Sufficient System Requirements for Supporting the PLpc Memory Model

Sarita V. Adve
Kourosh Gharachorloo
Anoop Gupta
John L. Hennessy
Mark D. Hill

Technical Report #1200

December 1993

Sufficient System Requirements for Supporting the PLpc Memory Model *

Sarita V. Adve[†], Kourosh Gharachorloo[‡],
Anoop Gupta[‡], John L. Hennessy[‡], and Mark D. Hill[†]

[†]Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706

[‡]Computer System Laboratory
Stanford University
Stanford, CA 94305

University of Wisconsin-Madison Computer Sciences Technical Report #1200
Stanford University Technical Report CSL-TR-93-595

Abstract

The paper, *Programming for Different Memory Consistency Models* [GAG+92], defines the PLpc memory model. This companion note formalizes the system requirements for PLpc along with a proof that shows these requirements are sufficient for supporting this model. In addition, we prove the correctness of the conditions presented in the original paper [GAG+92] for porting PLpc programs to the various hardware-centric models. The reader should be familiar with the material in the original paper on PLpc [GAG+92] before reading this supplement.

1 Introduction

This is a supplementary note to the paper that defines the PLpc memory model [GAG+92]. Section 2 provides a brief overview of the PLpc model along with the key definitions; a few definitions are changed slightly compared to the original versions to either address subtle correctness issues or to increase clarity. Section 3 specifies the sufficient system requirements for supporting PLpc; this specification is based upon a general framework we have proposed for specifying system conditions for different memory models [GAG+93]. Section 4 briefly describes the conditions presented in the PLpc paper [GAG+92] for porting PLpc programs to hardware-centric models. Appendices A, B, and C provide extended forms for some of the conditions discussed in the paper. Finally, Appendix D provides the correctness proof for the system requirements to support PLpc and Appendix E proves the correctness of the porting conditions.

*The University of Wisconsin authors are supported in part by a National Science Foundation Presidential Young Investigator Award (MIPS-8957278) with matching funds from A.T. & T. Bell Laboratories, Cray Research Foundation and Digital Equipment Corporation. Sarita Adve is also supported by an IBM graduate fellowship. The Stanford University authors are supported by DARPA contract N00039-91-C-0138. Anoop Gupta is partly supported by a NSF Presidential Young Investigator Award.

1200

2 The PLpc Model

This section presents a brief overview of the relevant definitions for the PLpc model as they appear in the original paper on PLpc [GAG+92]. We refer the reader to the original paper for the motivation, intuition, and full description of the PLpc model. Some of the definitions and conditions presented below are slightly different as compared to those in the original paper [GAG+92]. (Some changes are made for clarity, while others are for correctness or are restrictions on the original conditions because we were unable to do the proofs for the more aggressive conditions.)

We begin by providing some terminology used in the PLpc framework [GAG+92]. For every execution of a program, the program text defines a partial order, called *program order* ($\xrightarrow{po}$), on the memory accesses of each process in the execution [SS88]. A system is sequentially consistent (SC) if the result of every execution on it can be obtained by some total order ($\xrightarrow{to}$) of the memory accesses of the execution such that $\xrightarrow{to}$ obeys $\xrightarrow{po}$ [Lam79]. An *SC execution* refers to an execution of a program on an SC system. Two accesses are considered *conflicting* if they are to the same location and at least one of them is a write [SS88].

Definitions 2.1 and 2.2 below define an *ordering chain* and the notion of *competing* and *non-competing* accesses.

Definition 2.1: Ordering Chain
For two conflicting accesses u and v of an SC execution with a total order $\xrightarrow{to}$, an ordering chain exists from access u to access v if $u \xrightarrow{po}$ v or $u \xrightarrow{po} w_1 \xrightarrow{to} r_1 \xrightarrow{po} w_2 \xrightarrow{to} r_2 \xrightarrow{po} w_3 \ldots \xrightarrow{to} r_n \xrightarrow{po} v$ where $n \geq 1$, w_i is a write access, r_j is a read access, and w_i and r_j are to the same location if $i = j$. If all accesses in the above chain are to the same location, then u may be the same as w_1, and v may be the same as r_n as long as there is at least one $\xrightarrow{po}$ arc in the chain.

Definition 2.2: Competing and Non-competing Accesses
Two conflicting accesses of an SC execution of a program form a *competing pair* if there is no ordering chain between them. An access is a *competing access* if there is at least one SC execution in which this access belongs to a competing pair. Otherwise, it is *non-competing*.

The next definition describes a *synchronization loop construct*. This is the simple definition; the more general definition appears in Appendix A.

Definition 2.3: Synchronization Loop Construct
A *synchronization loop construct* is a sequence of instructions in a program that satisfies the following. (i) The construct executes a read or a read-modify-write to a specific location. Depending on whether the value returned is one of certain specified values, the construct either terminates or repeats the above. (ii) If the construct executes a read-modify-write, then the writes of all but the last read-modify-write store values that are returned by the corresponding reads. (iii) The construct terminates in every SC execution.

As in the original definition, we assume that the unsuccessful reads of a synchronization loop construct do not contribute to the result of the program [GAG+92]. Thus, we assume that if all accesses of a synchronization loop construct are replaced with only the last read or read-modify-write that exited the loop, we still get a SC execution with the same result as before. Therefore, in analyzing SC executions, we treat a synchronization loop construct as a single access which is the last read or read-modify-write that terminates the loop construct.

The definition of the synchronization loop construct (Definition 2.3) can be further constrained to simplify the system specification for PLpc. The additional constraint is as follows. Consider any read R of a synchronization loop construct in execution E1. If the instruction instance corresponding to R occurs in any other execution E2, then the values that can terminate the instance of the synchronization loop construct corresponding to R are the same in E1 and E2.[1] Imposing the above additional constraint results in a simplification of the reach relation discussed in Appendix B. Appendix B specifies the reach relation with and without the above additional constraint. Furthermore, the above constraint can be applied to the more general definition for a synchronization loop construct provided in Appendix A.

[1] The intuitive notions of an execution and that of an instruction instance of one execution occurring in another execution are formalized in Section 3.1 and Appendix B respectively.

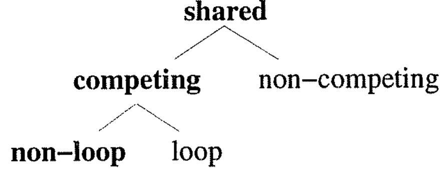

Figure 1: Categorization of read and write accesses to shared data.

Given the above definitions, *loop* and *non-loop* accesses are defined as follows.[2] [3] [4] Unless mentioned otherwise, terms such as *last*, *after*, and *between* used below refer to the ordering by the execution order. In addition, we assume that the initial value of each memory location is modeled by a hypothetical write to that location that writes the initial value at the beginning of the execution; we assume this write occurs before (in the execution order) all other conflicting operations to the given memory location.

Definition 2.4: Loop and Non-loop Reads
A competing read (R) is a *loop read* if in any SC execution
(i) it is the final read of a synchronization loop construct that terminates the construct,
(ii) it competes with at most one write,
(iii) whenever it competes with a write (W), the write is necessary to make the synchronization loop construct terminate; i.e., the read returns the value of that write and the immediately preceding write would not make the loop construct terminate, and
(iv) let W' be the last conflicting write (if any) before W whose value could terminate the loop construct; if there exists any competing write W" to the same location between W' and W (by definition, W" fails to terminate the loop construct), then there is an ordering chain from W" to R that ends with a $\xrightarrow{po}$.
A competing read that is not a loop read is a *non-loop read.*

Definition 2.5: Loop and Non-loop Writes
A competing write W is a *loop write* if in any SC execution:
(i) it competes only with loop reads, and
(ii) for any non-loop read R that conflicts with W and is after W, there is an ordering chain from W to R that ends with a $\xrightarrow{po}$.
A competing write that is not a loop write is a *non-loop write.*

Figure 1 shows the categorization of read and write accesses to shared data in PLpc. We use the abstraction of a *label* associated with every access to identify the category of the access. In PLpc, the valid labels are *non-competing*, *loop*, and *non-loop*. We use the subscript $_L$ to distinguish the label from the intrinsic category of the access. We assume the labels can be conveyed either through special instructions or through the address of the memory operation (or both). The following definition (Definition 2.6) formalizes when a program is properly labeled according to PLpc.[5] Definition 2.7 defines the PLpc memory model.

Definition 2.6: Properly Labeled Programs (PLpc Programs)
A program is *properly labeled* (PLpc) if (i) all accesses labeled *non-competing* $_L$ are *non-competing* and (ii) all accesses labeled *loop* $_L$ are either loop accesses or non-competing.

[2]The third clause (iii) in the definition of loop read (Definition 2.4) is phrased differently in the original definition [GAG +92]. It turns out that this makes a subtle semantic difference and we have changed the wording to correct this. In addition, clause (iv) is missing in the original definition; we need this extra condition to prove the correctness of the optimizations we intended for PLpc systems and the ports to hardware-centric models. Similarly, clause (ii) is missing in the original definition of loop writes (Definition 2.5).

[3]The following is a more aggressive definition for clause (iv) of Definition 2.4 (it uses the $\xrightarrow{co}$ relation defined in Section 3.1): let W' be the last conflicting write (if any) before W whose value could terminate the loop construct; if there exists a $\xrightarrow{po} \cup \xrightarrow{co}$ path from any competing write W" between W' and W to R that begins with $\xrightarrow{co}$, ends with $\xrightarrow{po}$, and consists of write-to-read $\xrightarrow{co}$'s only, then there should be an ordering chain from W" to R that ends with a $\xrightarrow{po}$.

[4]The following is a more aggressive definition for clause (ii) of Definition 2.5: for any non-loop read R that conflicts with W and for which there is a $\xrightarrow{po} \cup \xrightarrow{co}$ path from W to R that begins with $\xrightarrow{co}$, ends with $\xrightarrow{po}$, and consists of write-to-read $\xrightarrow{co}$'s only, then there should be an ordering chain from W to R that ends with a $\xrightarrow{po}$.

[5]There is a minor omission in the second clause of Definition 2.6 in the original paper [GAG +92]; the original definition did not say that non-competing accesses may be labeled loop $_L$ even though the description in the text mentioned this.

Definition 2.7: The PLpc Memory Model
A system obeys the PLpc memory model if all executions of any PLpc program on the system are SC executions.

3 Sufficient System Requirements for PLpc

This section provides a set of sufficient system conditions that satisfy the PLpc model. These conditions are presented using a general formalism and framework that we have proposed for specifying system requirements for memory models [GAG+93]. The first part of the section provides a brief background on the specification framework. We then proceed to present the specific conditions that satisfy PLpc. The proof of correctness for the system conditions for PLpc is given in Appendix D.

3.1 Background on Specification of System Requirements

This section provides a brief overview of the framework we use for specifying the system requirements. Only the relevant terminology and definitions are provided below; we refer the reader to the paper that proposes this framework [GAG+93] for a more detailed treatment (much of what follows is paraphrased from that paper). The specification methodology described below naturally exposes the architecture and compiler optimizations allowed by a memory model, thus leading to a relatively simple translation of the conditions into correct and efficient implementations.

The specification assumes the following system abstraction. The system consists of n processors, $P_1, \ldots, P_n$. A shared read operation R by P_i is comprised of a single atomic sub-operation, *R(i)*. A shared write operation W by P_i is comprised of *(n+1)* atomic sub-operations: the initial write sub-operation $W_{init}(i)$ and n sub-operations *W(1)*, ..., *W(n)* (all sub-operations of W access the same location and write the same value). Definition 3.1 below defines the notion of an *execution* and Conditions 3.1 and 3.2 specify restrictions that should be obeyed by any execution. (Appendix C provides a more aggressive version of Condition 3.1 that applies to the system specification for PLpc.)

Definition 3.1: Execution
An *execution* of a program consists of the following three components and should obey Conditions 3.1 and 3.2 specified below.
(i) A (possibly infinite) set I of instruction instances where each instruction instance is associated with the processor that issued it, registers or memory locations it accessed, and values it read or wrote corresponding to the accessed registers and memory locations.
(ii) A set S of shared-memory read and write sub-operations that contains exactly the following components. For every instruction instance i in I that reads a shared-memory location, the set S contains the memory sub-operation for the reads of i with the same addresses and values as in i. For every instruction instance i in I that writes a shared-memory location, the set S contains the W_{init} sub-operation for the writes of i with the same addresses and values as in i, and possibly other sub-operations of these writes in the memory copies of different processors.
(iii) A partial order, called the program order (denoted $\xrightarrow{po}$), on the instruction instances in I, where the partial order is total for the instruction instances issued by the same processor.

Condition 3.1: Uniprocessor Correctness Condition
For any processor P_i, the instruction instances of P_i in the set I (including the associated locations and values) belonging to an execution, and its corresponding program order $\xrightarrow{po}$, should be the same as the instruction instances and program order of a conceptual execution of the code of processor P_i on a *correct* uniprocessor, given that the shared-memory reads of P_i (i.e., the R(i)'s) on the uniprocessor are made to return the same values as those specified in the set I.

Condition 3.2: Value Condition

Consider the set S of shared-memory read and write sub-operations. If set S is part of an execution, then there must exist a total order called the *execution order* (denoted $\xrightarrow{xo}$) on the memory sub-operations of S such that (a) only a finite number of sub-operations are ordered by $\xrightarrow{xo}$ before any given sub-operation, and (b) any read sub-operation R(i) by P_i returns a value that satisfies the following conditions. If there is a write operation W by P_i to the same location as R(i) such that $W_{init}(i) \xrightarrow{xo} R(i)$ and $R(i) \xrightarrow{xo} W(i)$, 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 $\xrightarrow{xo}$.

The specific requirements for a memory model are presented as restrictions on the execution order (defined in Condition 3.2). The following three conditions impose these restrictions.

Condition 3.3: Initiation Condition (for memory sub-operations)

Given two operations by P_i to the same location, the following must be true. If R $\xrightarrow{po}$ W, then R(i) $\xrightarrow{xo}$ $W_{init}(i)$. If W $\xrightarrow{po}$ R, then $W_{init}(i) \xrightarrow{xo} R(i)$. If W1 $\xrightarrow{po}$ W2, then $W1_{init}(i) \xrightarrow{xo} W2_{init}(i)$.

Condition 3.4: $\xrightarrow{sxo}$ Condition

Let X and Y be memory operations. If X $\xrightarrow{sxo}$ Y, then X(i) must appear before Y(j) in the execution order for some i,j pairs (the i,j pairs used are specific to each model). The $\xrightarrow{sxo}$ relation is individually specified for each memory model.

Condition 3.5: Termination Condition (for memory sub-operations)

Suppose a write sub-operation $W_{init}(i)$ (belonging to operation W) by P_i appears in the execution. The termination condition requires that the other n corresponding sub-operations, *W(1) ... W(n)*, appear in the execution as well. A memory model may restrict this condition to a subset of all write sub-operations.

The system conditions for PLpc presented in the next section specify the components that comprise the $\xrightarrow{sxo}$ condition (Condition 3.4). We will use the following terminology in specifying the conditions. A condition like "X(i) $\xrightarrow{xo}$ Y(j) for all i,j" implicitly refers to pairs of values for i and j for which both X(i) and Y(j) are defined. In addition, we implicitly assume that X(i) and Y(j) appear in the execution for all such pairs. Two sub-operations conflict if their corresponding operations conflict with one another. The conflict order ($\xrightarrow{co}$) is defined between two conflicting operations X and Y (X $\xrightarrow{co}$ Y) if X(i) $\xrightarrow{xo}$ Y(i) holds for some i. Given X $\xrightarrow{co}$ Y, if X and Y are on different processors, then X and Y are ordered by the interprocessor conflict order ($\xrightarrow{co'}$) as well (i.e., X $\xrightarrow{co'}$ Y). RMW denotes a read-modify-write operation and AR and AW denote its individual read and write operations.

Given the above conditions on the execution order, a *valid execution order* is one that satisfies Conditions 3.3, 3.4, and 3.5 for a given memory model. An execution is a *valid execution* for a given model iff there is *at least one* valid execution order that can be constructed for it. Finally, a system correctly implements a memory model iff *every* execution allowed by the system is a *valid* execution. (The above correspond to Definitions 2, 3, and 4 in the specification framework paper [GAG+93].)

3.2 System Requirements for PLpc

This section presents the system requirements for PLpc based on the specification framework [GAG+93] discussed in the previous section. Figure 2 presents the sufficient system requirements for PLpc. We use the following notation. Rc and Wc denote competing read and write memory operations, respectively, to shared writable locations. Rnl and Wnl are non-loop operations. R and W refer to any read or write operations (including, e.g., Rnl and Wnl) to shared writable locations. The relations used in the specification are as follows (the significance of these relations is discussed in [GAG+93]). The $\xrightarrow{spo}$ and $\xrightarrow{sco}$ relations capture the specific $\xrightarrow{po}$ and $\xrightarrow{co}$ arcs that are used to define the $\xrightarrow{sxo}$ relation. In turn, the $\xrightarrow{sxo}$ relation captures certain $\xrightarrow{po} \cup \xrightarrow{co}$ orders that are used in the $\xrightarrow{sxo}$ condition to constrain the execution order. Note that the $\xrightarrow{sxo}$ relation and condition impose orders among conflicting accesses only. This is in compliance with the philosophy of imposing as few constraints as possible in order to expose more optimizations [GAG+93]. The figure enumerates the three constraints (i.e., initiation, $\xrightarrow{sxo}$, and termination conditions) imposed on the execution order ($\xrightarrow{xo}$).

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

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

$Rc \xrightarrow{po} Rnl$

$Rc \xrightarrow{po} Wc$

$Wnl \xrightarrow{po} Rnl$, to *different* locations

$Wnl \xrightarrow{po} Wc$

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

$Rc \xrightarrow{po} RW$

$RW \xrightarrow{po} Wc$

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

$Wc \xrightarrow{co'} Rc$

$Rnl \xrightarrow{co'} Wnl$

$Wnl \xrightarrow{co'} Wnl$

$Rnl \xrightarrow{co'} Wnl \xrightarrow{co'} Rc$

define $\xrightarrow{sxo}$: $X \xrightarrow{sxo} Y$ if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: $RW \xrightarrow{po} W$

coherence: $Wnl \xrightarrow{co'} Wnl$

multiprocessor dependence chain: one of

$Wc \xrightarrow{co'} Rc \xrightarrow{spo} RW$

$RW \xrightarrow{spo} \{Wc \xrightarrow{sco} Rc \xrightarrow{spo'}\}* \ \{Wc \xrightarrow{sco} Rc \xrightarrow{spo}\} \ RW$

$RWnl \xrightarrow{spo'} \{A \xrightarrow{sco} B \xrightarrow{spo'}\}+ \ RWnl$

$Wnl \xrightarrow{sco} Rc \xrightarrow{spo'} \{A \xrightarrow{sco} B \xrightarrow{spo'}\}+ \ Rnl$

atomic read-modify-write (AR,AW):

if W conflicts with AR,AW, then either ($W \xrightarrow{sxo} AR$ and $W \xrightarrow{sxo} AW$) or ($AW \xrightarrow{sxo} W$)

reach: $R \xrightarrow{rch} \{W \xrightarrow{co'} R \xrightarrow{rch}\}+ \ W$

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if $X \xrightarrow{sxo} Y$, then $X(i) \xrightarrow{xo} Y(i)$ for all i.

Termination condition holds for all *competing* sub-operations.

Figure 2: Sufficient conditions for PLpc.

We now briefly describe the various components of the $\xrightarrow{sxo}$ relation. The uniprocessor dependence condition captures the order among operations on the same processor and to the same location. The coherence condition represents the order among non-loop writes; its effect on the execution order (through the $\xrightarrow{sxo}$ condition) is to require non-loop write sub-operations for each location to execute in the same order with respect to every processor. The multiprocessor dependence chain condition captures the $\xrightarrow{po} \cup \xrightarrow{co}$ orders among certain operations. The first two chains roughly correspond to the ordering chain in PLpc (Definition 2.1). The last two chains capture the order among non-loop operations. The "$\{A \xrightarrow{sco} B \xrightarrow{spo}\}+$" notation represents one or more occurrences of the given pattern within the chain (this is similar to regular expression notation; we also use "*" to denote zero or more occurrences). The atomic read-modify-write condition simply captures the atomicity of such operations with respect to other writes to the same location. Finally, the reach condition captures the order among operations arising from uniprocessor data and control dependences and certain multiprocessor dependences. We further discuss the reach condition and the reach relation ($\xrightarrow{rch}$) below.

The main purpose for the reach condition is to disallow anomalous executions that arise if we allow "speculative" write sub-operations to take effect with respect to other processors. For example, consider the program segment shown in Figure 3. If we assume all operations are labeled non-competing by default, the program shown is a PLpc program. Even though it seems that there may a possibility for the reads to compete with the writes, it turns out that the writes to locations A and B do not occur in any SC execution. The non-SC execution can occur, however, if we allow the two writes to execute speculatively by predicting that both if statements will

```
P1                    P2

if (A == 1) {         if (B == 1) {
   B = 1;                A = 1;
}                     }
```

Figure 3: Example to illustrate the reach condition.

Table 1: Mapping of PLpc accesses to hardware-centric models.

Hardware-Centric Model	*Mapping of PLpc Labels*
TSO	for every Wnl $\xrightarrow{po}$ Rnl, at least one access is mapped to a RMW
PC	every Rnl is mapped to a RMW
PSO	same as TSO, plus every Wc is preceded by a STBAR
WO	every Rc and Wc is mapped to a synchronization access
RCsc	every Rc is mapped to an acquire and every Wc is mapped to a release
RCpc	same as PC plus RCsc (Rnl mapped to RMW with R an acquire and W an nsync)

execute their then clauses. Without the reach condition, none of the other constraints in Figure 2 disallow this anomalous execution. However, to satisfy the PLpc model (which requires all executions of the program to be sequentially consistent), we need to disallow any execution in which either of the two write operations occurs. In most previous work, the conditions to disallow such executions are informal: "intraprocessor dependencies are preserved" [AH90] or "uniprocessor control and data dependences are respected" [GLL+90]. Even though some proofs of correctness (e.g., proof of correctness for PL programs executing on the RCsc model [GLL+90]) formalized certain aspects of this condition, the full condition was never presented in precise terms. The work by Adve and Hill in the context of the DRF1 model [AH92] specifies the condition more explicitly. In this paper, we have further formalized this condition and present a more aggressive form as part of specifying the sufficient conditions for PLpc.

The full description of the reach relation ($\xrightarrow{rch}$) is presented in Appendix B. Below, we provide the intuition behind this relation. A read operation reaches a write operation (R $\xrightarrow{rch}$ W) that follows it in program order 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 or address of another memory operation that is before W in program order and is related to W by certain program order arcs (e.g., $\xrightarrow{spo}$) that are used in the $\xrightarrow{sxo}$ relation. Referring back to the example program in Figure 3, the $\xrightarrow{rch}$ relation holds between the read and write operation on each processor because the read controls whether the write operation occurs through conditional execution. In this example, the reach relation in conjunction with the reach condition in Figure 2 disallow the execution in which the writes to locations A and B occur.

The proof of correctness that shows the above system conditions satisfy the PLpc memory model is provided in Appendix D. To better understand the implications of the above conditions on architecture and compiler optimizations, we refer the reader to the general discussion on implementations presented in the specification framework paper [GAG+93].

4 Porting PLpc Programs to Hardware-Centric Models

This section briefly presents the conditions that were described in the original PLpc paper [GAG+92] for correctly porting PLpc programs onto various hardware-centric models. Table 1 summarizes the mappings of accesses in a PLpc program to accesses recognized by the various hardware-centric models. The proof of correctness for the porting conditions is provided in Appendix E.

As discussed in the original PLpc paper [GAG+92], the TSO, PSO, PC, and RCpc models do not provide a direct mechanism for imposing the program order between a non-loop write followed by a non-loop read (i.e., Wnl $\xrightarrow{po}$ Rnl). Thus, we impose the required order by employing read-modify-write (RMW) operations. Typically, an access that needs to be mapped to a RMW is already part of such an operation and this requirement is naturally satisfied. However, in some cases, the access has to actually be transformed into a RMW operation. The alternative is to augment the above models with direct mechanisms for obtaining the appropriate order [GAG+92]. For TSO and PSO, we need a fence mechanism [GLL+90] that delays future reads for previous writes. Then, the alternative mapping for TSO and PSO is to place such a fence between any pair of Wnl $\xrightarrow{po}$ Rnl (for PSO, Wc still needs to be preceded by a STBAR as before). For PC, we also require the extra ability to specify certain writes to appear atomic. The alternative mapping for PC is to place a fence between any pair of Wnl $\xrightarrow{po}$ Rnl and to specify every Wnl as an atomic write. The mapping for RCpc can be made more aggressive. RCpc requires the fence to delay future non-loop reads for previous non-loop writes. The alternative for RCpc is to place this weaker fence between any pair of Wnl $\xrightarrow{po}$ Rnl and to specify every Wnl as an atomic write (as before, all Rc and Wc are specified as acquires and releases, respectively).

Among the above models, RCpc provides the most efficient platform for PLpc programs. Nevertheless, the sufficient conditions for PLpc in Figure 2 allow for a more aggressive implementation than any of the above hardware-centric systems.

5 Acknowledgements

We would like to thank Allan Gottlieb for bringing to our attention an omission in Definition 6 in the original paper [GAG+92] which is corrected in this note (Definition 2.6). We would also like to thank Michael Merritt whose comments led us to clarify the notion of competing in Definition 2 (Definition 2.2 in this note).

References

[Adv93] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors.* PhD thesis, Computer Sciences Technical Report #1198, University of Wisconsin - Madison, December 1993.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[AH92] Sarita V. Adve and Mark D. Hill. Sufficient conditions for implementing the data-race-free-1 memory model. Technical Report #1107, Computer Sciences, University of Wisconsin - Madison, September 1992.

[AH93] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GAG+92] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[GAG+93] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Specifying system requirements for memory consistency models. Technical Report CSL-TR-93-594, Stanford University, December 1993. Also available as Computer Sciences Technical Report #1199, University of Wisconsin - Madison.

[GLL+90] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

Appendix A: Extended Definition for Synchronization Loop Construct

Below, we give the more general definition than Definition 2.3 for a synchronization loop construct. The conditions below are almost identical to the general conditions provided in (the appendix of) the original paper on PLpc [GAG+92], with the main exception that clauses (vi) and (vii) were not present earlier.

Definition A.1: Loop Construct
A *loop construct* is a sequence of instructions in a program that would be repeatedly executed until a specific read in the sequence (the *exit read*) reads a specific location (the *exit location*) and returns one of certain values (the *exit read values*). If the exit read is part of a read-modify-write, then the write of the read-modify-write is called the *exit write* and the value it writes is called the *exit write value.*

Definition A.2: Synchronization Loop Construct
A loop construct in a program is a *synchronization loop construct* iff it always terminates in every SC execution of the program and the following conditions hold. Consider a modification of the program so that it executes beginning at the loop construct. Add another process to the program that randomly changes the data memory. Consider every SC execution with every possible initial state of the data memory and processor registers. At the beginning of every such SC execution, the exit read, exit location, and exit read values should only be a function of the initial state of memory and registers and of the program text. The exit write value can additionally be a function of the value that the exit read returns. Then, for every such SC execution,
(i) except for the final exit write, loop instructions should not change the value of any shared memory location,
(ii) the values of registers or private memory changed by any loop instruction cannot be accessed by any instruction not in the loop construct,
(iii) a loop instruction cannot modify the exit read, exit location, exit read values, or the exit write values corresponding to a particular exit read value,
(iv) the loop terminates only when the exit read returns one of the exit read values from the exit location and the exit write stores the exit write value corresponding to the exit read value returned,
(v) if exit read values persist in the exit location, then the loop eventually exits,
(vi) the first instruction instance program ordered after the loop is the same in every other SC execution that begins with the same initial state of data memory and processor registers, and
(vii) the only shared-memory operations in the iteration that terminates the loop should be the exit read and exit write (if the exit read is part of a read-modify-write).

When analyzing an SC execution, the accesses of a synchronization loop construct can be replaced by the final successful exit read and exit write (if any). The unsuccessful accesses can be labeled non-competing. The above definition is fairly general and allows for implementations such as locks using a test&test&set technique to be considered a synchronization loop construct.

Appendix B: Reach Relation

This appendix defines the reach relation ($\xrightarrow{rch}$). To define the reach relation, we need to consider the set of instruction instances, I, of the execution. To formalize the reach relation, we first present an abstraction for the program instruction statements. We classify instructions into three types: *computation*, *memory*, and *control*. Computation instructions read a set of registers (*register read set*) and map the read values into new values that are written to another (possibly same) set of registers (*register write set*). Memory instructions are used to read and write memory locations (both private and shared). A memory read instruction reads the address to be read from a register, reads the specified memory location, and writes the return value into a register. A memory write instruction reads the address and value to be written from registers and writes the value into the specified memory location. A memory read-modify-write instruction is both a memory read and a memory write instruction that reads and writes the same location. Therefore, for the read instruction, the register read set comprises of the address register and the register write set is the destination register. For the write instruction, the register read set comprises of the address and value registers and there is no register write set. Finally, control instructions (e.g., branch instructions) change the control flow by appropriately adjusting the program counter. The register read set for a control instruction is a set of registers whose values determine the change in the control flow. If an

Definition B.2: Multiprocessor Reach Dependence
Let X and Y be instruction instances in an execution E of program Prog. Let Y be an instruction instance that accesses shared-memory. X $\xrightarrow{Mdep}$ Y in E iff any of the following are true.
(a) X $\xrightarrow{po}$ Y in E and X and Y occur in another possible SC execution E' of Prog, where X $\xrightarrow{Udep}$+ Z $\xrightarrow{rpo}$ Y in E', Z is an instance of a shared-memory instruction, for any A $\xrightarrow{dd}$ B constituting the $\xrightarrow{Udep}$+ path from X to Z in E', B also occurs in E, and either Z does not occur in E, or Z occurs in E but is to a different address in E and E', or Z is a write to the same address in E and E' but writes a different value in E and E'.
(b) X $\xrightarrow{po}$ Y in E and X $\xrightarrow{Udep}$ Z $\xrightarrow{rpo}$ Y in E.
(c) X $\xrightarrow{po}$ Y in E and X $\xrightarrow{Mdep}$ Z $\xrightarrow{rpo}$ Y in E.
(d) X $\xrightarrow{po}$ Y or X is the same as Y. Further, X generates a competing read R within a synchronization loop construct and Y generates an operation O (different from R) such that R $\xrightarrow{rpo}$+ Y.

Definition B.3: Reach' Relation
Given an execution E and instruction instances X and Y in E (where X may or may not be the same as Y), X $\xrightarrow{rch'}$ Y in E iff X and Y are instances of memory instructions, X generates a shared-memory read, Y generates a shared-memory write, and X $\{\xrightarrow{Udep} \mid \xrightarrow{Mdep}\}+$ Y in E. For two different memory operations, X' and Y', from instruction instances X and Y respectively, X' $\xrightarrow{rch'}$ Y' iff X' is a read, Y' is a write, X' $\xrightarrow{po}$ Y' and X $\xrightarrow{rch'}$ Y.

Definition B.4: Reach Relation
Given an execution E and instruction instances X and Y in E, X $\xrightarrow{rch}$ Y in E iff X $\xrightarrow{rch'}$ Y and X generates a memory read that reads the value of another processor's write. (The $\xrightarrow{rch}$ relation among memory operations is defined in an analogous manner to $\xrightarrow{rch'}$.)

The reach relation is a transitive closure of the uniprocessor reach dependence ($\xrightarrow{Udep}$) and the multiprocessor reach dependence ($\xrightarrow{Mdep}$) relations. The uniprocessor component corresponds to uniprocessor data and control dependence, while the multiprocessor component corresponds to dependences that are present due to the memory consistency model. The components that make up $\xrightarrow{Udep}$ and $\xrightarrow{Mdep}$ are defined for a given execution E. Both relations also require considering other sequentially consistent executions of the program, and determining if an instruction in one execution occurs in the other execution.[6] For an instruction instance from one execution to occur in another, we do not require that locations accessed or the values read and written by the corresponding instruction instances in the two executions be the same; we are only concerned with whether the specific instances appear in the execution. In the absence of constructs such as loops and recursion, it is straightforward to determine if an instance that appears in one execution also appears in another. In the presence of constructs such as loops and recursion, care has to be taken to match consistent pairs of instruction instances. Instruction instances between two executions are matched consistently if the set of instances that are considered to appear in both executions have the same program order relation between them in both executions, and are the maximal such sets. (A set S with property P is a maximal set satisfying property P if there is no other set that is a superset of S and also satisfies property P.)

Definition B.5 defines the $\xrightarrow{rpo}$ relation which is used as part of the definition for $\xrightarrow{Mdep}$.

Definition B.5: $\xrightarrow{rpo}$ Relation
Let X and Y be instances of shared-memory instructions in an execution E, and let X' and Y' be the memory operations corresponding to X and Y respectively in E. X $\xrightarrow{rpo}$ Y in E iff either
(a) X' $\xrightarrow{spo}$ Y' in E, or
(b) X' $\xrightarrow{po}$ Y' in E and X' $\xrightarrow{po}$ Y' is in the uniprocessor dependence part of the $\xrightarrow{sxo}$ condition, or
(c) X'=W $\xrightarrow{po}$ Y'=R in E and the initiation condition requires that $W_{init}(i)$ $\xrightarrow{xo}$ R(i) in E.

We have shown that the $\xrightarrow{rch}$ relation as formalized above provides a sufficient condition for satisfying the PLpc model as part of the specification presented in Figure 2. However, we do not know that it is necessary and

[6] It may be simpler to be conservative and consider all possible executions of the program and not just SC executions for E' in the definitions of $\xrightarrow{Udep}$ and $\xrightarrow{Mdep}$.

believe that certain aspects such as requiring W $\xrightarrow{po}$ R as part of $\xrightarrow{rpo}$ in part (c) of definition B.2 may be overly conservative. Therefore, it may be possible to relax the $\xrightarrow{rch}$ relation in the future without violating the PLpc model.

Appendix C: Aggressive Form of Uniprocessor Correctness Condition

This appendix discusses a relaxation of the uniprocessor correctness condition (Condition 3.1) given in Section 3.1. the notion of a correct uniprocessor used in Condition 3.1 assumes the following: given an execution, E, of a correct uniprocessor, if an instruction instance, *i*, is in E, then the number of instruction instances in E that are ordered before *i* by program order is finite. This can potentially restrict implementations by prohibiting the aggressive execution of operations that may be ordered after potentially non-terminating loops by program order. In this appendix, we present a more aggressive condition that allows certain operations to execute before a preceding loop may terminate. Definition C.1 below formalizes the notion of a preceding loop; Definition C.2 formalizes the condition that determines whether an operation can execute before its preceding loop terminates.

Definition C.1: Loop
A *loop* in a control flow graph refers to a cycle in the control flow graph. A loop L *does not terminate* in an execution iff the number of instances of instructions from loop L in the execution is infinite.

Condition C.2: Infinite Execution Condition
Consider an execution E of program Prog that contains instruction instance j of instruction j', and j' is a write instruction.
(a) If *j'* follows loop L that does not terminate in some SC execution, then the number of instances of instructions in E that are from loop L and that are ordered by program order before *j* is finite.
(b) The number of instruction instances that are ordered before *j* by $\xrightarrow{rch'}$ in E is finite.

With the above condition, a processor can execute a read operation before it is known whether the previous loops in program order will terminate. For a write operation, the processor is allowed to execute it before a previous loop as long as the loop is known to terminate in every SC execution and as long as no memory operations from the loop are ordered before the write by $\xrightarrow{rch'}$. Most programs are written so that either they will terminate in all SC executions, or there are no shared-memory operations that follow a potentially non-terminating loop. In addition, the information about whether a loop will always terminate in an SC execution is often known to the programmer and can be easily obtained. Thus, the above relaxation of the uniprocessor correctness condition is applicable to a large class of programs.

Appendix D: Proof of Correctness of System Requirements for PLpc

This appendix proves that the system requirements presented in Figure 2 are sufficient for obeying the PLpc memory model. The proof is based on a general framework for proving the correctness of generic programmer-centric models developed by Adve in her thesis [Adv93]. Adve and Hill describe programmer-centric models such as PLpc as SCNF (sequential consistency normal form) models [Adv93, AH93], and so we refer to the general framework (from [Adv93]) as the SCNF framework. The SCNF framework [Adv93] characterizes an SCNF model in terms of a property called the valid paths of the model, gives a generic set of system requirements for an SCNF model primarily based on these valid paths, and proves the correctness of these system requirements. Section D.1 reviews concepts of the SCNF framework necessary to prove the correctness of PLpc system conditions. In Section D.2, we use the SCNF framework to derive a set of sufficient valid paths for the PLpc model. Section D.3 uses the SCNF framework to show that given the derived valid paths, the system requirements presented in Figure 2 are indeed sufficient for satisfying the PLpc model.

D.1: Review of the SCNF Framework [Adv93]

This section briefly and somewhat informally reviews concepts of the SCNF framework [Adv93] that are required to understand the proof of this appendix. We refer the reader to Chapter 7 of Adve's thesis [Adv93] for a more thorough understanding of these concepts.

The key notions in the SCNF framework are those of critical paths and valid paths, where critical paths characterize a program and valid paths characterize a system and memory model. The following first gives three basic definitions (from [Adv93]) that are then used to informally motivate the notions of critical and valid paths. The informal discussion is followed by more formal definitions of critical paths, valid paths, a characterization of a generic SCNF memory model in terms of valid paths, and a generic system specification for a generic SCNF memory model.[7]

Definition D.1:
The *program/conflict graph* for an execution E is a directed graph where the vertices are the (dynamic) operations of the execution and the edges represent the program order and conflict order relations on the operations.

Definition D.2:
A path in the program/conflict graph from operation X to operation Y is called a *race path* iff X $\xrightarrow{co}$ Y and there is no path from X to Y in the program/conflict graph that contains at least one program order edge.

Definition D.3:
An *ordering path* is a path in the program/conflict graph that is between two conflicting operations, and either it has at least one program order edge or it is a race path between two writes.

We say an ordering path from operation X to operation Y is executed *safely* if X(i) $\xrightarrow{xo}$ Y(i) for all i. Sequential consistency is trivially guaranteed if all ordering paths of an execution are executed safely [Adv93]. The SCNF framework shows how to guarantee sequential consistency by safely executing only a subset of the ordering paths, thereby resulting in system optimizations. Informally, for a given program, the subset of ordering paths that need to be executed safely to guarantee sequential consistency are called the critical paths for the program. A memory model is characterized in terms of the ordering paths that it guarantees to execute safely; these are called the valid paths of the model. Thus, the valid paths characterize the system, whereas the critical paths characterize the program. An SCNF memory model guarantees sequential consistency to a program if all critical paths in all sequentially consistent executions of the program are valid paths for the model.

More formally, the notion of critical paths uses the following concepts from [Adv93]. Some definitions are slightly customized to only consist of parts relevant to the PLpc model.

[7] A minor, subtle difference between the framework used in Section 3.1 (from [GAG+93]) and the SCNF framework [Adv93] is that the former requires the existence of an execution order for an execution while the latter associates a single execution order with an execution. Thus, an execution in [GAG+93] can have many execution orders and correspondingly many program/conflict graphs, whereas each such execution order corresponds to a different execution in [Adv93]. This appendix assumes the latter convention.

Definition D.4:
An operation in an execution is *unessential* if it is from a synchronization loop construct and is not from the final read or read-modify-write that terminates the loop construct in the execution. All operations in an execution that are not unessential are *essential.*

Definition D.5:
The essential reads from a synchronization loop construct are *self-ordered* reads. Synchronization loop constructs are also called *self-ordered loops.*

Definition D.6:
Two conflicting operations, O1 and O2, in a sequentially consistent execution are called *consecutive* conflicting operations iff there is no write W ordered between O1 and O2 by the execution order where W conflicts with O1 and O2.

Definition D.7 below describes the notion of critical sets and critical paths [Adv93]. (Refer to [Adv93] for the motivation of the different parts of the following definition and for a simpler, but slightly more restrictive form of the definition.) Unless mentioned otherwise, terms such as *last*, *after*, and *between* refer to the ordering by the execution order. Further, part (2) of Definition D.7 requires considering the initial values of a location. To model the effect of initial values, assume that there is a hypothetical write to each memory location that writes the initial value of the location at the beginning of the execution order. Such hypothetical writes are not considered unless explicitly mentioned.

Definition D.7:
A *critical set* for a sequentially consistent execution, Es, is a set of ordering paths of the execution that obey the following properties. Let X and Y be any two consecutive conflicting operations such that there is an ordering path or race path from X to Y in Es. Ignore all unessential operations in Es.
(1) If Y is not a self-ordered read and if there is an ordering path from X to Y, then one such path is in the critical set.
(2) Suppose Y is a self-ordered read. Let W be the last write (including the hypothetical initial write) before X such that W conflicts with Y, W writes an exit value of Y, and the following is true. If W1 (conflicting with X) is between W and X and writes an exit value of Y, then Y is a read from a read-modify-write, the first conflicting write (W2) after W1 writes a non-exit value of Y, W2 is from a synchronization loop whose exit read is competing, and the write of Y's read-modify-write writes a non-exit value for W2's loop. If W exists and if there is an ordering path from any write after W to Y that ends in a program order arc, then one such path is in the critical set.
For every sequentially consistent execution, we consider one specific critical set, and refer to the paths in that set as the *critical paths.*

Definition D.8 below defines a generic SCNF memory model in terms of its valid paths [Adv93].

Definition D.8: A Generic SCNF Memory Model
An SCNF memory model specifies a characteristic set of ordering paths called the *valid paths* of the model. A program is a *valid program* for an SCNF memory model iff for all sequentially consistent executions of the program, a critical set of ordering paths for the execution are valid paths of the model. A system obeys an SCNF memory model iff it appears sequentially consistent to all programs that are valid programs for the model.

The PLpc model does not directly map into the definition of SCNF models because it does not explicitly specify a set of valid paths. Instead, the model specifies the the set of valid programs directly (i.e., PLpc programs). However, we can use the SCNF framework for our purpose by deriving a set of *sufficient* valid paths that ensure that the critical set of ordering paths for PLpc programs are covered.

The system requirements for satisfying a generic SCNF model are described below [Adv93]. Adve has proven that given a set of valid paths that characterize an SCNF model, a system obeys the model if it satisfies Condition D.1 [Adv93]. Condition D.1 uses the notions of a control path and control relation. Definition D.9 defines the notion of a control path. The control relation in the SCNF framework [Adv93] is specified in terms of a set of properties it should obey. Adve has developed a more general version of the reach relation presented in Appendix B for generic SCNF models that obeys the requirements of the control relation [Adv93]. Therefore, the reach relation of Appendix B can be shown to trivially obey the requirements of the control relation for the sufficient valid paths we derive for the PLpc model (Section D.2) (for PLpc, the $\xrightarrow{rch'}$ relation can be conservatively

substituted for $\xrightarrow{ctl}$ used below). For this reason, we do not repeat the reach or control relations from [Adv93] here.

Definition D.9:
A *control/semi-causal-conflict graph* of an execution is a graph where the vertices are the (dynamic) memory operations of the execution, and the edges are due to the transitive closure of the control (denoted $\xrightarrow{ctl}$) relation of the execution, or of the type *Write* $\xrightarrow{co}$ *Read.*
A *control path* for an execution is a path between two conflicting operations in the control/semi-causal-conflict graph of the execution such that no read on the path returns the value of its own processor's write in the execution.

Condition D.1: System Specification of a Generic SCNF Model
Consider a program Prog. Let Es represent a sequentially consistent execution of program Prog. The system should satisfy the following requirements.
Valid Path:

If there is a valid path of the model from X to Y and if either X and Y are from the same processor, or if X is a write, or if X is a read that does not return the value of its own processor's write, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Control:

(1) *Critical set requirement:* If there is a control path from R to W, then R(i) $\xrightarrow{xo}$ W(i) for all i.
(2) *Finite speculation:*

(a) The number of memory operations that are ordered before any write operation by $\xrightarrow{ctl}$ + in E is finite.
(b) Let j be an instance of any instruction j' in E that writes shared-memory or writes to an output interface in E. If j' follows (in E) an instance L of a loop that does not terminate in some Es, then the number of instances of instructions that are from the loop instance L and that are ordered by program order before j in E is finite.

(3) *Write termination:*

(a) Let operation X and write W be in E and in some Es. Let X and W be essential in Es. If there is a race path between X and W in Es, then W's sub-operation in the memory copy of X's processor must be in E.
(b) Let R and W be in E and in some Es. Let R and W be essential in Es. If R is an exit read in E from a self-ordered loop that does not terminate in E, W writes the exit value read by R in Es, and no ordering path from W to R is valid in Es, then W's sub-operation in the memory copy of R's processor must be in E.

(4) *Loop Coherence:* If W1 $\xrightarrow{co}$ W2 and one of W1 or W2 is from a synchronization loop construct, then W1(i) $\xrightarrow{xo}$ W2(i) for all i.

To prove that the sufficient system conditions for PLpc in Figure 2 obey the PLpc model, we need to determine a set of critical paths for PLpc programs, and then show that with these critical paths as the valid paths, the system conditions of Figure 2 obey the generic system conditions specified in Condition D.1. [8]

The SCNF framework [Adv93] uses a slightly different system abstraction for specifying system conditions than the abstraction (from [GAG+93]) used in Figure 2. Informally, the main difference in the abstractions is that the former assumes that if R(i) returns the value of W(i), then W(i) $\xrightarrow{xo}$ R(i). This is not necessarily true for the specification in Figure 2 if R and W are from the same processor. The abstraction used in the SCNF framework [Adv93] is based directly on Collier's work and will be referred to as Collier's abstraction, whereas the abstraction used in Figure 2 extends Collier's work and is therefore referred to as the extended abstraction.

To allow us to directly use the SCNF framework, Figure 4 presents a specification of the sufficient conditions for PLpc using Collier's abstraction. This specification differs from the specification in Figure 4 in the first part of the $\xrightarrow{sco}$ relation, the uniprocessor dependence part of the $\xrightarrow{sxo}$ relation, and the $\xrightarrow{sxo}$ condition. It can be shown

[8] In Condition D.1(3), only those loops that are *exploited* as self-ordered are considered in the generic system condition; i.e., the critical path to exit reads of the considered loops obey part (2) of the definition of a critical set.

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

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

$Rc \xrightarrow{po} Rnl$

$Rc \xrightarrow{po} Wc$

$Wnl \xrightarrow{po} Rnl$, to *different* locations

$Wnl \xrightarrow{po} Wc$

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

$Rc \xrightarrow{po} RW$

$RW \xrightarrow{po} Wc$

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

$Wc \xrightarrow{co'} Rc$ where Rc returns the value of another processor's write

$Rnl \xrightarrow{co'} Wnl$

$Wnl \xrightarrow{co'} Wnl$

$Rnl \xrightarrow{co'} Wnl \xrightarrow{co'} Rc$

define $\xrightarrow{sxo}$: $X \xrightarrow{sxo} Y$ if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: $RW \xrightarrow{po} RW$

coherence: $Wnl \xrightarrow{co'} Wnl$

multiprocessor dependence chain: one of

$Wc \xrightarrow{co'} Rc \xrightarrow{spo} RW$

$RW \xrightarrow{spo} \{Wc \xrightarrow{sco} Rc \xrightarrow{spo'}\}* \ \{Wc \xrightarrow{sco} Rc \xrightarrow{spo}\} \ RW$

$RWnl \xrightarrow{spo'} \{A \xrightarrow{sco} B \xrightarrow{spo'}\}+ \ RWnl$

$Wnl \xrightarrow{sco} Rc \xrightarrow{spo'} \{A \xrightarrow{sco} B \xrightarrow{spo'}\}+ \ Rnl$

atomic read-modify-write (AR,AW):

if W conflicts with AR,AW, then either ($W \xrightarrow{sxo} AR$ and $W \xrightarrow{sxo} AW$) or ($AW \xrightarrow{sxo} W$)

reach: $R \xrightarrow{rch} \{W \xrightarrow{co'} R \xrightarrow{rch}\}+ \ W$

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if $X \xrightarrow{sxo} Y$ and either X,Y are from the same processor, or X is a write, or X is a read that returns the value of another processor's write, then $X(i) \xrightarrow{xo} Y(i)$ for all i.

Termination condition holds for all *competing* sub-operations.

Figure 4: Sufficient conditions for PLpc with Collier's abstraction.

that the conditions in Figure 4 are strictly weaker (i.e., more aggressive) than the conditions in Figure 2 (see Appendix H in [Adv93]). Therefore, to prove that the conditions in Figure 2 are sufficient for obeying the PLpc model, it is sufficient to show that the conditions in Figure 4 obey Condition D.1 given the valid paths we derive for PLpc. Therefore, in the remaining sections, we will assume Collier's abstraction and refer to the conditions in Figure 4.

The next section enumerates the critical paths for PLpc programs and uses this to derive the sufficient set of valid paths for the PLpc model. Section D.3 shows that given the derived valid paths, the system requirements presented in Figure 2 obey Condition D.1 and are therefore sufficient for satisfying the PLpc model.

D.2: Derivation of Valid paths for PLpc

This section derives a set of sufficient valid paths for the PLpc model. Theorem 1 below states the sufficient valid paths and the remainder of the section is dedicated to proving this theorem. Definition D.10 defines the relations used in the theorem for specifying the valid paths.

Definition D.10:
X $\xrightarrow{vco1}$ Y iff X and Y are from different processors and are respectively the first and last operations in one of

Rnl $\xrightarrow{co}$ Wnl
Wc $\xrightarrow{co}$ Rc, the last conflicting write before Rc is from a different processor than Rc
Wnl $\xrightarrow{co}$ Wnl
Rnl $\xrightarrow{co}$ Wnl $\xrightarrow{vco1}$ Rc.

X $\xrightarrow{vco2}$ Y iff X and Y are from different processors and are respectively the first and last operations in

Wc $\xrightarrow{co}$ Rc, the last conflicting write before Rc is from a different processor than Rc.

Theorem 1: Valid paths of PLpc
The following types of ordering paths (Definition D.3) *suffice* as a set of valid paths for PLpc.
(1) Xnl ($\xrightarrow{po}$ ∪ $\xrightarrow{vco1}$)+ Ynl
(2) X $\xrightarrow{po}$ (A ($\xrightarrow{po}$ ∪ $\xrightarrow{vco2}$)* B) $\xrightarrow{po}$ Y
(3) Wc $\xrightarrow{vco2}$ Rc $\xrightarrow{po}$ Y
(4) X ($\xrightarrow{po}$ ∪ $\xrightarrow{vco1}$)+ Y, X is a write from a read-modify-write whose read is non-loop, Y is a read from a read-modify-write whose write is non-loop
(5) X $\xrightarrow{po}$ Y
Further, in all of the paths, no two consecutive edges are both of the same type.

To prove the above theorem correct, we need to prove that for any sequentially consistent execution of a PLpc program, a critical set of paths of the execution are covered by the valid paths shown in Theorem 1. Let Prog be a PLpc program and let Es denote a sequentially consistent execution of Prog. Ignore all unessential operations in Es. We use Lemmas 1, 2, and 3 given below in the proof of the theorem. Below, a path of type 1, 2, 3, 4, or 5 refers to the paths specified in Theorem 1. Also, we say that an operation is of a particular category (where the category may be competing, non-competing, loop, or non-loop) if the operation is labeled as that category (a competing operation is an operation either labeled as loop or non-loop). We say that an operation is *intrinsically* of a particular category if the operation obeys the definition for that category (i.e., the label is not conservative). The notion of competing used below is from Definition 2.2 in Section 2. R, R1, R2, etc. denote reads and W, W1, W2, etc. denote writes. Recall that we assume Collier's abstraction and so if R returns the value of W, then W(i) $\xrightarrow{xo}$ R(i) for all i. Finally, since we only consider sequentially consistent executions below, the execution order can be assumed to be on operations rather than on sub-operations.

Lemma 1: Let X $\xrightarrow{co}$ Y. Suppose there is a race path from X to Y, or X and Y are consecutive conflicting operations that compete in Es. Then X $\xrightarrow{vco1}$ Y.

Proof:

First note that X cannot be an intrinsic loop read because it competes with Y but does not return the value of Y. Thus, X must be intrinsically a non-loop read or a write.

Y cannot be an intrinsic loop write because it competes with an operation which is not an intrinsic loop read. Thus, if X is a read, then X $\xrightarrow{vco1}$ Y and the lemma follows.

Therefore, assume that X is a write. If Y is also a write, then neither X nor Y can be intrinsic loop writes because a loop write does not compete with other writes. Thus, both are non-loops and the lemma follows.

The only remaining case is where X is a write and Y is a read. Since there is either a race path from X to Y in Es or X and Y are consecutive conflicting operations that compete, it follows that Y cannot return the value of its own processor's write in Es. Therefore, X $\xrightarrow{vco1}$ Y and the lemma follows.

Definition D.11:
Define the relation *rel* on the memory operations of an execution as follows. X $\xrightarrow{rel}$ Y iff X $\xrightarrow{po}$ Y, or if X is a write and Y is a read that returns the value written by X. We also view *rel* as a graph where the vertices are the operations of the execution and the edges are either $\xrightarrow{po}$ edges or W $\xrightarrow{co}$ R edges where R returns the value written by W. Thus, a path in *rel* refers to a path in the above graph.

Lemma 2: If there is a path in *rel* from X to Y in Es with at least one $\xrightarrow{po}$ arc, then there is a path in ($\xrightarrow{po} \cup \xrightarrow{vco2}$)+ from X to Y in Es with at least one $\xrightarrow{po}$ arc.

Proof:

Consider the paths in *rel* from X to Y that have the maximum number of edges and such that no two consecutive edges are $\xrightarrow{po}$. If there is no *rel* path from X to Y of the type required by the lemma, then on all the *rel* paths considered above, there must be a W $\xrightarrow{co}$ R arc on the path such that at least one of W or R is non-competing. From the above considered paths, consider one path that has the fewest number of W $\xrightarrow{co}$ R edges where at least one of W or R is non-competing. Let W1 $\xrightarrow{co}$ R1 be one of the arcs on the considered path where at least one of W1 or R1 is non-competing. Let O be the set of all operations after W1 that have a rel path to R1 with at least one $\xrightarrow{po}$ arc. Note that W1 or any operation program ordered after W1 cannot be in O since then the chosen path is not the longest possible. Consider a modification of the execution order of Es where the operations in O and R1 are moved to just after W1, retaining their original relative order. Consider the resulting order until R1. This resulting order is still consistent with program order, and the last conflicting write before any read is still the same as with the original execution order. Thus, the resulting order until R1 is the prefix of the execution order of some sequentially consistent execution where all operations until R1 are labeled as in Es and are also essential. We show next that W1 and R1 must compete in this execution. This is a contradiction since at least one of W1 or R1 is labeled as non-competing.

Suppose W1 and R1 do not compete in the above execution. Then there must be an ordering chain from W1 to R1. There cannot be an ordering chain that begins with a $\xrightarrow{po}$ arc. Therefore, the ordering chain must be of the type where all operations on the path are to the same location. However, since W1 is the last conflicting write before R1 in Es, it follows that there must be an ordering chain of the type W1 $\xrightarrow{co}$ R2 $\xrightarrow{po}$ R1, where W1 and R2 are competing and R2 returns the value of W2. Therefore, the arc W1 $\xrightarrow{co}$ R1 on the chosen *rel* path can be replaced with W1 $\xrightarrow{co}$ R2 to get a path with fewer $\xrightarrow{co}$ arcs with non-competing operations than the chosen path, a contradiction.

Lemma 3: If there is a path in *rel* from X to Y in Es that begins and ends in a $\xrightarrow{po}$ arc, then there is a path of type (2) from X to Y in Es that begins and ends with a $\xrightarrow{po}$ arc.

Proof:

The proof is identical to that of lemma 2 except that the considered *rel* paths should begin and end with $\xrightarrow{po}$ arcs.

We next use the above lemmas to prove Theorem 1. For this, we need to show that for every pair of consecutive conflicting operations, X, Y where X $\xrightarrow{co}$ Y, if a critical path is required for X and Y, then one candidate for the critical path is of type 1-5. If X and Y are from the same processor, then clearly there is a path of type 5 from X to Y. Therefore, assume that X and Y are from different processors. Also assume that a critical path is required for X and Y. The following four cases are possible.

Case 1: Both X and Y are non-loops.

Proof:

There must be an ordering path from X to Y where the $\xrightarrow{co}$ arcs are race paths and no two adjacent arcs are both $\xrightarrow{po}$. Then the $\xrightarrow{co}$ arcs are $\xrightarrow{vco1}$ arcs by lemma 1. Consider the shortest path such that the $\xrightarrow{co}$ arcs are $\xrightarrow{vco1}$ and no two adjacent arcs are both $\xrightarrow{po}$. This path is of type 1 and can be considered as the critical path.

Case 2: At least one of X or Y is non-competing.

Proof:

If there is a path of type 2 from X to Y, then this case is proved. So assume that there is no path of type 2 from X to Y. Then by lemma 3, there cannot be a *rel* path from X to Y that begins and ends

in a $\xrightarrow{po}$ arc. Let X' be the first operation after X by $\xrightarrow{po}$. Let O be the set of operations that are after X and have a *rel* path to Y in Es. X' cannot be in O. Consider a modification of the execution order of Es where the operations in O and Y are moved to just after X, retaining their relative order. Consider the resulting order until Y. This resulting order is still consistent with program order, and the last conflicting write before any read is still the same as with the original execution order. It follows that the resulting order until Y is the prefix of an execution order of some sequentially consistent execution where the operations until Y are labeled as in Es and are essential. Since X and Y cannot compete in this new execution, there must be an ordering chain from X to Y in the new execution. The only ordering chain possible is of the type X $\xrightarrow{co}$ R1 $\xrightarrow{po}$ Y. There should be a chain of the type X $\xrightarrow{co}$ R2 $\xrightarrow{po}$ Y where X and R2 compete. Further, R2 cannot return the value of its own processor's write since X and Y are consecutive conflicting operations in Es. Therefore X $\xrightarrow{vco2}$ R2 in Es. The above chain, therefore, is a path of type 3. This is also present in Es and can be chosen as the critical path for X and Y.

Case 3: Both X and Y are competing, at least one of X or Y is not an intrinsic non-loop operation, and Y is not an exit read from a synchronization loop construct.

Proof:

If there is an ordering path of type 2 from X to Y, then this case is proved. If there is no ordering path of type 2 from X to Y, then (by lemma 3) there cannot be a *rel* path from X to Y that begins and ends in a $\xrightarrow{po}$ arc. Let X' be the first operation after X by $\xrightarrow{po}$. Let O be the set of operations that are after X and have a *rel* path to Y in Es. X' cannot be in O. Consider a modification of the execution order of Es where the operations in O and Y are moved to just after X, retaining their relative order. Consider the resulting order until Y. This resulting order is still consistent with program order, and the last conflicting write before any read is still the same as with the original execution order. It follows that the resulting order until Y is the prefix of an execution order of some sequentially consistent execution where the operations until Y are labeled as in Es and are essential. There are two sub-cases depending on whether X and Y compete in this new execution.

Sub-case 3a: X and Y do not compete in the new execution.

There must be an ordering chain from X to Y in the new execution. The only ordering chain possible is of the type X $\xrightarrow{co}$ R1 $\xrightarrow{po}$ Y. There should be a chain of the type X $\xrightarrow{co}$ R2 $\xrightarrow{po}$ Y where X and R2 compete. Further, R2 cannot return the value of its own processor's write since X and Y are consecutive conflicting operations in Es. Therefore X $\xrightarrow{vco2}$ R2 in Es. The above chain, therefore, is a path of type 3. This path is also present in Es, and can be chosen as the critical path for X and Y.

Sub-case 3b: X and Y compete in the new execution.

Since X and Y are consecutive conflicting operations in Es and in the new execution, by lemma 1, it follows that X $\xrightarrow{vco1}$ Y in Es and the new execution. In this case, both X and Y are not non-loops. Therefore, the only way that X $\xrightarrow{vco1}$ Y is if X is a competing write and Y is a competing read. Since Y is not from a synchronization loop construct, it follows that Y must be an intrinsic non-loop read and must be labeled non-loop. But then X cannot be an intrinsic loop write and must also be labeled non-loop, a contradiction.

Case 4: Both X and Y are competing, at least one of X or Y is not an intrinsic non-loop operation, and Y is an exit read of a synchronization loop construct.

Proof:

Let Wc be the first conflicting write in Es (including initial hypothetical writes) from which there can be a critical path to Y corresponding to X and Y. Then the write Ws just before Wc writes an exit value for Y. There must be a Ws because otherwise a critical path is not required. Let Wu be any conflicting write in Es between Wc and X (including Wc and X). Let Y' be the first operation before Y by $\xrightarrow{po}$.

Sub-case 4a: There does not exist a *rel* path from any Wu to Y'.

Let O be the set of operations that are after Wc and have a *rel* path to Y' in Es. Consider a modification of the execution order of Es where the operations in O and Y' are moved to just before Wc, retaining their relative order. Next, if Wc is part of a read-modify-write, move the corresponding read to just before Wc. Consider the resulting order until before Wc. This resulting order is still consistent with program order, and the last conflicting write before any read is still the same as with the original execution order. (Note that O cannot consist of any write conflicting with Wc.) Call the above order the current order. The following sub-cases are possible.

Sub-case 4a1: Both Wc and Y are non-loop operations.

There must be an ordering path from Wc to Y that ends in a $\xrightarrow{po}$ arc in Es (otherwise, a critical path is not required). Therefore, there must be an ordering path from Wc to Y that ends in a $\xrightarrow{po}$ arc and where the $\xrightarrow{co}$ arcs are race paths and no two adjacent arcs are both $\xrightarrow{po}$. Then the $\xrightarrow{co}$ arcs are $\xrightarrow{vcol}$ arcs by lemma 1. Consider the shortest path such that it ends in a $\xrightarrow{po}$ arc, the $\xrightarrow{co}$ arcs are $\xrightarrow{vcol}$, and no two adjacent arcs are both $\xrightarrow{po}$. This path is of type 1 and can be chosen as critical.

Sub-case 4a2: At least one of Wc or Y is a loop operation and Y is not part of a read-modify-write.

Modify the current order so that Y is just before Wc. Make Y return the value of the last conflicting write before it (i.e., the value of Ws). The resulting order until and including Wc is the prefix of an execution order of some sequentially consistent execution where the operations until and including Wc are labeled similar to Es, and are essential in Es. In the new execution, Y competes with Wc, and Wc does not make Y's loop terminate. Therefore, Y must be a non-loop read. Therefore, Wc must be a non-loop write. Thus both Wc and Y are non-loops, a contradiction.

Sub-case 4a3: At least one of Wc or Y is a loop operation and Wc is not part of a read-modify-write.

Modify the current order so that Y and the write of Y's read-modify-write (if any) are just before Wc. Make Y return the value of the last conflicting write before it (i.e., the value of Ws). The resulting order until and including Wc is the prefix of an execution order of some sequentially consistent execution where the operations until and including Wc are labeled similar to Es, and are essential in Es. In the new execution, Y competes with Wc, and Wc does not make Y's loop terminate. Therefore, Y must be a non-loop read. Therefore, Wc must be a non-loop write. Thus both Wc and Y are non-loops, a contradiction.

Sub-case 4a4: At least one of Wc or Y is a loop operation, both Y and Wc are parts of read-modify-writes, Wc is from a synchronization loop construct, and the write of Y's read-modify-write does not write an exit read value for Wc's loop.

Y must be an intrinsic competing operation. If Wc does not write an exit value for Y, then it follows that Wc is not the first write from which a critical path to Y is allowed (by definition of a critical set). Therefore Wc must write an exit value for Y. Modify the current order so that Y is just after Wc. Make Y return the value of Wc. The resulting order until and including Y is the prefix of an execution order of some sequentially consistent execution where the operations until and including Y are labeled similar to Es, and are essential in Es. Y competes with Wc in the new execution. Further, Wc and Ws both write exit values of Y. Therefore, Y is a non-loop read and so Wc is a non-loop write also, a contradiction.

Sub-case 4a5: At least one of Wc or Y is a loop operation, both Y and Wc are parts of read-modify-writes, and either Wc is not from a synchronization loop construct or the write of Y's read-modify-write writes an exit read value for Wc's loop.

Modify the current order so that Y and the write of Y's read-modify-write are just before Wc and the read of Wc's read-modify-write. Make Y return the

value of the last conflicting write before it (i.e., the value of Ws). Make the read of Wc's read-modify-write return the value of Y's read-modify-write. The resulting order until and including Wc is the prefix of an execution order of some sequentially consistent execution where the operations until and including Wc are labeled similar to Es, and are essential in Es. The read of Wc's read-modify-write competes with the write of Y's read-modify-write in the new execution. The above write and Ws both write exit values of the read of Wc's read-modify-write. Therefore, the read of Wc's read-modify-write must be non-loop. Therefore, the write of Y's read-modify-write must be non-loop. There must be an ordering path from Wc to Y that ends in a $\xrightarrow{po}$ arc in Es. Therefore, there must be an ordering path from Wc to Y that ends in a $\xrightarrow{po}$ arc and where the $\xrightarrow{co}$ arcs are race paths and no two adjacent arcs are both $\xrightarrow{po}$. Then the $\xrightarrow{co}$ arcs are $\xrightarrow{vco1}$ arcs by lemma 1. Consider the shortest path such that it ends in a $\xrightarrow{po}$ arc, the $\xrightarrow{co}$ arcs are $\xrightarrow{vco1}$, and no two adjacent arcs are both $\xrightarrow{po}$. This path is of type 4 and can be chosen as the critical path.

Sub-case 4b: There exists a *rel* path from some Wu to Y'.

Consider the last such Wu. If there is a path of type 2 from Wu to Y, then this case is proved. Therefore, assume there is no path of type 2 from Wu to Y. Let Wu' be the first operation after Wu by program order in Es. There cannot be a *rel* path from Wu' to Y' in Es because this implies a path of type 2 from Wu to Y in Es (by lemma 3). Therefore, every *rel* path from Wu to Y' must begin with a $\xrightarrow{co}$. Therefore, Wu is a competing operation (by lemma 2).

Let O be the set of operations that are after Wu and have a *rel* path to Y' in Es. Consider a modification of the execution order of Es where the operations in O and Y' are moved to just after Wu, retaining their relative order. The resulting order until Y' is still consistent with program order, and the last conflicting write before any read is still the same as with the original execution order. Thus, it represents the prefix of the execution order of an SC execution where all operations until Y' and Y are labeled similar to Es and all operations until Y' are essential. Now extend the resulting order to represent an SC execution where Y is also essential and Y returns the value of the first write after Wu that writes its exit value. There are two sub-cases.

Sub-case 4b1: Wu and Y are both non-loop.

There is an ordering path from Wu to Y that ends in a $\xrightarrow{po}$ arc in Es. Therefore, there must be an ordering path from Wu to Y that ends in a $\xrightarrow{po}$ arc and where the $\xrightarrow{co}$ arcs are race paths and no two adjacent arcs are both $\xrightarrow{po}$. Then the $\xrightarrow{co}$ arcs are $\xrightarrow{vco1}$ arcs by lemma 1. Consider the shortest path such that it ends in a $\xrightarrow{po}$ arc, the $\xrightarrow{co}$ arcs are $\xrightarrow{vco1}$, and no two adjacent arcs are both $\xrightarrow{po}$. This path is of type 1 and can be chosen as the critical path.

Sub-case 4b2: At least one of Y or Wu is intrinsic loop.

There is a *rel* path from Wu to Y' in the new execution as well. So one ordering chain from Wu to Y must end in a $\xrightarrow{po}$ arc. This chain cannot begin with a $\xrightarrow{po}$ arc. So it must be of the type Wu $\xrightarrow{co}$ R $\xrightarrow{po}$ Y where Wu and R are competing and Wu is the last conflicting write before R. This is a path of type 3. It is also present in Es and qualifies as a critical path corresponding to X and Y in Es.

Thus, for all cases, there is always a critical path of the type specified in Theorem 1 and so Theorem 1 is correct.

D.3: Proof of Correctness for the System Conditions for PLpc

We now show that the conditions in Figure 2 are sufficient for obeying the PLpc model. As discussed in Section D.1, the conditions in Figure 4 are strictly weaker (i.e., more aggressive) than those in Figure 2. Therefore, it is

sufficient to show that the conditions in Figure 4 obey Condition D.1 for the derived valid paths for PLpc.

By inspection, it follows that the specification in Figure 4 upholds the sufficient valid paths derived for PLpc. Thus, the valid path clause of Condition D.1 is satisfied.

Condition D.1(1) is satisfied as follows. Condition D.1(1) depends on the control relation. As discussed in Section D.1, the reach relation ($\xrightarrow{rch'}$) of Appendix B obeys the properties of the control relation. Therefore, the reach part of the $\xrightarrow{sxo}$ relation in figure 4 ensures that Condition D.1(1) is satisfied.

Condition D.1(2) is satisfied because the specification in Figure 4 obeys the aggressive form of the uniprocessor correctness condition in Appendix C; the infinite execution condition in Appendix C is exactly the same as the finite speculation condition.

Condition D.1(3) is satisfied because of the following. This condition requires the existence of all sub-operations of writes that can be involved in race paths and writes whose value is read by exit reads of self-ordered loops. From the derivation in the previous section, such writes are competing writes. The termination condition in Figure 4 ensures that all sub-operations of these writes are in the execution.

Finally, the specification in Figure 4 obeys the Condition D.1(4) trivially due to the atomic read-modify-write part of the $\xrightarrow{sxo}$ relation in Figure 4.

Appendix E: Proof of Correctness of Conditions for Porting PLpc

This appendix provides the correctness proofs that show PLpc programs that are ported to various hardware-centric models as prescribed in Section 4 satisfy the PLpc model (i.e., executions of such programs will be sequentially consistent). We use the system requirements specified in Section 3 as the requirements to be satisfied by a PLpc system. To determine whether these are satisfied, we need specifications for the hardware-centric models as well. We chose to use the specifications developed in [GAG+93] because they are expressed using the same specification framework used in Section 3.[9] The uniform framework across the specifications simplifies the task of proving the hardware-centric implementations satisfy the PLpc system requirements.

Figures 5-10 show the specifications for TSO, PSO, WO, RCsc, PC, and RCpc.[10] Figure 11 shows the specification of PC augmented with a fence and ability to specify certain writes as atomic. This provides an example of how the models may be augmented as was suggested in Section 4 to avoid the need for mapping Rnl's to RMW's. Below, we show that if programs are ported as specified in Section 4, the constraints imposed by each of these models strictly satisfies the sufficient constraints for PLpc.

5.1 Porting to TSO

We start by comparing the specification for TSO (Figure 5) with that of PLpc (Figure 2). Our goal is to show that the conditions for TSO strictly satisfy those of PLpc. Consider the constraints on $\xrightarrow{xo}$ first. Both specifications obey the value condition. For the termination condition, the TSO specification is stricter since all sub-operations are specified as opposed to only competing sub-operations. Finally, the manner in which the $\xrightarrow{sxo}$ relation constrains $\xrightarrow{xo}$ is the same.

We now consider the various components in the $\xrightarrow{sxo}$ relation. The uniprocessor dependence and atomic read-modify-write conditions are identical. The coherence condition of TSO strictly satisfies that of PLpc. The reach condition of PLpc is also strictly satisfied by the second component in the multiprocessor dependence chain of TSO.

Now consider the multiprocessor dependence chains under $\xrightarrow{sxo}$; we identify the chains specified under this category as 1st chain, 2nd chain, and so on. These chains are composed of the $\xrightarrow{spo}$ and $\xrightarrow{sco}$ relations. The $\xrightarrow{sco}$ relation of TSO is a superset of the PLpc $\xrightarrow{sco}$ relation. Same is true for the $\xrightarrow{spo}$ relation except TSO is missing Wnl $\xrightarrow{po}$ Rnl. Given the above, the first chain of PLpc is strictly satisfied by the combination of the 1st chain in TSO and TSO's coherence and uniprocessor dependence conditions. Below, we consider the 2nd, 3rd, and 4th chains of PLpc given the various mapping options (from Section 4).

First consider the mapping where the Wnl in every Wnl $\xrightarrow{po}$ Rnl is a RMW. In this case, the $\xrightarrow{spo}$ from Wnl to Rnl is defined in TSO (by third case under $\xrightarrow{spo}$) and the 2nd and 3rd chains of TSO strictly satisfy the 2nd, 3rd, and 4th chains in PLpc.

Now consider the mapping where the Rnl in every Wnl $\xrightarrow{po}$ Rnl is a RMW. Therefore, we have Wnl $\xrightarrow{po}$ RMW. In TSO, W $\xrightarrow{po}$ W constitutes an $\xrightarrow{spo}$. Let AR be the read and AW be the write in RMW. Therefore, Wnl $\xrightarrow{spo}$ AW. Consider the case when Wnl $\xrightarrow{po}$ Rnl is in the beginning or middle of the 2nd, 3rd, or 4th chains in PLpc. The Wnl $\xrightarrow{po}$ Rnl can participate in the chain only if the next access is a Wnl' such that Rnl $\xrightarrow{co'}$ Wnl'. Since Rnl is part of a RMW, the atomic read-modify-write condition requires AW $\xrightarrow{co}$ Wnl' if Rnl $\xrightarrow{co}$ Wnl'. In addition, AW $\xrightarrow{co}$ Wnl' constitutes an $\xrightarrow{sco}$ in TSO. Therefore, the chain is upheld through Wnl $\xrightarrow{spo}$ AW $\xrightarrow{sco}$ Wnl'. Now consider the case when Wnl $\xrightarrow{po}$ Rnl is at the end of the 2nd, 3rd, or 4th chains in PLpc. Therefore, the chain begins with a W to the same location as Rnl. We know W $\xrightarrow{sxo}$ AW in TSO if the chain is in the form of the 2nd, 3rd, or 4th chains in PLpc. Again, because of the atomic read-modify-write condition, this translates to W $\xrightarrow{sxo}$ AR. Therefore, the $\xrightarrow{sxo}$ relation is maintained in TSO for this case as well.

It is relatively simple to argue that the case when some Wnl's are transformed to RMW's and some Rnl's are

[9] Proofs that these specifications are equivalent to the original specifications of the hardware-centric models is presented in [GAG +93].

[10] The reach relation for WO, RCsc, and RCpc and the infinite execution condition for PC, TSO, PSO, WO, RCsc, and RCpc are specified in [GAG+93] and differ from the corresponding relation (Appendix B) and condition (Appendix C) for PLpc. The reach relation for the hardware-centric models trivially obeys the reach relation for PLpc with the given mappings. Similarly, the infinite execution condition is stricter for the hardware-centric models and therefore obeys the condition for PLpc.

transformed to RMW's for every Wnl $\xrightarrow{po}$ Rnl satisfies the PLpc requirements also.

Now consider the alternative mapping where TSO is augmented with a fence that delays reads for previous writes. The fence make Wnl $\xrightarrow{spo}$ Rnl to hold under TSO and the argument that this satisfies PLpc is similar to the case above where the Wnl in every Wnl $\xrightarrow{po}$ Rnl is mapped to a RMW.

5.2 Porting to PSO

The difference between the PSO and TSO proofs is that W $\xrightarrow{po}$ Wc does not constitute an $\xrightarrow{spo}$ under PSO (see Figure 6). However, our mapping requires that every Wc be preceded with a STBAR. Then, by the second case under $\xrightarrow{spo}$ in PSO, we have W $\xrightarrow{spo}$ Wc. Given this, the proof of TSO can be used for PSO.

5.3 Porting to WO

Given that every Rc and Wc is mapped into a synchronization operation (Rs or Ws) for WO, it is relatively simple to show that the conditions in PLpc are strictly satisfied by WO (Figure 7). Note that the reach condition component of $\xrightarrow{sxo}$ under PLpc is strictly satisfied by the 2nd chain under multiprocessor dependence chain of WO (since $\xrightarrow{rch}$ in WO is included in its $\xrightarrow{spo}$ definition).

5.4 Porting to RCsc

Figure 8 shows the conditions for RCsc. Similar to WO, given that every Rc is mapped into an acquire (Rc_acq) and every Wc is mapped into a release (Wc_rel), it is relatively simple to show that the conditions in PLpc are strictly satisfied by RCsc.

5.5 Porting to PC

The specification for PC is given in Figure 9. There are two main issues to consider for satisfying the PLpc conditions. Similar to TSO, PC does not have an $\xrightarrow{spo}$ defined for Wnl $\xrightarrow{po}$ Rnl. In addition, PC's $\xrightarrow{sco}$ does not include Rnl $\xrightarrow{co'}$ Wnl $\xrightarrow{co'}$ Rc as a component. Finally, there is no chain in the multiprocessor dependence chains for PC that starts with a W $\xrightarrow{sco}$ R.

The mapping for PC requires every Rnl to be a RMW. Let AR be the read and AW be the write in RMW. We first discuss how this mapping alleviates the lack of Rnl $\xrightarrow{co}$ Wnl $\xrightarrow{co}$ Rc in the $\xrightarrow{sco}$ definition of PC. Consider X $\xrightarrow{po}$ R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2, where R1 is Rnl which is part of a RMW. Given R1 $\xrightarrow{co}$ W and the atomic read-modify-write condition, we know AW $\xrightarrow{co}$ W and therefore, AW $\xrightarrow{co}$ R2. Also, for any X, X $\xrightarrow{po}$ AW defines an $\xrightarrow{spo}$. Therefore, X $\xrightarrow{po}$ R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 implies X $\xrightarrow{spo}$ AW $\xrightarrow{co}$ R2, which can replace X $\xrightarrow{po}$ R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 in the chain.

The argument for lack of $\xrightarrow{spo}$ between Wnl $\xrightarrow{po}$ Rnl is similar to the argument presented for TSO where every Rnl is mapped to a RMW.

Finally, we argue below that the lack of a multiprocessor chain in PC, that starts with a W $\xrightarrow{sco}$ R, is not an issue. In PLpc, this chain ends with an Rnl, which we know is mapped to a RMW for PC. Assume a chain of the form W1 $\xrightarrow{sco}$ R1 ... R2, where R2 is the Rnl (which is a RMW). The second multiprocessor dependence chain in PC implies R1 $\xrightarrow{sxo}$ AW (of the RMW). Therefore, we know W1(i) $\xrightarrow{xo}$ AW(i) for all i. By the atomic read-modify-write condition, we know W1(i) $\xrightarrow{xo}$ AR(i) for all i also. Therefore, the order which would be maintained by the 4th multiprocessor dependence chain in PLpc is strictly maintained in PC.

We now consider the augmented PC model discussed in Section 4 with a fence to delay future reads for previous writes and an atomic read-modify-write. The specification of this augmented PC is shown in Figure 11. The mapping conditions require a fence between every Wnl $\xrightarrow{po}$ Rnl pair and require all Wnl's to be mapped to W_atomic. Given this, it is relatively simple to see that the conditions in Figure 11 strictly satisfy the PLpc conditions.

5.6 Porting to RCpc

The proof for RCpc is quite similar to that of PC given that every Rc is mapped to an acquire and every Wc is mapped into a release.

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

R $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ W
AW (in RMW) $\xrightarrow{po}$ R
W $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y
R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
  W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R
  RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW
  W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ R
atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.
$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.
Termination condition holds for all sub-operations.

Figure 5: Specification of TSO.

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

R $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ W
AW (in RMW) $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y
R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
  W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R
  RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW
  W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ R
atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.
$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.
Termination condition holds for all sub-operations.

Figure 6: Specification of PSO.

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

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

$RWs \xrightarrow{po} RWs$

$RW \xrightarrow{po} RWs$

$RWs \xrightarrow{po} RW$

$X \xrightarrow{spo'} Y$ if $X \xrightarrow{spo''} A \; \{\xrightarrow{rch} \mid \xrightarrow{spo''}\}^* \; Y$

$X \xrightarrow{spo} Y$ if $X \; \{\xrightarrow{rch} \mid \xrightarrow{spo'}\}+ \; Y$

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

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

$X \xrightarrow{co} Y$

$R1 \xrightarrow{co} W \xrightarrow{co} R2$ where R1,R2 are on the same processor

$X \xrightarrow{sco'} Y$ if X,Y are the first and last operations in $R1 \xrightarrow{co} W \xrightarrow{co} R2$ where R1,R2 are on different processors

define $\xrightarrow{sxo}$: $X \xrightarrow{sxo} Y$ if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: $RW \xrightarrow{po} W$

coherence: $W \xrightarrow{co} W$

multiprocessor dependence chain: one of

$W \xrightarrow{co} R \xrightarrow{spo} R$

$RW \xrightarrow{spo} \{(A \xrightarrow{sco} B \xrightarrow{spo}) \mid (A \xrightarrow{sco'} B \xrightarrow{spo'})\}+ \; RW$

$W \xrightarrow{sco} R \xrightarrow{spo'} \{(A \xrightarrow{sco} B \xrightarrow{spo}) \mid (A \xrightarrow{sco'} B \xrightarrow{spo'})\}+ \; R$

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either $W \xrightarrow{sxo} AR$ or $AW \xrightarrow{sxo} W$

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if $X \xrightarrow{sxo} Y$, then $X(i) \xrightarrow{xo} Y(i)$ for all i.

Termination condition holds for all sub-operations.

Figure 7: Specification of WO.

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

X $\xrightarrow{spo'''}$ Y if X and Y are *competing* and Xc $\xrightarrow{po}$ Yc

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

RW $\xrightarrow{po}$ Wc_rel

Rc_acq $\xrightarrow{po}$ RW

X $\xrightarrow{spo'}$ Y if X $\xrightarrow{spo'''}$ A $\{\xrightarrow{rch} \mid \xrightarrow{spo''} \mid \xrightarrow{spo'''}\}*$ Y

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} \mid \xrightarrow{spo''} \mid \xrightarrow{spo'''}\}+$ Y

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

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

X $\xrightarrow{co}$ Y

R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on the same processor

X $\xrightarrow{sco'}$ Y if X,Y are the first and last operations in R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on different processors

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R

RW $\xrightarrow{spo}$ $\{(A \xrightarrow{sco} B \xrightarrow{spo}) \mid (A \xrightarrow{sco'} B \xrightarrow{spo'})\}+$ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo'}$ $\{(A \xrightarrow{sco} B \xrightarrow{spo}) \mid (A \xrightarrow{sco'} B \xrightarrow{spo'})\}+$ R

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Termination condition holds for all *competing* sub-operations.

Figure 8: Specification of RCsc.

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

R $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ W

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X $\xrightarrow{co}$ Y

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R
RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW
atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.
$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.
Termination condition holds for all sub-operations.

Figure 9: Specification of PC.

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

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of
Rc $\xrightarrow{po}$ RWc
Wc $\xrightarrow{po}$ Wc
RW $\xrightarrow{po}$ Wc_rel
Rc_acq $\xrightarrow{po}$ RW
X $\xrightarrow{spo}$ Y if X {$\xrightarrow{rch}$ | $\xrightarrow{spo'}$ }+ Y

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y
R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on the same processor

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R
RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW
atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.
$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.
Termination condition holds for all *competing* sub-operations.

Figure 10: Specification of RCpc.

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

R $\xrightarrow{po}$ RW

W $\xrightarrow{po}$ W

W $\xrightarrow{po}$ Fence $\xrightarrow{po}$ R

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R $\xrightarrow{co}$ W_atomic $\xrightarrow{co}$ R

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R

RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW

W_atomic $\xrightarrow{sco}$ R $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ R

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$**:**

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Termination condition holds for all sub-operations.

Figure 11: Specification of PC augmented with a special fence and atomic writes.