JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 14, 785-808 (1998)

A New Relaxed Memory Consistency Model for Shared-Memory Multiprocessors with Parallel-Multithreaded Processing Elements

CHAO-CHIN WU AND CHENG CHEN
Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan 300, R. O. C.
E-mail: ccwu@csie.nctu.edu.tw
E-mail: cchen@eicpca5.csie.nctu.edu.tw

The release consistency model is the generally accepted hardware-centric relaxed memory consistency model because of its performance and implementation complexity. By extending the release consistency model, in this paper, we propose a hardware-centric memory consistency model particularly for shared-memory multiprocessor systems with parallel-multithreaded processing elements. The new model uses a new categorization for memory references and utilizes the feature of parallel multithreaded processors (PMPs). We further partition acquire and release references into three sub-categories: one for lock-unlock pairs, one for barrier synchronization, and the last for others. According to the semantic of each synchronization primitive, each sub-category has its own relaxed restrictions. On the other hand, the feature of a PMP is that it is capable of executing more than one thread at the same time, where all parallel threads share only one cache hierarchy. Under the new model, we can use *dual write-caches* to reduce write traffic and synchronization time.

We have used five benchmarks in the SPLASH suite to evaluate the performance gain for the new model. According to the simulation results, the new model is superior to the release consistency model at best by about 11%.

Keywords: memory consistency model, multithread, multiprocessor, write cache, synchronization, PSC model, barrier, performance evaluation.

1. INTRODUCTION

Improvements in semiconductor technology have allowed more and more transistors to reside on a single chip, leading to rapid progress in advanced microprocessor architecture design, including superscalar [1] and multithreaded architectures [2], etc. Parallel multithreaded processors (PMPs) are capable of executing more than one thread at the same time in a single chip [3-5]. The advantage of PMPs is that they enable greater hardware utilization because all the parallel running threads share the functional units in this processor. Therefore, we expect that this type of processor will become one of the most popular single-processor designs [5]. For huge computations, however, PMP architectures are not powerful enough because a single chip cannot execute tens or even hundreds of parallel threads. One possible way to upgrade their performance is to employ the PMP-

Received May 24, 1997; accepted February 24, 1998.
Communicated by Youn-Long Lin.

based multiprocessor system (PMP-MP). The PMP-MP is a cache-coherent, non-uniformed memory architecture similar to the conventional shared-memory multiprocessor system except that the processing element (PE) is PMP architecture. The key feature of the PMP-MP is that several threads share only one cache in each PE.

Since multiple threads are allowed to simultaneously read and write the same memory locations, programmers need a memory consistency model for the semantics of memory operations to allow them to correctly use the shared memory. A memory consistency model determines the order in which memory references can be executed by the system and greatly affects the implementation and performance of a system [6-14]. The most commonly assumed memory model is sequential consistency (SC) [6]. While SC provides a simple model for programmers, it imposes many constrains on architecture and compiler optimizations that exploit the reordering and overlap of memory references. To achieve better system performance, several relaxed memory models have been proposed, for example, processor consistency (PC) [7], weak consistency (WC) [8, 9], and release consistency (RC) [10, 11]. These are referred to as *hardware-centric* models because they are defined in terms of relatively low-level hardware constraints on the ordering of memory references [15].

In this paper, we propose a new hardware-centric memory model, the *PMP-MP-specific consistency model* (*PSC*), which is suitable especially for PMP-MPs. We further divide the *acquire and release references* into three groups, according to the program-level objects for which they are implemented, e.g., critical sections and barriers. The release reference for a critical section can be performed if references within the critical section have all been performed. It does not matter whether memory references prior to the critical section have all been performed or not. On the other hand, because more than one thread is executing on each processing element in a PMP-MP system, we can relax the ordering constrain on releases references for barrier synchronizations. A release reference of this type can be performed independently of memory references prior to it if the issuing thread is not the last one arriving at the barrier synchronization from the same processor.

The new parallelism exploited by the new categorization can be utilized by incorporating *dual write-caches* in PMP-MPs. Write caches can merge all the write references belonging to the same block into a single write miss request [16, 17]. Consequently, the amount of write traffic can be reduced significantly. The dual write-cache not only reduces the write traffic, but also shortens the synchronization time. We have used five benchmarks in the SPLASH suite to evaluate the performance gain for the new model. According to the simulation results, the new model is superior to the release consistency model at best by about 11%.

The rest of this paper is organized as follows. Section 2 introduces the framework proposed by Gharachorloo et al. [13, 15] and the release consistency model. Section 3 specifies the ordering requirements of memory references for the PSC model. Section 4 gives an example to illustrate how the PSC model can be implemented. In section 5, we present our use of a simulation environment to evaluate the performance benefit propped by the PSC model. Some concluding remarks are given at the end of this paper.

2. RELATED WORK

Release consistency is the generally accepted hardware-centric relaxed memory consistency model because of its performance and implementation complexity [14]. It categorizes memory references, as depicted in Fig. 1, in order to relax restrictions on memory access ordering. Basically, release consistency takes two forms that depend on whether special references are sequential consistent (RC_{SC}) or processor consistent (RC_{PC}) [10]. In the following, we use the framework [13, 15] proposed by Gharachorloo et al. to specify the system requirements of RC_{SC}. The framework provides a formal methodology for specifying sufficient system requirements that precisely capture the semantics of various memory consistency models. In addition, we can compare various models through the formal framework. For more details about the framework, we refer the reader to [13, 15].

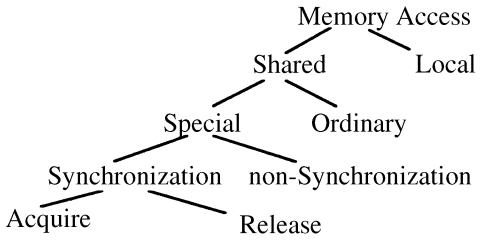

Fig. 1. Categorization of memory references.

The specification assumes the following system abstraction. The system consists of n processors, $P_1, \ldots, P_n$, and P_i contains t_i threads. A read operation **R** shared by P_i is composed of an initial sub-operation $R_{init}(i)$ and a read sub-operation $R(i)$. A write operation **W** shared by P_i is composed of $(n + 1)$ atomic sub-operations: the initial write sub-operation $W_{init}(i)$ and n sub-operations $W(1), \ldots, W(n)$. All sub-operations of **W** access the same location and write the same value.

To specify restrictions on the execution order under the RC_{SC} model, we illustrate several notations in Fig. 2, and the reach condition will be explained in some detail here. The main purpose of the reach condition is to disallow anomalous executions that arise if, for example, systems have the ability to overlap or reorder a read reference with respect to a subsequent write reference. Informally, a read reference reaches a write reference that follows it in program order (denoted by $R \xrightarrow{rch} W$) if the read determines whether the write will execute, the address accessed by the write, or the value written by it. In addition, $R \xrightarrow{rch} W$ if the read controls the execution, address, or value written of another memory reference that is before W in program order and is related to W by certain program orders. We refer the reader to [13, 15] for a formal definition of the reach condition.

Fig. 3 shows the sufficient system requirements for RC_{SC}. Two relations, $\xrightarrow{spo}$ and $\xrightarrow{spo'}$, are defined so as to capture the specific $\xrightarrow{po}$ arcs that are used to define the $\xrightarrow{xo}$ relation. The condition in part (a) on $\xrightarrow{xo}$ includes four general constraints as described in the following Conditions 1 to 4.

- R: a read access, including an acquire memory operation;
- W: a write access, including a release memory operation;
- RW: either a read or a write memory operation;
- X, Y: either a read or a write memory operation;
- X(i), Y(i): either a read sub-operation R(i) or a write sub-opertion W(i), excluding the initial read or write sub-operations, which are always denoted explicitly;
- RW_c: a competing read or write memory operation;
- R_{c_acq}: an acquire memory operation;
- W_{c_rel}: a release memory operation;
- $\xrightarrow{po}$: *program order*;
- $\xrightarrow{xo}$: *execution order*;
- $\xrightarrow{co}$: *conflict order* ($X \xrightarrow{co} Y$ iff $X(i) \xrightarrow{xo} Y(i)$ holds for any i and X and Y are two conflicting memory operations);
- $\xrightarrow{rch}$: *reach condition*;
- $X(i) \xrightarrow{xo} Y(i)$ for all i, j: pairs of values for i and j for which both X(i) and Y(j) are defined.

Fig. 2. Listing of notation (1).

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$:

$X \xrightarrow{spo'} Y$ if X and Y are the first and last operations in one of
$RW_c \xrightarrow{po} RW_c$;

$X \xrightarrow{spo} Y$ if X and Y are the first ad last operations in one of
$R_{c\_acq} \xrightarrow{po} RW$
$RW \xrightarrow{po} W_{c\_rel}$.

Conditions on $\xrightarrow{xo}$:

(a) the following conditions must be obeyed:
Condition 1: initiation condition for reads and writes.
Condition 2: termination condition for writes; applies to *all write sub-operations.*
Condition 3: return value for read sub-operations.
Comdition 4: atomicity of read-modify-write operations.

(b) uniprocessor dependence: if X and Y conflict and are the first and last operations in $RW \xrightarrow{po} W$ from P_k, then $X(k) \xrightarrow{xo} Y(k)$.

(c) coherence: if $W1 \xrightarrow{co} W2$, then $W1(i) \xrightarrow{xo} W2(i)$ for all i.

(d) multiprocesor dependence: given that X and Y are the first and last operations in one of the following conditions, and Y is from P_k:
$X \xrightarrow{spo'} Y$
$X \xrightarrow{spo} Y$
$W \xrightarrow{co} R \xrightarrow{spo'} RW$.
If Y=R then $X(i) \xrightarrow{xo} Y(j)$ for all i, j, and if Y=W, then $X(i) \xrightarrow{xo} Y(j)$ for all i, j except j=k.

(e) reach: if $R \xrightarrow{rch} W$, where R and W are from P_k, then $R(i) \xrightarrow{xo} W(j)$ for all i, j except j=k.

Fig. 3. Sufficient conditions for RC_{SC}.

Condition 1: Initiation Condition for Reads and Writes

Given memory operations performed by P_i to the same location, the following conditions hold. If $R \xrightarrow{po} W$, then $R_{init}(i) \xrightarrow{xo} W_{init}(i)$. If $W \xrightarrow{po} R$, then $W_{init}(i) \xrightarrow{xo} R_{init}(i)$. If $W \xrightarrow{po} W'$, then $W_{init}(i) \xrightarrow{xo} W'_{init}(i)$.

Condition 2: Termination Condition for Writes

Suppose a write sub-operation $W_{init}(i)$ (corresponding to operation W) performed by P_i appears in the execution. The termination condition requires that the other *n* corresponding sub-operations, W(1), …, W(n), also appear in the execution.

Condition 3: Return Value for Read Sub-Operations

A read sub-operation R(i) performed by P_i returns a value that satisfies the following conditions. If there is a write operation W performed by Pi to the same location as R(i) such that $W_{init}(i) \xrightarrow{xo} R_{init}(i)$, then R(i) returns the value of the last such $W_{init}(i)$ in $\xrightarrow{xo}$. Otherwise, R(i) returns the value of W′(i) (from any processor) such that W'(i) is the last write sub-operation to the same location that is ordered before R(i) by $\xrightarrow{xo}$. If there are no writes that satisfy either of the above two categories, then R(i) returns the initial value of the location.

Condition 4: Atomicity of Read-Modify-Write Operations

If R and W are the constituent read and write operations for an atomic read-modify-write ($R \xrightarrow{po} W$ by definition) on P_j, then for every *conflicting* write operation W' from a different processor P_k, *either* $W'(i) \xrightarrow{xo} R(i)$ and $W'(i) \xrightarrow{xo} W(i)$ for all i *or* $R(i) \xrightarrow{xo} W'(i)$ and $W(i) \xrightarrow{xo} W'(i)$ for all i.

The condition in part (b) is uniprocessor dependence, which captures the order of operations from the same thread and to the same location. The condition in part (c) is coherence, which ensures that write sub-operations to the same location occur in the same order with respect to every memory copy. The condition in part (d) is multiprocessor dependence chain, which captures the relations among operations that are ordered according to program order and conflict order. The multiprocessor dependence chain specifies the restrictions for a given memory model. Finally, the execution order has to obey the *reach condition* in part (e). In the following section, we will use the above framework to describe a new model for PMP-MP systems that are extensions from the release consistency model.

3. THE PMP-MP-SPECIFIC CONSISTENCY MODEL (PSC)

In this section, we will extend the release consistency model to obtain a more relaxed one that is more suitable for PMP-MP systems. First, we will explain how to utilize the different semantics of critical section and barrier synchronization to further categorize the shared memory references for the release consistency model in Section 3.1. Next, we will propose our new model in Section 3.2 by presenting its constraint on memory access ordering. In Section 3.3, we will introduce how to properly label programs for the proposed model to ensure correctness.

3.1 Critical section and barrier synchronization

In the SPLASH benchmark suite, synchronization references are divided into locks, unlocks and barriers [18]. Barriers require that all processors reach a consistent state before the codes following them are executed. When a thread reaches a barrier, it typically perform the following three steps: (1) it marks itself as present at the barrier; (2) it delays until all other processes reach that same barrier; (3) after all involved threads have arrived at the barrier, it proceeds past the barrier. Shared data accessed within a barrier must be made consistent only at the points where the barrier computation proceeds from one phase to the next. Within a phase, there are no consistency guarantees for data updated during that phase (unless other lock synchronizations are used), so threads must assume that only data from previous phases has reached a consistent state.

The purpose of locks (acquire references) is to gain permission to access a set of shared locations; the purpose of unlocks (release references) is to grant permission for accesses. Usually, locks and unlocks together ensure exclusive accesses within critical sections. A critical section is bounded by a pair of synchronization references to a synchronization variable. A lock occurs at the beginning of a critical section and is used to gain access to a set of shared memory locations. An unlock occurs at the end of a critical section and is used to signal that access is available. Lock and unlock references together protect the shared data within a critical section from concurrent accessing. When a processor executes a lock reference to enter a critical section, it will later execute an unlock reference.

To sum up, before we release a lock, data that can be accessed within the corresponding critical section must be made consistent with all processors. On the other hand, before barriers can be performed, we have to make all shared data consistent with all threads. Based on the above observation, we propose a new consistency model for PMP-MP systems, called the *PMP-MP-specific consistency* (*PSC*) *model*. Because barriers are also implemented using acquire and release references [19], we have to further classify acquire and release references into different groups. Acquires and releases implemented for critical sections and not for barrier operations are called *CS-acquires* and *CS-releases*, respectively. Each paired *CS-acquire* and *CS-release* is executed by the same thread. The acquires and releases which it needs to mark itself as present at barriers (step 1 in implementing a barrier as mentioned above) are called *bar-acquires* and *bar-releases*, respectively. Other acquires and releases are called *ord-acquires* and *ord-releases*, respectively.

We can relax the restrictions on memory access ordering by utilizing the suggested categorization and the feature of PMP architecture: (1) shared data prior to a critical section in program order can be inconsistent with other memory copies even when the lock to the critical section has been released; (2) shared data prior to a barrier operation in program order can be not consistent with other memory copies until the issuing thread has passed the barrier; (3) multiple threads on the same processor share only one memory copy. An important observation is that no thread can pass the barrier synchronization if any participating thread has not reached the barrier. Consequently, shared data accessed by a processor prior to a barrier in program order can be inconsistent with other memory copies if any participating thread on the processor has not arrived at the barrier. In the following section, we will introduce the PSC model more precisely and formally.

3.2 The PSC model

Like the release consistency model, PSC has PSC_{SC} and PSC_{PC} two forms. It depends on whether the special references are sequential or processor consistent. In order to explain our PSC_{SC} model clearly, we will first introduce some notations used in this model as shown in Fig. 4. (Other notations have the same meanings as defined in the previous section.)

- $R_{c\_CS\_acq}$: a CS-acquire access
- $W_{c\_CS\_rel}$: a CS-release access
- $R_{c\_bar\_acq}$: a bar-acquire access
- $W_{c\_bar\_rel}$: a bar-release access
- $R_{c\_ord\_acq}$: an ord-acquire access
- $W_{c\_ord\_rel}$: an ord-release access

Fig. 4. List of notation (2).

Fig. 5 shows the sufficient conditions for the PSC_{SC} model. We define the following three relations to capture the specific $\xrightarrow{po}$ arcs.

- The $\xrightarrow{spo''}$ relation holds for either
  - (i) a memory references within a critical section and the immediately following CS-release reference ordered by $\xrightarrow{po}$, or
  - (ii) a memory reference and the immediately following bar-release reference ordered by $\xrightarrow{po}$, where the memory reference is enclosed by the bar-release reference and the paired bar-acquire reference.
- The $\xrightarrow{spo'}$ relation holds for any competing pair ordered by $\xrightarrow{po}$.
- The $\xrightarrow{spo'}$ relation holds for any pair of operations ordered by $\xrightarrow{po}$
  - (i) if the first operation is an ord-acquire reference, or
  - (ii) if the second operation is an ord-release reference.

The conditions in parts (a), (b), (c) and (e) are identical in the two specifications for PSC_{SC} and RC_{SC}. The difference between PSC_{SC} and RC_{SC} lies in the condition in part (d) and the additional condition in part (f), particularly for PSC_{SC}. The multiprocessor dependence chain in Fig. 5 specifies the following restrictions for PSC_{SC} systems.

(i) For each pair of memory references ordered by $\xrightarrow{spo''}$, before a CS-release reference or a bar-release reference can take effect in every memory copy, a memory reference protected by the release reference must have occurred with respect to every memory copy.

(ii) Each pair of competing references ordered in program order ($X \xrightarrow{spo'} Y$) must take effect in the same order as the program order in every memory copy.

(iii) If an ord-acquire reference is prior to a memory reference in program order

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$, $\xrightarrow{spo''}$:

X $\xrightarrow{spo''}$ Y if $R_{c\_CS\_acq}$ and $W_{c\_CS\_rel}$ together enclose a critical section, $R_{c\_bar\_acq}$ and $W_{c\_ar\_rel}$ together mark a thread as present at a barrier, X and Y are the second and last operations in one of

$R_{c\_CS\_acq} \xrightarrow{po} RW \xrightarrow{po} W_{c\_CS\_rel}$

$R_{c\_bar\_acq} \xrightarrow{po} RW \xrightarrow{po} W_{c\_bar\_rel}$.

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

$RW_c \xrightarrow{po} RW_c$.

X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

$R_{c\_ord\_acq} \xrightarrow{po} RW$

$RW \xrightarrow{po} W_{c\_ord\_rel}$.

Conditions on $\xrightarrow{xo}$:

(a) the following conditions must be obeyed:

Condition 1: initiation condition for reads and writes.

Condition 2: termination condition for writes; applies to *all write sub-operations.*

Condition 3: return value for read sub-operations.

Condition 4: atomicity of read-modify-write operations.

(b) uniprocessor dependence: if X and Y conflict and are the first and last operations in RW $\xrightarrow{po}$ W from P_k, then X(k) $\xrightarrow{xo}$ Y(k).

(c) coherence: if W1 $\xrightarrow{co}$ W2, then W1(i) $\xrightarrow{xo}$ W2(i) for all i.

(d) multiprocesor dependence: given that X and Y are the first and last operations in one of the following conditions, and Y is from P_k:

X $\xrightarrow{spo''}$ Y

X $\xrightarrow{spo'}$ Y

X $\xrightarrow{spo}$ Y

W $\xrightarrow{co}$ R $\xrightarrow{spo'}$ RW,

if Y=R then X(i) $\xrightarrow{xo}$ Y(j) for all i, j, and if Y=W, then X(i) $\xrightarrow{xo}$ Y(j) for all i, j except j=k.

(e) reach: if R $\xrightarrow{rch}$ W, where R and W are from P_k, then R(i) $\xrightarrow{xo}$ W(j) for all i, j except j=k.

(f) intra-processor snchronization: if X $\xrightarrow{ipo}$ $W_{c\_bar\_rel}$ and $ARR_{(B,\,k)} = COMP_{(B,\,k)} - \{r\}$, then X(i) $\xrightarrow{xo}$ $W_{c\_bar\_rel}$(j) for all i, j except j=k, where X is from T_r on P_k, and $W_{c\_bar\_rel}$ is for the barrier B.

Fig. 5. Sufficient conditions for PSC_{SC}.

($R_{c\_ord\_acq} \xrightarrow{po} RW$), then the ord-acquire reference must occur before the subsequent memory reference with respect to every memory copy.

(iv) If a memory reference is prior to a ord-release reference in program order ($RW \xrightarrow{po} W_{c\_ord\_rel}$), then the memory reference has to take effect before the ord-release reference in every memory copy.

(v) If a write reference conflicts with a competing read reference and the read reference is prior to a competing memory reference in program order, then the write reference must take effect before the last memory reference in every memory copy.

The final condition in part (f) relaxes the restrictions on memory access ordering between a memory reference and barrier synchronization reference following it on the same processing element. Shared data accessed by the same processor, P_k, have to be consistent with other memory copies only when all the participating threads on Pk have arrived at a barrier. In the following, we will define the relation of intra-processor program order (denoted by $\xrightarrow{ipo}$) to specify which memory references have to be consistent with other memory copies before passing a barrier.

A memory reference is prior to either an $R_{c\_bar\_acq}$ reference or a $W_{c\_bar\_rel}$ reference ordered by $\xrightarrow{ipo}$ if

(i) the memory reference and its subsequent $R_{c\_bar\_acq}$ or $W_{c\_bar\_rel}$ reference are issued from the same processor but not necessarily from the same thread, and

(ii) the memory reference is prior to a barrier synchronization, B, ordered by, $\xrightarrow{po}$ and

(iii) the $R_{c\_bar\_acq}$ reference or the $W_{c\_bar\_rel}$ reference will be executed so as to pass the barrier B.

Formally, we give the following definition of intra-processor program order in Definition 5 and we illustrate use of the notation using an example in Fig. 6.

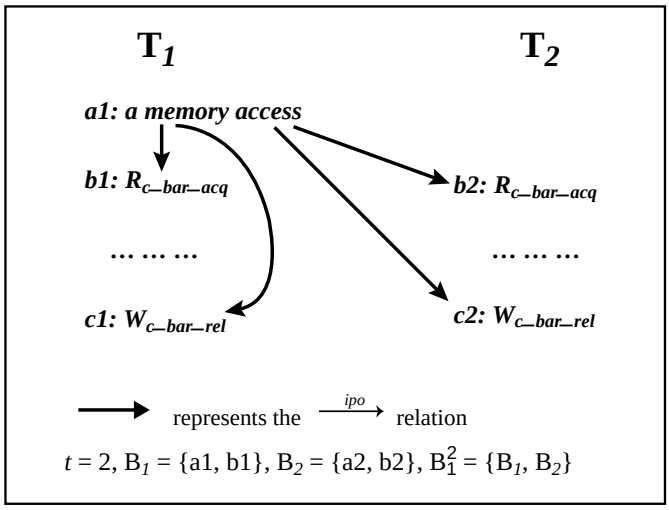

Fig. 6. An example illustrating the $\xrightarrow{ipo}$ relation.

Definition 5: Intra-Processor Program Order

Assume that *t* threads are participating in a barrier synchronization instance, B, on processor P_k. A set denoted as B_r is composed of $R_{c\_bar\_acq}$ and $W_{c\_bar\_rel}$ instruction instances for thread *r*, *Tr*, to pass instance B, where r ranges from 1 to t. A set denoted as B_1^t is the union of B_1, B_2, …, and B_t. We define the *intra-processor program* order, denoted by $\xrightarrow{ipo}$, as an order on the dynamic instruction instances from different threads on the same processor in a run of the program. If a memory operation, *o*, is prior to all instruction instances in B_r in program order, then for any element, *b*, in B_1^t, o $\xrightarrow{ipo}$ b.

Definition 6 below specifies (i) the comparison set (denoted by $COMP_{(B, k)}$) that consists of the identities of the participating threads for barrier instance B on processor P_k, and (ii) the arrival set (denoted by $ARR_{(B, k)}$) that is comprised of the identities of the participating threads that have reached barrier instance B on processor P_k at the run time. The intra-processor synchronization condition in part (f) imposes the following ordering restriction: the final issued $W_{c\_bar\_rel}$ reference on P_k cannot be performed until all the memory references prior to the $W_{c\_bar\_rel}$ reference ordered by $\xrightarrow{ipo}$ have taken effect in every memory copy. The final issued $W_{c\_bar\_rel}$ reference is identified by comparing the equivalence between the set $(COMP_{(B, k)} - \{r\})$ and the set $ARR_{(B, k)}$, where the $W_{c\_bar\_rel}$ reference is issued from thread *r*. Fig. 7 shows an example to illustrate the comparison set and the arrival set. There are two threads, T_1 and T_2, on processor P_0. Therefore, $t = 2$, and the comparison set is made up of the identities of T_1 and T_2, that is, 1 and 2. On the other hand, because only the references from T_1 for barrier instance B have all been performed, the arrival set is composed of the identity of T_1. Now, we will give the formal definitions for the comparison set and the arrival set below.

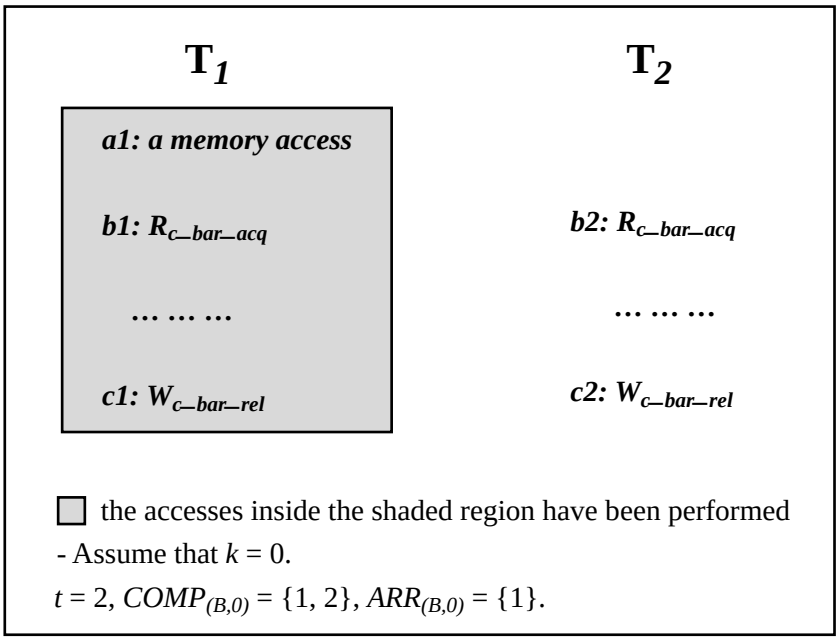

Fig. 7. An example illustrating the comparison set and the arrival set.

Definition 6: Comparison Set and Arrival Set of Barrier Synchronization Instance

Assume *t* threads are participating in a barrier synchronization instance, B, on processor P_k. A comparison set, $COMP_{(B, k)}$, is made up of the identities of the *t* threads. An arrival set of a barrier synchronization instance, $ARR_{(B, k)}$, for processor Pk to pass instance B is initially an empty set. Once a $W_{c\_bar\_rel}$ instruction instance from thread *r* is performed, the element, *r*, is added to the arrival set $ARR_{(B, k)}$.

In summary, the PSC_{SC} model relaxes two ordering restrictions imposed by RC_{SC}. (1) A CS-release reference does not wait for ordinary references prior to the paired CS-acquire reference in program order. Because a CS-release reference gives permission to other process to read/write the shared data protected by the corresponding critical section, it only needs to be delayed for the references within its corresponding critical section. (2) If a bar-release reference is not the final one reaching the corresponding barrier instance on a processor, then it need not be delayed for the ordinary references prior to the paired bar-acquire reference ordered in program order. This is reasonable because even when the bar-release reference has completed, the issuing thread cannot pass the corresponding barrier instance.

Because we relax the release consistency model by means of different semantics of critical section and barrier synchronization and we assure that the PSC model enforces the original semantics of these two synchronization primitives, a program can be executed correctly under the PSC model if memory references can be labeled properly. In the next section, we will introduce the proper-labeling technique.

3.3 Proper labeling technique

The categorization of shared memory references described in Section 3.2 is based on the intrinsic property of an reference. Like the release consistent system, the programmer or the compiler has the responsibility of labeling memory references. The goal of labeling is to ensure that a program can be executed correctly under relaxed consistent systems [10]. The label represents what is asserted about the categorization of the reference. It needs to have a proper relationship with the actual category of an reference. Fig. 8 shows the labels used for memory references in PSC systems, where the subscript L denotes that these are labels. The labels that are in the same level are disjoint, and a parent label includes all properties of its leaf labels.

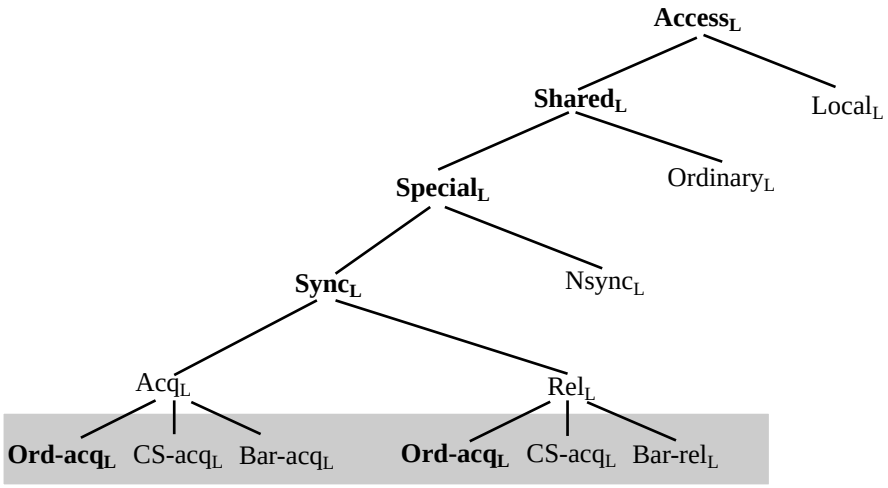

Fig. 8. Labels for memory references.

If every reference is labeled with the category it actually belongs to, the labeling is clearly correct. However, the conventional categories may be difficult to determine for certain references. Therefore, we allow such labeling to be conservative. In other words, the references labeled by our categories will have more restrictions on reference ordering than those of the conventional categories. In those cases, we ensure correctness at the cost of not fully exploiting the potential for performance. Conservative labels are shown in bold type in Fig. 8. The following definition formalizes when a program is properly labeled.

Definition 7: Properly Labeled Programs for the PSC model (PL_{PSC} Programs)

A program is properly labeled for the PSC model if (i) all references labeled $shared_L$ are *shared* references, (ii) all references labeled $special_L$ are special references, (iii) all references labeled $sync_L$ are *synchronization* references, (iv) all references labeled $ord\text{-}acq_L$ are *ord-acquire* references, and (v) all references labeled $ord\text{-}rel_L$ are *ord-release* references.

Fig. 9 shows the implementation of two common synchronization primitives [13] that we will use to illustrate properly labeling. Figure 9(a) shows a critical section bounded by a pair of lock and unlock operations, where we use a test-and-set to lock a synchronization variable and a write to unset the lock. The test is labeled as a CS-acquire reference and the write used to unset the lock is labeled as a CS-release reference. In addition, the set of the test-and-set does not compete with any other operations and can, therefore, be labeled as ordinary.

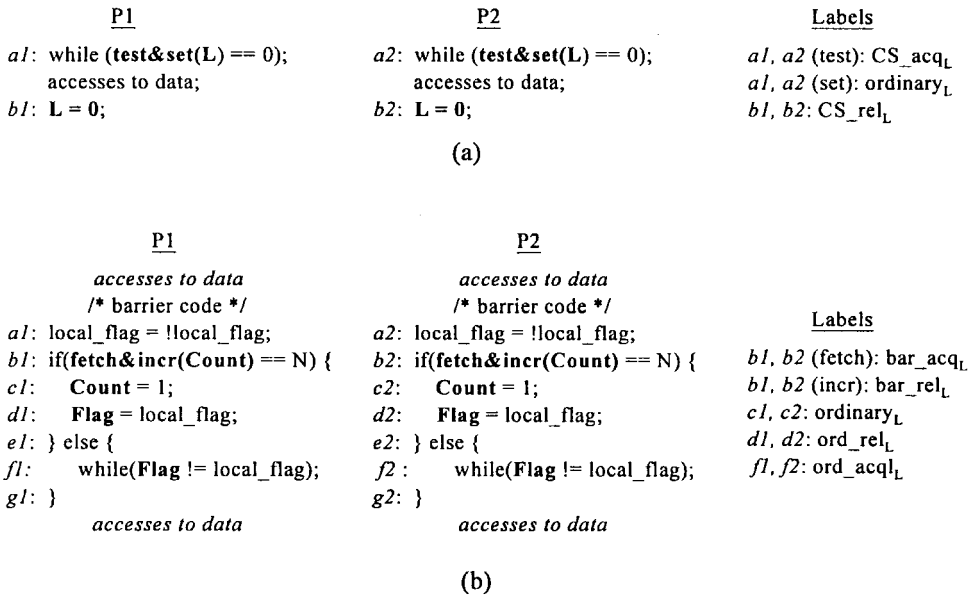

Fig. 9. Example of program segments: (a) critical section, (b) barrier.

Fig. 9(b) shows the implementation of a barrier [20] using an atomic fetch-and-increment operation. Because the fetch-and-set operation marks the corresponding thread as present in a barrier, the fetch is a bar-acquire reference, and the increment is a bar-release reference. The write to Count is ordinary because it is executed by the final thread reaching the barrier to reset the counter, based on which the number of threads having arrived at the barrier will be calculated. Therefore, it is non-competing. The write to Flag is used to inform all other threads that they can pass the barrier. Consequently, this write is qualified as an ord-release because it needs to be delayed until all preceding memory references have all performed. The read to Flag is qualified as an ord-acquire reference.

We will next discuss how the category of an reference can be conveyed in PL_{PSC} programs. The PARMACS macros from the Argonne National Laboratory [19] used in the SPLASH suite extend conventional sequential languages through parallel runtime libraries. The macros provide a set of predefined high-level synchronization constructs which programmers can use to order all conflicting memory references. Programmers only have to understand the exact semantic of each predefined synchronization primitive and use these constructs to ensure that all memory references are race-free or non-competing. Only the writers of the high-level constructs need to see the full complexity of a PL_{PSC} program. Compared with labeling for release consistency, they only have to label the instructions differently in the macros for barrier synchronization and the lock-unlock pair for the critical section according to their own semantics. Therefore, it is easy to transform a program labeled for release consistency into one labeled for the PSC model. On the other hand, because acquire (release) references in the release consistency and ord-acquire (ord-release) references in the PSC model have the same ordering restrictions, a PL_{PSC} program can also be easily executed in the release consistent system. We only need to recognize the ord-acquire (ord-release), CS-acquire (CS-release), and bar-acquire (bar-release) references as ord-acquire (ord-release) ones. In the following section, we will introduce how the PS_{CSC} model can be implemented in a PMP-MP architecture.

4. IMPLEMENTATION OF PSC MODEL

To reduce the complexity of implementation, we do not optimize all types of barrier synchronization. We call a barrier operation a Type I barrier if it is participated by all threads; otherwise we call a Type II barrier. We will investigate Type I barriers in order to improve the performance of PMP-MPs under the PSC_{SC} model for the following reasons. (1) They are used much more frequently in the SPLASH suite programs [18]. (2) Hardware can easily decide when all the participating threads have arrived at a barrier. (We will discuss this shortly.) In other words, we label Type II barriers only with ord-acquire and ord-release references. Consequently, to distinguish between Type I and Type II barriers, we have to provide programmers with two different barrier macros. In addition, to distinguish three types of acquire and release references, we can use different macros and instructions for various references.

The PMP-MP we are studying is a cache-coherent, non-uniformed memory architecture similar to that described in [17] except that the processing element (PE)

is PMP instead of a RISC processor as illustrated in Fig. 10. The key feature of the PMP-MP is that several threads share only one cache hierarchy in each PE. It hides write latencies because each PE has a two-level cache in which each level has one write buffer to store write misses that cannot be serviced immediately. Moreover, a *dual write-cache* (DWC) is placed next to the second level cache.

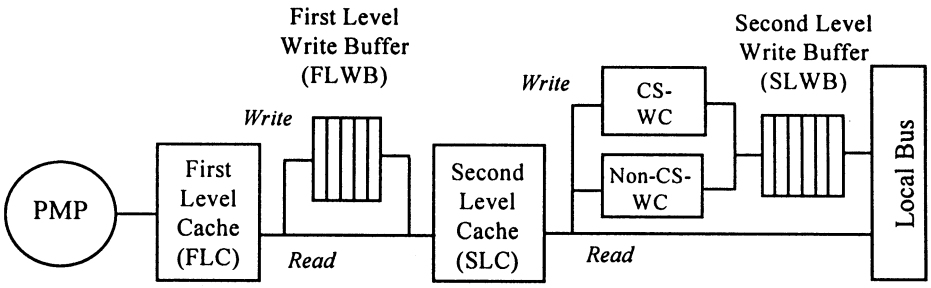

Fig. 10. Processing node architecture.

A DWC is composed of two separate write-caches (WC). Write-caches (WCs) were first proposed by Bray and Flynn for uniprocessors [16] and are used in allocate-on-write-miss, write-back, and no-allocate-on-a-read-miss strategies with a single, combined dirty/valid bit per word. A WC allocates a block frame to a write reference. The write reference is performed in the WC by setting the corresponding dirty bit and writing the data. Thereafter, all the write references belonging to the same block will be merged into a single write miss request. If the block is replaced, only dirty words need to be transferred to the next level in the memory hierarchy. Consequently, temporal and spatial locality in write references will result in less write traffic [16].

However, WCs have a different effect on shared-memory multiprocessor systems. Unlike uniprocessor systems, memory reference ordering requirements must be enforced by some underlying memory consistency model. In weak or release consistency systems, because every ordinary reference can be performed independently of other ordinary references, WCs have to be flushed only when a synchronization reference or a release reference arrives. Nevertheless, because an acquire reference request usually incurs a long synchronization waiting period, we also flush the write cache whenever an acquire reference request is encountered. Consequently, write caches can merge write reference requests between any two consecutive synchronization points, thus boosting the performance substantially [17].

DWCs are composed of two separate write-caches. One is for buffering writes issued from the code segments enclosed by CS-acquire-and-CS-release pairs or bar-acquire-and-bar-release pairs, and the other is for other write references. We call the first the *critical-section write-cache* (*CS-WC*), and the second the *non-critical-section write-cache* (*non-CS-WC*). DWCs are the key components used to implement the PSC_{SC} model in PMP-MP systems. In the following, we will describe the handling method and the function of DWCs in some detail.

Three counters on each processor are used to handle DWCs: *the switch counter, the comparison counter, and the arrival counter*. Their initial values are all zero.

The switch counter records how many CS-acquire and bar-acquire references have arrived at the SLC. When a CS-acquire or bar-acquire reference arrives, the switch counter is increased by one. In contrast, when a CS-release or a bar-release reference arrives, the switch counter is decreased by one. According to our categorization, a CS-release reference must follow a CS-acquire reference. Similarly, a bar-release reference must follow a bar-acquire reference. Consequently, the value of the switch counter is always non-negative. The value of the switch counter determines which write cache is allocated for subsequent write references. A positive value indicates that at least one thread has encountered a CS-acquire or a bar-release operation, but that the paired CS-release or bar-release reference has not yet arrived. To ensure that the memory references enclosed by the paired references are buffered in the CS-WC, a positive value switches from the non-CS-WC to the CS-WC. On the other hand, the zero value switches from the CS-WC to the non-CS-WC. Of course, the CS-WC needs to be flushed when a CS-release or a bar-release is encountered.

The purpose of the comparison counter is to record how many threads on the processor will participate in a barrier instance. Remember that we investigate only the Type I barriers for optimization. Therefore, the number of participating threads on a processor for a Type I barrier is equal to the number of threads on the processor. The value of the comparison counter is maintained by the operating system.

Finally, the arrival counter records how many threads on the processor have reached a barrier instance. Because each thread will mark itself as present at a barrier instance only one time, the arrival counter records how many bar-release references have arrived. Whenever a bar-release is encountered, the arrival counter is increased by one. If the values of the comparison counter and the arrival counter are equal, then we reset the arrival counter after flushing the CS-WC and the non-CS-WC.

The non-CS-WC and the CS-WC are also flushed to the write buffer when an ord-release reference arrives. However, unlike the handling method proposed in [17], we do not flush DWCs whenever acquire references of any type arrive. The reason is that the long synchronization waiting period incurred by an acquire reference request can be hidden by the parallel executions of other threads on the same processor for PMP-MP systems.

A DWC lookup is performed on every reference to the SLC. When a block is evicted from the DWC, all modified words in the block have to be bookkept in the SLWB. In the following, we will describe how the DWC handles different read and write conditions.

(1) If a read hits in the SLC or a write reference is done to a dirty SLC copy, no write action is taken.
(2) If a read misses in the SLC but hits in the DWC, the DWC can supply the processor with the data.
(3) If the requested word for a read-miss is valid neither in the SLC nor in the DWC, a request is sent to the memory system. When the block arrives, it is filled in the FLC and the SLC but not filled in the DWC, after the block has merged with words that are valid in the DWC.
(4) If the SLC copy is invalid or shared, a write reference is performed in the DWC by first making sure that a block frame has been allocated. We must update the DWC and set the dirty/valid bit of the corresponding word. If the block

frame has not been allocated, another DWC block might have to be evicted before the write reference can be implemented in the DWC. Because a DWC must have at most one write request for each memory word, when a write is allocated to the CS-WC, we must invalidate the corresponding word in the non-CS-WC. Similarly, when a write is allocated to the non-CS-WC, we must invalidate the corresponding word in the CS-WC.

(5) The DWC is not affected by any incoming invalidation requests because the data modified by the write requests in the DWC are invisible to other processors.

We use the following algorithm to show the interaction between the DWC and the three counters mentioned above.

Algorithm: *Interaction between the DWC and the three counters*

```
declare integer  C, A, S;
C ← the number of parallel threads on the PE;
A ← 0;
S ← 0;
while(1) {
    if (a CS-acquire or a bar-acquire reference request arrives) {
        S = S + 1;
        switch the current write cache to the CS-WC;
    }
    else if (a CS-release or a bar-release reference request arrives) {
        flush the CS-WC;
        S = S − 1;
        if (S == 0)
            switch the current cache to the non-WC;
        if (the reference is a bar-release one) {
            if (A-1 == C) {
                flush the CS-WC and the non-CS-WC;
                A = 0;
            }
            else
                A = A + 1;
        }
    }
    else if (an ord-release reference request arrives) {
            flush the non-CS-WC
}
```

In summary, to implement a PSC system, we need to provide three more macros for (i) Type I barriers, and (ii) locking and unlocking for critical sections. Moreover, we have to provide additional instructions for (i) CS-acquire and CS-release references, (ii) bar-acquire and bar-release references. The macros are implemented by means of different instructions according to their own semantics. Programmers can easily use the predefined macros to write their applications. In addition, we use three counters and a dual write-cache per processor to utilize the new parallelism

exploited by the PSC model. Because the block size of a write-cache is usually small (about 4 to 8 blocks), the hardware cost is small.

5. SIMULATION RESULTS

In order to evaluate the performance of the PSC_{SC} model for PMP-MPs, we have constructed a simulation environment named SEESMA (a simulation and evaluation environment for shared-memory multiprocessor architecture). It is a program-driven simulator consisting of a memory reference generator (front end) and a target system simulator (back end) as shown in Fig. 11. The former is basically the MINT package [21], and it models the execution of an application program on some number of processors. The latter models memory hierarchy and system interconnect. Once the program performs a memory reference, the front-end sends an event to the back-end. As soon as the event is completed, the back-end signals the front-end that the corresponding process can continue.

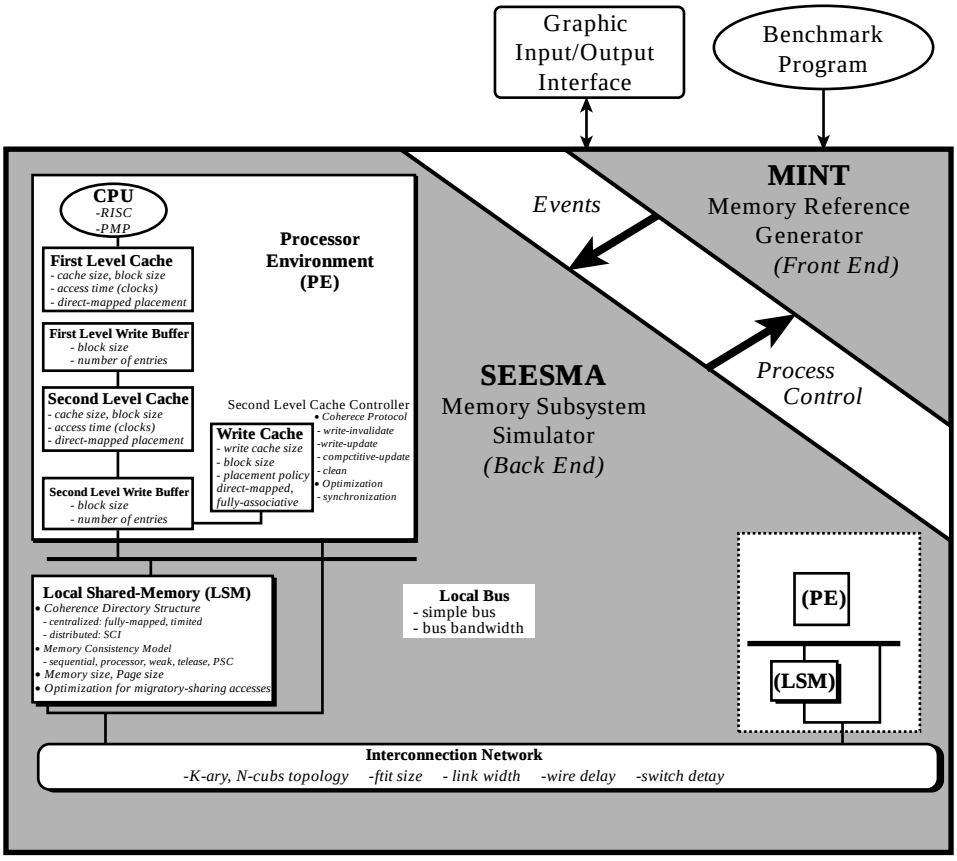

Fig. 11. Overview of the SEESMA package.

SEESMA aids investigation of shared-memory multiprocessor architecture by means of a user friendly interface. It is a software platform for education and research purposes. To achieve the objectives, SEESMA supports the following simulation environment:

(1) two-level cache;
(2) interconnection network;
(3) memory consistency models;
(4) cache coherence protocols;
(5) parallel multithreaded processor, etc.

Each sub-environment supports various options for investigating the interactions among these options. Users are provided with an X-window interface to specify the system architecture and benchmarks in addition to friendly on-line help. SEESMA runs a simulation based on a user specified architecture and outputs the evaluated results. Finally, users can analyze the interaction among system modules for the sake of cost/effective system design.

We assume that all memory references to code and private data always hit on the first level cache (FLC) and take a single processor clock. We summarize several important architecture parameters in Table 1, and others are as follows: (1) the processor is blocked on read misses but write misses; (2) the clock rate of the processor is 100 MHz; (3) the reference times of FLC and SLC are 1 and 3 processor clocks; (4) the size of each WC is 8 blocks; (5) FLC, SLC, and DWC are all direct-mapped; (6) the size of the memory page is 4 Kbytes, and the memory pages are distributed in a round-robin fashion; (7) the interconnection network is 4-by-4 torus; and (8) the linked width of the network is 64 bits.

Table 1. Architecture parameters.

Parameter	Value
Number of Processing nodes	16
Number of threads per PE	4
Sixe of FLC	16 Kbytes
Size of SLC	256 Kbytes
Block size of FLC and SLC	32 bytes
Nubmer of entries in FLWB	16
Number of entries in SLWB	32

As for the cache coherence protocol, we adopt the clean protocol, which is a fully-mapped directory-based protocol. The clean protocol is similar to the write-invalidate protocol except that the memory copy is kept clean until a write request arrives from the only PE that has a cached copy. Dahlgren and Stenstrom concluded that write caches can improve the performance of write-update, clean, and competi-

tive-update protocols [17]. However, only clean protocols with write caches (Clean-WC) and competitive-update protocols with write caches (Comp-WC) were shown to be superior to write-invalidate protocols without write caches. Moreover, because Clean-WC and Comp-WC have almost the same performance, and the competitive-update protocol requires one counter per cache block, we chose the clean protocol in order to investigate the impact of write caches on PMP-MPs after considering the cost/performance tradeoff.

We used five applications as benchmark programs, which are summarized in Table 2. All applications were written in C using the PARMACS macros from the Argonne National Laboratory [19] and have been compiled using cc under IRIS version 3 with optimization level 2 on a SGI workstation. All statistics were gathered in the parallel sections of the benchmarks.

Table 2. Benchmark programs.

Benchmark	Description	Data sets
MP3D	3-D particle-based wind-tunnel sumulator	50K particles, 10 time steps
Cholesky	Cholesky factorization of a sparse matrix	The matrix bcsstk 14
Pthor	Distributed time digital circuit simulator	RISC circuit, 1000 time step
Ocean	Ocean basin simulator	130x130 grid, tolerance 10^{-7}
Barnes	Hierarchical N-body gravitation simulator	1024 bodies, 3 steps

Fig. 12 shows a network traffic comparison for RC_{SC} and PSC_{SC}. For the sake of brevity, we will use the terms RC to denote RC_{SC} and PSC to denote PSC_{SC} programs in the subsequent discussion and figures. The network traffic for each application under the PSC model is normalized relative to that under the RC model.

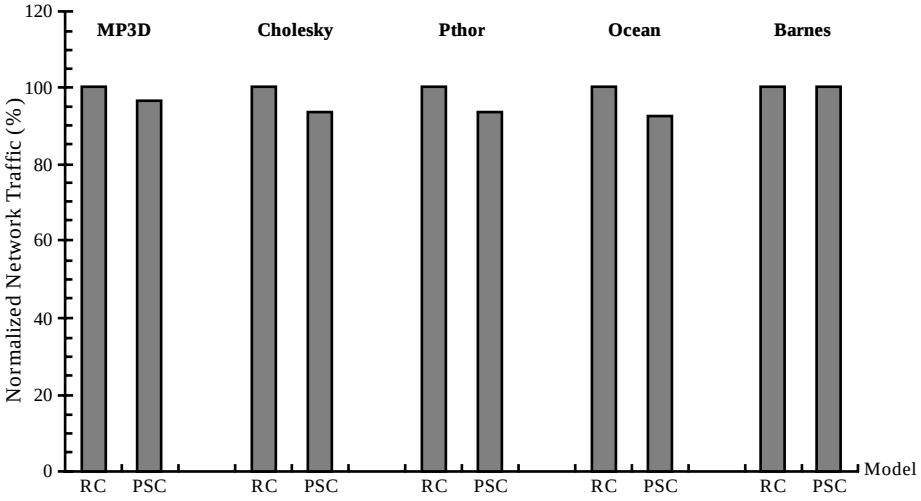

Fig. 12. Normalized network traffic for the RC and PSC models.

The network traffic is reduced for the PSC by further merging more write requests. The PSC model provides more opportunities for merging write traffic. The first reason is that DWC is not flushed whenever an acquire reference is encountered. The second reason is that a bar-release reference will not incur a flush for the DWC. Because the frequency of flushing the DWC is reduced, references from different threads on the same processor are likely to be merged.

We show a performance comparison for RC_{SC} and PSC_{SC} in Fig. 13. The execution times for each application under the PSC model were normalized relative to that under the RC model. In addition, we have decomposed each execution-time bar into five sections: the busy time (the bottom section); the read-stall time, i.e., the time spent servicing cache misses; the acquire-stall time, i.e., the time spent waiting for a lock to be acquired; the contention time, i.e., the time before permission is granted to reference the FLC; and at the top, the buffer-stall time, i.e., the time during which the processor is stalled due to a full first-level write buffer (FLWB).

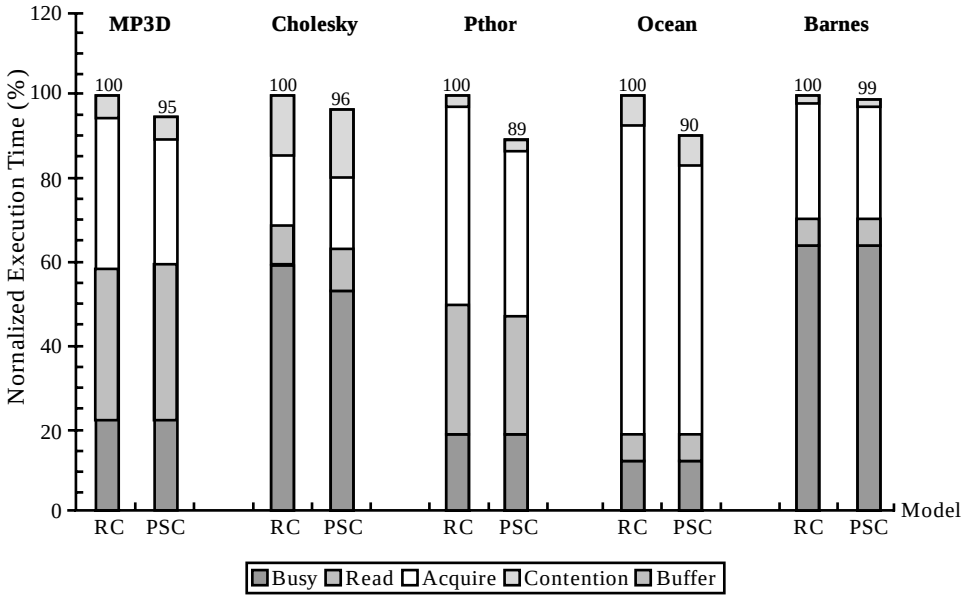

Fig. 13. Normalized execution times for RC and PSC models.

We can see that the PSC model is superior to the RC model for all application programs for the following reasons. (1) The PSC model merges more write traffic as shown in Fig. 12; thus, it reduces the network contention. (2) Acquire references have no need to wait until all the DWCs have been flushed. Therefore, the acquire-stall time is reduced. (3) A CS-release reference has to be delayed only for the references enclosed within the corresponding critical section instead of for all the references prior to the CS-release reference in program order. Consequently, CS-release references can release locks earlier and the execution time for the critical section can then be reduced. (4) Only the bar-release reference from the final

participating thread on a processor will flush the DWC. Bar-release references from other threads on the processor can be performed after the references enclosed by them and their paired bar-acquires have completed.

In the following, we will examine the different effects on total performance of various numbers of threads for each processing element. Fig. 14 shows a performance comparison between the RC model and the PSC model. The speedup is derived from the ratio of the execution times between these two models. It becomes larger when the number of threads is increased. This is because more threads on a processor provide more opportunities to merge write traffic and to optimize bar-release references in the PSC model. The exception is that the release consistency model outperforms the PSC model for the Cholesky program when the number of threads on a processor is 2. This is because Cholesky is dynamically scheduled during run time and its busy time may be not equal for different runs. For a configuration of 2 threads on each processor, the busy time for the PSC model is larger than that for the RC model.

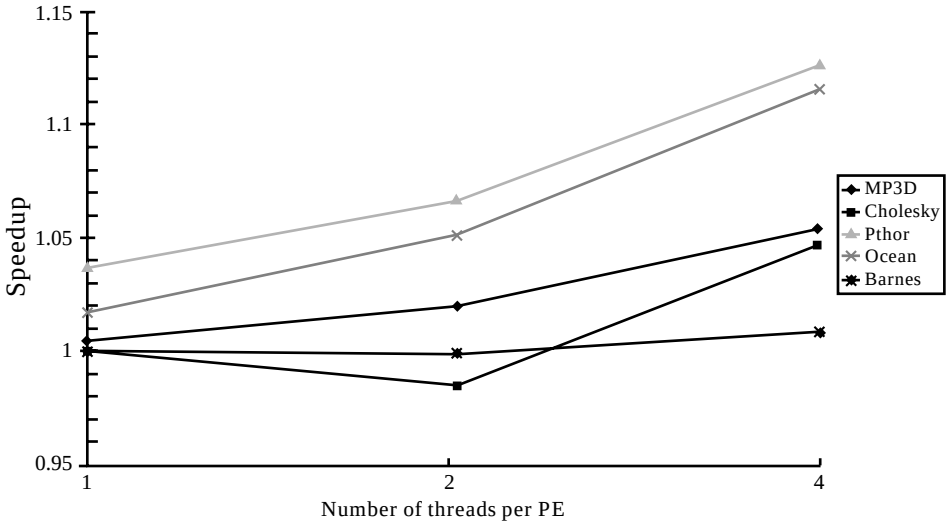

Fig. 14. Performance speedup for various numbers of threads per processor.

Another important observation is that the PSC model has better performance than the RC model for a configuration of 1 thread per PE. The PSC model has the same restrictions on ord-release references as does the RC model because there is only one thread on each PE. Therefore, the performance gain is only due to the relaxation on CS-release references. A CS-release reference has no need to wait for its previous references to all be performed. Consequently, the synchronization waiting time is shortened; thus, the system performance is boosted.

6. CONCLUDING REMARKS

In multiprocessor systems, it is very important to provide an efficient memory consistency model. In this paper, we have proposed a hardware-centric model, called the PMP-MP-specific consistency (PSC) model. This new model extends the release consistency model by partitioning acquire and release references into three sub-categories. It makes the data consistent under different reference ordering requirements of lock and barrier synchronization. Locks protect only the data that can be accessed within critical sections enclosed by it and its paired unlocks. Consequently, only the protected data need be made consistent before locks are released. On the other hand, barriers enforce all data prior to them in the program order to become consistent before subsequent codes are executed.

According to the categorization, fewer restrictions on memory reference ordering can be imposed on each synchronization reference. Thus, the PSC model not only allows more pipelining and buffering, but also exploits new parallelism. We have implemented the PSC model by incorporating DWCs into PMP-MP systems to utilize the full performance potential as much as possible. DWCs can merge more write traffic and reduce the time spent waiting for locks to be acquired. According to our simulation results on five application programs in the SPLASH suite, the PSC model outperforms the release consistency model at best by about 11% for a configuration of 4 threads on a processor. In addition, the PSC model has better performance gain when the number of threads on a processor is increased.

In the future, we will study aggressive conditions for PSC_{SC} that impose constraints on the execution order among conflicting sub-operations only. The aggressive specification does not prohibit practical optimization that does not violate the semantics of the model. Consequently, it is easier for the system designer to implement the model efficiently. Furthermore, we will study the original and aggressive conditions for the PSC_{PC} model.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Ting-Lu Huang for his helpful suggestions. This research was supported by the National Science Council of the Republic of China under contract number: NSC86-2213-E009-094.

REFERENCES

1. M. Johnson, *Superscalar Micropropcessor Design*, Prentice Hall, Englewood Cliff, New Jersey, 1993.
2. R.A. Iannucci, G.R. Gao, R.H. Halstead and B. Smith, *Multithreading: A Summary of the State of the Art,* Kluwer Academic Publishers, 1993.
3. H. Hirata, K. Kimura, S. Nagamine and Y. Mochizuki, "An elementary processor architecture with simultaneous instruction issuing from multiple threads," *The 19th Annual International Symposium on Computer Architecture,* 1992, pp. 136-145.

4. S.W. Keckler and W.J. Dally, "Processor coupling: integrating compiler time and runtime scheduling for parallelism," *The 19th Annual International Symposium on Computer Architecture*, 1992, pp. 202-213.
5. M. Bekerman, A. Mendelson and G. Sheaffer, "Performance and hardware complexity tradeoffs in designing multithreaded architectures," in *Proceedings of Parallel Architecture and Compiler Techniques*, 1996, pp. 24-34.
6. L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocessor programs," *IEEE Transactions on Computer*, Vol. 28, No. 9, 1979, pp. 241-248.
7. Goodman. J. R., "Cache consistency and sequential consistency," Technical Report 61, SCI Committee, 1989.
8. M. Dubois, C. Scheurich, and F. Briggs, "Memory reference buffering in multiprocessors," *The 13th Annual International Symposium on Computer Architecture*, 1986, pp. 434-442.
9. V.S. Adve and M.D. Hill, "Weak ordering- a new definition," *The 17th Annual International Symposium on Computer Architecture*, 1990, pp. 2-14.
10. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons A. Gupta and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *The 17th Annual International Symposium on Computer Architecture*, 1990, pp. 15-26.
11. K. Gharachorloo, A. Gupta and J. Hennessy, Revision to "Memory consistency and event ordering in scalable shared-memory multiprocessors," Technical Report, CSL-TR-93-568, Department of Computer Science, Stanford University, 1993.
12. S.V. Adve, "Designing memory consistency models for shared-memory multiprocessors," Ph. D. dissertation, Computer Science Department, University of Wisconsin-Madison, 1993.
13. K. Gharachorloo, "Memory consistency models for shared-memory multiprocessors," Ph. D. dissertation, Department of Electrical Engineering, Stanford University, 1995.
14. K. Gharachorloo, A. Gupta and J. Hennessy, "Performance evaluation of memory consistency models for shared-memory multiprocessors," in *Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 245-257.
15. K. Gharachorloo, S.V. Adve, A. Gupta, J. Hennessy and M.D. Hill, "Specifying system requirements for memory consistency models," Technical Report, CSL-TR-93-594, Department of Computer Science, Stanford University, 1993.
16. B.K. Bray and M.J. Flynn, "Write-caches as an alternative to write buffers," Technical Report CSL-TR-91-470, Computer Systems Laboratory, Stanford University, 1991.
17. F. Dahlgren and P. Stenstrom, "Using write-caches to improve performance of cache coherence protocols in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 26, No. 2, 1995, pp. 193-210.
18. J.P. Singh, W-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," *ACM SIGARCH Computer Architecture News*, Vol. 20, No. 1, 1992, pp. 5-44.

19. J. Boyle, R. Bulter, T. Disz, et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc., New York, 1987.
20. J.M. Mellor-Crummey and M.L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, Vol. 1, No. 9, 1991, pp. 21-65.
21. J.E. Veenstra and R.J. Fowler, "MINT tutorial and user manual," Technical Report 452, The University of Rochester, 1994.

Chao-Chin Wu（伍朝欽）received the B.S. degree in Computer Science and Engineering from the Tatung Institute of Technology, Taiwan, in 1990 and the M.S. degree in Computer Science and Information Engineering from National Chiao Tung University, Taiwan, 1992. Currently, he is a Ph.D. candidate in the Department of Computer Science and Information Engineering, National Chiao Tung University. His research interests include computer architecture and parallel processing.

Cheng Chen（陳正）is a professor in the Department of Computer Science and Information Engineering at National Chiao Tung University in Taiwan, R.O.C. Since 1972, he has been a member of the faculty of National Chiao Tung University. He served as the chairman of the Department of Computer Science and Information Engineering from 1987 to 1988, and as the deputy director of the Microelectronics and Information Systems Research Center (MIRC), National Chiao Tung University, from 1990 to 1994. He spent his sabbatical leaves in the Department of Computer Science, the University of Illinois at Urbana-Champior from 1980 to 1981, and later, in the Department of Computer Science, Carnegie-Mellon University, in the 1988 academic year. His current research interests include computer architectures, parallel processing, parallelizing compiler, and VOD server architecture. He received his M.S. degree from the Institute of Electronics, National Chiao Tung University, Hsinchu, Taiwan, in 1971.

Prof. Chen is a member of the Chinese Institute of Electrical Engineering and the Computer Society of the Republic of China.