Master Thesis

Operational Characterization of Weak Memory Consistency Models

Maximilian Senftleben

March 26, 2013

Department of Computer Science,
University of Kaiserslautern,
D 67653 Kaiserslautern,
Germany

Examiner: Prof. Dr. Klaus Schneider
Dipl.-Technoinform. Daniel Baudisch

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Operational Characterization of Weak Memory Consistency Models" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 26. März 2013

____
Maximilian Senftleben

Abstract

This thesis deals with the most common memory consistency models and provides comparable definitions for some of them. The relationship between these models is analysed and the difference between them explained by providing characterising examples. The primary focus of this thesis is to provide operational semantics for the most relevant consistency models. This is achieved by implementing machines which are based on the consistency model defintions using the synchronous programming language Quartz. The provided implementations are constructed to conform with the corresponding definitions in terms of correctness and completeness. The latter property, completeness, is desirable for further work which may aim on reducing more sophisticated implementations on the constructed ones to show their correctness.

Zusammenfassung

Diese Arbeit behandelt die meistgebräuchlichsten Speicherkonsistenzmodelle und gibt vergleichbare Definition für einige dieser. Die Beziehungen der Modell wird analysisert und derren Unterschiede mittels charakterisierender Beispiele erklärt. Der primäre Fokus der Arbeit ist die Bereitstellung von ausführbarer Semantik für die relevantesten Konsistenzmodelle. Dies wird erreicht durch die Implementierung von Maschinen, welche auf den Definitionen der Speichermodelle basieren, unter der Verwendung der synchronen Programmiersprache Quartz. Die vorgestellten Implementierungen wurden sowohl hinsichtlich Korrektheit als auch Vollständigkeit anhand der zugehörigen Definitionen erstellt. Die Vollständigkeitseigenschaft ist interessant um in möglicher künftiger Arbeit den Versuch unternehmen zu können ausgeklügeltere Implementierung auf eine der vorgestellten zurückzuführen um deren Korrektheit zu zeigen.

Contents

1. Introduction **1**

2. Related Work **5**

3. Consistency Models **7**
3.1. Local Consistency (LC) 8
3.2. Slow Consistency 8
3.3. Pipelined-RAM (PRAM) Consistency / Global Process Order (GPO) 10
3.4. PRAM-M Consistency 11
3.5. Cache Consistency (CC) / Global Data Order (GDO) 12
3.6. Global Write-read-write Order (GWO) 13
3.7. Global Anti-Order (GAO) 14
3.8. Causal Consistency / GPO+GWO 15
3.9. Processor Consistency by Goodman (PC-G) 15
3.10. GPO + GDO Consistency 17
3.11. Processor Consistency by DASH (PC-D) 18
3.12. Partial Store Ordering (PSO) 19
3.13. Total Store Ordering (TSO) 20
3.14. Sequential Consistency (SC) 21
3.15. Overview/Relationship 23

4. Reference Machines **31**
4.1. Common Structural Elements 31
4.2. Local Consistency Reference Machine 33
4.3. Slow Consistency Reference Machine 35
4.4. PRAM Consistency Reference Machine 36
4.5. Cache Consistency Reference Machine 38
4.6. Causal Consistency Reference Machine 40
4.7. Processor (PC-G) Consistency Reference Machine 43
4.8. PSO Consistency Reference Machine 45
4.9. TSO Consistency Reference Machine 45
4.10. Sequential Consistency Reference Machine 46

5. Implementations **49**
5.1. Environment 49
5.2. Interface 49
5.3. Non-Determinism 49
5.4. (Un)Bounded Buffer 49
5.5. Improvements 50

6. Conclusions and Further Work 51

A. Quartz Implementations 53

A.1. Remarks . . . 53
A.2. Shared modules . . . 53
A.3. RefLocal : Local Consistency Reference Machine . . . 60
A.4. RefSlow : Slow Consistency Reference Machine . . . 62
A.5. RefPRAM : PRAM Consistency Reference Machine . . . 64
A.6. RefCache : Cache Consistency Reference Machine . . . 66
A.7. RefCausal : Causal Consistency Reference Machine . . . 67
A.8. RefProcessor : PC-G Consistency Reference Machine . . . 70
A.9. RefPSO : PSO Consistency Reference Machine . . . 72
A.10.RefTSO : TSO Consistency Reference Machine . . . 74
A.11.RefSequential : Sequential Consistency Reference Machine . . . 76

Bibliography 79

1. Introduction

With multiprocessor systems, the historical assumption of a sequential consistent memory system becomes more and more a bottleneck for memory latency. While some programs rely on sequential consistency, many programs could execute properly with weaker consistency constraints and therefore justify the research of such models.

Historically, a computer architecture was considered to consist of a single processing unit and a single memory connected via a single bus (Von Neumann architecture 1945). The single bus enforces that each memory read operation returns the value most recently written to that location. Even if the processor alternatingly executes multiple processes, the memory operations take place one after the other and form a sequence.

Nowadays a computer architecture may consist of multiple processors which share a common main memory. Early multiprocessor systems connected multiple processors via a single bus to a shared memory such that processors had to compete for bus access. Therefore the memory operations of all processes formed a sequence as well. Newer multiprocessor systems became more complex as the individual processors started to cache memory operations or otherwise tried to increase their performance. In distributed systems, the assumption of a single memory is given up in favor of multiple memories which are shared between all processors as so called distributed shared memory.

Multiple processes interact via shared memory to synchronize with each other. For instance, if two processes share a resource which only may be accessed by one process at a time, then it must be ensured that one process does not use the resource if it is already in use by the other one. As an example, the following simple program tries to solve that problem:

```
bool p1_in_cs, p2_in_cs;

process P1 {                          process P2 {
  if(p2_in_cs == false) {               if(p1_in_cs == false) {
    p1_in_cs = true;                      p2_in_cs = true;
    critical_operations;                  critical_operations;
    p1_in_cs = false;                     p2_in_cs = false;
  }                                     }
}                                     }
```

The problem that may occur is that process P1 reads variable $p2_in_cs = false$, then P2 reads variable $p1_in_cs = false$ and then both enter the critical section, both assuming that they are the only one.

This problem is called the mutual exclusion problem and was first correctly solved by T.J. Dekker and presented by E.W. Dijkstra in 1965 [Dijk68]. The resulting algorithm, known as Dekker's algorithm, is as follows:

```
// initially false
bool flag1,flag2,turn;

process P1 {
  flag1 = true;
  while(flag2 == true) {
    if(turn != false) {
      flag1 = false;
      while(turn != false) {}
      flag1 = true;
    }
  }

      critical_operations;

      turn = true;
      flag0 = false;

      noncritical_operations;
}
```

```
process P2 {
  flag2 = true;
  while(flag1 == true) {
    if(turn != true) {
      flag2 = false;
      while(turn != true) {}
      flag2 = true;
    }
  }

      critical_operations;

      turn = false;
      flag1 = false;

      noncritical_operations;
}
```

This algorithm enables a process to detect when an other process has the intention to enter or currently resides in the critical section. If both processes did not enter the critical section yet then the algorithm determines which process may proceed and which one has to wait using variable *turn*.

Dekker's algorithm works for multi-threaded uniprocessor and the mentioned single-bus-based multiprocessor systems, but it may fail on other multiprocessor systems, e.g. systems with processors that buffer memory operations in write-back caches, if no further constraints for the memory systems are given. In such a system, both process P1 and P2 may set their flag variable to true but both keep the write operation in their cache and read the value of the other's flag variable from main memory as false. As a consequence both processes enter the critical section. Dekker's algorithm failed for the considered multiprocessor system.

There are two ways to handle this problem:

- Either the multiprocessor system's memory management must be modified to behave equivalent to a multi-threaded uniprocessor. This behaviour is called sequential consistency and if a multiprocessor system provides sequential consistent behaviour then Dekker's algorithm works as intended. A technique to enable multiprocessor systems which use write-back caches to provide sequential consistent behaviour is the so called MESI protocol [PaPa98]. MESI is named after the four states each cache line can have: Modified, Exclusive, Shared, and Invalid. Variations of MESI are used in many modern CPUs, e.g. Intel's Pentium 4 [IA313].
- An alternative way to handle the problem is to accept that there may exist multiprocessor systems with non-sequential memory behaviour. To utilise these systems despite their lack of sequential consistency, each system has to define its memory behaviour. These definitions can act as some sort of interface that guarantees programmers the cor-

rect execution of their programs. Such interfaces are called memory consistency models and determine which relaxations or optimizations are possible and in which way the system behaves different to a sequential consistent one.

A memory consistency model can be defined in different ways. It can be defined in an operational way by providing a machine structure and related interconnection rules or by revealing an implementation of such a system. For instance, Lipton and Sandberg [LiSa88] provided an implementation for PRAM by defining its structure and communication rules. The consistency model of such a machine is defined as the set of executions it may produce for given programs.

A method of defining a consistency model indirectly is to define a set of rules for programmers to ensure sequential consistency, for example with Sequential Consistency Normal Form (SCNF) by Adve [Adve93]. An implementation of a memory system complies with the consistency model if each program satisfying the given rules is guaranteed to behave like a sequential consistent system. This type of definition has the benefit that if a given system can be optimized in a way that programs which respect the definition's rules still behave sequential consistent, then the optimized version can replace the prior version without requiring the programmer to make adjustments.

Another way of defining a consistency model is to define which ordering of memory operations the memory and processes are allowed to observe. One possibility is to define the model by providing rules which must be satisfied by the ordering of all operations as observed by the main memory. For example this is done for the SPARC memory models TSO and PSO in an axiomatic way. This is done by providing 6 axioms (TSO) respectively 7 axioms (PSO) which if satisfied by an ordering on all memory operations imply a consistent execution. Yet another possibility is a view-based definition which defines rules on the ordering of each process' own operations and others' write operations. Such definitions (e.g. given by Steinke and Nutt in [StNu04]) allow a simple comparison of different memory models and abstract from the inner structure of memory systems.

Weak memory models emerged from the endeavours to optimize the latency of memory systems. Improvements weakened the underlying sequential model in a way that some inconsistent states became reachable. These improvements and new concepts became new weak memory models. To use the benefits of the new models they must be defined in a way that programmers can write or adjust their programs such that potential inconsistent states of the memory system are avoided and the executions produce the intended results.

In this thesis, some known consistency models described in the literature [ABHN91, BaBe97, Good91, HuAh90, Lamp79, LiSa88, SPAR91] are characterized in an operational manner by specifying reference machines. These reference machines are obtained by deriving memory system implementations directly from the consistency model definitions. The resulting reference machines do not aim to be efficient but try to be as minimalistic as possible (in terms of different components and structures, not necessarily size) to simplify verification of correctness and completeness of the implementation.

Apart from the benefit of such easy-to-understand definitions of different memory consistency models for educational purposes, such reference machines may be the basis of attempts to classify or verify the consistency class of an arbitrary memory system.

2. Related Work

Most parts of this thesis are based on the Unified Theory of Shared Memory Consistency by Steinke and Nutt [StNu04] which offers view-based definitions for many common memory consistency models. Furthermore it reveals underlying properties which suffice to describe the presented models.

Similar to this work Higham, Kawash and Verwaal [HiKV98] provided definitions and comparisons of some consistency models and defined machines for the models. Their implementations lack some details (e.g. "switch" functionality) and the work leaves some questions unanswered (e.g. sizes of FIFOs).

[BaBe97] characterizes different consistency models in a view based way for non-synchronized ("pure") memory models and presents formal definitions for some synchronized ("hybrid") memory models as well. Ahamad et al. [ABJK+93] compares different view-based definitions of processor consistency by Goodman and the DASH system.

Sindhu, Frailong and Cekleov [SiFC92] present a formal framework for axiomatic definitions of memory consistency models and provide definitions for sequential, PSO and TSO consistency.

[Mosb93a] gives a brief overview of some weak memory models and discusses their influence on programming language and compiler design. Adve [Adve93] provides sequential consistency normal form, a programmer-centric approach to specify memory models. Adve and Hill [AdHi93] describe a shared-memory model that guarantees sequential consistency for data-race-free programs and which unifies four earlier models. Adve and Gharachorloo [AdGh96] give an overview over memory consistency related issues, describe existing and possible relaxations of a sequential consistency.

In [BoPe09] an approach to formalize a memory model as a part of weak operational semantics is presented and proved to work as intended for data-race free programs.

Loewenstein et al. [LoCM06] presents a technique to verify a memory model's (TSO) behaviour against its axiomatic definition. Linden and Wolper [LiWo11] describe an approach to memory fence insertion in programs to execute correctly under weak memory systems (TSO,PSO) based on verification techniques (finite state automata based verification tool). The TSOtool described by Hangal et al. [HVML+04] offers a test platform to check pseudo-randomly generated test programs run on a memory system against the TSO specification.

Owens et al. describe x86-TSO [OwSS09], a new model for x86 processors which is based on the SPARCv8 TSO model [SPAR91]. Sewell et al. [SSON+10] present and compare x86-TSO with recent Intel and AMD specifications and address data-race-freedom for x86-TSO.

3. Consistency Models

Consistency models can be defined by the set of executions they allow for a given set of processes.

In the following section, a formalism for specifying memory models is introduced, formal definitions of common memory consistency models are given, and their relationships are analysed.

To argue about memory models, a formalism is needed which is able to cover all aspects of a memory operation:

- access mode (read, write)
- issuer (process issuing the operation)
- location (memory address)
- value

The following "view"-based formalism is taken from Steinke and Nutt which itself is based on Ahamad et al.[ABJK+93] and Bataller and Bernabeu[BaBe97]: A "view"-based definition focuses on the view of a process with respect to the memory system. If a process can observe a behaviour which conflicts with the consistency properties, then the execution may be rejected. This kind of definition does not give any implementation details but only requires the system to provide a view for each connected process which satisfies the consistency properties.

Definition 3.1 (**Shared Memory Formalism**). [StNu04, Appendix A]
An **execution** is a set of processes P, a set of shared variables V, a set of operations O, and two partial orders on O: process order $<_{PO}$, and writes-to order $\mapsto$.
An **operation** is a tuple (op,i,x,v) where $op \in \{r, w\}$ (read,write), $i \in P$, $x \in V$, and v is a valid value for x. The operation $(w, \epsilon, x, \perp)$ is called the **initial write** of x.

Several functions are defined on the operation tuple: $type(o)$ returns the type of operation o, $var(o)$ returns the variable of operation o, $val(o)$ returns the value of an operation o and $proc(o)$ returns the process of operation o.

Local order for process i $<_i$ is a relation such that
$\left(\forall_{o_j,o_k \in (r,*,*,*)} (o_j <_i o_k \oplus o_k <_i o_j)\right) \bigwedge \left(\forall_{x \in V, o \in (*,i,*,*)} (w, \epsilon, x, \perp) <_i o\right)$.
Process order $<_{PO}$ is a relation such that $<_{PO} = \bigcup_{i \in P} <_i$.
Writes-to order $\mapsto$ is a relation such that $\forall_{o \in (r,*,*,*)} \exists_{i \in P} \left[(w, i, var(o), val(o)) \mapsto o\right]$.

A relation $<_{serial}$ is a **serial view** $SerialView\,(< | O')$ on operations $O' \subseteq O$ which respects $<$ iff
$< \subseteq <_{serial} \wedge \forall_{o_1,o_2 \in O'} \,(o_1 <_{serial} o_2 \oplus o_2 <_{serial} o_1)$ and
$\forall_{r \in O'} \exists_{w \in O'} \,(w \mapsto r \wedge w <_{serial} r \wedge \nexists_{w_2 \in O'} \,(var(w_2) = var(r) \wedge w <_{serial} w_2 <_{serial} r))$

In this formalism, an execution represents the result and context of multiple processes (P) that issue operations (O) on shared variables (V) to a memory system. Each operation is assigned to a single process and is either a read from a given memory address or a write with a target address and value. The formalism defines three orders: 1) the local order ($<_i$) defines the sequence in which operations of a single process (i) are issued to the memory system, 2) the process order is the union of all processes' local orders, and 3) the writes-to order orders each read operation after a write operation which writes the value to the memory the read operation reads. Important for the following definitions is the notion of a serial view. A serial view (SerialView) on an execution is a total order (view) on a subset of the execution's operations which represents the order in which the process sees the memory operations. A read in a serial view must read from the most recent preceding write to the same variable (serial).

3.1. Local Consistency (LC)

Local consistency was first defined by Heddaya and Sinha [HeSi92] as the weakest constraint that could be required of a shared memory system. It requires that each process observes all operations as if they were executed on a single processor. It enforces that a process' write can be observed by itself.

Local Consistency [HeSi92, BaBe97] is defined by Steinke and Nutt [StNu04, Theorem 3.8] as follows:

> **Definition 3.2.** An execution is **locally consistent** iff
>
> $$\forall_{i \in P} \exists SerialView\,(<_i \mid (*, i, *, *) \cup (w, *, *, *))$$

An execution is locally consistent iff each process observes operations in a serial view which orders its own operations in program order and other processes' write operations in an arbitrary order.

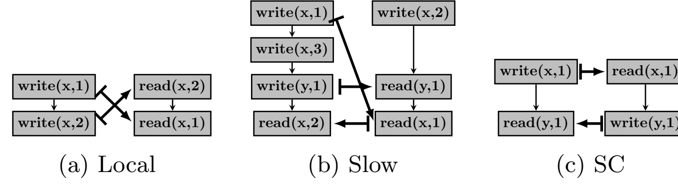
Example Figure 3.1a shows an execution which is locally consistent because of the existence of the following SerialViews:

[$P1$]: write(x,1), write(x,2)
[$P2$]: write(x,2), read(x,2), write(x,1), read(x,1)

Remark: The example is not slow consistent which is shown later.

3.2. Slow Consistency

Hutto and Ahamad [HuAh90] introduced slow consistency to eliminate consistency maintenance which leads to high latency. Having other weak memory models in mind they showed that even for their weaker model, slow consistency, techniques exist to solve the exclusion and dictionary problem. With those techniques, slow consistency may be used to increase



(a) Local (b) Slow (c) SC

Figure 3.1.: Examples for locally, slowly, and sequentially consistent executions

memory latency for many multiprocess programs relying on the mentioned problems. Slowly consistent memory allows writes to propagate slowly to other processes.

Slow consistency [HuAh90] is defined as follows: A read returns a previously written value, and successive reads to the same location may not return writes issued earlier (by the process that issued the earlier write) than the read one. Furthermore, local writes must be visible immediately. Slow consistency is formally defined in Steinke and Nutt's formalism for shared memory consistency models [StNu04, Theorem 3.7]:

Definition 3.3. An execution is **slowly consistent** iff

$$\forall_{i \in P, x \in V} \exists SerialView\,(<_{PO}|\;(*, i, x, *) \cup (w, *, x, *))$$

An algorithm which solves the n-process exclusion problem for slowly consistent systems was provided by Hutto and Ahamad [HuAh90] and is given in pseudo-code as follows:

```
// initially false
bool[n] req,ack;

process P_i {
  req[i] = true;
  while(ack[i] == false) {}
  critical_operations;
  ack[i] = false;
  noncritical_ops;
}
```

```
process MutexServer {
  while(true) {
    for(int i = 0 .. n-1) {
      if(req[i] == true) {
        req[i] = false;
        ack[i] = true;
        while(ack[i] == true) {}
      }
    }
  }
}
```

Example Figure 3.1b shows an execution which is slowly consistent because the following SerialViews exist:

$[P1, x]$: write(x,1), write(x,3), write(x,2), read(x,2) $[P1, y]$: write(y,1)
$[P2, x]$: write(x,2), write(x,1), read(x,1), write(x,3) $[P2, y]$: write(y,1), read(y,1)

Remark: The example is neither PRAM nor cache consistent which is shown later.

Figure 3.1a is not slowly consistent because the $\mapsto$ relation requires read(x,2) to happen after write(x,2) but read(x,1) to happen before write(x,2), which is not allowed due to $read(x, 2) <_{PO} read(x, 1)$.

3.3. Pipelined-RAM (PRAM) Consistency / Global Process Order (GPO)

One of the first common weak memory models described was PRAM (Pipelined RAM), which was presented 1988 by Lipton and Sandberg [LiSa88, LiSa94]. They show that their shared memory system PRAM scales better than sequentially consistent systems as it is immune to high network latency. Additionally, synchronization costs remain low while performance increases significantly.

Original PRAM [LiSa88] definition by Lipton and Sandberg [LiSa88]:

> **Definition 3.4.** Consider n processes $P_1, \ldots, P_n$ with a local memory $M_1, \ldots, M_n$ each. Process k executes a read from location i "by performing a normal read from location i" of its own memory M_k. Process k executes a write to location i with value v "by performing a local action and initializing a global action. Locally, it does a normal write to $[M_k]$ at location i with value v. Globally, it sends a message $< i, v >$ to all the other processors."

PRAM consistency (equivalent to GPO) based on the Steinke and Nutt [StNu04, Theorem 3.2, Definition 4.1] (which itself uses the definition of Ahamad et al. [ABJK+93]) is as follows:

> **Definition 3.5.** An execution is **PRAM consistent** / Global Process Order (**GPO**) iff
>
> $$\forall_{i \in P} \exists SerialView \, (<_{PO} |(*, i, *, *) \cup (w, *, *, *))$$

In a PRAM consistent execution, every process observes the writes of an other process in the order they were issued. But two different processes may see writes of several processes in a different order.

A system implementing PRAM consistency therefore only has to assure that the communication from one process to another does not reorder or lose writes, while the transmission delay is not critical.

Example Figure 3.2a shows an execution which is PRAM consistent:

[$P1$]: write(x,1), read(x,1), write(x,2) [$P2$]: write(x,2), read(x,2), write(x,1)

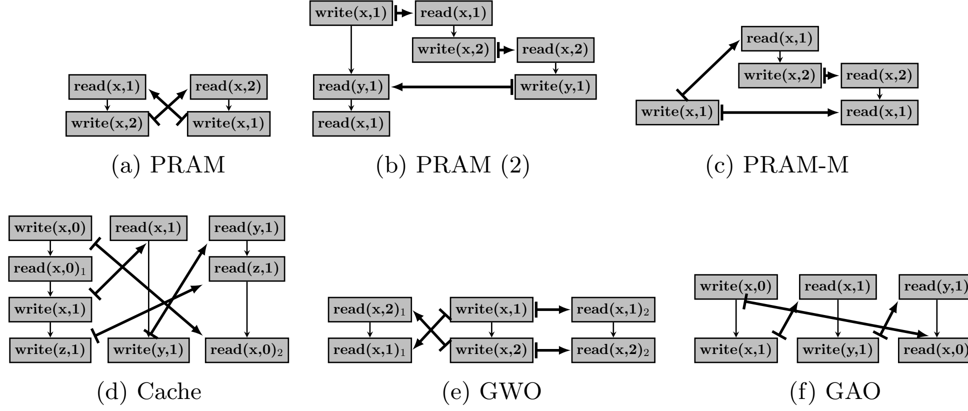
Remark: The example is neither GWO nor cache consistent (and therefore neither causal nor processor consistent either) which is shown in the corresponding sections.

Figure 3.2b shows another execution which is PRAM consistent:

[$P1$]: write(x,2), write(x,1), write(y,1), read(y,1), read(x,1)
[$P2$]: write(x,1), read(x,1), write(x,2), write(y,1) [$P3$]: write(x,1), write(x,2), read(x,2), write(y,1)

Remark: The example is however not PRAM-M consistent which is shown later.

Figure 3.1b is not PRAM consistent because the $\mapsto$ relation requires the write(y,1) to happen before the read(y,1) and the read(x,1) before the write(x,3), which is not possible because write(x,3) $<_{PO}$ write(y,1) and read(y,1) $<_{PO}$ read(x,1).



(a) PRAM (b) PRAM (2) (c) PRAM-M

(d) Cache (e) GWO (f) GAO

Figure 3.2.: Examples for PRAM, cache, GWO, and GAO consistent executions

Figure 3.2d shows an execution which is not PRAM consistent because $write(x,1) <_{PO} write(z,1) \mapsto read(z,1) <_{PO} read(x,0)_2$ implies $write(x,1)$ appears before $read(x,0)$ such that no SerialView exists for P3.

Figure 3.2e gives an example of an execution which is not PRAM consistent because the view P3 has on the writes of P2 does not comply with $<_{PO}$.

Figure 3.4a shows another example of an execution which is not PSO consistent because in a SerialView for P1 the following order on y would be required: $read(y,1) < write(y,2) < read(y,2) < write(y,3) < read(y,3)$, but the constraints $write(x,1) < write(y,3) < read(y,3) < read(x,0)$ and $write(x,0) < read(y,1) < write(y,2) < write(x,1)$ would not allow a SerialView respecting $<_{PO}$.

3.4. PRAM-M Consistency

The informal definition (Definition 3.4) of PRAM consistency by Lipton and Sandberg [LiSa88] led to different interpretations which represent different consistency models. In contrast to the preceding definition of PRAM based on Ahamad et al., the definition of Mosberger [Mosb93b] assumes that a process k's write first updates the local memory M_k and afterwards broadcasts it to other processes and that reads are explicitly blocking. To distinguish this definition from the previous one, it is named PRAM-M in this thesis.

Definition 3.6. Two operations are ordered by local write-read-write order, $o_1 <_{lwo}^{(i)} o_2$, iff $process(o_1) = i \wedge (o_1, o_2) \in (\mapsto \cup <_{PO})^*$
(* means the transitive closure)

Definition 3.7. An execution is **PRAM-M consistent** iff

$$\forall_{i \in P} \exists SerialView \left(<_{PO} \cup <_{lwo}^{(i)} |(*, i, *, *) \cup (w, *, *, *) \right)$$

Example Figure 3.2c gives an example of a PRAM-M consistent execution:

[$P1$]: write(x,1), write(x,2) [$P2$]: write(x,1), read(x,1), write(x,2)
[$P3$]: write(x,2), read(x,2), write(x,1), read(x,1)

Remark: The example is however neither cache nor GWO consistent which is shown in the corresponding sections.

Figure 3.2b is not PRAM-M consistent as for P1 no SerialView exists as $write(x,1) <_{lwo}^{(i)} write(x,2) <_{lwo}^{(i)} write(y,1)$ conflicts with $read(y,1) <_i read(x,1)$.

3.5. Cache Consistency (CC) / Global Data Order (GDO)

In 1989 Goodman [Good91] provided a definition for cache consistency, which he called Weak Consistency as he implied that it would be the weakest form of consistency. Furthermore he mentioned that no synchronization guarantees would be possible with CC which is disproved by the existence of algorithms for exclusion even for slow consistency.

Cache consistency [Good91] and the equivalent GDO consistency are formally defined in Steinke and Nutt [StNu04, Theorem 3.3,Definition 4.4] as follows:

Definition 3.8. An execution is **cache consistent** iff
$\forall_{x \in V} \exists SerialView \left(<_{PO} \mid (*, *, x, *) \right)$

Definition 3.9. Two operations are ordered by data order, $o_1 <_{DO} o_2$, iff $var(o_1) = var(o_2)$ and either: $o_1 <_{PO} o_2$ or $o_1 \mapsto o_2$ or $\exists_{r \in (r,*,*,*)} var(r) = var(o_1) \wedge val(r) \neq val(o_1) \wedge o_1 <_{PO} r \wedge o_2 \mapsto r$ or $\exists_{o \in O} o_1 <_{DO} o <_{DO} o_2$

Definition 3.10. An execution is Global Data Order (**GDO**) iff:
$\forall_{i \in P} \exists SerialView \left(<_i \cup <_{DO} \mid (*, i, *, *) \cup (w, *, *, *) \right)$

Each process observes the same ordering on memory operations regarding the same memory location, but processes may see operations regarding different memory locations in different orders.

Example Figure 3.2d shows an execution which is cache consistent:

[x]: write(x,0), read(x,0)$_1$, read(x,0)$_2$, write(x,1), read(x,1)
[y]: write(y,1), read(y,1) [y]: write(z,1), read(z,1)
[$P1$]: write(x,0), read(x,0)$_1$, write(x,1), write(z,1), write(y,1)
[$P2$]: write(x,0), write(x,1), read(x,1), write(z,1), write(y,1)
[$P3$]: write(y,1), read(y,1), write(z,1), read(z,1), write(x,0), read(x,0)$_2$, write(x,1)

Remark: The example is neither PRAM nor GWO nor PSO (and therefore neither PC-G nor PC-D nor causal) consistent which is shown in the corresponding sections.

Figure 3.1b shows an execution which is not cache consistent because the $\mapsto$ relation requires the read(x,1) to happen before the write(x,3) and the write(x,2) after the write(x,3), which is not possible because write(x,2) $<_{PO}$ read(x,1).

Figure 3.2a cannot be cache consistent because $<_{PO} \cup \mapsto |(*, *, x, *)$ forms a cycle so that no Serialization may exist.

Figure 3.2e gives an example of an execution which is not cache consistent because no total order respecting $<_{PO}$ on the operations regarding variable x can be formed: no matter which order is chosen for write(x,1) and write(x,2), it either conflicts with P1's or P3's operations.

Figure 3.3a shows an execution which is not cache consistent because P2 and P3 see a different ordering of the writes, so that no total order of the operations regarding x is possible.

Figure 3.2c is not cache consistent because there exists no SerialView for x as $write(x, 1)$ has to be ordered before $write(x, 2)$ and $read(x, 1)$ after $write(x, 2)$.

3.6. Global Write-read-write Order (GWO)

GWO was defined by Steinke and Nutt in their Unified theory of Shared Memory Consistency [StNu04]. They introduced the GWO property along with three other properties (GPO,GDO,GAO) which can be combined to define different consistency models.

Definition 3.11. Two writes are ordered by **write-read-write order**, $w_1 <_{WO} w_2$, iff there exists a read r, such that $w_1 \mapsto r <_{PO} w_2$

Definition 3.12. An execution is Global Write-read-write Order (**GWO**) iff
$\forall_{i \in P} \exists SerialView\,(<_i \cup <_{WO} |\,(*, i, *, *) \cup (w, *, *, *))$

The write-read-write order orders two writes if a read exists which reads from the first write and is issued before the second write by the same processor. An execution is GWO if a process sees all writes which are ordered by write-read-write order in the given order along all own operations in the order it issued them.

Example Figure 3.2e gives an example of an execution which is GWO consistent:
[$P1$]: write(x,2), read(x,2), write(x,1), read(x,1) [$P2$]: write(x,1), write(x,2)
[$P3$]: write(x,1), read(x,1), write(x,2), read(x,2)

Remark: The example is neither PRAM nor Cache nor PSO (and therefore neither PC-G nor PC-D nor causal) consistent which is shown in the corresponding sections.

Figure 3.2a is not GWO consistent because $<_0 \cup <_{WO} \cup \mapsto |(*, i, *, *) \cup (w, *, *, *)$ forms a cycle which excludes the existence of a SerialView.

Figure 3.2d shows an execution which is not GWO consistent as $write(x, 0) <_{WO} write(x, 1) <_{WO} write(y, 1)$ and $read(y, 1) < read(x, 0)_2$ exclude the existence of a SerialView.

An example for an execution which is not GWO is given in Figure 3.3b. It cannot be GWO consistent because write(x,1) $<_{WO}$ write(x,2) $<_{WO}$ write(z,1) which conflicts with read(z,1) $<_i$ read(x,1).

Figure 3.4b shows another example of an execution which is not GWO consistent as no SerialView for P2 respecting $write(x,0) <_{WO} write(x,1) <_{WO} write(y,1)$ and $read(y,1) <_{PO} read(x,0)$ can exist, the example cannot be GWO consistent.

The execution in Figure 3.4c is not GWO as $write(y,0) <_{WO} write(y,1) <_{WO} write(x,1) <_{WO} write(y,2)$ prevents a SerialView of P1 which respects $<_{WO}$ and $<_i$.

Figure 3.2c is not GWO consistent because there exists no SerialView for P3 as $write(x,1)$ has to be ordered before $write(x,2)$ and $read(x,1)$ after $write(x,2)$.

3.7. Global Anti-Order (GAO)

GAO was the fourth basic property defined by Steinke and Nutt [StNu04]. It is an extension to GDO, as they showed that GPO, GDO, and GWO together are not sufficient to describe sequential consistency. GPO+GAO+GWO however is equivalent to sequential consistency.

Definition 3.13. A **Serial Order** $<_{SO}$ is a minimum partial order that satisfies:

$$\forall_{w,r \in O} \left((var(w) = var(r) \land val(w) \neq val(r)) \Rightarrow \left(w <_{SO} w' \mapsto r \lor r <_{SO} w \right) \right)$$

Definition 3.14. Two writes are ordered by **Anti Order**, $w_1 <_{AO(<_{SO})} w_2$, iff

$$\begin{aligned} \exists_{r_1, r_2 \in O} \big[& \\ & w_1 \mapsto r_1 <_{PO} r_2 <_{DO} w_2 \\ \lor\ & w_1 \mapsto r_1 <_{PO} r_2 <_{SO} w_2 \\ \lor\ & w_1 \mapsto r_1 <_{SO} w_2 \\ \lor\ & w_1 <_{PO} r_1 <_{DO} w_2 \\ \lor\ & w_1 <_{PO} r_1 <_{SO} w_2 \quad \big] \end{aligned}$$

Definition 3.15. An execution is Global Anti-Order (**GAO**) iff

$$\forall_{i \in P} \exists SerialView \left(<_i \cup <_{SO} \cup <_{AO(<_{SO})} | \ (*, i, *, *) \cup (w, *, *, *) \right)$$

Example Figure 3.2f gives an example of an execution which is GAO consistent.

Remark: The example is not PSO consistent which is shown in the corresponding section.

Figure 3.4d gives an example of an execution that is not GAO consistent: For $<_{SO}$ only the pairs $write(x,1), read(x,0)$ and $write(y,1), read(y,0)$ and $write(y,0), read(y,0)$ must be considered:

$write(x,1) <_{SO} write(x,0)$ conflicts with P1's $<_i$ so choose $read(x,0) <_{SO} write(x,1)$.
$write(y,1) <_{SO} write(y,0)$ conflicts with P2's $<_i$ so choose $read(y,0) <_{SO} write(y,1)$.
$read(y,1) <_{SO} write(y,0)$ with $<_i$ would not allow a SerialView for P1
so choose $write(y,0) <_{SO} write(y,1)$ instead.
$write(x,0) <_{PO} write(x,1) <_{PO} read(y,0) <_{DO} write(y,1)$ leads to:
$write(x,0) <_{AO(<_{SO})} write(y,1)$ and $write(x,1) <_{AO(<_{SO})} write(y,1)$
$read(x,0) <_{SO} write(x,1)$, $read(y,0) <_{SO} write(y,1)$ implies:
$write(y,0) <_{AO(<_{SO})} write(y,1)$ and $write(x,0) <_{AO(<_{SO})} write(x,1)$
For P2: $<_{AO(<_{SO})}$ and $<_i$ require $write(x,0)$ to occur before $write(x,1)$ and $write(x,1)$ before $read(x,0)$ which excludes the existence of a SerialView which respects $<_{AO(<_{SO})}$ and $<_i$.

3.8. Causal Consistency / GPO+GWO

Causal memory is based on potential causality defined by Lamport [Lamp78] which defines a partial order on all memory operations. The partial order orders operations that are causally related, which enhances the former mentioned write-read-write order by process order (for other processes). Causal memory enforces that if a process writes a value after reading some writes' values, which potentially influenced the write, then all processes reading that write must have observed those other writes before as well.

The definition of causal consistency [ABHN91] taken from Steinke and Nutt [StNu04] is as follows:

Definition 3.16. An execution is **causally consistent** iff

$$\forall_{i \in P} \exists SerialView\,(<_i \cup <_{PO} \cup <_{WO} | (*, i, *, *) \cup (w, *, *, *))$$

Example Figure 3.3a shows an execution which is causally consistent:

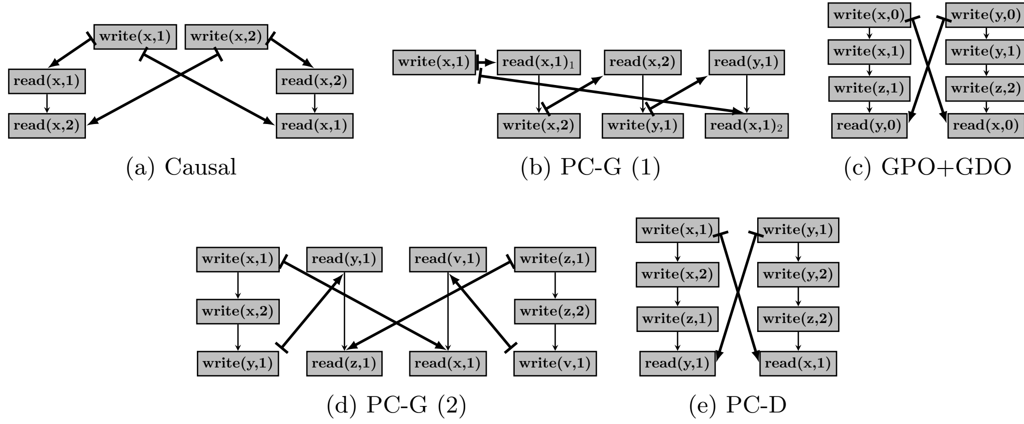
$[P1]$: write(x,1), read(x,1), write(x,2), read(x,2) $[P2]$: write(x,1), write(x,2)
$[P4]$: write(x,2), read(x,2), write(x,1), read(x,1) $[P3]$: write(x,2), write(x,1)

Remark: The example is not cache consistent (and therefore not PC-G either) which is shown in the corresponding section.

3.9. Processor Consistency by Goodman (PC-G)

Goodman gave a definition of processor consistency which is a consistency model stronger than both cache consistency and PRAM consistency but weaker than sequential consistency. As there exists multiple consistency models that differ slightly from Goodman's definition but are also called processor consistency the abbreviation PC-G is used for Goodman's definition.

Goodman's definition of processor consistency [Good91] and the equivalent GPO+GDO' consistency based on Theorem 3.4 from Steinke and Nutt [StNu04] is as follows:



(a) Causal (b) PC-G (1) (c) GPO+GDO

(d) PC-G (2) (e) PC-D

Figure 3.3.: Examples for causally, PC-G, GPO+GDO, and PC-D consistent executions

Definition 3.17. An execution is **processor consistent (PC-G)** iff

$$\forall_{x \in V} \exists <_x = SerialView\left(<_{PO}\mid (*,*,x,*)\right)$$
$$\wedge \forall_{i \in P} \exists SerialView\left(\left(\cup_{x \in V} <_x\right) \bigcup <_{PO} \Big| (*,i,*,*) \cup (w,*,*,*)\right)$$

Definition 3.18. An order is an augmented data order $<_{DO'}$ iff: $<_{DO} \subseteq <_{DO'}$ and $\forall_{o_1,o_2 \in O} \left(var(o_1) = var(o_2) \Rightarrow o_1 <_{DO'} o_2 \vee o_2 <_{DO'} o_1\right)$

Definition 3.19. An execution is **GPO+GDO'** iff

$$\forall_{i \in P} \exists SerialView\left(<_{PO} \cup <_{DO'} \mid (*,i,*,*) \cup (w,*,*,*)\right)$$

Example An example for a PC-G consistent execution is given in Figure 3.3b:

$[x]$: write(x,1), read(x,1), read(x,1), write(x,2), read(x,2)
$[y]$: write(y,1), read(y,1)
$[P1]$: write(x,1), write(x,2), write(y,1)
$[P2]$: write(x,1), read(x,1), write(x,2), write(y,1)
$[P3]$: write(x,1), write(x,2), read(x,2), write(y,1)
$[P4]$: write(x,1), write(y,1), read(y,1), read_2(x,1), write(x,2)

Remark: The example is neither GWO nor PSO consistent which is shown in the corresponding sections.

Figure 3.3d is PC-G consistent, because the required SerialViews exist:

$[x]$: $write(x,1), read(x,1), write(x,2)$ $[y]$: $write(y,1), read(y,1)$
$[z]$: $write(z,1), read(z,1), write(z,2)$ $[v]$: $write(v,1), read(v,1)$
$[P1]$: $write(x,1), write(x,2), write(y,1), write(z,1), write(z,2), write(v,1)$
$[P2]$: $write(x,1), write(x,2), write(y,1), read(y,1), write(z,1), read(z,1), write(z,2), write(v,1)$
$[P3]$: $write(z,1), write(z,2), write(v,1), read(v,1), write(x,1), read(x,1), write(x,2), write(y,1)$
$[P4]$: $write(z,1), write(z,2), write(v,1), write(x,1), write(x,2), write(y,1)$

Remark: The example is not PC-D consistent which is shown in the corresponding section.

Figure 3.3c cannot be PC-G because PC-G requires a total order on all operations on z, but if $write(z,1)$ is ordered before $write(z,2)$ then $write(x,1)$ would be ordered before $read(x,0)$, otherwise if $write(z,2)$ is ordered before $write(z,1)$ then $write(y,1)$ would be before $read(y,0)$.

Another example for an execution which is not PC-G is given in Figure 3.4c: PC-G requires a SerialView on each variable:

$[x]_1$: $write(x,0), read(x,0), write(x,1), read(x,1)$
$[x]_2$: $write(x,1), read(x,1), write(x,0), read(x,0)$
$[y]$: $write(y,0), read(y,0), read(y,0), write(y,1), read(y,1), read(y,1), write(y,2), read(y,2)$

Neither one of the possible SerialViews $[x]_1$ nor $[x]_2$ allows P1 to have a SerialView that respects it together with $[y]$ and $<_{PO}$. Therefore it cannot be PC-G.

Figure 3.3e is not PC-G consistent as there exists no SerialView for z which allows SerialViews for both processes. If $write(z,1)$ is ordered before $write(z,2)$ then no SerialView for P2 exists as $write(x,1) <_{PO} write(x,2) <_{PO} write(z,1)$ conflicts with $write(z,2) <_{PO} read(x,1)$, otherwise if $write(z,2)$ is ordered before $write(z,1)$ then no SerialView for P1 exists as $write(y,1) <_{PO} write(y,2) <_{PO} write(z,2)$ conflicts with $write(z,1) <_{PO} read(y,1)$. Therefore, the example cannot be PC-G consistent.

3.10. GPO + GDO Consistency

GPO+GDO is a model described by Steinke and Nutt [StNu04] as combination of the properties GPO and GDO (PRAM and cache consistency). It is however not equivalent to PC-G as there exist executions which are GPO+GDO but not PC-G consistent. However, it is very close to PC-G, and Steinke and Nutt argue that GPO+GDO should be chosen as definition.

The definition of GPO+GDO [StNu04] and an equivalent modified version of PC-G is as follows:

Definition 3.20. An execution is **GPO+GDO** iff

$$\forall_{i \in P} \exists SerialView\left(<_i \cup <_{PO} \cup <_{DO} \big| (*,i,*,*) \cup (w,*,*,*)\right)$$

Definition 3.21. An execution is **PC-G'** iff

$$\forall_{i \in P} \Bigg(\forall_{x \in V} \exists <_x = SerialView\left(<_{PO} \big| (*,*,x,*)\right) \\ \wedge \exists SerialView\left((\cup_{x \in V} <_x) \bigcup <_{PO} \Big| (*,i,*,*) \cup (w,*,*,*) \right) \Bigg)$$

An example for an execution which is GPO+GDO is given in Figure 3.3c.

Remark: The example is not PC-G which is shown in the corresponding section. It is also an example of a cache and PRAM consistent execution which is not PC-G.

3.11. Processor Consistency by DASH (PC-D)

The Stanford DASH multiprocessor system [LLGW+92] implements a variation of processor consistency hereafter called PC-D which is incomparable with PC-G [ABJK+93, GLLG+90]. Gharachorloo et al.'s [GLLG+90] definition differs from PC-G by weakening process order by allowing a process' read to outrun a write of the same process but on a different location. Furthermore PC-D enforces a different ordering on write operations.

View-based definition of PC-D [GLLG+90] by Ahamad et al. [ABJK+93]:

Definition 3.22. Two operations are ordered by weak-process-order, $o_1 <_{wpo} o_2$, iff

$$\begin{aligned} & o_1 <_{PO} o_2 \\ & \wedge \big(var(o_1) = var(o_2) \vee type(o_1) = type(o_2) \\ & \quad \vee (type(o_1) = r \wedge type(o_2) = w) \vee \exists_{o'} o_1 <_{wpo} o' <_{wpo} o_2\big) \end{aligned}$$

Definition 3.23. Two operations are ordered by weak-writes-before-order, $o_1 <_{wwb} o_2$, iff

$$type(o_1) = w \wedge type(o_2) = r \wedge \exists_{o'} \left(o_1 <_{wpo} o' \wedge o' \mapsto o_2\right)$$

Definition 3.24. Two operations are ordered by weak-reads-before-order, $o_1 <_{wrb} o_2$, with respect to the SerialViews $<_x, x \in V$ iff

$$type(o_1) = r \wedge type(o_2) = w \wedge \exists_{o' \in (w,*,*,*)} \left(o_1 (\cup_{x\in V} <_x) o' \wedge o' <_{wpo} o_2\right)$$

Definition 3.25. An execution is **processor consistent (PC-D)** iff

$$\begin{aligned} & \forall_{x\in V} \exists <_x = SerialView\,(<_{PO}|\ (*,*,x,*)) \\ & \wedge \forall_{i\in P} \exists SerialView\,(<_{wpo} \cup <_{wwb} \cup <_{wrb} \cup ((\cup_{x\in V} <_x)|(w,*,*,*))|(*,i,*,*) \cup (w,*,*,*)) \end{aligned}$$

Example Figure 3.3e gives an example of a PC-D consistent execution.

3.12. Partial Store Ordering (PSO)

Partial Store Ordering is one memory model used in SPARC architectures. It provides a better performance than the default TSO, but is defined as optional only in the architecture manual, so not all SPARC architectures may provide PSO. It allows to reorder writes after writes of different locations and writes after reads.

In the following, a simplified and modified axiomatic definition from the SPARC Architecture manual [SPAR91, SiFC92] for PSO [SPAR91] consistency is given. 'Simplified' means that the original definition uses 7 axioms, but three of them (Atomicity, Termination, StoreStore) are irrelevant for this consideration. Atomicity only concerns Swap operations which are not covered in this pure model definition of TSO, Termination gives a guarantee that a store will eventually be written to the memory and removed from the store buffer, and StoreStore considers STBAR instructions which are also not covered by this pure model definition. 'Modified' means that the axioms have been rewritten to fit the formalism used in this paper:

Definition 3.26. An execution is PSO if there exists a memory order $\leq$ which respects:

- Order: $(w, i, x, v_1) \leq (w, j, y, v_2) \vee (w, i, y, v_2) \leq (w, j, x, v_1)$
- Value:

$$val\big((r, i, a, x)\big) = val\big(Max_{\leq}\big[\,\{(w, j, a, y) | (w, j, a, y) <_{PO} (r, i, a, x)\} \cup \{(w, j, a, y) | (w, j, a, y) \leq (r, i, a, x)\}\,\big]\big)$$

- LoadOp: $r \in (r, i, *, *), o \in (*, i, *, *)\ r <_{PO} o \Rightarrow r \leq o$
- StoreStoreEq: $\forall_{x \in V} \forall_{w_1, w_2 \in (write, i, x, *)} w_1 <_{PO} w_2 \Rightarrow w_1 \leq w_2$

PSO consistency is best explained by describing a possible computer architecture: Each process has a store buffer for each memory location which buffers writes before writing them to memory. If a process reads a location for which a write exists in the corresponding store buffer then it reads the latest write's value from that store buffer, otherwise it reads the value from memory.

An execution is PSO consistent if a process observes its own operations in program order and there exists a total order on all write operations which is observed by all processes regarding others' writes. Also a process may not observe all writes of other processes as these may be hidden by own writes in its store buffer.

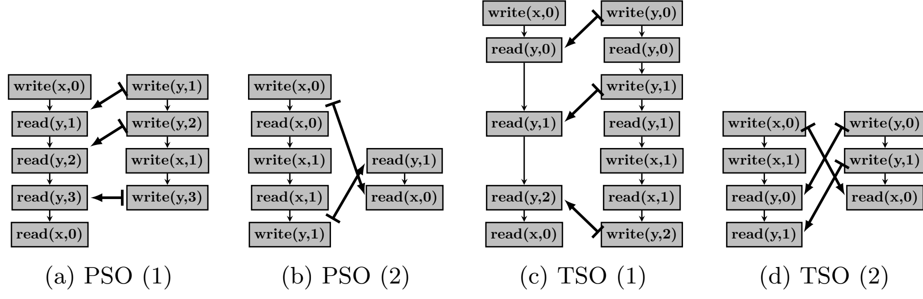
Example Figure 3.4a shows an example of an execution which is PSO consistent what can easily be seen if $write(x, 0)$ is buffered by P1.

Remark: The example is not PRAM consistent which is shown in the corresponding section.

It can be seen that the execution in Figure 3.4b is PSO consistent if $write(x, 1)$ is buffered by P1.

Remark: The example is not GWO consistent which is shown in the corresponding section.

Figure 3.2d shows an execution which is not PSO consistent because P3 has no writes and



(a) PSO (1) (b) PSO (2) (c) TSO (1) (d) TSO (2)

Figure 3.4.: Examples for PSO, and TSO consistent executions

this way no writes may be hidden and therefore it is implied that $read(x, 1)$ appears before $read(x, 0)_2$ in the Serialization which therefore cannot be a SerialView.

Figure 3.2e gives an example of an execution which is not PSO as P1 and P3 have different views on the writes of P2 where no serialization $<_W$ on all writes may exist.

Figure 3.2f gives another example of an execution which is not PSO consistent because if P2 reads $write(x, 1)$, both writes of P1 have already been written to main memory and if P3 reads $write(y, 1)$ all three writes of P1 and P2 have also been written to memory, therefore P3 cannot read $write(x, 0)$.

Another example for an execution which is not PSO is given in Figure 3.3b.

3.13. Total Store Ordering (TSO)

Total Store Ordering is the default memory model of SPARC architectures. The SPARC architecture manual [SPAR91] states that every implementation has to offer TSO. TSO allows the reordering of stores after loads.

In the following a simplified and modified axiomatic definition by [SPAR91, SiFC92] for TSO [SPAR91] consistency is given. 'Simplified' means that the original definition uses 6 axioms, but two of them (Atomicity, Termination) are irrelevant for this consideration. Atomicity only concerns Swap operations which are not covered in this definition of TSO and Termination gives a guarantee that a store will eventually be written to the memory and removed from the store buffer. 'Modified' means that the axioms have been rewritten to fit the formalism used in this paper:

Definition 3.27. An execution is TSO if there exists a memory order $\leq$ which respects:

- Order: $(w, i, x, v_1) \leq (w, j, y, v_2) \vee (w, i, y, v_2) \leq (w, j, x, v_1)$
- Value:

$$val\big((r, i, a, x)\big) = val\Big(Max_{\leq}\big[\{(w, j, a, y) | (w, j, a, y) <_{PO} (r, i, a, x)\} \cup \{(w, j, a, y) | (w, j, a, y) \leq (r, i, a, x)\}\big]\Big)$$

- LoadOp: $r \in (r, i, *, *), o \in (*, i, *, *)\ r <_{PO} o \Rightarrow r \leq o$
- StoreStore: $w_1, w_2 \in (write, i, *, *)\ w_1 <_{PO} w_2 \Rightarrow w_1 \leq w_2$

A view-based definition of TSO [SPAR91] consistency based on the axioms in [SPAR91, SiFC92, LoCM06] is proposed as follows:

Definition 3.28.

$$W_{Inv}^{(i)} = \big\{w \,|proc(w) \neq i \wedge \nexists r \in (r,i,*,*) : w \mapsto r \\ \wedge \big[\exists w' \in (w,i,*,*) : w <_W w' \wedge var(w) = var(w') \\ \wedge \nexists w'' \in (w,j,*,*), j \neq i, r \in (r,i,*,*) : w <_W w'' <_W w' \wedge w'' \mapsto r \wedge r <_i w'\big]\big\}$$

Proposition 3.1. An execution is **TSO consistent** iff

$$\exists <_W = SerialView\,(<_{PO}|(w,*,*,*)) \\ \wedge \\ \forall_{i \in P} \exists SerialView\,\Big(<_i \cup (<_W \setminus (w,*,*,*) \times (w,i,*,*))\Big|\Big((w,*,*,*) \setminus W_{inv}^{(i)}\Big) \cup (*,i,*,*)\Big)$$

TSO consistency is best explained by describing a possible architecture: Each process has a store buffer which buffers writes before writing them to memory in order. If a process reads a location for which a write exists in its store buffer, then it reads the latest value from the store buffer, otherwise it reads the value from memory.

An execution is TSO consistent if a process observes its own operations in program order and there exists a total order on all write operations which respects program order and is observed by all processes regarding others' writes. Also a process may not observe all writes of other processes as they may be hidden by own writes in its store buffer.

Example An example for a TSO consistent execution is given in Figure 3.4c. It can be easily seen that the example is TSO consistent if only $write(x,0)$ is buffered.

Remark: The example is neither PC-G nor GWO consistent which is shown in the corresponding sections.

Figure 3.4d gives an example of an execution that is TSO consistent. If only $write(x,1)$ is kept in the store buffer then it is clear that the execution is TSO.

Remark: The example is not GAO consistent which is shown in the corresponding section.

3.14. Sequential Consistency (SC)

Sequential consistency was defined by Lamport [Lamp79]. He states that an execution on a multiprocessor system may not produce the desired results while each process itself executes its program correctly. Therefore he introduces two implementation requirements which enforce sequential consistency:

- R1: Each processor issues memory requests in the order specified by its program.

- R2: Memory request from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request in this queue.

The definition of sequential consistency [Lamp79] according to [StNu04] is:

Definition 3.29. An execution is **sequentially consistent** iff
$\exists SerialView(<_{PO})$

A system is sequentially consistent if for all executions, there exists a sequential order which respects process order and leads to the same result as the execution.

Example Figure 3.1c gives an example of a sequentially consistent execution. It is SC as the following SerialView respecting $<_{PO}$ exists: $write(x, 1), read(x, 1), write(y, 1), read(y, 1)$.

3.15. Overview/Relationship

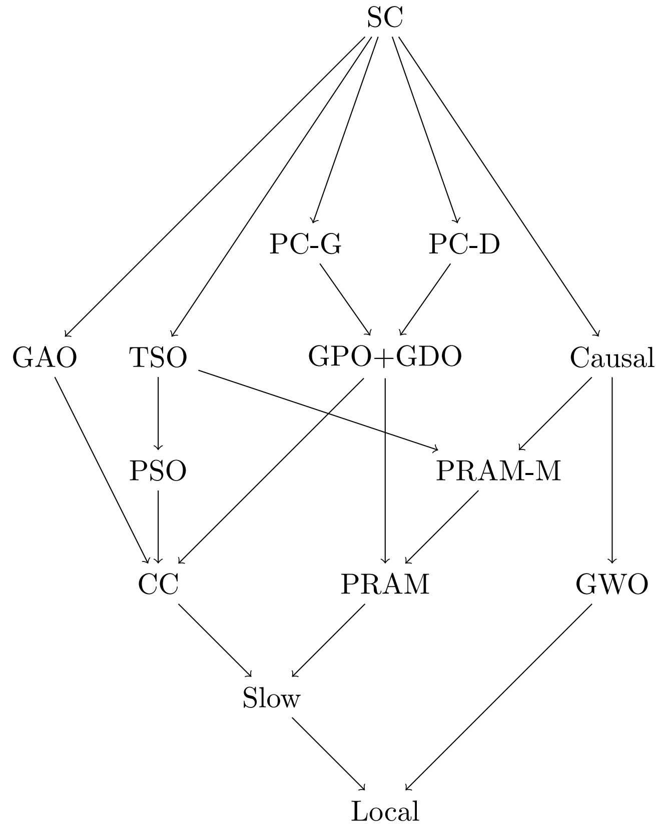


Figure 3.5.: Relationship overview over the presented consistency models

Figure 3.5 illustrates the relationship between the described consistency models. An arrow indicates that the model the arrow originates from implies the model the arrow points at. A model implies an other model if all executions accepted by this model are excepted by the other one.

The relationship of local, slow, PRAM (GPO), cache (GDO), GWO, causal (GPO+GWO), GPO+GDO and sequential consistency is shown by Steinke and Nutt [StNu04].

In the following, the relationship of PSO and TSO to the other models is shown.

3.15.1. PSO

Theorem 3.1. *TSO ⇒ PSO : all TSO consistent executions are PSO consistent.*

Proof. It is easy to see that a TSO system behaves like a PSO system which writes values from its store buffers to main memory in the same order the process' single store buffer

of the TSO system does. □

Theorem 3.2. *PSO $\not\Rightarrow$ TSO : some PSO consistent executions are not TSO consistent.*

Proof. Figure 3.4b gives a counterexample which is PSO consistent but not TSO consistent, it is explained in Section 3.12 that the example is PSO. It cannot be TSO because when P2 reads the value of $write(y, 1)$ all of P1's writes must have been written back from the store buffer, therefore P2 observes $write(x, 0) <_{PO} write(x, 1)$ and cannot observe $read(x, 0)$ afterwards. □

Theorem 3.3. *PSO $\Rightarrow$ CC : all PSO consistent executions are cache consistent.*

Proof. Assume a PSO consistent execution. Then the main memory sees a sequential order $<_W$ of all writes and all unbuffered reads. If all operations from this order are excluded except the operations regarding variable x, then the resulting order $<^x_W$ clearly respects $<_{PO}$. If $<_{PO}$ is limited to only operations regarding variable x then buffered reads always follow the corresponding writes without any other intermediate operations regarding x in between. If those buffered reads are inserted directly after their corresponding write into $<^x_W$, then the resulting order $<_x$ is a sequentialization of all operations regarding x which respects $<_{PO}$. Therefore, the execution is cache consistent. □

Theorem 3.4. *CC $\not\Rightarrow$ PSO - some cache consistent executions are not PSO consistent.*

Proof. Figure 3.2d shows an example of an execution which is cache consistent but not PSO consistent, which is explained in Section 3.5 and 3.12. □

Theorem 3.5. *PSO $\not\Rightarrow$ PRAM - some PSO consistent executions are not PRAM consistent.*

Proof. Figure 3.4a shows an example of an execution which is PSO consistent but not PRAM consistent, which is explained in Section 3.12 and 3.3. □

Theorem 3.6. *PRAM $\not\Rightarrow$ PSO - some PRAM consistent executions are not PSO consistent.*

Proof. Assuming $PRAM \Rightarrow PSO$. Using $PSO \Rightarrow CC$ follows $PRAM \Rightarrow CC$ ↯ because PRAM is incomparable to CC as seen in the examples at Sections 3.5 and 3.3. Therefore, the assumption must be incorrect. □

Theorem 3.7. *$PSO \not\Rightarrow GWO$ - some PSO consistent executions are not GWO consistent.*

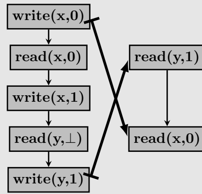
Proof. Figure 3.4b shows an example of an execution which is PSO consistent, but not GWO consistent, which is explained in Sections 3.12 and 3.6. □

Theorem 3.8. *$GWO \not\Rightarrow PSO$ - some GWO consistent executions are not PSO consistent.*

Proof. Assuming $GWO \Rightarrow PSO$. Using $PSO \Rightarrow CC$ follows $GWO \Rightarrow CC$ ↯ because GWO is incomparable to CC as seen in the examples at Sections 3.5 and 3.6. Therefore, the assumption must be incorrect. □

Theorem 3.9. *$PSO \not\Rightarrow GAO$ - some PSO consistent executions are not GAO consistent.*

Proof. Counterexample:



The example is clearly PSO if only write(x,1) is kept in the corresponding store buffer. It is not GAO because $write(x,0) <_{PO} read(x,0) <_{DO} write(x,1) <_{PO} read(y,\bot) <_{DO} write(y,1)$ implies $write(x,0) <_{AO(<_{SO})} write(x,1) <_{AO(<_{SO})} write(y,1)$, and thus no SerialView for P2 exists which respects $<_{AO(<_{SO})}$ and $<_i$ as $read(y,1) <_i read(x,0)$ conflicts with the mentioned $<_{AO(<_{SO})}$. □

Theorem 3.10. *$GAO \not\Rightarrow PSO$ - some GAO consistent executions are not PSO consistent.*

Proof. Figure 3.2f shows an example of an execution which is GAO consistent, but not PSO consistent, which is explained in Sections 3.7 and 3.12. □

3.15.2. TSO

Theorem 3.11. *$TSO \Rightarrow PRAM$ - all TSO consistent executions are PRAM consistent.*

Proof. Assuming TSO consistency.
TSO implies the existence of a total order $<_W$ on all writes which respects $<_{PO}$. As every process sees its own writes in program order $<_i$ and all other processes' write in

order $<_W$, it is implied that every process sees all writes in an order which respects $<_{PO}$ which implies PRAM consistency. □

Theorem 3.12. *$PRAM \not\Rightarrow TSO$ - some PRAM consistent executions are not TSO consistent.*

Proof. Assuming $PRAM \Rightarrow TSO$. Using $TSO \Rightarrow PSO$ and $PSO \Rightarrow CC$ follows $PRAM \Rightarrow CC$ ↯ because PRAM is incomparable to CC as seen in the examples at Sections 3.5 and 3.3. Therefore, the assumption must be incorrect. □

Theorem 3.13. *$TSO \not\Rightarrow GWO$ - some TSO consistent executions are not GWO consistent.*

Proof. Figure 3.4c shows an example of an execution which is TSO consistent, but not GWO consistent, which is explained in Sections 3.13 and 3.6. □

Theorem 3.14. *$GWO \not\Rightarrow TSO$ - some GWO consistent executions are not TSO consistent.*

Proof. Assuming $GWO \Rightarrow TSO$. Using $TSO \Rightarrow PSO$ follows $GAO \Rightarrow PSO$. ↯ This assumption is incorrect as it would conflict with $GWO \not\Rightarrow PSO$ shown in Theorem 3.8 □

Theorem 3.15. *$GAO \not\Rightarrow TSO$ - some GAO consistent executions are not TSO consistent.*

Proof. Assuming $GAO \Rightarrow TSO$. Using $TSO \Rightarrow PSO$ follows $GAO \Rightarrow PSO$. ↯ This assumption is incorrect as it would conflict with $GAO \not\Rightarrow PSO$ shown in Theorem 3.10 □

Theorem 3.16. *$TSO \not\Rightarrow GAO$ - some TSO consistent executions are not GAO consistent.*

Proof. Figure 3.4d shows an example of an execution which is TSO consistent, but not GAO consistent, which is explained in Sections 3.13 and 3.7. □

Theorem 3.17. *$TSO \not\Rightarrow PC\text{-}G$ - some TSO consistent executions are not PC-G consistent.*

Proof. Figure 3.4c shows an example of an execution which is TSO consistent, but not PC-G consistent, which is explained in Sections 3.13 and 3.9. □

3.15.3. PRAM-M

Theorem 3.18. *PRAM-M* $\Rightarrow$ *PRAM - all PRAM-M consistent executions are PRAM consistent.*

Proof. Consider a PRAM-M consistent execution, then there exists a SerialView on own operations and others' writes which respects $<_{PO} \cup <_{lwo}^{(i)}$ for each process. A SerialView respecting $<_{PO} \cup <_{lwo}^{(i)}$ obiously respects $<_{PO}$ as well. Therefore, all PRAM-M consistent executions must be PRAM consistent. □

Theorem 3.19. *PRAM* $\not\Rightarrow$ *PRAM-M - some PRAM consistent executions are not PRAM-M consistent.*

Proof. Figure 3.2b shows an example of an execution which is PRAM consistent, but not PRAM-M consistent, which is explained in Sections 3.3 and 3.4. □

Theorem 3.20. *PRAM-M* $\not\Rightarrow$ *CC - some PRAM-M consistent executions are not CC consistent.*

Proof. Figure 3.2c shows an example of an execution which is PRAM-M consistent, but not CC consistent, which is explained in Sections 3.4 and 3.5. □

Theorem 3.21. *CC* $\not\Rightarrow$ *PRAM-M - some CC consistent executions are not PRAM-M consistent.*

Proof. Assuming $CC \Rightarrow$ PRAM-M. Using PRAM-M $\Rightarrow PRAM$ follows $CC \Rightarrow PRAM$. ↯ This assumption is incorrect as PRAM is incomparable to CC as seen in the examples at Sections 3.5 and 3.3. □

Theorem 3.22. *PRAM-M* $\not\Rightarrow$ *GWO - some PRAM-M consistent executions are not GWO consistent.*

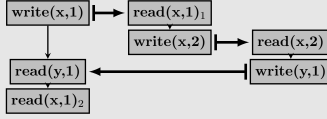
Proof. Figure 3.2c shows an example of an execution which is PRAM-M consistent, but not GWO consistent, which is explained in Sections 3.4 and 3.6. □

Theorem 3.23. *GWO* $\not\Rightarrow$ *PRAM-M - some GWO consistent executions are not PRAM-M consistent.*

Proof. Assuming $GWO \Rightarrow$ PRAM-M. Using PRAM-M $\Rightarrow PRAM$ follows $GWO \Rightarrow PRAM$. ↯ This assumption is incorrect as PRAM is incomparable to GWO as seen in the examples at Sections 3.6 and 3.3. □

Theorem 3.24. *PC-G $\not\Rightarrow$ PRAM-M - some PC-G (and therefore GPO+GDO) consistent executions are not PRAM-M consistent.*

Proof. Example:



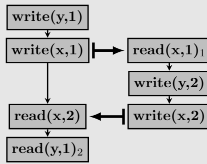
The example is PC-G consistent and therefore GPO+GDO consistent as well:

$[x]$: $write(x,1), read(x,1)_1, read(x,1)_2, write(x,2), read(x,2)$
$[y]$: $write(y,1), read(y,1)$
$[P1]$: $write(x,1), write(y,1), read(y,1), read(x,1)_2, write(x,2)$
$[P2]$: $write(x,1), read(x,1), write(x,2), write(y,1)$
$[P3]$: $write(x,1), write(x,2), read(x,2), write(y,1)$

But not PRAM-M as for P1: $write(x,1) <_{lwo}^{(i)} write(x,2) <_{lwo}^{(i)} write(y,1)$ allows no SerialView with $write(x,1) <_i read(y,1) <_i read(x,1)_2$ □

Theorem 3.25. *PSO $\not\Rightarrow$ PRAM-M - some PSO consistent executions are not PRAM-M consistent.*

Proof. Example:



The example is PSO consistent as it can be easily seen if $write(y,1)$ is buffered.

But not PRAM-M as for P1: $write(y,1) <_{lwo}^{(i)} write(y,2) <_{lwo}^{(i)} write(x,2)$ allows no SerialView with $write(y,1) <_i write(x,1) <_i read(x,2) <_i read(y,1)_2$ □

Proposition 3.2. *Causal* $\Rightarrow$ PRAM-M - all causal consistent executions are PRAM-M consistent.

Proposition 3.3. *TSO* $\Rightarrow$ PRAM-M - all TSO consistent executions are PRAM-M consistent.

3.15.4. PC-D

Theorem 3.26. *PC-D $\not\Rightarrow$ PC-G - some PC-D consistent executions are not PC-G consistent.*

Proof. Figure 3.3e shows an example of an execution which is PC-D consistent, but not PC-G consistent, which is explained in Sections 3.11 and 3.9. □

Theorem 3.27. *PC-G $\not\Rightarrow$ PC-D - some PC-G consistent executions are not PC-D consistent.*

Proof. Figure 3.3d shows an example of an execution which is PC-G consistent, but not PC-D consistent, which is explained in Sections 3.9 and 3.11. □

Proposition 3.4. PC-D $\Rightarrow$ *GPO+GDO* - all PC-D consistent executions are GPO+GDO consistent.

Proposition 3.5. *GPO + GDO* $\not\Rightarrow$ PC-D - some GPO+GDO consistent executions are not PC-D consistent.

Proposition 3.6. *TSO* $\not\Rightarrow$ PC-D - some TSO (and therefore PSO) consistent executions are not PC-D consistent.

Proposition 3.7. *Causal* $\not\Rightarrow$ PC-D - some causally consistent executions are not PC-D consistent.

4. Reference Machines

Given the set of processes P, shared variables V, operations O and the process order $\leq_{PO}$ an implementation of a memory system determines the writes-to order $\mapsto$, from which the read values can be derived.

In this chapter, reference machines for different memory consistency models are introduced. These machines determine for each input (P,V,O,$\leq_{PO}$) a writes-to order which is consistent with its memory consistency model. Furthermore, it is shown that the machines cover every possible writes-to order which is consistent with its memory model for a given input.

These reference machines illustrate the definition of different memory models in an operational way.

4.1. Common Structural Elements

The reference machines are constructed by components which are introduced in the following.

4.1.1. FIFO

The FIFO component is a First-In-First-Out Buffer which buffers memory operations as tuples. It holds the operation type (read or write), the issuing process' id, the memory address, and in case of a write operation the value to be written.

The FIFO component is considered to be an unbounded buffer for completeness proofs. It can be bounded for finite operation sets. Boundedness is further discussed in Section 5.4.

An implementation is given as Quartz code in Appendix A.2.1. The implementation is bounded because of programming language restrictions.

4.1.2. FIFOwClock

FIFOwClock is an extended FIFO buffer which endores entries by an additional clock value. A clock value is a natural number which is considered unbounded for theoretical purposes.

An implementation is given as Quartz code in Appendix A.2.2. In the implementation the clock value is a bounded natural number because of programming language restrictions.

4.1.3. FIFOwClocks

FIFOwClocks is an extended FIFOwClock buffer which does not only contain one clock for each entry, but N clocks, where N is the number of processes. An implementation is given as Quartz code in Appendix A.2.3.

4.1.4. FIFOwReadForwarding

The FIFOwReadForwarding components is a FIFO buffer which is extended by a read forwarding mechanism. This mechanism enables to check if an entry regarding a given memory location is present in the buffer and if so to obtain that entries value. An implementation is given as Quartz code in Appendix A.2.4.

4.1.5. MemUnit

The MemUnit component represents a memory unit which stores values to a given location and retrieves the last written values on a read operation. An implementation is given as Quartz code in Appendix A.2.5.

4.1.6. MemUnitSingleCell

MemUnitSingleCell behaves equivalent to MemUnit but consists only of one memory cell which corresponds to a single memory location. An implementation is given as Quartz code in Appendix A.2.6.

4.1.7. Store Buffer (SB)

Store buffers (SB) consist of a MemUnitSingleCell component and additional logic. It receives memory operations and inserts writes into the MemUnitSingleCell and forwards them to the arbiter if requested. On received read operations it checks the MemUnitSingleCell if it contains a write to that memory location and returns the latest one if it exists and otherwise forwards the read to the arbiter if requested.

4.1.8. Distributor (Dist)

A distributor (Dist) is responsible for distributing memory operations to the connected components while respecting some rules. As these rules differ from memory model to memory model, the semantics and implementations differ for each model.

4.1.9. Arbiter

The arbiter is typically the characterising component of a memory model. It determines the order in which memory operations are passed from the connected components based on consistency model dependent rules.

If it chooses a component which is not ready to deliver an operation (e.g. in case of an empty FIFO) the arbiter is considered to idle and to make a new choice in the next cycle. This behaviour could be optimized but is required to achieve the completeness property.

The arbiter's behaviour of most models differ significantly and therefore the implementations are model specific.

4.1.10. Receiver (Rec)

A receiver (Rec) is responsible to receive read results from multiple components and to deliver them to its process.

4.2. Local Consistency Reference Machine

4.2.1. Structure

The structure of the local consistency reference machine is shown in Figure 4.1 for a given set P of n processes and m memory locations. For each process $P_i \in P$ the memory system has a distributor $Dist_i$, an arbiter $Arbiter_i$, a memory unit Mem_i and n different FIFO buffers $FIFO_{i,j}, j \in \{1 \dots n\}$ and an additional FIFO buffer $FIFOloop_i$. A distributor $Dist_i$ broadcasts received writes to all corresponding FIFOs $FIFO_{j,i}, j \in \{1 \dots n\}$, and sends all received reads to the FIFO $FIFO_{i,i}$. The arbiters choose non-deterministically from the connected FIFOs to read from. If $Arbiter_i$ selected $FIFO_{i,i}$ the operation is passed to the memory unit. If any other $FIFO_{i,j}, i \neq j$ or $FIFOloop_i$ is selected then the arbiter chooses non-deterministically to either pass it to the memory unit or to insert it in $FIFOloop_i$.

4.2.2. Correctness

Theorem 4.1. *The given reference machine is correct: All executions it may produce for a given input are Local consistent.*

Proof. For each process, its arbiter generates a SerialView while maintaining $<_i$. The own memory commands are kept in order in the FIFO and are sent directly in order to the memory if the corresponding FIFO is selected. The SerialView covers all own operations and all write operations as no write is lost, writes from other processes are inserted into their corresponding FIFO and either send to the memory unit or put in the feedback FIFO until further processing from there. □

4.2.3. Completeness

Theorem 4.2. *The given reference machine is complete: For a given input, it covers every Local consistent execution.*

Proof. Consider a local consistent execution (by Definition 3.2):
$\forall_{i \in P} \exists SerialView\,(<_i|\ (*, i, *, *) \cup (w, *, *, *))$
As the arbiter selects nondeterministically, it may use its process SerialView (which must exist according to the definition) as selection criteria: If according to the SerialView, operation $\widetilde{o}$ is to be processed next, it moves operations from the corresponding FIFO into the feedback FIFO until it fetches $\widetilde{o}$ which it passed to the memory unit. Using this selection order, the resulting writes-to order $\mapsto$ is the same as the one of the assumed

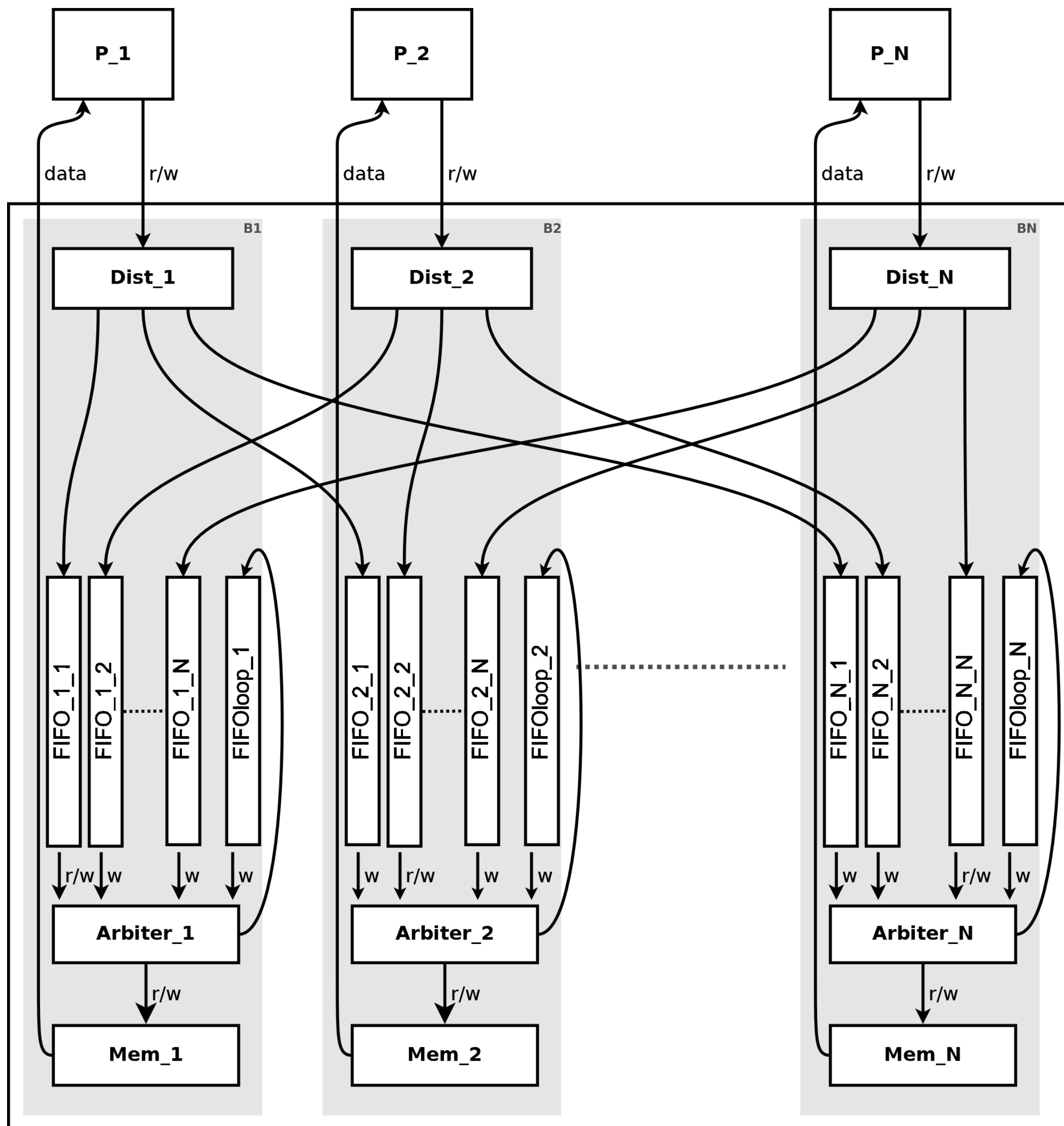


Figure 4.1.: Local consistency reference machine

execution. Therefore every Local consistent execution is covered by the reference machine. □

4.3. Slow Consistency Reference Machine

4.3.1. Structure

The structure of the slow consistency reference machine is shown in Figure 4.2 for a given set P of n processes and m memory locations. For each process $P_i \in P$ the memory system has a distributor $Dist_i$ and n different distributors $Dist_{i,j}, j \in \{1 \dots n\}$, an arbiter $Arbiter_i$, a memory unit Mem_i and $n \times m$ different FIFO buffers $FIFO_{i,j,k}, j \in \{1 \dots n\}, k \in \{1 \dots m\}$. A distributor $Dist_i$ broadcasts received writes to all corresponding distributors $Dist_{j,i}, j \in \{1 \dots n\}$, and sends all received reads to the distributor $Dist_{i,i}$. A sub-distributor $Dist_{i,j}$ broadcasts received memory commands for memory cell k to the corresponding FIFO $FIFO_{i,j,k}$. The arbiters choose non-deterministically one of the connected FIFOs to read from.

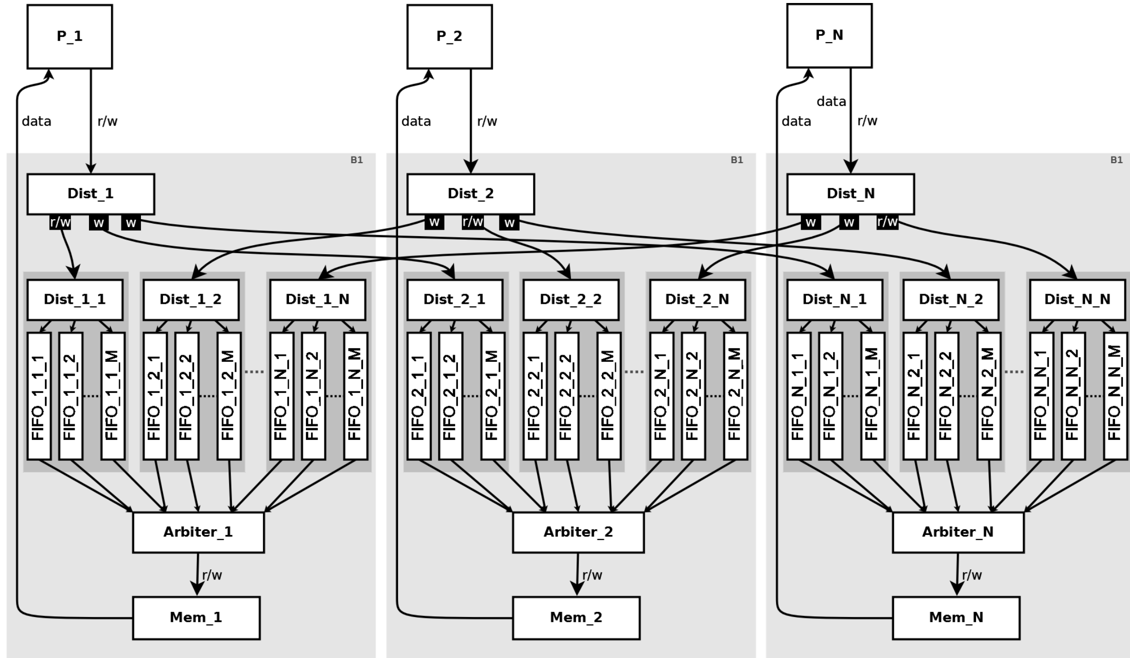


Figure 4.2.: Slow consistency reference machine

4.3.2. Correctness

Theorem 4.3. *The given reference machine is correct: All executions it may produce for a given input are slowly consistent.*

Proof. The distributor fills the FIFOs in the order of the incoming memory operations. Usage of FIFO buffers ensures by construction that the read and write operations of each process are kept in order for each memory location (maintains $\leq_{PO}$

$|(*, i, x, *) \cup (w, *, x, *))$. The arbiter takes elements from the top of a FIFO buffer and issues the operation to the memory unit. Therefore, the memory unit has a serial view on all processes' write operations and the read operations of its corresponding process, and because this property holds for each memory unit, the execution, consisting of (P,V,O,$\leq_{PO}$,$\mapsto$), is slowly consistent according to Definition 3.3. □

4.3.3. Completeness

Theorem 4.4. *The given reference machine is complete: For a given input, it covers every slowly consistent execution.*

Proof. Consider a slowly consistent execution (by Definition 3.3):
$\forall_{i \in P, x \in V} \exists SerialView(\leq_{PO} |(*, i, x, *) \cup (w, *, x, *))$
If each arbiter uses the SerialViews (one for each memory location) of its process (which exist according to the definition) as selection order (which is a valid selection order because the arbiter chooses non-deterministically), then the resulting writes-to order $\mapsto$ is the same as the one of the assumed execution. ↯ Therefore every slowly consistent execution is covered by the reference machine. □

4.4. PRAM Consistency Reference Machine

M_{PRAM} shows a possible implementation of PRAM consistency (Section 3.3). The model suits for distributed systems as seen in M_{PRAM} because each process has its own local memory. Another advantage for use in distributed system is the lack of transmission delay dependencies: no process has to wait for the arrival of another process' write to proceed.

4.4.1. Structure

The structure of the PRAM consistency reference machine is shown in Figure 4.3 for a given set P of n processes. For each process $P_i \in P$ the memory system has a distributor $Dist_i$, an arbiter $Arbiter_i$, a memory unit Mem_i and n different FIFO buffers $FIFO_{i,j}, j \in \{1 \ldots n\}$. A distributor $Dist_i$ broadcasts received writes to all corresponding FIFOs $FIFO_{j,i}, j \in \{1 \ldots n\}$, and sends all received reads to the FIFO $FIFO_{i,i}$. The arbiters choose non-deterministically from the connected FIFOs to read from.

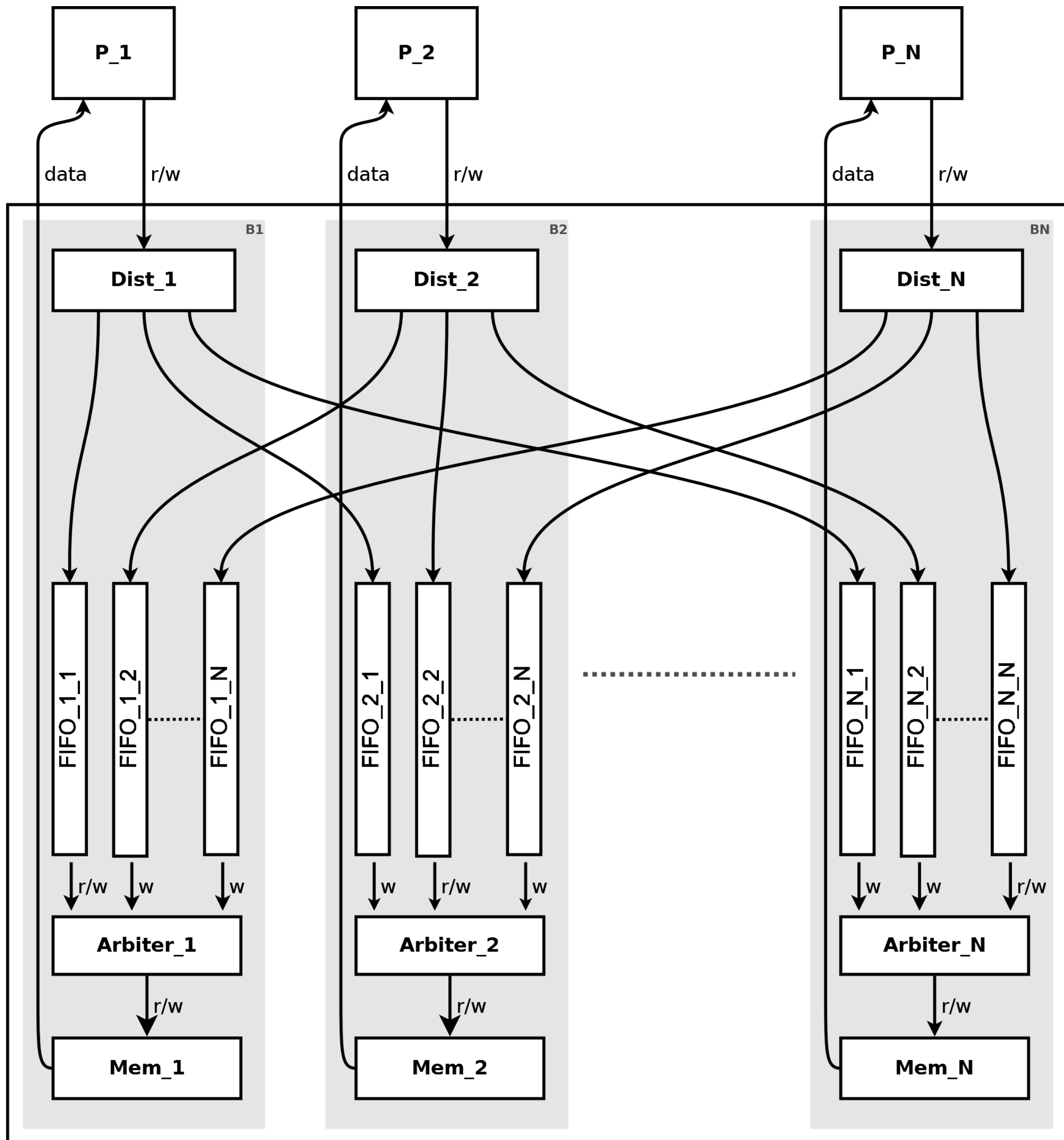


Figure 4.3.: M_{PRAM} - PRAM consistency reference machine

4.4.2. Correctness

Theorem 4.5. *The given reference machine is correct: All executions it may produce for a given input are PRAM consistent.*

Proof. Using FIFO buffers ensures by construction that the read and write operations of each process are kept in order (maintains $\leq_{PO}$). The arbiter takes elements from the top of a FIFO buffer and issues the operation to the memory unit. Therefore, the memory unit has a serial view on write operations of all processes and the read operations of its

corresponding process. Due to this property holds for each memory unit, the execution, consisting of (P,V,O,$\leq_{PO}$,$\mapsto$), is PRAM consistent according to the Definition 3.5. □

4.4.3. Completeness

Theorem 4.6. *The given reference machine is complete: For a given input, it covers every PRAM consistent execution.*

Proof. Consider a PRAM consistent execution (by Definition 3.5):
$\forall_{i \in P} \exists SerialView(\leq_{PO} |(*, i, *, *) \cup (w, *, *, *))$
If each arbiter uses the SerialView of its process (which exists according to the definition) as selection order (which is a valid selection order because the arbiter chooses non-deterministically), then the resulting writes-to order $\mapsto$ is the same as the one of the assumed execution. ↯ Therefore every PRAM consistent execution is covered by the reference machine. □

4.5. Cache Consistency Reference Machine

M_{Cache} implements Cache consistency (Section 3.5) in a comprehensible way as for each variable an arbiter generates the variable order.

4.5.1. Structure

The structure of the cache consistency reference machine is shown in Figure 4.4 for a given set P of n processes. For each process $P_i \in P$ the memory system has a distributor $Dist_i$, a receiver Rec_i and m different FIFO buffers $FIFO_{i,j}, j \in \{1 \ldots m\}$. For each memory cell M_j the memory system provides a memory unit Mem_j and an arbiter $Arbiter_j$. The memory units represent one memory cell, writes are immediately written to it when they are received and the result of a read is sent to the corresponding processes receiver Rec_i. A distributor $Dist_i$ passes the received memory command for memory cell M_j to the corresponding $FIFO_{j,i}$. The receiver Rec_i receives reads for its process and sends them to the processes data interface. The arbiters choose non-deterministically from the connected FIFOs to read from.

4.5.2. Correctness

Theorem 4.7. *The given reference machine is correct: For a given input, all executions it may produce are cache consistent.*

Proof. Usage of FIFO buffers ensures by construction that the read and write operations regarding a specific memory location of each process are kept in order (maintains $\leq_{PO}$ per variable). The arbiter $Arbiter_j$ takes elements from the top of the FIFO buffers which hold the operations for the memory location j and issues the operation to the memory

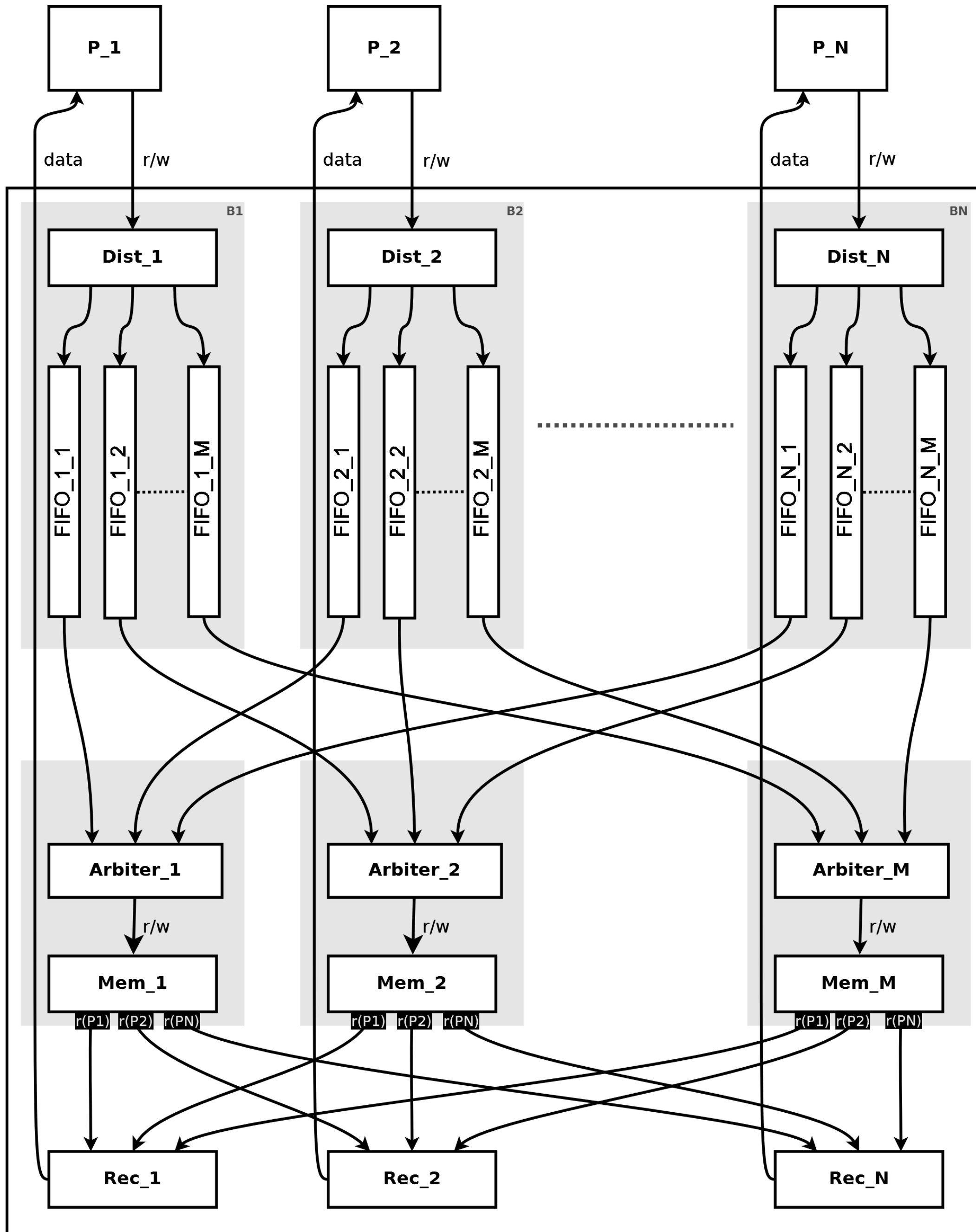


Figure 4.4.: M_{Cache} - Cache consistency reference machine

unit of that memory location. Therefore that memory unit has a serial view on all read and write operations regarding its memory location, and because this property holds for each memory unit and location, the execution consisting of (P,V,O,$\leq_{PO}$,$\mapsto$) is cache consistent according to the Definition 3.8. □

4.5.3. Completeness

Theorem 4.8. *The given reference machine is complete: For a given input, it covers every cache consistent execution.*

Proof. Consider a cache consistent execution (by Definition 3.8):
$\forall_{x \in V} \exists SerialView(\leq_{PO} |(*, *, x, *))$
If each arbiter uses the SerialView of its memory location (which exists according to the definition) as selection order (which is a valid selection order because the arbiter chooses non-deterministically), then the resulting writes-to order $\mapsto$ is the same as the one of the assumed execution. $\lightning$ Therefore, every cache consistent execution is covered by the reference machine. □

4.6. Causal Consistency Reference Machine

Machine M_{Causal} is a reference machine for causal consistency (Section 3.8). Causal consistency requires the machine to keep track of causal dependencies created by writes following reads. This tracking is achieved by introducing clocks which represents the progress a process already observed from other processes.

4.6.1. Structure

The structure of the causal consistency reference machine is shown in Figure 4.5 for a given set P of n processes. It is based on the 'Simple Algorithm' described by Ahamad et. al [ABHN91]. For each process $P_i \in P$, the memory system has an arbiter $Arbit_i$, a memory unit Mem_i and $n-1$ FIFOs $FIFO_{i,j}, j \in \{1 \dots m\}, i \neq j$. The arbiters hold a clock vector $t_i \in \mathbb{N}^n$ which is used to determine the execution order of received writes and is appended to the writes sent to other processes. Before 'sending' writes to the other processes, the arbiter increases the clock vector's value of the entry corresponding to its process $t_i[i]$. Upon 'retrieving' a write from an other process, the arbiter updates the clock vector's value of the entry corresponding to the sending process $t_i[j]$. A write is only retrieved from $FIFO_{i,j}$ if its clock is lower or equal than the arbiters clock with the writes clock entry replaced: $t_w[k] \leq t_i[k], k \neq j$

4.6.2. Correctness

Theorem 4.9. *The given reference machine is correct: For a given input, all executions it may produce are causally consistent.*

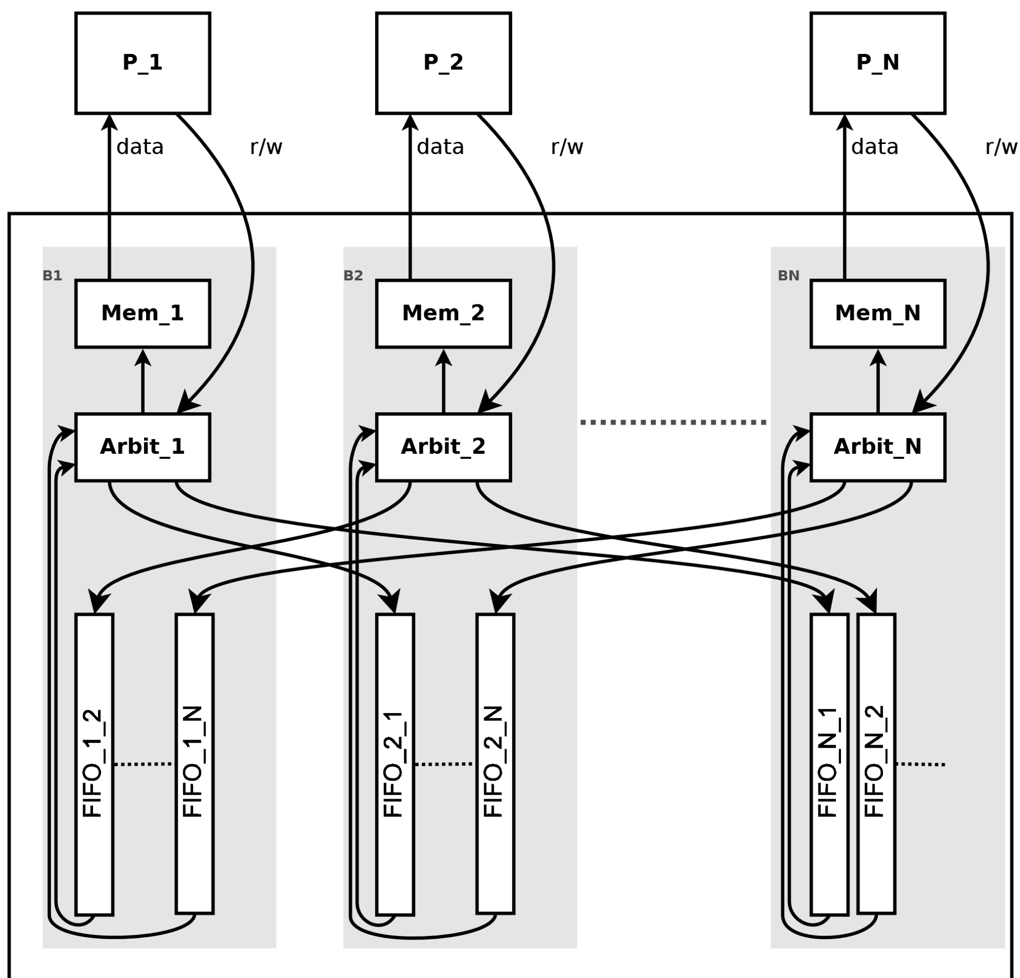


Figure 4.5.: M_{Causal} - Causal consistency reference machine

Proof. The order in which the operations are passed to a process memory unit provides a SerialView on all writes and the process' own reads.

Assume that an execution produces by the machine is not causal correct. Then, the given SerialView must violate either $<_{PO}$ or $<_{WO}$. As the arbiter forwards a process' own operations inorder to its memory location, the SerialView clearly respects $<_i$. As the others processes' writes are inserted inorder into FIFOs and passed to the memory unit inorder from the FIFOs the SerialView respects $<_{PO}$ as well. If the SerialView violates $<_{WO}$, then there exists a write w_1 which writes to a read r process ordered before another write w_2 with the SerialView ordering w_2 before w_1. If process P_i's w_1 was read by the process P_j before it issues w_2 then P_j increased its clock value $t_j[i]$ and w_2 was sent with a clock vector which contained the new value. Another process P_k only can read w_2 if its clock vector value is greater or equal to the value w_2 was tagged with: $t_k[i] \geq t_j[i]$. But as only writes received from P_i can increase $t_k[i]$ and writes are tagged with increasing clock values regarding process order and as shown before writes appear in process order in the SerialView, $<_{WO}$ could not have been violated. ↯ Therefore the assumption must be incorrect and all possible executions the machine can generate are causal correct. □

4.6.3. Completeness

Theorem 4.10. *The given reference machine is complete: For a given input, it covers every causally consistent execution.*

Proof. Assumption: There exists a causally consistent execution (P,V,O,$\leq_{PO}$,$\mapsto$) which cannot be covered by the machine.
As the execution is causally consistent, we know that (by Definition 3.16):
$\forall_{i \in P} \exists SerialView \left(<_i \cup <_{PO} \cup <_{WO} | (*, i, *, *) \cup (w, *, *, *) \right)$
If each arbiter uses the SerialView of its memory location (which exists according to the definition) as selection order, then the resulting writes-to order $\mapsto$ is the same as the one of the assumed execution. It remains to show that the selection order is a valid one. Since the arbiter may choose non-deterministically, the only restriction is its clock vector. If it is not a valid selection order for process i then the operation to select next would need to be a write from another process as all own operations may be selected without checking the clock vector. Without loss of generality, we may say that this operation is a write from process j called w_1. The write w_1 would need to have a non native clock value which is greater than process i's corresponding clock vector value: $t_{w_1}[k] > t_i[k], k \neq j$. As clock vector values are only increased if a write is performed, that would imply that process j observed a write w_2 from process k before issuing w_1 and that process i did not observe w_2 before w_1. But that would contradict $<_{WO}$ and therefore the assumption must be incorrect. ↯
Therefore every causally consistent execution is covered by the reference machine. □

4.6.4. Alternative

To improve performance, the presented machine could be modified to write new clock values upon retrieved writes to a temporary clock vector instead of updating clock values directly and to update a clock value with its temporary counterpart whenever the process reads from that location. An equivalent behaviour is already covered by the given machine via non-determinism. However, the modification would allow a process to retrieve some writes, which would have not been allowed in the previously presented machine.

4.7. Processor (PC-G) Consistency Reference Machine

4.7.1. Structure

The structure of the processor consistency (PC-G) reference machine is shown in Figure 4.6 for a given set P of n processes. For each process $P_i \in P$, the memory system has M FIFOs $FIFO_{i,x}$, one for each memory cell x, an arbiter $Arbiter_i$ and a memory unit Mem_i. The system has an additional arbiter which non-deterministically takes memory commands from the processes and inserts them into the corresponding FIFOs. Reads from process i to memory cell x are inserted into the process' $FIFO_{i,x}$ only, writes to memory cell x are inserted into all $FIFO_{i,x}, i \in P$. The main arbiter keeps a clock vector $t_i \in \mathbb{N}^n, i \in P$ and tags each write from process i with a tuple (i, t_i) and increases t_i after distributing the write. The processes' arbiters $Arbiter_i$ keep each a clock vector $t_{i,j} \in \mathbb{N}^n, i, j \in P$, too. They select non-deterministically one of the FIFOs to read from but only pop an element from the selected FIFO if its tag's clock is the next element to be processed for that process: $clock\big((k,t)\big) = t_{i,k} + 1$

4.7.2. Correctness

Theorem 4.11. *The given reference machine is correct: All executions it may produce for a given input are PC-G consistent.*

Proof. The order in which the main arbiter passes commands to its corresponding FIFOs cells corresponds to a total order $<_x$ on all memory commands regarding this memory address. All of a process' write operations are tagged with a steadily increasing counter. As this counter reflects their order in $<_{PO}$ and the operations are only passed to the memory units in that particular order, the $<_{PO}$ is maintained. Therefore the processes arbiter construct each a serial view on all read operations of their corresponding process and all processes' write operations which respects $<_{PO}$ and $\bigcup_{x \in V} <_x$ and as a consequence the executions of the reference machine are Processor consistent. □

4.7.3. Completeness

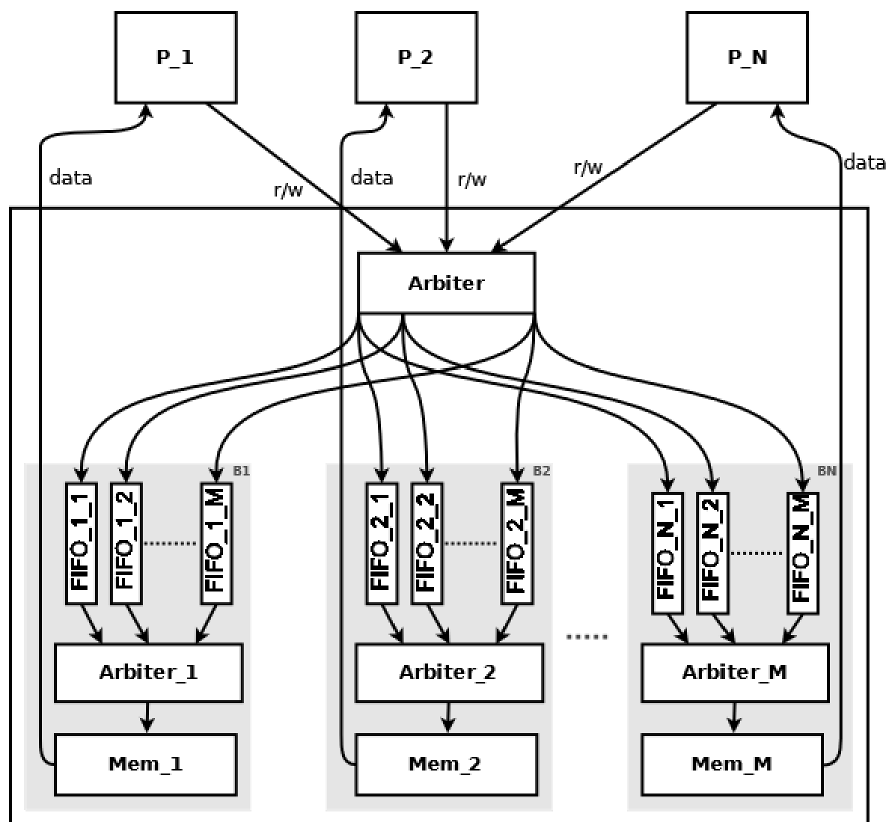


Figure 4.6.: M_{PC} - Processor consistency reference machine

Theorem 4.12. *The given reference machine is complete: For a given input, it covers every PC-G consistent execution.*

Proof. Consider a PC-G consistent execution (by Definition 3.17):
$\forall_{x \in V} \exists <_x = SerialView\,(<_{PO}|(*, *, x, *))$
$\wedge \forall_{i \in P} \exists SerialView\,((\cup_{x \in V} <_x) \bigcup <_{PO}|(*, i, *, *) \cup (w, *, *, *))$.
If the main arbiter uses the orders $<_x$ as selection criteria (it only selects an operation if its predecessor in $<_x$ has already been passed to the FIFOs before) then the FIFOs maintain $<_x$. As the processes' arbiters use an operations' tag as selection criteria they maintain $<_{PO}$ by the way the tags are generated. Therefore the execution is covered by the machine. The assumption must be incorrect.
Therefore every PC-G consistent execution is covered by the reference machine. □

4.8. PSO Consistency Reference Machine

4.8.1. Structure

M_{PSO} shown in Figure 4.7 illustrates the reference machine for PSO consistency. The reference machine consists of M store buffers $SB_{i,j}$ (see Section 4.1.7), one for each memory location, and one distributor for each connected process, an arbiter, and a memory unit. The distributors pass the memory operations to the corresponding store buffer. The arbiter selects non-deterministically between the store buffers and writes the next buffered value back to the main memory unit. If the store buffer wants to issue a read, the read may non-deterministically be processed before the next store buffer entry.

4.8.2. Correctness

Proposition 4.1. The given reference machine is correct: For a given input, all executions it may produce are PSO consistent.

4.8.3. Completeness

Proposition 4.2. The given reference machine is complete: For a given input, it covers every PSO consistent execution.

4.9. TSO Consistency Reference Machine

4.9.1. Structure

M_{TSO} shown in Figure 4.8 illustrates the reference machine for TSO consistency. The reference machine consists of a store buffer SB_i (see Section 4.1.7) for each connected process,

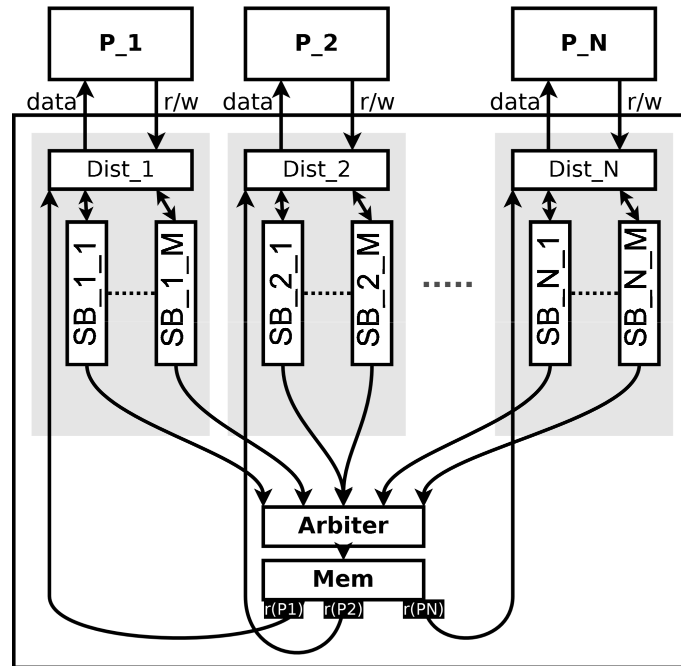


Figure 4.7.: M_{PSO} - PSO consistency reference machine

an arbiter and a memory unit. The store buffers receive operations from the processes and return read results back to them. The arbiter selects non-deterministically between the store buffers and writes the next buffered value back to the main memory unit. If the store buffer wants to issue a read, the read may (chosen non-deterministically) be processed before the next store buffer entry.

4.9.2. Correctness

Proposition 4.3. The given reference machine is correct: For a given input, all executions it may produce are TSO consistent.

4.9.3. Completeness

Proposition 4.4. The given reference machine is complete: For a given input, it covers every TSO consistent execution.

4.10. Sequential Consistency Reference Machine

Both M_{SCb} and M_{SCnb} are implementations of sequential consistency. Sequential consistency has been long time the assumed memory model for programmers as it represents an uniprocessors memory behaviour.

There exist more performant implementations not discussed here, e.g. using caches and the MESI protocol.

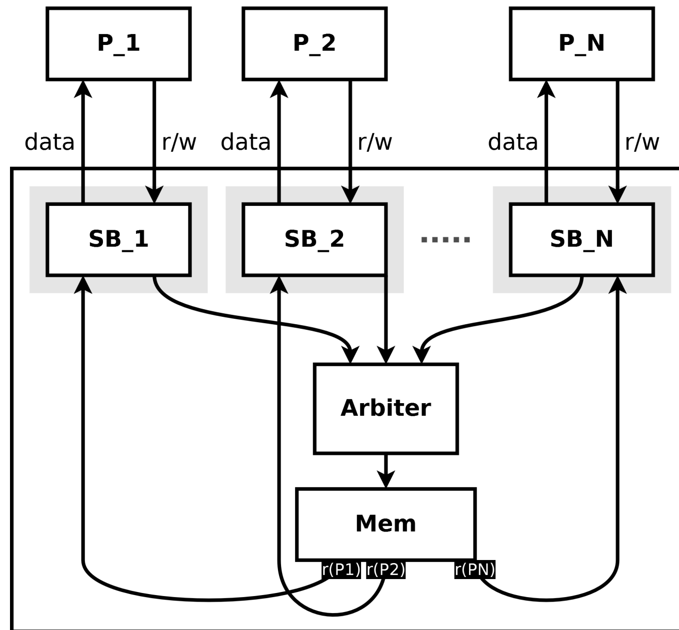


Figure 4.8.: M_{TSO} - TSO consistency reference machine

4.10.1. Structure

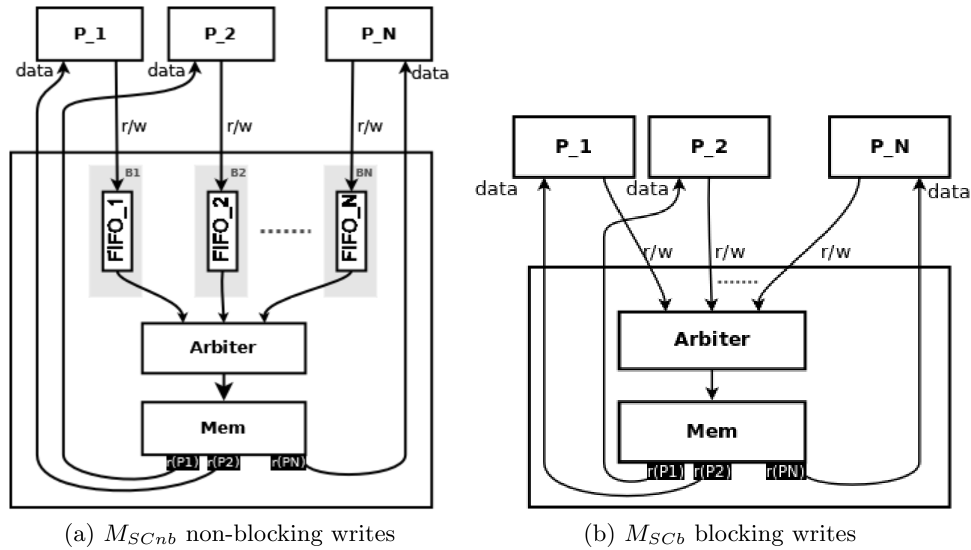
Figure 4.9 depicts two possible implementations for a sequential consistent reference machine. As some of the previously presented reference machines offer non-blocking write access, the machine M_{SCnb} in Figure 4.9a offers them, too. The machine M_{SCb} with blocking reads in Figure 4.9b however is simpler in its design and lacks problems introduced with unbounded buffers.

M_{SCnb} has an unbounded FIFO buffer for each connected process which holds read and write operations. Both machines have an arbiter which selects non-deterministically from its connected components and passes the memory command to the memory unit. If the selected component has no memory command ready to pass then the arbiter stops and starts its selection again in the next cycle. The memory unit passes processed reads to the process it was issued from.

4.10.2. Correctness

Theorem 4.13. *The given reference machines are correct: For a given input, all executions they may produce are sequentially consistent.*

Proof. In M_{SCb}, all memory commands are blocking and are served one after another, in M_{SCnb} they are inserted into a FIFO. Therefore, in both machines the arbiter serves the memory commands without violating $<_{PO}$. The arbiter generates a serialization of all issued memory commands which are issued to the memory unit. Clearly this satisfies



(a) M_{SCnb} non-blocking writes (b) M_{SCb} blocking writes

Figure 4.9.: Sequential Consistency Reference Machines

Definition 3.29. □

4.10.3. Completeness

Theorem 4.14. *The given reference machines are complete: For a given input, they cover every sequentially consistent execution.*

Proof. If an arbitrary execution is sequentially consistent, then following from Definition 3.29 a serial view exists which respects $<_{PO}$. If the arbiter uses this view as oracle for its non-deterministically choices, then the resulting behaviour is equivalent to the considered execution. Therefore, all sequentially consistent executions are covered by the reference machines. □

5. Implementations

5.1. Environment

For the implementation, the synchronous programming language Quartz [Schn09] from the reactive system framework Averest [Embe] is used. Quartz is very similar to Esterel [Berr00], but provides language constructs for asynchronous parallel execution, explicitly defined non-deterministic choices and delayed data manipulation. The Averest framework offers interfaces for simulation, verification (SMV [McMi92]) and synthesis (Verilog [IEEE96, Moor92], SystemC [IEEE05]).

5.2. Interface

The processor interface provided by the built memory systems is the same used by the Abacus processor family built by the Embedded Systems Group of TU Kaiserslautern [Embe]. The Abacus processor series is built using Quartz for educational and research interests and up to now covers a single-cycle, a pipelined and an out-of-order. It uses the instruction set Abacus which is similar to the MIPS architecture. The memory interface consists of an input bus for the address, input buses for the read, write and request memory flags, output buses for the request acknowledgement and memory done flags and an inout bus for data.

5.3. Non-Determinism

Non-determinism is needed to achieve completeness in the given reference machines. If for example two processes issue each a write operation on a sequential reference machine, then there exist two possible executions (each of both operations could be executed before the other one) which are both sequential consistent. Because we want a complete reference machine it must be able to produce all possible executions, and therefore it must use non-determinism.

5.4. (Un)Bounded Buffer

As the FIFOs in the reference machines are considered unbounded the question arises whether that assumption is necessary.

For a finite number of issued memory commands, the FIFOs can be bounded. For example the FIFOs of the PRAM reference machine can be bounded by the length of the longest process: $\max_{i \in P} |\{o \in O | process(o) = i\}|$.

But as real applications have to be considered non-terminating, the FIFOs must be considered

unbounded, iff the reference model should satisfy completeness. If, for example, two processes both issue an unbounded number of memory operations to a PRAM reference machine with FIFOs which have a maximum size of N entries, the reference system can't cover all possible executions any more: If process1 emits $(N+1)$ operations: $w^1_, \ldots, w^1_N$ and process2 emits $(N+1)$ operations: $w^2_0 \ldots w^2_N$ then there exists a PRAM consistent execution for which process1 has the serial view: $w^2_1 < \ldots < w^2_N < w^1_1 < \ldots < w^1_n$ and process2 has the serial view $w^1_1 < \ldots < w^1_N < w^2_1 < \ldots < w^2_n$. To generate this view, the first FIFO belonging to process1 would have to hold all $(N+1)$ commands of process1, or the second FIFO belonging to process2 would have to hold $(N+1)$ commands, but because the FIFOs are bounded by N this execution cannot be produced by the bounded reference machine.

Another interesting question is, what would happen if a given reference machine's buffer would be bounded. The machine would most likely not satisfy completeness any more. But as long as data loss (by trying to insert entries into an already full buffer) is prevented by signalising the buffer status to the process, the constraints defined by the consistency models stay intact, therefore the execution produced remains consistent.

Real implementations make use of finite buffers. Thus, if the reference machines are used as specifications, one can also use bounded buffers of sufficiently large size.

5.5. Improvements

If a process notifies the arbiter when it stops submitting memory commands then the arbiters can exclude the process' corresponding FIFOs from the selection process to increase throughput without violating completeness. An arbiter could exclude empty FIFOs from the selection process to increase throughput, however that would destroy the completeness property. Similarly it could use statistically selection criteria, fairness and liveness properties for the selection process.

6. Conclusions and Further Work

This thesis provides a summary and formal presentation of memory consitency models. It is the basis for future work on this topic and serves as a reference document for educational purposes. The most common memory consistency models were presented in a comparable way and their definitions were explained by examples. Where previous publications only considered a subset of the analysed consistency models, this thesis includes some additional ones, namely PSO, TSO, PRAM-M, PC-DASH, and describes their relationship to each other.

Operational semantics for the most relevant memory consistency models were presented by providing implementations in the language Quartz. The structures and semantics of the implementations were shown to be equivalent to the formal consistency model definition in terms of correctness and completeness.

The reference machines were tested with a set of test cases which resulted in expected behaviour.

As the implementations focused on equivalence with the formal definitions, there may exist smaller or more efficient implementations that are still correct but lack the completeness property. It is worth some effort to implement optimised versions of the memory models in the future. Methods should be considered to show that the behaviour of those optimised versions are covered by the reference machines. Such methods could then be extended to prove correctness of real world implementations of memory systems towards given model reference machines.

This thesis focused on ‘pure’ memory consistency models. In contrast to the described pure consistency models, hybrid models have different classes of operations: special and normal operations. Hybrid models enforce a more restrictive consistency model for special operations than the underlying weak memory model for the normal operations. This way, the programmer can take performance advantages of the weaker consistency model and use the special instructions to enforce the more restrictive model when required to synchronise. View-based formal definitions and reference machines for such hybrid systems may be developed in future work.

A. Quartz Implementations

A.1. Remarks

The mechanism for explicit non-determinism in Quartz ('choose') was replaced by oracle variables which are added to the modules interface. This has been done for verification purposes and to enable the reproducibility of simulation results.

A.2. Shared modules

A.2.1. FIFO

This is a FIFO buffer with elements ($writeFlag, originProcess, memoryTarget, value$). The tuple field *writeFlag* determiness if the entry is a write or read operation, Tuple field *originProcess* contains the ID of the process which issued the operation, Field *memoryTarget* is the memory address the operation operates on and the field *value* contains the value to be written to memory in case of a write operation.

```
package Architecture.ConsistencyModels.Structure;

macro ProcessCount = 3;
macro DataWidth =  8;
macro MemSize = 8;

macro BufferSize = 6;

module FIFO(
   event ?pop,
   event ?push,
   event !isempty,
   event isfull,
   // input : writeCommand & target & value
   event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) ?inp,
   // output : writeCommand & target & value
   event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) !outp
   ) {

   [BufferSize] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) fifo;
   nat{BufferSize} head;
   nat{BufferSize} nxxt;
   bool empty;

   empty = true;

   always {
      if(empty) {
         emit(isempty);
      }
      if((head==nxxt) & !empty) {
         emit(isfull);
```

```
        }

        if(!empty) {
            outp = fifo[head];
        }

        if(pop & !empty) {
            if(head==BufferSize-1) { next(head) = 0; } else { next(head) = head+1; }

            if(((head==BufferSize-1 & nxxt==0) | (head+1==nxxt))) {
                if(!(push & !isfull)) {
                    next(empty) = true;
                }
            }
        }

        if(push & !isfull) {
            next(empty) = false;
            next(fifo[nxxt]) = inp;

            if(nxxt!=BufferSize-1) {
                next(nxxt) = nxxt+1;
            } else {
                next(nxxt) = 0;
            }
        }
    }
}

drivenby FiFoBufferTest {
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
    inp = (true, 1, 1, 0b00100110);emit(push);pause;
    inp = (false, 1, 2, 0b00100110);emit(push);pause;
    inp = (false, 1, 1, 0b00100110);emit(push);emit(pop);pause;
    inp = (true, 1, 3, 0b00100110);emit(push);pause;
    inp = (true, 1, 3, 0b00100110);emit(push);pause;
    inp = (true, 1, 3, 0b00100110);emit(push);pause;
    inp = (true, 1, 3, 0b00100110);emit(push);pause;
    inp = (true, 1, 3, 0b00100110);emit(push);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
}
```

A.2.2. FIFOwClock

This is a FIFO buffer with elements $(writeFlag, originProcess, memoryTarget, value, clock)$. The interface of FIFO was extended by adding field *clock* which holds a clock value (natural number). The other fields behave like the counterpart in module FIFO.

```
package Architecture.ConsistencyModels.Structure;

macro ProcessCount = 3;
macro DataWidth =  8;
macro MemSize = 8;
```

```
macro MaxClock = 127;

macro BufferSize = 6;

module FIFOwClock(
   event ?pop,
   event ?push,
   event !isempty,
   event isfull,
   // input : writeCommand & target & value & clock
   event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * nat{MaxClock}) →
       ?inp,
   // output : writeCommand & target & value & clock
   event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * nat{MaxClock}) →
       !outp
   ) {

   [BufferSize] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * →
       nat{MaxClock}) fifo;
   nat{BufferSize} head;
   nat{BufferSize} nxxt;
   bool empty;

   empty = true;

   always {
      if(empty) {
         emit(isempty);
      }
      if((head==nxxt) & !empty) {
         emit(isfull);
      }

      if(!empty) {
         outp = fifo[head];
      }

      if(pop & !empty) {
         if(head==BufferSize-1) { next(head) = 0; } else { next(head) = head+1; }

         if(((head==BufferSize-1 & nxxt==0) | (head+1==nxxt))) {
            if(!(push & !isfull)) {
               next(empty) = true;
            }
         }
      }

      if(push & !isfull) {
         next(empty) = false;
         next(fifo[nxxt]) = inp;

         if(nxxt!=BufferSize-1) {
            next(nxxt) = nxxt+1;
         } else {
            next(nxxt) = 0;
         }
      }
   }
}
```

A.2.3. FIFOwClocks

This is a FIFO buffer with elements (*writeFlag*, *originProcess*, *memoryTarget*, *value*, *clocks*). The interface of FIFO was extended by adding field *clocks* which holds a tuple of clock values (natural numbers), one for each process. The other fields behave like the counterpart in module FIFO.

```
package Architecture.ConsistencyModels.Structure;

macro ProcessCount = 3;
macro DataWidth =  8;
macro MemSize = 8;

macro MaxClock = 127;

macro BufferSize = 6;

module FIFOwClocks(
   event ?pop,
   event ?push,
   event !isempty,
   event isfull ,
   // input : writeCommand & origin & target & value
   event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * →
       [ProcessCount]nat{MaxClock}) ?inp,
   // output : writeCommand & origin & target & value
   event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * →
       [ProcessCount]nat{MaxClock}) !outp
   ) {

   [BufferSize] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth} * →
       [ProcessCount]nat{MaxClock}) fifo;
   nat{BufferSize} head;
   nat{BufferSize} nxxt;
   bool empty;

   empty = true;

   always {
      if(empty) {
         emit(isempty);
      }
      if((head==nxxt) & !empty) {
         emit(isfull);
      }

      if(!empty) {
         outp = fifo[head];
      }

      if(pop & !empty) {
         if(head==BufferSize-1) { next(head) = 0; } else { next(head) = head+1; }

         if(((head==BufferSize-1 & nxxt==0) | (head+1==nxxt))) {
            if(!(push & !isfull)) {
               next(empty) = true;
            }
         }
      }

      if(push & !isfull) {
         next(empty) = false;
         next(fifo[nxxt]) = inp;

         if(nxxt!=BufferSize-1) {
```

```
          next(nxxt) = nxxt+1;
        } else {
          next(nxxt) = 0;
        }
      }
    }
}
```

A.2.4. FIFOwReadForwarding

This is a FIFO buffer with elements ($writeFlag, originProcess, memoryTarget, value$). The fields behave like the counterpart in module FIFO.

This buffer additionally offers an interface and mechanisms to retrieve the most recent write's value if available for a given memory address.

```
package Architecture.ConsistencyModels.Structure;

macro ProcessCount = 3;
macro DataWidth =  8;
macro MemSize = 8;

macro BufferSize = 6;

module FIFOwReadForwarding(
  event ?pop,
  event ?push,
  event !isempty,
  event isfull ,
  // input : writeCommand & target & value
  event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) ?inp,
  // output : writeCommand & target & value
  event (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) !outp,

  // Read Forward Mechanisms
  // readIn : valid & address
  event (bool * nat{MemSize}) ?readIn,
  // readOut : success & value
  event (bool * bv{DataWidth}) !readOut
  ) {

  // FIFO variables
    [BufferSize] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) fifo;
    nat{BufferSize} head;
    nat{BufferSize} nxxt;
    bool empty;

  // Read Forward Mechanism variables
    nat{BufferSize} tail;
    event [BufferSize] nat{BufferSize} readpos;
    event nat{BufferSize} headreadpos;
    event [BufferSize] bool readdone;

  // Initialze FIFO empty
  empty = true;

  always {
    if(empty) {
      emit(isempty);
    }
    if((head==nxxt) & !empty) {
      emit(isfull);
    }
```

```
if(!empty) {
   outp = fifo[head];
}

if(pop & !empty) {
   if(head==BufferSize-1) { next(head) = 0; } else { next(head) = head+1; }

   if(((head==BufferSize-1 & nxxt==0) | (head+1==nxxt))) {
      if(!(push & !isfull)) {
         next(empty) = true;
      }
   }
}

if(push & !isfull) {
   next(empty) = false;
   next(fifo[nxxt]) = inp;

   if(nxxt!=BufferSize-1) {
      next(nxxt) = nxxt+1;
   } else {
      next(nxxt) = 0;
   }
}


// Read Forward Mechanism //

//tail = (nxxt==0?BufferSize-1:nxxt-1);
if(nxxt<=0) {
   tail = BufferSize-1;
} else {
   tail = nxxt-1;
}

for(i = 0 .. BufferSize-1) do || {
   //readpos[i] = (tail>=i?tail-i:tail+BufferSize-i);
   if(tail>=i) {
      readpos[i] = tail-i;
   } else {
      readpos[i] = tail+BufferSize-i;
   }
}
if(readIn.0 & !empty) { // If read request and FIFO not empty
   for(i = 0 .. BufferSize-1) do || {
      let(pos = readpos[i])
      {
         if(i == head) { headreadpos = pos; }

         if(pos <= headreadpos) { // exclude invalid entries
            if(pos == 0) { // start from nxt-1 % size
               if(readIn.1 == fifo[i].2) {
                  emit(readdone[0]);
                  readOut = (true, fifo[i].3);
               }
            } else {
               if((!readdone[pos-1]) & readIn.1 == fifo[i].2) {
                  emit(readdone[pos]);
                  readOut = (true, fifo[i].3);
               } else {
                  readdone[pos] = readdone[pos-1];
               }
            }
         }
      }
   }
```

```
        }
    }
}

drivenby StBfBufferTest {
    emit(pop);pause;
    emit(pop);pause;
    readIn = (true,1);
    emit(pop);pause;
    inp = (true, 1, 1, 0b00000001);emit(push);pause;
    inp = (false, 1, 2, 0b00000010);emit(push);pause;
    readIn = (true,2);pause;
    readIn = (true,3);pause;
    inp = (false, 1, 1, 0b00000011);emit(push);emit(pop);pause;
    inp = (true, 1, 3, 0b00000100);emit(push);pause;
    inp = (true, 1, 3, 0b00000101);emit(push);pause;
    inp = (true, 1, 3, 0b00000110);emit(push);pause;
    inp = (true, 1, 3, 0b00000111);emit(push);pause;
    readIn = (true,1);pause;
    readIn = (true,3);pause;
    inp = (true, 1, 3, 0b00001000);emit(push);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
    readIn = (true,3);pause;
    emit(pop);pause;
    readIn = (true,3);pause;
    emit(pop);pause;
    emit(pop);pause;
    emit(pop);pause;
}
```

A.2.5. MemUnit

This is a memory unit which processes memory operations and returns read results. The values are stored in a bit-vector array.

```
package Architecture.ConsistencyModels.Structure;

macro DataWidth =  8;
macro MemSize = 8;

macro ProcessCount = 3;

module MemUnit(
        // input : issue & (writeCommand & origin & target & value)
        event (bool * (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth})) →
            ?arbiterOut,

        // output (doneRead & origin & value)
        event (bool *  nat{ProcessCount} * bv{DataWidth}) !readResult
    ) {

     [MemSize]bv{DataWidth} Mem;

     always {
         immediate await(arbiterOut.0);

       if((arbiterOut.1).0) { // write
          Mem[(arbiterOut.1).2] = (arbiterOut.1).3;
       } else {
          readResult = (true, (arbiterOut.1).1, Mem[(arbiterOut.1).2]);
```

```
      }
    }
}
```

A.2.6. MemUnitSingleCell

A single cell memory unit which processes memory operations and returns read results of a single memory location. The value is stored in a bit-vector.

```
package Architecture.ConsistencyModels.Structure;

macro DataWidth =  8;
macro MemSize = 8;

macro ProcessCount = 3;

module MemUnitSingleCell(
      // input : issue & (writeCommand & origin & target & value)
      event (bool * (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth})) →
          ?arbiterOut,

      // output (doneRead & origin & value)
      event (bool *  nat{ProcessCount} * bv{DataWidth}) !readResult
   ) {

    bv{DataWidth} Mem;

    always {
        immediate await(arbiterOut.0);

      if((arbiterOut.1).0) { // write
         Mem = (arbiterOut.1).3;
      } else {
         readResult = (true, (arbiterOut.1).1, Mem);
      }
    }
}
```

A.3. RefLocal : Local Consistency Reference Machine

```
package Architecture.ConsistencyModels.RefLocal;

import Architecture.ConsistencyModels.Structure.FIFO;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth =  8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefLocal(
   // address for memory access
   event [ProcessCount] nat{MemSize} ?adrBus,
   // data for memory access
   event [ProcessCount] bv{DataWidth} dataBus,
   // whether data is read or written to memory
   event [ProcessCount] bool ?readMem,
   event [ProcessCount] bool ?writeMem,
   // signals for memory transaction
   event [ProcessCount] bool ?reqMem,
```

```
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,

    // processor terminated
    [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event [ProcessCount] nat{ProcessCount+1} ?oracle,
    event [ProcessCount] nat{2} ?oracle2
) {
    // FIFO
        event [ProcessCount][ProcessCount] bool FIFOpop;
        event [ProcessCount][ProcessCount] bool FIFOpush;
        event [ProcessCount][ProcessCount] bool FIFOisempty;
        event [ProcessCount][ProcessCount] bool FIFOisfull;
        // input : writeCommand & origin & target & value
        event [ProcessCount][ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOinp;
        // output : writeCommand & origin & target & value
        event [ProcessCount][ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOoutp;

        event [ProcessCount] bool someFIFOfull;

    // FIFOloop
        event [ProcessCount] bool FIFOloopPop;
        event [ProcessCount] bool FIFOloopPush;
        event [ProcessCount] bool FIFOloopIsempty;
        event [ProcessCount] bool FIFOloopIsfull;
        // input : writeCommand & origin & target & value
        event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) →
            FIFOloopInp;
        // output : writeCommand & origin & target & value
        event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) →
            FIFOloopOutp;

    // Arbiter
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) arbiterTemp;

    // Mem
        // memIn : valid/issue & (writeCommand & origin & target & value)
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) memIn;
        // readResult : valid & origin & value
        event [ProcessCount] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

    for(i = 0 .. ProcessCount-1) do || {
        always { // Distributor (broadcasts write to all connected FIFOs and sends →
            reads to own FIFO)
            if(reqMem[i] & !someFIFOfull[i]) {
                emit(ackMem[i]);
                if(writeMem[i]) {
                    emit(doneMem[i]);
                }

                for(j = 0 .. ProcessCount-1) {
                    if((j==i) | writeMem[i]) {
                        // REMARK: notify mixed indices.
                        FIFOpush[j][i] = true;
                        FIFOinp[j][i] = (writeMem[i], i, adrBus[i], dataBus[i]);
                    }
                }
            }

        }
```

```
        ||
        for(j = 0 .. ProcessCount-1) do || {
           always {
              if(FIFOisfull[j][i]) {
                 emit(someFIFOfull[i]);
              }
           }
           ||
           fifo: FIFO(FIFOpop[i][j], FIFOpush[i][j], FIFOisempty[i][j], →
               FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[i][j]);
        }
        ||
        fifoloop: FIFO(FIFOloopPop[i], FIFOloopPush[i], FIFOloopIsempty[i], →
            FIFOloopIsfull[i], FIFOloopInp[i], FIFOloopOutp[i]);
        ||
        always { // Arbiter ("Shuffle" others, pass own)
           let(o1 = oracle[i])
           let(o2 = oracle2[i])
           {
              //choose(o1 = 0 .. ProcessCount) {
                 if(o1 < ProcessCount) { // read from FIFOs
                    if(!FIFOisempty[i][o1]) {
                       arbiterTemp[i] = (true, FIFOoutp[i][o1]);
                       emit(FIFOpop[i][o1]);
                    }
                 } else { // read from feedback
                    if(!FIFOloopIsempty[i]) {
                       arbiterTemp[i] = (true, FIFOloopOutp[i]);
                       emit(FIFOloopPop[i]);
                    }
                 }
              //}
              if(arbiterTemp[i].0) {
                 //choose(o2 = 0 .. 1) {
                    if(o1 == i | o2 == 0) { // pass to Mem
                       memIn[i] = arbiterTemp[i];
                    } else { // feedback
                       if(!FIFOloopIsfull[i]) {
                          FIFOloopInp[i] = arbiterTemp[i].1;
                          emit(FIFOloopPush[i]);
                       }
                    }
                 //}
              }
           }
        }
        ||
        memunit: MemUnit(memIn[i], readResult[i]);
        ||
        always {
           if(readResult[i].0) {
              emit(doneMem[i]);
              dataBus[i] = readResult[i].2;
           }
        }
    }
}
```

A.4. RefSlow : Slow Consistency Reference Machine

```
package Architecture.ConsistencyModels.RefSlow;

import Architecture.ConsistencyModels.Structure.FIFO;
import Architecture.ConsistencyModels.Structure.MemUnit;
```

```
macro DataWidth =  8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefSlow(
  // address for memory access
  event [ProcessCount] nat{MemSize} ?adrBus,
  // data for memory access
  event [ProcessCount] bv{DataWidth} dataBus,
  // whether data is read or written to memory
  event [ProcessCount] bool ?readMem,
  event [ProcessCount] bool ?writeMem,
  // signals for memory transaction
  event [ProcessCount] bool ?reqMem,
  event [ProcessCount] bool ackMem,
  event [ProcessCount] bool !doneMem,

  // processor terminated
  [ProcessCount] bool ?terminated,

  // oracle (choose replacement)
  event [ProcessCount] nat{ProcessCount} ?oracle,
  event [ProcessCount] nat{MemSize} ?oracle2
) {
  // FIFO
    event [ProcessCount][ProcessCount][MemSize] bool FIFOpop;
    event [ProcessCount][ProcessCount][MemSize] bool FIFOpush;
    event [ProcessCount][ProcessCount][MemSize] bool FIFOisempty;
    event [ProcessCount][ProcessCount][MemSize] bool FIFOisfull;
    // input : writeCommand & origin target & value
    event [ProcessCount][ProcessCount][MemSize] (bool * nat{ProcessCount} * →
        nat{MemSize} * bv{DataWidth}) FIFOinp;
    // output : writeCommand & origin target & value
    event [ProcessCount][ProcessCount][MemSize] (bool * nat{ProcessCount} * →
        nat{MemSize} * bv{DataWidth}) FIFOoutp;

    event [ProcessCount] bool someFIFOfull;

  // Memory Units
    // memIn : valid/issue & (writeCommand & origin & target & value)
    event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
        bv{DataWidth})) memIn;
    // readResult : valid & origin & value
    event [ProcessCount] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

  for(i = 0 .. ProcessCount-1) do || {
    always {
      // Distributor: broadcasts writes to all connected FIFOs
          // and sends reads to own FIFO
      // SubDistributor (splits memory cells)
      if(reqMem[i] & !someFIFOfull[i]) {
        emit(ackMem[i]);
        if(writeMem[i]) {
          emit(doneMem[i]);
        }
        for(j = 0 .. ProcessCount-1) {
          if((j==i) | writeMem[i]) {
            // REMARK: notify mixed indices.
            FIFOpush[j][i][ adrBus[i] ] = true;
            FIFOinp[j][i][ adrBus[i] ] =
              (writeMem[i], i, adrBus[i], dataBus[i]);
          }
        }
      }
```

```
        }
        ||
        for(j = 0 .. ProcessCount-1) do || { // FIFO
            for(k = 0 .. MemSize-1) do || {
                always {
                    if(FIFOisfull[i][j][k]) {
                        emit(someFIFOfull[j]);
                    }
                }
                ||
                fifo: FIFO(FIFOpop[i][j][k], FIFOpush[i][j][k], FIFOisempty[i][j][k],
                           FIFOisfull[i][j][k], FIFOinp[i][j][k], FIFOoutp[i][j][k]);
            }
        }
        ||
        always { // Arbiter (Simply choose from NxM components)
            let(o1 = oracle[i])
            let(o2 = oracle2[i])
            {
            //choose(o1 = 0 .. ProcessCount-1) choose(o2 = 0 .. MemSize-1) {
                if(!FIFOisempty[i][o1][o2]) {
                    memIn[i] = (true, FIFOoutp[i][o1][o2]);
                    emit(FIFOpop[i][o1][o2]);
                }
            //}
            }
        }
        ||
        memunit: MemUnit(memIn[i], readResult[i]);
        ||
        always {
            if(readResult[i].0) {
                emit(doneMem[i]);
                dataBus[i] = readResult[i].2;
            }
        }
    }
}
```

A.5. RefPRAM : PRAM Consistency Reference Machine

```
package Architecture.ConsistencyModels.RefPRAM;

import Architecture.ConsistencyModels.Structure.FIFO;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth =  8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefPRAM(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,

    // processor terminated
```

```
    [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event [ProcessCount] nat{ProcessCount} ?oracle
) {
    // FIFO
        event [ProcessCount][ProcessCount] bool FIFOpop;
        event [ProcessCount][ProcessCount] bool FIFOpush;
        event [ProcessCount][ProcessCount] bool FIFOisempty;
        event [ProcessCount][ProcessCount] bool FIFOisfull;
        // input : writeCommand & origin & target & value
        event [ProcessCount][ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOinp;
        // output : writeCommand & origin & target & value
        event [ProcessCount][ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth}) FIFOoutp;

        event [ProcessCount] bool someFIFOfull;

    // Arbiter
        // output : issue & (writeCommand & target & value)
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) arbiterOut;

    // Memory Units
        // memIn : valid/issue & (writeCommand & origin & target & value)
        event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
            bv{DataWidth})) memIn;
        // readResult : valid & origin & value
        event [ProcessCount] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

    for(i = 0 .. ProcessCount-1) do || {
        always {
            // Distributor: broadcasts writes to all connected FIFOs
                // and sends reads to own FIFO
            if(reqMem[i] & !someFIFOfull[i]) {
                emit(ackMem[i]);
                if(writeMem[i]) {
                    emit(doneMem[i]);
                }

                for(j = 0 .. ProcessCount-1) {
                    if((j==i) | writeMem[i]) {
                        // REMARK: notify mixed indices.
                        FIFOpush[j][i] = true;
                        FIFOinp[j][i] = (writeMem[i], i, adrBus[i], dataBus[i]);
                    }
                }
            }

        }
        ||
        for(j = 0 .. ProcessCount-1) do || { // FIFO
            always {
                if(FIFOisfull[j][i]) {
                    emit(someFIFOfull[i]);
                }
            }
            ||
            fifo: FIFO(FIFOpop[i][j], FIFOpush[i][j], FIFOisempty[i][j],
                       FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[i][j]);
        }
        ||
        always { // Arbiter (Simply choose from N components)
            let(o1 = oracle[i]) {
            //choose(o1 = 0 .. ProcessCount-1) {
```

```
            if(!FIFOisempty[i][o1]) {
               memIn[i] = (true, FIFOoutp[i][o1]);
               emit(FIFOpop[i][o1]);
            }
         //}
         }
      }
      ||
      memunit: MemUnit(memIn[i], readResult[i]);
      ||
      always { // returns memory unit read result to connected process
         if(readResult[i].0) {
            emit(doneMem[i]);
            dataBus[i] = readResult[i].2;
         }
      }
   }
}
```

A.6. RefCache : Cache Consistency Reference Machine

```
package Architecture.ConsistencyModels.RefCache;

import Architecture.ConsistencyModels.Structure.FIFO;
import Architecture.ConsistencyModels.Structure.MemUnitSingleCell;

macro DataWidth =  8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefCache(
   // address for memory access
   event [ProcessCount] nat{MemSize} ?adrBus,
   // data for memory access
   event [ProcessCount] bv{DataWidth} dataBus,
   // whether data is read or written to memory
   event [ProcessCount] bool ?readMem,
   event [ProcessCount] bool ?writeMem,
   // signals for memory transaction
   event [ProcessCount] bool ?reqMem,
   event [ProcessCount] bool ackMem,
   event [ProcessCount] bool !doneMem,

   // processor terminated flags
   [ProcessCount] bool ?terminated,

   // oracle (replacement for choose)
   event [MemSize] nat{ProcessCount} ?oracle
) {
   // FIFO
      event [MemSize][ProcessCount] bool FIFOpop;
      event [ProcessCount][MemSize] bool FIFOpush;
      event [MemSize][ProcessCount] bool FIFOisempty;
      event [ProcessCount][MemSize] bool FIFOisfull;
      // input : writeCommand & origin & target & value
      event [ProcessCount][MemSize] (bool * nat{ProcessCount} * nat{MemSize} * →
          bv{DataWidth}) FIFOinp;
      // output : writeCommand & origin & target & value
      event [MemSize][ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * →
          bv{DataWidth}) FIFOoutp;

      event [ProcessCount] bool someFIFOfull;

   // Memory Units
```

```
      // memIn : valid/issue & (writeCommand & origin & target & value)
      event [MemSize] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
          bv{DataWidth})) memIn;
      // readResult : valid & origin & value
      event [MemSize] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

   for(i = 0 .. ProcessCount-1) do || {
      always { // Distributor (Split memory cells)
         if(reqMem[i] & !someFIFOfull[i]) {
            emit(ackMem[i]);
            if(writeMem[i]) {
               emit(doneMem[i]);
            }

            FIFOpush[i][adrBus[i]] = true;
            FIFOinp[i][adrBus[i]] = (writeMem[i], i, adrBus[i], dataBus[i]);
         }
      }
      ||
      for(j = 0 .. MemSize-1) do || {
         always {
            if(FIFOisfull[i][j]) {
               emit(someFIFOfull[i]);
            }
         }
         ||
         // REMARK: notify mixed indices.
         fifo: FIFO(FIFOpop[j][i], FIFOpush[i][j], FIFOisempty[j][i],
                   FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[j][i]);
      }
   }
   ||
   for(i = 0 .. MemSize-1) do || {
      always { // Arbiter (Simply choose from M components)
         let(o1 = oracle[i]) {
         //choose(o1 = 0 .. ProcessCount-1) {
            if(!FIFOisempty[i][o1]) {
               memIn[i] = (true, FIFOoutp[i][o1]);
               emit(FIFOpop[i][o1]);
            }
         //}
         }
      }
      ||
      memunit: MemUnitSingleCell(memIn[i], readResult[i]);
      ||
      always {
         if(readResult[i].0) {
            emit(doneMem[readResult[i].1]);
            dataBus[readResult[i].1] = readResult[i].2;
         }
      }
   }
}
```

A.7. RefCausal : Causal Consistency Reference Machine

```
package Architecture.ConsistencyModels.RefCausal;

import Architecture.ConsistencyModels.Structure.FIFOwClocks;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth = 8;
macro MemSize = 8;
```

```
macro ProcessCount = 3;

macro MaxClock = 127;

module RefCausal(
   // address for memory access
   event [ProcessCount] nat{MemSize} ?adrBus,
   // data for memory access
   event [ProcessCount] bv{DataWidth} dataBus,
   // whether data is read or written to memory
   event [ProcessCount] bool ?readMem,
   event [ProcessCount] bool ?writeMem,
   // signals for memory transaction
   event [ProcessCount] bool ?reqMem,
   event [ProcessCount] bool ackMem,
   event [ProcessCount] bool !doneMem,

   // processor terminated
   event [ProcessCount] bool ?terminated,

   // oracle (choose replacement)
   event [ProcessCount] nat{ProcessCount} ?oracle
) {
   // Clocks
      [ProcessCount][ProcessCount] nat{MaxClock} clocks;
      event [ProcessCount][ProcessCount] nat{MaxClock} tempclocks;
      event [ProcessCount] bool clockgreater;

   // Dist
      event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
          bv{DataWidth} * [ProcessCount]nat{MaxClock})) distIn;
      event [ProcessCount][ProcessCount-1] (bool * (bool * nat{ProcessCount} * →
          nat{MemSize} * bv{DataWidth} * [ProcessCount]nat{MaxClock})) distOut;

   // FIFO
      event [ProcessCount][ProcessCount-1] bool FIFOpop;
      event [ProcessCount][ProcessCount-1] bool FIFOpush;
      event [ProcessCount][ProcessCount-1] bool FIFOisempty;
      event [ProcessCount][ProcessCount-1] bool FIFOisfull;
      // input : writeCommand & target & value
      event [ProcessCount][ProcessCount-1] (bool * nat{ProcessCount} * nat{MemSize} →
          * bv{DataWidth} * [ProcessCount]nat{MaxClock}) FIFOinp;
      // output : writeCommand & target & value
      event [ProcessCount][ProcessCount-1] (bool * nat{ProcessCount} * nat{MemSize} →
          * bv{DataWidth} * [ProcessCount]nat{MaxClock}) FIFOoutp;

      event [ProcessCount] bool someFIFOfull;

   // Mem
      // input: issue & (writeCommand & target & value)
      event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
          bv{DataWidth})) memIn;
      event [ProcessCount] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

   for(i = 0 .. ProcessCount-1) do || {
      always { // Arbiter
         //choose(o1 = 0 .. ProcessCount-1) {
         let(o1 = oracle[i]) {
            if(o1 == i) { // check for r/w
               if(reqMem[i] & !someFIFOfull[i]) {
                  emit(ackMem[i]);

                  // pass memory command to memory
                  memIn[i] = (true, (writeMem[i], i, adrBus[i], dataBus[i]));

                  if(writeMem[i]) {
```

```
                emit(doneMem[i]);

                for(k = 0 .. ProcessCount-1) {
                   if(k==i) {
                      tempclocks[i][k] = clocks[i][k]+1;
                   } else {
                      tempclocks[i][k] = clocks[i][k];
                   }
                }

                // Send write to FIFOs
                distIn[i] = (true,
                          (writeMem[i], i, adrBus[i], dataBus[i],
                           tempclocks[i]));
                next(clocks[i]) = tempclocks[i];
             } else {
                // read : await memory response
                immediate await(readResult[i].0);
             }
          }
       } else if(o1 != i) { // check FIFO
          let(o1m = (o1>=i & o1!=0?o1-1:o1)) {
             if(!FIFOisempty[i][o1m]) {
                let(clock = FIFOoutp[i][o1m].4) {
                   // check if operations clock is leq than own clock
                   for(k = 0 .. ProcessCount-1) {
                      if(k!=o1) {
                         if(clock[k] > clocks[i][k]) {
                            emit(clockgreater[i]);
                         }
                      }
                   }
                   if(!clockgreater[i]) {
                      emit(FIFOpop[i][o1m]);

                      // pass memory command to memory
                      memIn[i] = (true,
                                (FIFOoutp[i][o1m].0, FIFOoutp[i][o1m].1,
                                 FIFOoutp[i][o1m].2, FIFOoutp[i][o1m].3));

                      // copy clock origin component to own clock
                      next(clocks[i][o1]) = clock[o1];
                   }
                }
             }
          }
       }
    }
}
||
always { // DistBroadcastClockedW
   if(distIn[i].0) {
      for(j = 0 .. ProcessCount-1) {
         if(i!=j) {
            let(k = (j<=i & i!=0?i-1:i)) {
               FIFOinp[j][k] = distIn[i].1;
               emit(FIFOpush[j][k]);
            }
         }
      }
   }
}
||
for(j = 0 .. ProcessCount-2) do || {
   fifo: FIFOwClocks(FIFOpop[i][j], FIFOpush[i][j], FIFOisempty[i][j],
                FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[i][j]);
```

```
            if(FIFOisfull[i][j]) {
               emit(someFIFOfull[(i>j?j:j+1)]);
            }
         }
         ||
         memunit: MemUnit(memIn[i], readResult[i]);
         ||
         always { // return read results
            if(readResult[i].0) {
               emit(doneMem[i]);
               dataBus[i] = readResult[i].2;
            }
         }
      }
}
```

A.8. RefProcessor : PC-G Consistency Reference Machine

```
package Architecture.ConsistencyModels.RefProcessor;

import Architecture.ConsistencyModels.Structure.FIFOwClock;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth =  8;
macro MemSize = 8;

macro ProcessCount = 3;

macro MaxClock = 127;

module RefProcessor(
   // address for memory access
   event [ProcessCount] nat{MemSize} ?adrBus,
   // data for memory access
   event [ProcessCount] bv{DataWidth} dataBus,
   // whether data is read or written to memory
   event [ProcessCount] bool ?readMem,
   event [ProcessCount] bool ?writeMem,
   // signals for memory transaction
   event [ProcessCount] bool ?reqMem,
   event [ProcessCount] bool ackMem,
   event [ProcessCount] bool !doneMem,

   // processor terminated
   [ProcessCount] bool ?terminated,

   // oracle (choose replacement)
   event nat{ProcessCount} ?oracle,
   event [ProcessCount] nat{MemSize} ?oracles2
) {

   // FIFO
      event [ProcessCount][MemSize] bool FIFOpop;
      event [ProcessCount][MemSize] bool FIFOpush;
      event [ProcessCount][MemSize] bool FIFOisempty;
      event [ProcessCount][MemSize] bool FIFOisfull;
      // input : writeCommand & origin & target & value & clock
      event [ProcessCount][MemSize] (bool * nat{ProcessCount} * nat{MemSize} * →
          bv{DataWidth} * nat{MaxClock}) FIFOinp;
      // output : writeCommand & origin & target & value & clock
      event [ProcessCount][MemSize] (bool * nat{ProcessCount} * nat{MemSize} * →
          bv{DataWidth} * nat{MaxClock}) FIFOoutp;

      event [MemSize] bool someFIFOfull;
```

```
// Arbiter
   [ProcessCount] nat{MaxClock} mainArbiterClocks;
   [ProcessCount][ProcessCount] nat{MaxClock} subArbiterClocks;
   event [ProcessCount] (bool * (bool * nat{ProcessCount} * nat{MemSize} * →
       bv{DataWidth})) subArbiterOut;

// Mem
   event [ProcessCount] (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

always { // MainArbiter
   //choose(oracle = 0 .. ProcessCount-1) {
      if(reqMem[oracle]) {
         emit(ackMem[oracle]);

         let(adr = adrBus[oracle])
         let(data = dataBus[oracle])
         let(write = writeMem[oracle])
         let(read = readMem[oracle])
         {
            if(write) {
               immediate await(!someFIFOfull[adr]);
               for(i = 0 .. ProcessCount-1) do || {
                  FIFOinp[i][adr] =
                     (true, oracle, adr, data, mainArbiterClocks[oracle]);
                  emit(FIFOpush[i][adr]);
               }
               emit(doneMem[oracle]);
               next(mainArbiterClocks[oracle]) = mainArbiterClocks[oracle]+1;
            } else if(read) {
               immediate await(!FIFOisfull[oracle][adr]);
               FIFOinp[oracle][adr] =
                  (false, oracle, adr, data, mainArbiterClocks[oracle]);
               emit(FIFOpush[oracle][adr]);
            }
         }
      }
   //}
}
||
for(i = 0 .. ProcessCount-1) do || {
   for(j = 0 .. MemSize-1) do || {
      fifo: FIFOwClock(FIFOpop[i][j], FIFOpush[i][j], FIFOisempty[i][j],
                  FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[i][j]);
      ||
      always {
         if(FIFOisfull[i][j]) {
            emit(someFIFOfull[j]);
         }
      }
   }
   ||
   always { // SubArbiter
      //choose(oracle2 = 0 .. ProcessCount-1) {
         let(oracle2 = oracles2[i])
         let(entry = FIFOoutp[i][oracle2])
         if(!FIFOisempty[i][oracle2]) {
            if(entry.0) { // write
               if(subArbiterClocks[i][entry.1] == entry.4) {
                  subArbiterOut[i] = (true, (entry.0,entry.1,entry.2,entry.3));
                  emit(FIFOpop[i][oracle2]);
                  next(subArbiterClocks[i][entry.1]) =
                     subArbiterClocks[i][entry.1]+1;
               }
            } else { // read
               subArbiterOut[i] = (true, (entry.0,entry.1,entry.2,entry.3));
```

```
                emit(FIFOpop[i][oracle2]);
            }
        }
      //}
    }
    ||
    memunit: MemUnit(subArbiterOut[i], readResult[i]);
    ||
    always {
      if(readResult[i].0) {
        emit(doneMem[i]);
        dataBus[i] = readResult[i].2;
      }
    }
  }
}
```

A.9. RefPSO : PSO Consistency Reference Machine

```
package Architecture.ConsistencyModels.RefPSO;

import Architecture.ConsistencyModels.Structure.MemUnit;
import Architecture.ConsistencyModels.Structure.FIFOwReadForwarding;


macro DataWidth =  8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefPSO(
  // address for memory access
  event [ProcessCount] nat{MemSize} ?adrBus,
  // data for memory access
  event [ProcessCount] bv{DataWidth} dataBus,
  // whether data is read or written to memory
  event [ProcessCount] bool ?readMem,
  event [ProcessCount] bool ?writeMem,
  // signals for memory transaction
  event [ProcessCount] bool ?reqMem,
  event [ProcessCount] bool ackMem,
  event [ProcessCount] bool !doneMem,

  // processor terminated
  [ProcessCount] bool ?terminated,

  // oracle (choose replacement)
  event nat{ProcessCount+1} ?oracle,
  event nat{MemSize+1} ?oracle2
) {
  // FIFOwReadForwarding interface variables
    event [ProcessCount][MemSize] bool FIFOpop;
    event [ProcessCount][MemSize] bool FIFOpush;
    event [ProcessCount][MemSize] bool FIFOisempty;
    event [ProcessCount][MemSize] bool FIFOisfull;
    // input : writeCommand & target & value
    event [ProcessCount][MemSize] (bool * nat{ProcessCount} * nat{MemSize} * →
       bv{DataWidth}) FIFOinp;
    // output : writeCommand & target & value
    event [ProcessCount][MemSize] (bool * nat{ProcessCount} * nat{MemSize} * →
       bv{DataWidth}) FIFOoutp;

    // readIn : valid & address
    event [ProcessCount][MemSize] (bool * nat{MemSize}) FIFOreadIn;
    // readOut : success & value
```

```
    event [ProcessCount][MemSize] (bool * bv{DataWidth}) FIFOreadOut;

// Arbiter variables
    // arbiterSelection: process which may proceed | arbiterSelection == →
        ProcessCount means idle
    nat{ProcessCount+1} arbiterSelection;
    // arbiterBufferSelection: process' buffer which may write back | →
        arbiterBufferSelection == MemSize means Read
    nat{MemSize+1} arbiterBufferSelection;

// Mem interface variables
    // memIn : valid/issue & (writeCommand & target & value)
    event (bool * (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth})) memIn;
    // readResult : valid & issuer & value)
    event (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

for(i = 0 .. ProcessCount-1) do || {
    always { // StoreBuffer
        if(reqMem[i]) {
            emit(ackMem[i]);

            if(writeMem[i]) { // write operation
                immediate await(!FIFOisfull[i][adrBus[i]]);
                FIFOinp[i][adrBus[i]] = (true, i, adrBus[i], dataBus[i]);
                emit(FIFOpush[i][adrBus[i]]);
                emit(doneMem[i]);
            }

            if(readMem[i]) { // read operation
                FIFOreadIn[i][adrBus[i]] = (true, adrBus[i]);
                if(FIFOreadOut[i][adrBus[i]].0) {
                    dataBus[i] = FIFOreadOut[i][adrBus[i]].1;
                    emit(doneMem[i]);
                } else {
                    immediate await(arbiterSelection == i
                                & arbiterBufferSelection == MemSize);
                    memIn = (true, (false, i, adrBus[i], dataBus[i]));
                }
            }
        }
    }
    ||
    always { // StoreBuffer write back / flush
        if(arbiterSelection == i & arbiterBufferSelection<MemSize) {
            if(!FIFOisempty[i][arbiterBufferSelection]) {
                memIn = (true, (true, i, arbiterBufferSelection,
                            FIFOoutp[i][arbiterBufferSelection].3));
            }
        }
    }
    ||
    for(j = 0 .. MemSize-1) do || {
        fifo: FIFOwReadForwarding(FIFOpop[i][j], FIFOpush[i][j], FIFOisempty[i][j],
                            FIFOisfull[i][j], FIFOinp[i][j], FIFOoutp[i][j],
                            FIFOreadIn[i][j], FIFOreadOut[i][j]);
    }
}
||
always { // Arbiter
    //choose(oracle = 0 .. ProcessCount) {
        arbiterSelection = oracle;
    //}
    //choose(oracle = 0 .. ProcessCount) {
        arbiterBufferSelection = oracle2;
    //}
}
```

```
    ||
    memunit: MemUnit(memIn, readResult);
    ||
    always { // Distribute Completed Read Operations to the corresponding process
        if(readResult.0) {
            dataBus[readResult.1] = readResult.2;
            emit(doneMem[readResult.1]);
        }
    }
}
```

A.10. RefTSO : TSO Consistency Reference Machine

```
package Architecture.ConsistencyModels.RefTSO;

import Architecture.ConsistencyModels.Structure.MemUnit;
import Architecture.ConsistencyModels.Structure.FIFOwReadForwarding;


macro DataWidth =  8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefTSO(
    // address for memory access
    event [ProcessCount] nat{MemSize} ?adrBus,
    // data for memory access
    event [ProcessCount] bv{DataWidth} dataBus,
    // whether data is read or written to memory
    event [ProcessCount] bool ?readMem,
    event [ProcessCount] bool ?writeMem,
    // signals for memory transaction
    event [ProcessCount] bool ?reqMem,
    event [ProcessCount] bool ackMem,
    event [ProcessCount] bool !doneMem,

    // processor terminated
    [ProcessCount] bool ?terminated,

    // oracle (choose replacement)
    event nat{ProcessCount+1} ?oracle,
    event bool ?oracle2
) {
    // FIFOwReadForwarding interface variables
        event [ProcessCount] bool FIFOpop;
        event [ProcessCount] bool FIFOpush;
        event [ProcessCount] bool FIFOisempty;
        event [ProcessCount] bool FIFOisfull;
        // input : writeCommand & target & value
        event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) →
            FIFOinp;
        // output : writeCommand & target & value
        event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) →
            FIFOoutp;

        // readIn : valid & address
        event [ProcessCount] (bool * nat{MemSize}) FIFOreadIn;
        // readOut : success & value
        event [ProcessCount] (bool * bv{DataWidth}) FIFOreadOut;

    // Arbiter variables
        // arbiterSelection: process which may proceed
            // | arbiterSelection == ProcessCount means idle
        nat{ProcessCount+1} arbiterSelection;
```

```
    // arbiterReadInsteadOfWb: should process read or do a WB
    bool arbiterReadInsteadOfWb;

  // Mem interface variables
    // memIn : valid/issue & (writeCommand & target & value)
    event (bool * (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth})) memIn;
    // readResult : valid & issuer & value)
    event (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

  for(i = 0 .. ProcessCount-1) do || {
    always { // StoreBuffer insert or read
      if(reqMem[i]) {
        emit(ackMem[i]);

        if(writeMem[i]) { // write operation
          immediate await(!FIFOisfull[i]);
          FIFOinp[i] = (true, i, adrBus[i], dataBus[i]);
          emit(FIFOpush[i]);
          emit(doneMem[i]);
        }

        if(readMem[i]) { // read operation
          FIFOreadIn[i] = (true, adrBus[i]);
          if(FIFOreadOut[i].0) {
            dataBus[i] = FIFOreadOut[i].1;
            emit(doneMem[i]);
          } else {
            immediate await((arbiterSelection == i) & arbiterReadInsteadOfWb);
            memIn = (true, (false, i, adrBus[i], dataBus[i]));
          }
        }
      }
    }
    ||
    always { // StoreBuffer write back / flush
      immediate await(arbiterSelection == i & !arbiterReadInsteadOfWb
                 & !FIFOisempty[i]);
      memIn = (true, (true, i, FIFOoutp[i].2, FIFOoutp[i].3));
    }
    ||
    fifo: FIFOwReadForwarding(FIFOpop[i], FIFOpush[i], FIFOisempty[i],
                       FIFOisfull[i], FIFOinp[i], FIFOoutp[i], FIFOreadIn[i], →
                           FIFOreadOut[i]);
  }
  ||
  always { // Arbiter
    //choose(oracle = 0 .. ProcessCount) {
      arbiterSelection = oracle;
    //}

    //choose {
    // arbiterReadInsteadOfWb = true;
    //} else {
    // arbiterReadInsteadOfWb = false;
    //}
    arbiterReadInsteadOfWb = oracle2;
  }
  ||
  memunit: MemUnit(memIn, readResult);
  ||
  always { // Distribute Completed Read Operations to the corresponding process
    if(readResult.0) {
      dataBus[readResult.1] = readResult.2;
      emit(doneMem[readResult.1]);
    }
  }
```

```
}
```

A.11. RefSequential : Sequential Consistency Reference Machine

```
package Architecture.ConsistencyModels.RefSequential;

import Architecture.ConsistencyModels.Structure.FIFO;
import Architecture.ConsistencyModels.Structure.MemUnit;

macro DataWidth =  8;
macro MemSize = 8;

macro ProcessCount = 3;

module RefSequential(
  // address for memory access
  event [ProcessCount] nat{MemSize} ?adrBus,
  // data for memory access
  event [ProcessCount] bv{DataWidth} dataBus,
  // whether data is read or written to memory
  event [ProcessCount] bool ?readMem,
  event [ProcessCount] bool ?writeMem,
  // signals for memory transaction
  event [ProcessCount] bool ?reqMem,
  event [ProcessCount] bool ackMem,
  event [ProcessCount] bool !doneMem,

  // processor terminated
  [ProcessCount] bool ?terminated,

  // oracle (choose replacement)
  event nat{ProcessCount} ?oracle
) {

  // FIFO
    event [ProcessCount] bool FIFOpop;
    event [ProcessCount] bool FIFOpush;
    event [ProcessCount] bool FIFOisempty;
    event [ProcessCount] bool FIFOisfull;
    // input : writeCommand & target & value
    event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) →
        FIFOinp;
    // output : writeCommand & target & value
    event [ProcessCount] (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth}) →
        FIFOoutp;

  // Mem
    // memIn : valid/issue & (writeCommand & origin & target & value)
    event (bool * (bool * nat{ProcessCount} * nat{MemSize} * bv{DataWidth})) memIn;
    // readResult : valid & origin & value
    event (bool * nat{ProcessCount} * bv{DataWidth}) readResult;

  for(i = 0 .. ProcessCount-1) do || {
    fifo: FIFO(FIFOpop[i], FIFOpush[i], FIFOisempty[i],
            FIFOisfull[i], FIFOinp[i], FIFOoutp[i]);
    ||
    always {
      if(reqMem[i] & !FIFOisfull[i]) {
        emit(ackMem[i]);

        FIFOinp[i] = (writeMem[i], i, adrBus[i], dataBus[i]);
        emit(FIFOpush[i]);

        if(writeMem[i]) {
          emit(doneMem[i]);
```

```
                }
            }
        }
    }
    ||
    always {
        let(o1 = oracle) { // Arbiter (Simply choose from N components)
        //choose(o1 = 0 .. ProcessCount-1) {
            if(!FIFOisempty[o1]) {
                memIn = (true, FIFOoutp[o1]);
                emit(FIFOpop[o1]);
            }
        //}
        }
    }
    ||
    memunit: MemUnit(memIn, readResult);
    ||
    always { // Distributor (distributes read results to corresponding process)
        if(readResult.0) {
            emit(doneMem[readResult.1]);
            dataBus[readResult.1] = readResult.2;
        }
    }
}
```

Bibliography

[ABHN91] M. Ahamad, J.E. Burns, P.W. Hutto, and G. Neigher. Causal memory. In S. Toueg, P.G. Spirakis, and L.M. Kirousis, editors, *International Workshop on Distributed Algorithms*, volume 579 of *LNCS*, pages 9–30, Delphi, Greece, 1991. Springer.

[ABJK+93] M. Ahamad, R.A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In L. Snyder, editor, *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 251–260, Velen, Germany, 1993. ACM.

[AdGh96] S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[AdHi93] S.V. Adve and M.D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 4(6):613–624, June 1993.

[Adve93] S.V. Adve. *Designing memory consistency models for shared-memory multiprocessors.* PhD thesis, University of Wisconsin at Madison, 1993. UMI Order No. GAX94-07354.

[BaBe97] J. Bataller and J.M. Bernabéu-Aubán. Synchronized DSM models. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *International Euro-Par Conference (Euro-Par)*, volume 1300 of *LNCS*, pages 468–475, Passau, Germany, 1997. Springer.

[Berr00] G. Berry. The Esterel v5 language primer, July 2000.

[BoPe09] G. Boudol and G. Petri. Relaxed memory models: an operational approach. In Z. Shao and B.C. Pierce, editors, *Principles of Programming Languages (POPL)*, pages 392–403, Savannah, Georgia, USA, 2009. ACM.

[Dijk68] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[Embe] University of Kaiserslautern Embedded Systems Group. The averest system. `http://www.averest.org/`. Accessed: 2013.02.28 `http://www.webcitation.org/6ElcrkFBi`, `http://www.webcitation.org/6Eld430ft`, `http://www.webcitation.org/6Eld430g3`.

[GLLG+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 15–26, Seattle, Washington, USA, 1990. IEEE Computer Society.

[Good91] J.R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Computer Sciences Department, University of Wisconsin-Madison,

February 1991.

[HeSi92] A. Heddaya and H. Sinha. Coherence, non-coherence and local consistency in distributed shared memory for parallel computing. Technical Report BU-CS-92-004, Department of Computer Science, Boston University, 1992. http://www.webcitation.org/6EnH7xvmI.

[HiKV98] L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models – part I: Definitions and comparisons. Technical Report 98/612/03, Department of Computer Science, University of Calgary, 1998.

[HuAh90] P.W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 302–309, Paris, France, 1990. IEEE Computer Society.

[HVML+04] S. Hangal, D. Vahia, C. Manovit, J.J. Lu, and S. Narayanan. TSOtool: A program for verifying memory systems using the memory consistency model. In *International Symposium on Computer Architecture (ISCA)*, pages 114–123, Munich, Germany, 2004. IEEE Computer Society.

[IA313] Intel Corporation. *Intel 64 and IA-32 Architectures: Software Developer's Manual*, January 2013. http://www.intel.com/products/processor/manuals/, http://www.webcitation.org/6EuoyZmHU.

[IEEE96] IEEE. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. New York, New York, USA, 1996. IEEE Std. 1394-1995.

[IEEE05] IEEE. *IEEE Standard SystemC Language Reference Manual*. New York, New York, USA, December 2005. IEEE Std. 1666-2005.

[Lamp78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM (CACM)*, 21(7):558–565, 1978.

[Lamp79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers (T-C)*, 28(9):690–691, September 1979.

[LiSa88] R.J. Lipton and J.S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, 1988.

[LiSa94] R.J. Lipton and J.S. Sandberg. Oblivious memory computer networking. Patent, January 1994. US 5276806.

[LiWo11] A. Linden and P. Wolper. A verification based approach to memory fence insertion in relaxed memory systems. In A. Groce and M. Musuvathi, editors, *Model Checking Software (SPIN)*, volume 6823 of *LNCS*, pages 144–160, Snowbird, Utah, USA, 2011. Springer.

[LLGW+92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[LoCM06] P. Loewenstein, S. Chaudhry, and R. CypherC. Manovit. Multiprocessor memory model verification. Unpublished. Automated Formal Meth-

ods (AFM), FLoC Workshop 2006. http://fm.csl.sri.com/AFM06/papers/4-Loewenstein.pdf, http://www.webcitation.org/6EleUJWBH, August 2006.

[McMi92] K.L. McMillan. The SMV system, symbolic model checking – an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[Moor92] P. Moorby. History of Verilog. *IEEE Design and Test of Computers*, pages 62–63, September 1992.

[Mosb93a] D. Mosberger. Memory consistency models. *ACM SIGOPS: Operating Systems Review*, 27(1):18–26, January 1993.

[Mosb93b] D. Mosberger. Memory consistency models. Technical Report TR 93/11, Department of Computer Science, The University of Arizona, Tucson, Arizona, USA, 1993.

[OwSS09] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 5674 of *LNCS*, pages 391–407, Munich, Germany, 2009. Springer.

[PaPa98] M.S. Papamarcos and J.H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *25 Years of the International Symposia on Computer Architecture (ISCA)*, pages 284–290, Barcelona, Spain, 1998. ACM.

[Schn09] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[SiFC92] P.S. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. In M. Dubois and S.S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 25–41. Kluwer, 1992.

[SPAR91] SPARC International. *The SPARC Architecture Manual - Version 8*. Prentice-Hall, Inc., 1991.

[SSON+10] P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli, and M.O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM (CACM)*, 53(7):89–97, July 2010.

[StNu04] R.C. Steinke and G.J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM (JACM)*, 51(5):800–849, September 2004.