (acqo1) where all read and write operations that follow the acquire and precede the release on the same synchronization variable must be ordered after the acquire. Release-order (relo) is defined for RC and states that all read and write operations that precede the release must be ordered before the release. For ScC, we use a new order called release-order-1. Release-order-1 (relo1) states that all read and write operations that succeed the acquire on lock l must be ordered before the release on lock l. As stated before, the legal sequence (ls) imposes that a read operation must read the value written by the most recent write operation on the same address.

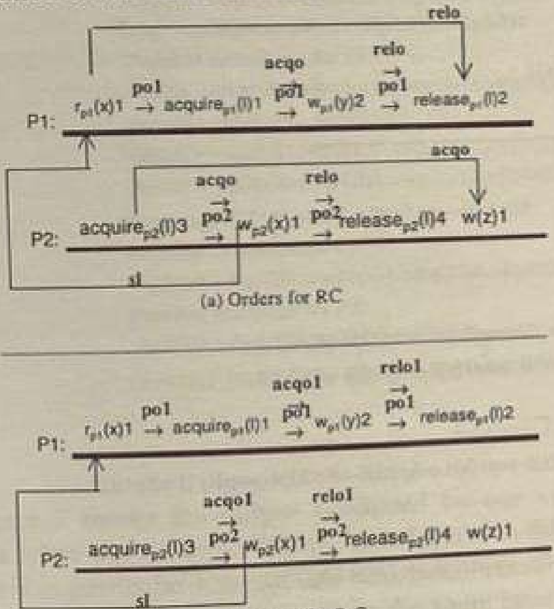


Figure 6 – Extracting orders on RC and ScC

Figure 6 shows the orders extracted from the same execution history for RC (6.a) and ScC (6.b).

B. Generating all Possible Synchronization Orders.

Analyzing the formal definitions of RC and ScC, it can be seen that the only restriction imposed is that all processors must agree on the order of all synchronization accesses (RC) or processors must agree on the order of all synchronization accesses to the same synchronization variable (ScC). Thus, in two different executions, synchronization orders may not be the same, due to race conditions to access synchronization variables, i. e., there can exist many valid synchronization orderings. The only restriction imposed is that program order must be respected for each synchronization operation (RC) or for each synchronization operation on the same synchronization variable (ScC).

In ScC, we consider that the same consistency scope must not be opened simultaneously by more than one process, as suggests [IFT96].

For instance, in figure 2, the restrictions are *acquire* $_{p1}(l)1 \rightarrow$ *release* $_{p1}(l)2$ and *acquire* $_{p2}(u)3 \rightarrow$ *release* $_{p2}(u)4$. Respecting these restrictions, there are 6 possible synchronization orders. To generate the possible synchronization orders, these restrictions are also modeled as posets.

C. Generating all Possible Linear Sequences.

The last condition imposed by both memory models is that all operations involved in each view must appear exactly once. Thus, we must generate possible permutations of operations that respect the synchronization order being considered and also respects the 5 orders described in section III.A.

In this part (and also in step 2), we used the algorithm developed by [PRU94] that generates linear sequences of a poset P in constant amortized time, i.e, in time O(e(P)) where e(P) is the number of linear sequences of P [PRU94]. For our problem, there can always exist a case where the total number of valid linear sequences is exponential with respect to the number of operations. As there is no practical sense to obtain so many valid operations' orderings, we limited of valid linear sequences to be produced to 20 for each possible synchronization order.

Although the algorithm [PRU94] is well-suited to solve our problem, it imposes a severe restriction. To construct a poset, all operations must be numbered from 1 to *n*, where *n* is the total number of operations. The algorithm imposes that the lexicographical order must be respected, i.e., the restriction *3 -> 1* is not possible. For this reason, we divided the generation process into 2 parts. In the first part, all linear sequences that respect the posets in lexicographical order are generated. In the second part, the linear sequences generated are examined one by one and are only accepted if the remaining posets are respected.

V. PROTOTYPE IMPLEMENTATION AND EXPERIMENTAL RESULTS

A prototype of our tool was implemented using C++. As input, the visualization tool receives an execution history and a memory consistency model. The operations that compose the history can be read (r), write(w), acquire(A) or release(R). The number of operations and the number of processors are not fixed.

Figure 6 shows the results obtained when providing the example execution history in RC. Each line lists two possible execution paths for processor Pi. Having received the execution history presented in the left upper corner in figure 6 and the memory consistency model (RC), our tool generates the orders to be respected (section III.A). The order restrictions generated are represented in the right upper corner in figure 6. Each possible synchronization ordering is analyzed and the linear sequences that respect all the orders are produced. Our tool only prints the synchronization orderings where there is at least one possible linear sequence for each processor view.

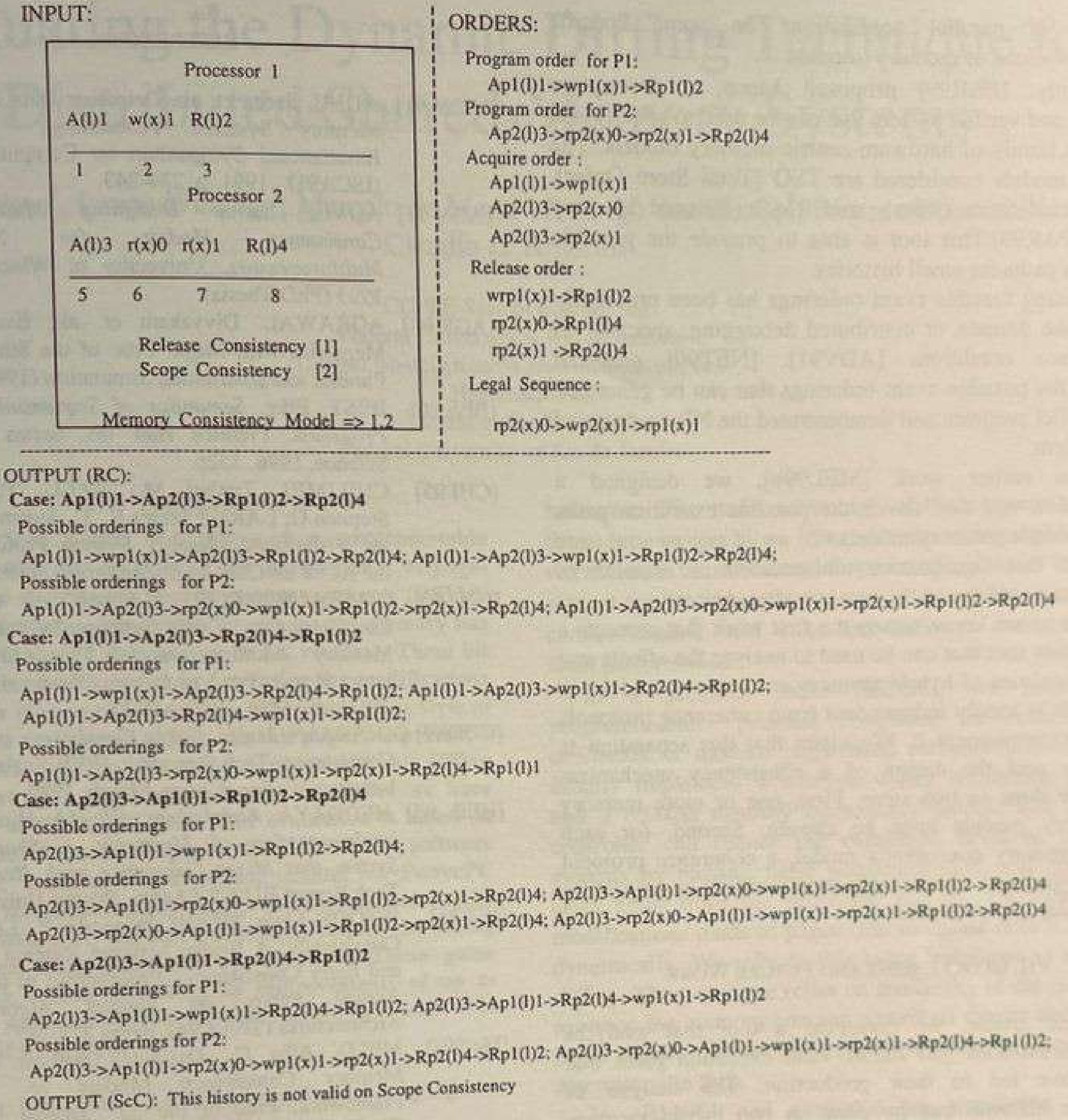
For the history in figure 6, there are not valid synchronization orderings where the first operation is $A_{p1}(l)1$ since the constraints imposed by RC on p_i state that $acquire_{p2}(l)3 \overset{acqo}{\rightarrow} w_{p2}(x)1 \overset{ls}{\rightarrow} r_{p1}(x)1 \overset{po}{\rightarrow} acquire_{p1}(l)1$. Also, the operation $r_{p1}(x)1$, that belongs only to p_i's view, imposes an order on $w_{p2}(x)1$ ($w_{p2}(x)1 \overset{ls}{\rightarrow} r_{p1}(x)1$).

```
INPUT:                                      ORDERS:
                                             Program order for P1:
        Processor 1                            rp1(x)1->Ap1(l)1->wp1(y)2->Rp1(l)2
                                             Program order for P2:
  r(x)1  A(l)1  w(y)2  R(l)2                   Ap2(l)3->wp2(x)1->Rp2(l)4
                                             Acquire order:
  1       2      3      4                      Ap1(l)1->wp1(y)2
        Processor 2                            Ap2(l)3->wp2(x)1
                                             Release order:
  A(l)3  w(x)1  R(l)4                          rp1(x)1->Rp1(l)2
                                               wp1(y)2->Rp1(l)2
  5       6      7                             wp2(x)1 ->Rp2(l)4

        Release Consistency [1]              Legal Sequence:
        Scope Consistency   [2]
                                               wp2(x)1->rp1(x)1
  Memory Consistency Model => 1

OUTPUT:
Case: Ap2(l)3->Ap1(l)1->Rp1(l)2->Rp2(l)4
 Possible orderings for P1:
 Ap2(l)3->wp2(x)1->rp1(x)1->Ap1(l)1->wp1(y)2->Rp1(l)2->Rp2(l)4
 Possible orderings for P2:
Ap2(l)3->Ap1(l)1->wp1(y)2->Rp1(l)2->wp2(x)1->Rp2(l)4; Ap2(l)3->Ap1(l)1->wp1(y)2->wp2(x)1->Rp1(l)2->Rp2(l)4
Ap2(l)3->Ap1(l)1->wp2(x)1->wp1(y)2->Rp1(l)2->Rp2(l)4; Ap2(l)3->wp2(x)1->Ap1(l)1->wp1(y)2->Rp1(l)2->Rp2(l)4
Case: Ap2(l)3->Ap1(l)1->Rp2(l)4->Rp1(l)2
 Possible orderings for P1:
Ap2(l)3->wp2(x)1->rp1(x)1->Ap1(l)1->wp1(y)2->Rp2(l)4->Rp1(l)2; Ap2(l)3->wp2(x)1->rp1(x)1->Ap1(l)1->Rp2(l)4->wp1(y)2->Rp1(l)2
 Possible orderings for P2:
Ap2(l)3->Ap1(l)1->wp1(y)2->wp2(x)1->Rp2(l)4->Rp1(l)2; Ap2(l)3->Ap1(l)1->wp2(x)1->wp1(y)2->Rp2(l)4->Rp1(l)2
Ap2(l)3->Ap1(l)1->wp2(x)1->Rp2(l)4->wp1(y)2->Rp1(l)2; Ap2(l)3->wp2(x)1->Ap1(l)1->wp1(y)2->Rp2(l)4->Rp1(l)2
Ap2(l)3->wp2(x)1->Ap1(l)1->Rp2(l)4->wp1(y)2->Rp1(l)2
Case: Ap2(l)3->Rp2(l)4->Ap1(l)1->Rp1(l)1
Possible orderings for P1:
 Ap2(l)3->wp2(x)1->rp1(x)1->Rp2(l)4->Ap1(l)1->wp1(y)2->Rp1(l)2; Ap2(l)3->wp2(x)1->Rp2(l)4->rp1(x)1->Ap1(l)1->wp1(y)2->Rp1(l)2
Possible orderings for P2:
 Ap2(l)3->wp2(x)1->Rp2(l)4->Ap1(l)1->wp1(y)2->Rp1(l)2
```

Figure 6. Possible Execution Orderings on RC



```
INPUT:                                        ORDERS:
                                               Program order  for P1:
              Processor 1                        Ap1(l)1->wp1(x)1->Rp1(l)2
                                               Program order  for P2:
 A(l)1   w(x)1   R(l)2                           Ap2(l)3->rp2(x)0->rp2(x)1->Rp2(l)4
 ______________________________                Acquire order :
 1       2       3                               Ap1(l)1->wp1(x)1
              Processor 2                        Ap2(l)3->rp2(x)0
                                                 Ap2(l)3->rp2(x)1
 A(l)3  r(x)0  r(x)1    R(l)4                  Release order :
 ______________________________                  wrp1(x)1->Rp1(l)2
 5      6       7       8                        rp2(x)0->Rp1(l)4
                                                 rp2(x)1 ->Rp2(l)4
        Release Consistency  [1]
        Scope Consistency    [2]               Legal Sequence :

   Memory Consistency Model => 1,2               rp2(x)0->wp2(x)1->rp1(x)1
---------------------------------------------------------------------------
OUTPUT (RC):
 Case: Ap1(l)1->Ap2(l)3->Rp1(l)2->Rp2(l)4
  Possible orderings   for P1:
   Ap1(l)1->wp1(x)1->Ap2(l)3->Rp1(l)2->Rp2(l)4; Ap1(l)1->Ap2(l)3->wp1(x)1->Rp1(l)2->Rp2(l)4;
  Possible orderings   for P2:
   Ap1(l)1->Ap2(l)3->rp2(x)0->wp1(x)1->Rp1(l)2->rp2(x)1->Rp2(l)4; Ap1(l)1->Ap2(l)3->rp2(x)0->wp1(x)1->rp2(x)1->Rp1(l)2->Rp2(l)4
 Case: Ap1(l)1->Ap2(l)3->Rp2(l)4->Rp1(l)2
  Possible orderings   for P1:
   Ap1(l)1->wp1(x)1->Ap2(l)3->Rp2(l)4->Rp1(l)2; Ap1(l)1->Ap2(l)3->wp1(x)1->Rp2(l)4->Rp1(l)2;
   Ap1(l)1->Ap2(l)3->Rp2(l)4->wp1(x)1->Rp1(l)2;
  Possible orderings   for P2:
   Ap1(l)1->Ap2(l)3->rp2(x)0->wp1(x)1->rp2(x)1->Rp2(l)4->Rp1(l)1
 Case: Ap2(l)3->Ap1(l)1->Rp1(l)2->Rp2(l)4
  Possible orderings   for P1:
   Ap2(l)3->Ap1(l)1->wp1(x)1->Rp1(l)2->Rp2(l)4;
  Possible orderings   for P2:
   Ap2(l)3->Ap1(l)1->rp2(x)0->wp1(x)1->Rp1(l)2->rp2(x)1->Rp2(l)4; Ap2(l)3->Ap1(l)1->rp2(x)0->wp1(x)1->rp2(x)1->Rp1(l)2->Rp2(l)4
   Ap2(l)3->rp2(x)0->Ap1(l)1->wp1(x)1->Rp1(l)2->rp2(x)1->Rp2(l)4; Ap2(l)3->rp2(x)0->Ap1(l)1->wp1(x)1->rp2(x)1->Rp1(l)2->Rp2(l)4
 Case: Ap2(l)3->Ap1(l)1->Rp2(l)4->Rp1(l)2
  Possible orderings for P1:
   Ap2(l)3->Ap1(l)1->wp1(x)1->Rp2(l)4->Rp1(l)2; Ap2(l)3->Ap1(l)1->Rp2(l)4->wp1(x)1->Rp1(l)2
  Possible orderings for P2:
   Ap2(l)3->Ap1(l)1->rp2(x)0->wp1(x)1->rp2(x)1->Rp2(l)4->Rp1(l)2; Ap2(l)3->rp2(x)0->Ap1(l)1->wp1(x)1->rp2(x)1->Rp2(l)4->Rp1(l)2;
  OUTPUT (ScC): This history is not valid on Scope Consistency
```

Figure 7. Possible Orderings on RC and ScC

Figure 7 shows the output produced by our visualization tool when the example history is analyzed in RC and ScC. For this particular history, RC and ScC orders imposed in acquire and release operations are the same because there is only one synchronization variable (lock *l*) and there are no ordinary operations outside the critical sections. However, as the formal definition of ScC used to extract the ordering restrictions does not allow the same consistency scope to be open simultaneously on more than one processor, there are fewer possible synchronization orderings in ScC. In fact, only the synchronization orders $A_{p1}(l)1 \rightarrow R_{p1}(l)2 \rightarrow Ap_2(l)3 \rightarrow R_{p2}(l)4$ and $A_{p2}(l)3 \rightarrow R_{p2}(l)4 \rightarrow A_{p1}(l)1 \rightarrow R_{p1}(l)2$ are valid on ScC. Using these orders, it is not possible to produce any valid interleaving of memory operations. Although there are six possible synchronization orderings for RC, only four synchronization orderings can produce valid results, as shown in figure 7.

VI. Related Work

[GHI95] presents a visualization tool called StormWatch that analyses the same application on different memory coherence protocols. StormWatch provides a trace view that is based on execution histories. Maya is a simulation plataform described in [AGR94] which analyses the behavior of coherence protocols that implement the following memory models: Sequential Consistency, Causal Consistency, PRAM Consistency and Entry Consistency. Basically, these two systems provide visualization of the behavior of parallel applications on some specific implementations of memory models.

Recently, [PAR99] proposed Murφ, a description language and verifier system that can be used to specify and analyse a family of hardware-centric memory models. The memory models considered are TSO (Total Store Order), PSO (Partial Store Order) and RMO (Relaxed Memory Order) [PAR99].This tool is able to provide the possible execution paths for small histories.

Analysing feasible event orderings has been previously done in the domain of distributed debugging, specially to detect race conditions [ADV91]. [NET90] analyzed formally the possible event orderings that can be generated by a parallel program and demonstrated the NP-hardness of this problem.

In an earlier work [MEL99b], we designed a visualization tool that shows the possible execution paths on very simple memory models. To avoid exponential state explosion, this visualization tool restricts the number of memory operations and the number of processors.

As far as we know, this is the first work that presents a visualization tool that can be used to analyze the effects and the potentialities of hybrid memory consistency models, in a way that is totally independent from coherence protocols that could implement it. We claim that this separation is necessary and the design of a consistency mechanism should be done in two steps. First, one or more memory consistency models must be chosen. Second, for each chosen memory consistency model, a coherence protocol must be specified. Our visualization tool will be helpful on the first step of this process.

VII. CONCLUSIONS AND FUTURE WORK

In this article, we presented a tool that analyses execution histories and shows valid execution paths that could have led to their production. The analysis of execution histories can be done on two hybrid memory consistency models. We claim that obtaining the possible interleavings of shared memory accesses is helpful for DSM-system designers and programmers in the task of choosing the most appropriate memory model. This can reduce considerably the number of unexpected results that are produced when using or designing a DSM system.

As future work, we intend to incorporate other memory models to our visualization tool and add a graphical interface to it. Also, we intend to define a language for specifying memory consistency models based on formal definitions. Using this language, the user can define its own memory models and interactively visualize the interleavings of memory operations that can produced on the newly defined memory consistency model. This facility will be helpful in memory model definition as well as in the comparison of distinct memory consistency models.

REFERENCES

[ADV 91] ADVE, Sarita et. al., Detecting Data Races on Weak Memory Systems, Proceedings of the 18th International Symposium on Computer Architecutre (ISCA91) , 1991, p. 234-243.

[ADV 93] ADVE, Sarita, *Designing Multiple Memory Consistency Models for Shared-Memory Multiprocessors,* University of Wisconsin-Madison, 1993 (PhD Thesis).

[AGR 94] AGRAWAL, Divyakant et. al.; Evaluating Weak Memories with Maya, Proc of the 8th Workshop on Parallel and Distributed Simulation (1994) 151-155.

[BES 96] BEST, Eike, *Semantics of Sequential and Parallel Programs,* Prentice Hall Int. Series in Computer Science, 1996, 352p.

[CHI 95] CHILIMBI, Trishul M.; BALL, Thomas; EICK Stephen G; LARUS James R. T., StormWatch: a Tool for Visualizing Memory System Protocols, Proc of the ACM Int Conf on Supercomputing'95, 1995.

[GHA90] GHARACHORLOO, Kourosh et al., Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, Proceedings of the International Symposium on Computer Architecture (ISCA90), May, 1990, p15-24.

[GOO89] GOODMAN, James, *Cache Consistency and Sequential Consistency,* Tech Report61, IEEE Scalable Coherent Interface WorkingGroup, March, 1989.

[HED 93] HEDDAYA Abdelsalam; SINHA Himanshu, *An Implementation of Mermera: a Shared Memory System that Mixes Coherence with Non-Coherence,* Tech Report BU-CS-93-006, Boston University, 1993.

[IFT 96] IFTODE, Liviu, SINGH, Jaswinder P.; LI, Kai, Scope Consistency: A Bridge between Release Consistency and Entry Consistency, Proceedings of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96), pages 277-287, June 1996.

[Mel99a] MELO, Alba C., Defining Uniform and Hybrid Memory Consistency Models on a Unified Framework, Proc of the 32nd Hawaii International Conference on System Sciences, January, 1999.

[MEL99b] MELO, Alba C.; CHAGAS Simone C., Visual-MCM: Visualising Execution Histories on Multiple Memory Consistency Models, in *Lecture Notes in Computer Science 1557,* February, 1999, p.500-509.

[MOS93] MOSBERGER, David, Memory Consistency Models, *Operating Systems Review*, pages 18-26, 1993.

[NET 90] NETZER, Robert, MILLER, Barton P., *On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions,* Technical Report TR-908, University of Wisconsin-Madison, (1990).

[PAR 99] PARK, Seungjoon; DILL David L., *An Executable Specification and Verifier for Relaxed Memory Order,* IEEE Transactions on Computers, v.48, n.2, february, 1999, p.227-235.

[PRU94] PREUSSE, Gara; RUSKEY Frank, *Generating Linear Extensions Fast,* SIAM J. Computing, Vol. 23, No. 2, pp. 373-386, April 1994.