ELSEVIER

Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE @DIRECT®

Information Processing Letters 90 (2004) 53–58

Information Processing Letters

www.elsevier.com/locate/ipl

Relationships between memory models

Vicent Cholvi [a,*], Josep Bernabéu [b]

[a] *Departamento de Lenguajes y Sistemas Informáticos, Universitat Jaume I, Castellón, Spain*
[b] *Instituto Tecnológico de Informática, Universitat Politécnica de València, València, Spain*

Received 3 August 2003; received in revised form 14 January 2004

Communicated by J.L. Fiadeiro

Abstract

There have been many proposals of shared memory systems, each one providing different types of memory coherence for interprocess communication. However, they have usually been defined using different formalisms. This makes it difficult to compare among them the different proposals put forward. In this paper we present a formal framework for specifying memory models with different coherency properties. We specify most of the known shared memory models using our framework, showing some of the relationships that hold among them.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Formal methods; Memory models; Consistency models; Distributed systems; Concurrency

1. Introduction

Shared memory (reading and writing of shared variables) is a mechanism used for inter-process communication in concurrent programs which has several important benefits. In the first place, it hides from the programmer the particular communication technique employed. Therefore, application developers do not need to be involved in the management of messages. In addition, it allows complex shared structures to be passed by reference, providing a simple and well-known paradigm.

However, while the semantics of reads and writes in sequential programs are clear, the situation is different for concurrent accesses to shared variables. There have been numerous proposals and implementations of shared memory systems providing different semantics [1–7].

It seems clear that "strong" memory models [1,2] make it simpler to write programs, since the return value of each read operation is more predictable. On the other hand, "weaker" memory models can be more efficiently implemented, since they allow more possible return values for each read operation [3–7], resulting in a lower coherence overhead. However, the simple programming style that programmers are accustomed to cannot be sacrificed for the sake of performance (that would eliminate the familiar programming paradigm that is one of the main advantages of shared memory). Therefore, this poses a tradeoff between simplicity and performance.

On the other hand, the different memory models have usually been defined using different formalisms.

* Corresponding author.
E-mail addresses: vcholvi@uji.es (V. Cholvi), josep@iti.upv.es (J. Bernabéu).

0020-0190/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2004.01.007

This makes it difficult to determine how those memory models are related and if a given memory model can be safely used for any task for which another memory model is satisfactory. In this paper we present a formal framework for specifying memory models with different coherency properties. We specify most of the known shared memory models using our framework, showing some of the relationships that hold among them.

The rest of the paper is organized as follows. In Section 2 we introduce the basic formalism and in Section 3 we present definitions for many of the most significant memory models. In Section 4 we show some of the relationships that hold among those memory models. Finally, in Section 5 we present our concluding remarks.

2. Preliminaries

In our formalism, we consider only operations for accessing shared resources (i.e., reads and writes). A write operation issued by process i to store the value u in the variable x is denoted as $w_i(x)u$. Similarly, a read operation reporting to process i that u is stored in the variable x is denoted as $r_i(x)u$. Sometimes, to describe an operation op, we use a pair of actions, start(op) and end(op), which respectively denote the start and end of that operation. We also take into account unfinished operations, which are described by using only the starting action.

A *computation* consists of a sequence of read and write operations. In this work, only those computations that follow the model of sequential processes executing blocking operations are taken into account, that is, a process computing an operation is forced to block until the operation completes. Therefore, computations will be formed by finished operations with the possible exception of the last one for each process, which can be unfinished. In order to reason with a program computation with unfinished operations, we can "finish" it by either removing each unfinished operation or replacing it by a finished operation (that captures the notion that some unfinished operations had "visible" effect, while the others did not). We say a computation fulfills a given property if at least one of its finished computations fulfills that property (note that a given computation may have several finished computations).

Now we provide the definitions of some relationships for the operations of a computation, which we use later on to characterize the computations forming part of a memory model (formally defined in the next section).

To simplify the notation, we assume that values are uniquely written in any variable. This assumption does not introduce new restrictions as it can be forced by associating a time-stamp with writes (there are logical implementation of clocks that provide bounded values [8]). Also, we assume that the initial values of the variables are set by using write operations.

Definition 1 (*Relationships*). Let op and op$'$ be two operations in a computation α. We define the *atomic*, *program* and *causal* relationships as follows:

- *Atomic*: $\mathrm{op} \prec^{\alpha}_{\mathrm{ATO}} \mathrm{op}'$ if end(op) happens before start(op$'$) in α.
- *Program*: $\mathrm{op} \prec^{\alpha}_{\mathrm{PO}} \mathrm{op}'$ if they are operations from the same process and $\mathrm{op} \prec^{\alpha}_{\mathrm{ATO}} \mathrm{op}'$.
- *Causal*: $\mathrm{op} \prec^{\alpha}_{\mathrm{CAU}} \mathrm{op}'$ if some of the following holds:
  (1) $\mathrm{op} \prec^{\alpha}_{\mathrm{PO}} \mathrm{op}'$,
  (2) $\mathrm{op} = w_i(x)u$ and $\mathrm{op}' = r_j(x)u$ (i.e., the read value is the written one),
  (3) $\exists\, \mathrm{op}''$: $\mathrm{op} \prec^{\alpha}_{\mathrm{CAU}} \mathrm{op}'' \prec^{\alpha}_{\mathrm{CAU}} \mathrm{op}'$.

All of them are partial relationships and, as it can be readily seen, $\mathrm{op} \prec^{\alpha}_{\mathrm{PO}} \mathrm{op}'$ implies both $\mathrm{op} \prec^{\alpha}_{\mathrm{CAU}} \mathrm{op}'$ and $\mathrm{op} \prec^{\alpha}_{\mathrm{ATO}} \mathrm{op}'$. However, whereas both the atomic and program relationships are orders, the causal relationship, since it is cyclic, is a preorder.

Basically, the atomic order [2] captures the "real-time" ordering of non-overlapping operations. The program order [5] relaxes the atomic order in such a way that operations follow the atomic order, but only those from the same process. Finally, the causal preorder [3] is defined in order to capture "causality" in the sense of [9].

3. Formalization of memory models

In this section, we show how to formalize some of the most widely known memory models proposed

in the literature. We do not consider memory models such as *release* [10], *entry* [11], etc. since they make use of synchronization operations.

First, we provide a definition of memory model completely general.

Definition 2 (*Memory model*). We define a *memory model M* as the set formed by all computations of type M.

Obviously, for this definition to make sense, it is necessary in each case to define what is a computation of type M.

To define those computations, first we introduce the *serial* computation concept (which has been also defined using the term *legal* [2] and *CMP* [12]). They follow the principle of reading always "the latest written value", not allowing the overlapping of operations (i.e., not allowing operations that cannot be compared by using the atomic order).

Definition 3 (*Serial computation*). A computation α is *serial* if it does not contain any overlapping operation and $\forall\, \mathrm{op} = r_i(x)u (\exists\, \mathrm{op}' = w_j(x)u$: $\mathrm{op}' \prec^{\alpha}_{\mathrm{ATO}} \mathrm{op}$ and $\nexists\, \mathrm{op}'' = w_k(x)v$: $\mathrm{op}' \prec^{\alpha}_{\mathrm{ATO}} \mathrm{op}'' \prec^{\alpha}_{\mathrm{ATO}} \mathrm{op})$.

Fig. 1 shows a serial computation. Serial computations are quite restrictive, since there is no real concurrency (operations do not overlap). However, what actually makes them interesting is the fact that, most of the computations (if not all) characterizing memory models are based on the O-*view* concept, where $O \in \{\mathrm{ATO}, \mathrm{PO}, \mathrm{CAU}\}$, which is based on the serial computation definition.

Definition 4 (O-*view*). A computation β is an O-*view* of another computation α, where $O \in \{\mathrm{ATO}, \mathrm{PO}, \mathrm{CAU}\}$, if it is formed by permuting the elements of α in such a way that it is serial and the relation $\prec^{\alpha}_{O}$ is preserved.[1]

Since in a distributed memory system, where processes work asynchronously, operations may not appear visible to all processes at the same time, an O-view of a computation α is nothing but an explanation of how it can be "serially perceived" from the point of view of processes (provided the relationship $\prec^{\alpha}_{O}$ is preserved).

For a given α, we denote the set of its O-views as V^{α}_{O}. From Definition 1, it is clear that $V^{\alpha}_{\mathrm{ATO}} \subseteq V^{\alpha}_{\mathrm{PO}}$ and $V^{\alpha}_{\mathrm{CAU}} \subseteq V^{\alpha}_{\mathrm{PO}}$.

Before we proceed with the definitions for the computations characterizing memory models, we will make use of the next notation.

Notation 1.

- α_i denotes a subsequence of α formed by removing the read operations from processes other than i.
- α_x denotes a subsequence of α formed by removing the operations on any variable other than x.
- $\alpha_{\langle x \rangle}$ denotes a subsequence of α formed by removing all the operations other than the write operations on variable x.
- α_* denotes a subsequence of α formed by removing the read operations which overlap with any write operation on the same variable.
- $V^{\alpha,\gamma}_{O}$ denotes the subset of V^{α}_{O} containing the computations that preserve $\prec^{\gamma}_{O}$.

Now, we define the computations characterizing memory models as follows:

Definition 5. A computation α is of the type specified below if for any prefix the following holds:

- Atomic [6,13,2]: $V^{\alpha}_{\mathrm{ATO}} \neq \emptyset$.
- Safe [6]: $V^{\alpha_*}_{\mathrm{ATO}} \neq \emptyset$.
- Regular [6]: $V^{\alpha_*}_{\mathrm{ATO}} \neq \emptyset$ and $\forall\, \mathrm{op} = r_i(x)u$ $(\exists\, \mathrm{op}' = w_j(x)u)$.
- Sequential [6]: $V^{\alpha}_{\mathrm{PO}} \neq \emptyset$.
- Cache [4]: $\forall x$ $(V^{\alpha_x}_{\mathrm{PO}} \neq \emptyset)$.
- pRAM [7]: $\forall i$ $(V^{\alpha_i}_{\mathrm{PO}} \neq \emptyset)$.
- Processor [4]: $\forall i, j$ $(\exists \beta \in V^{\alpha_i}_{\mathrm{PO}} \wedge \exists \beta' \in V^{\alpha_j}_{\mathrm{PO}}$: $\forall x$ $(\beta_{\langle x \rangle} = \beta'_{\langle x \rangle}))$.
- Causal [3]: $\forall i$ $(V^{\alpha_i,\alpha}_{\mathrm{CAU}} \neq \emptyset)$.

Note that those definitions have been stated based exclusively on the set of computations they allow independently of the architectural platform that can be use to implement them. We would like to remark that instead of directly using the original definitions,

[1] We say that β preserves $\prec^{\alpha}_{O}$ if $\forall\, \mathrm{op}, \mathrm{op}'$: $\mathrm{op} \prec^{\alpha}_{O} \mathrm{op}'$, $\mathrm{op} \prec^{\beta}_{O} \mathrm{op}'$.

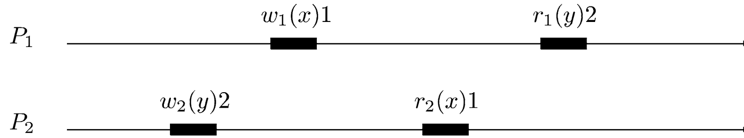

Fig. 1. Serial computation.

the above definitions have been based on the rationale behind the original definitions.

4. Relationships between memory models

An important issue regarding memory models consists in determining whether a given memory model can be safely used for any task for which another memory model is satisfactory, i.e., determining if a memory model "solves" another one. We say that a memory model *solves* another one if the set of computations that represent the first is included in the set that represent the latter.

In Fig. 6 we have summarized how memory models are related by the solvability relationship (they are formally proved from Proposition 1 to Proposition 8).

Proposition 1. *The atomic model solves the sequential model.*

Proof. Let α be an atomic computation. Therefore $V_{\mathrm{ATO}}^{\alpha} \neq \emptyset$. Since $V_{\mathrm{ATO}}^{\alpha} \subseteq V_{\mathrm{PO}}^{\alpha}$, we have that $V_{\mathrm{PO}}^{\alpha} \neq \emptyset$. Thus, α is sequential. □

Proposition 2. *The atomic model solves the regular model.*

Proof. Let α be an atomic computation. Therefore $V_{\mathrm{ATO}}^{\alpha} \neq \emptyset$, which implies that $V_{\mathrm{ATO}}^{\alpha *} \neq \emptyset$. Furthermore, as α is atomic then $\forall\, \mathrm{op} = r_i(x)u$ $(\exists\, \mathrm{op}' = w_j(x)u)$. Thus, α is regular. □

Proposition 3. *The regular model solves the safe model.*

Proof. Let α be a regular computation. Therefore $V_{\mathrm{ATO}}^{\alpha *} \neq \emptyset$, which implies that α is safe.

Proposition 4. *The sequential model solves the causal model.*

Proof. Let α be a sequential computation. Therefore $V_{\mathrm{PO}}^{\alpha} \neq \emptyset$, which implies that $\exists \beta \in V_{\mathrm{PO}}^{\alpha}$ (and consequently $\forall i$ $(\beta_i \in V_{\mathrm{PO}}^{\alpha_i,\alpha})$. On the other hand, for each process i, we have that β_i also preserves $\prec_{\mathrm{CAU}}^{\alpha}$. Thus $\beta_i \in V_{\mathrm{CAU}}^{\alpha_i,\alpha}$, which implies that $V_{\mathrm{CAU}}^{\alpha_i,\alpha} \neq \emptyset$ and, by Definition 5, α is causal. □

Proposition 5. *The sequential model solves the processor model.*

Proof. Let α be a sequential computation. Therefore $V_{\mathrm{PO}}^{\alpha} \neq \emptyset$. Let $\gamma \in V_{\mathrm{PO}}^{\alpha}$. It is enough to take $\beta = \gamma_i$ and $\beta' = \gamma_j$ in Definition 5 to ensure that α is a processor computation. □

Proposition 6. *The causal model solves the pRAM model.*

Proof. Let α be a causal computation. Therefore $\forall i$ $(V_{\mathrm{CAU}}^{\alpha_i,\alpha} \neq \emptyset)$. However, since $V_{\mathrm{PO}}^{\alpha_i} \subseteq V_{\mathrm{CAU}}^{\alpha_i,\alpha}$, we have that $\forall i$ $(V_{\mathrm{PO}}^{\alpha_i} \neq \emptyset)$. Thus, α is pRAM. □

Proposition 7. *The processor model solves the pRAM model.*

Proof. Let α be a processor computation. Therefore $\forall i, j$ $(\exists \beta \in V_{\mathrm{PO}}^{\alpha_i} \wedge \exists \beta' \in V_{\mathrm{PO}}^{\alpha_j}$: $\forall x$ $(\beta_{\langle x \rangle} = \beta'_{\langle x \rangle}))$ which implies that $\forall i$ $(V_{\mathrm{PO}}^{\alpha_i} \neq \emptyset)$. Thus, α is pRAM. □

Proposition 8. *The processor model solves the cache model.*

Proof. Let α be a processor computation. We can see that it is also cache in the following way. For each variable x, we construct a computation β, containing the write operations on variable x in the same order as in $\beta_{\langle x \rangle}$. The remaining operations can be easily placed in the right order to see that β_x is sequential. Hence, $\forall x$ $(V_{\mathrm{PO}}^{\alpha_x} \neq \emptyset)$ and, by Definition 5, we have that α is cache. □

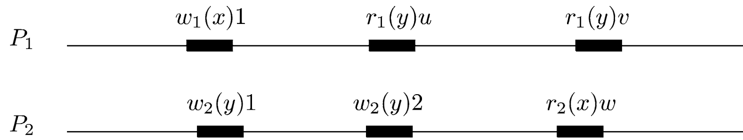

Fig. 2. Atomic ($u = 1$, $v = 2$ and $w = 1$), sequential ($u = 0$, $v = 0$ and $w = 1$), regular ($u = 0$, $v = 2$ and $w = 1$), safe ($u = 3$, $v = 2$ and $w = 1$) and cache ($u = 0$, $v = 0$ and $w = 0$) computation.

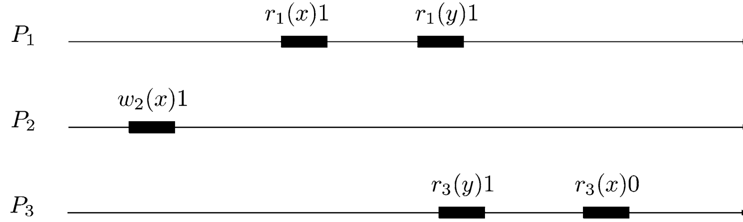

Fig. 3. pRAM computation.

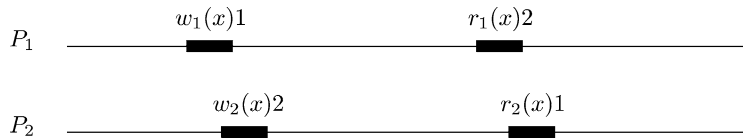

Fig. 4. Causal computation.

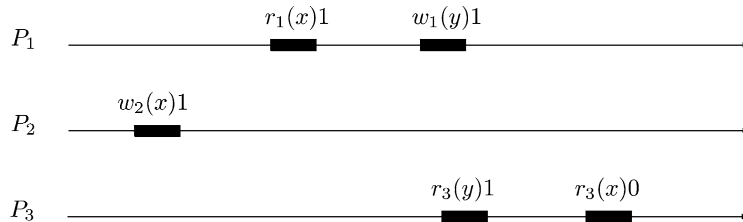

Fig. 5. Processor computation.

Clearly, not always a memory model solves another one. On the contrary that with the solvability relations, here we do not state in an independent manner each one of the non-solvability relationships. However, they can be easily proved by using contradictions taking the computations presented in Figs. 2–5 as counterexamples. For instance, since Fig. 6 shows that the causal model does not solve neither the atomic model, nor the sequential, nor the processor model, nor the cache, nor the regular, etc., therefore the causal computation depicted in Fig. 4 has been chosen so that it is not neither atomic, nor sequential, nor processor, nor cache, nor regular, etc.

Finally, as the two following propositions show, it is also possible to establish relationships among combinations of memory models in which it is not included one into another.

Proposition 9. *The intersection of the processor and causal models does not solve the sequential model.*

Proof. By contradiction. Consider the following computation $w_1(x)1\ r_1(x)1\ r_1(y)0\ w_2(y)1\ r_2(y)1\ r_2(x)0$ which is both processor and causal, and assume that it is sequential.

This computation, in order to be sequential, forces $r_1(y)$ to go before $w_2(y)$. However, that forces $r_2(x)$ to go after $w_1(x)$, which makes this computation non-sequential. □

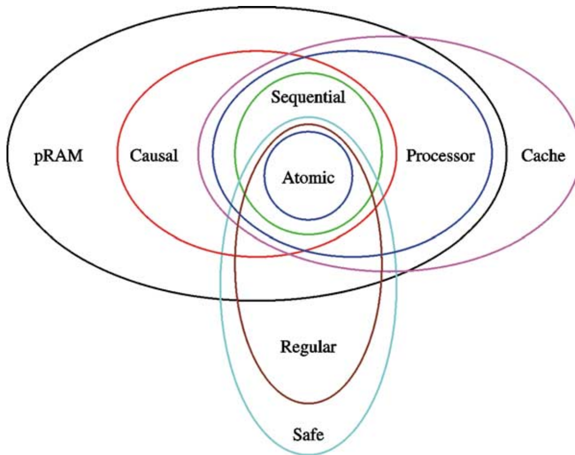

Fig. 6. Relationships between memory models.

Proposition 10. *The intersection of the pRAM and cache models does not solve neither the processor nor the causal nor the regular models.*

Proof. By contradiction. Consider the following computation $w_1(x)1$ $w_1(y)1$ $r_2(y)1$ $w_2(x)2$ $r_2(x)2$ $r_3(x)2$ $r_3(x)1$ which is both pRAM and cache.

This computation is not regular, since the operations $r_3(x)1$ should return a value 2.

Also, it is not processor. Indeed, process 2 forces the next ordering of write operations $w_1(x)1$ $w_1(y)1$ $w_2(x)2$ and process 3 forces $w_2(x)2$ $w_1(x)1$ $w_1(y)1$, which implies that $\beta_{\langle x\rangle} \neq \beta'_{\langle x\rangle}$.

Finally, it is not causal either since any "potential" causal view for process 3 forces $w_2(x)2$ to go before $w_1(x)1$, which does not preserve the causal preorder. □

5. Conclusions

In this paper we have presented a formal framework that can be used to study the way memory access operations take place.

We have shown how to define the most significant memory models. Furthermore, we have provided a "methodology" to specify new memory models. Namely, it consists on specifying the type of computations that the memory model allows. This, as we shown previously, is strongly based on the O-view concept (i.e., on how computations can be "serially perceived" from the point of view of processes).

The availability of an adequate formalism made it possible to establish some of the relationships that hold among them. Those relationships were inferred directly based exclusively on the set of computations they allow.

References

[1] H. Attiya, J. Welch, Sequential consistency versus linearizability, ACM Trans. Comput. Systems 12 (2) (1994) 91–122.

[2] M. Herlihy, J. Wing, Linearizability: A correctness condition for concurrent objects, ACM Trans. Programming Languages Systems 12 (3) (1990) 463–492.

[3] M. Ahamad, G. Neiger, J. Burns, P. Kohli, P. Hutto, Causal memory: Definitions, implementation and programming, Distrib. Comput. 9 (1) (1995) 37–49.

[4] J. Goodman, Cache consistency and sequential consistency, Technical Report 61, IEEE Scalable Coherence Interface Working Group (March 1989).

[5] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Trans. Comput. 28 (9) (1979) 690–691.

[6] L. Lamport, On interprocess communication I, II, Distrib. Comput. 1 (2) (1986) 77–101.

[7] R. Lipton, J. Sandberg, PRAM: A scalable shared memory, Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[8] A. Singh, Bounded timestamps in process networks, Parallel Process. Lett. 6 (2) (1996) 259–264.

[9] L. Lamport, Time, clocks and the ordering of events in a distributed system, Comm. ACM 21 (7) (1991) 558–565.

[10] P. Keleher, A. Cox, W. Zwaenepoel, Lazy release consistency for software distributed shared memory, in: Proceedings of the 19th Annual Symposium on Computer Architecture, 1992, pp. 13–21.

[11] B. Bershad, M. Zekauskas, Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors, Technical Report CMU-CS-91-170, Carnegie–Mellon University, Pittsburgh, PA, September 1991.

[12] W. Collier, Reasoning About Parallel Architectures, Prentice–Hall International Editions, Englewood Cliffs, NJ, 1992.

[13] J. Misra, Axioms for memory access in asynchronous hardware systems, ACM Trans. Programming Languages Systems 8 (1) (1986) 142–153.