TITLE

Memory Models

BYLINE

Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL
USA
sadve@illinois.edu

Hans-J. Boehm
HP Laboratories
Palo Alto, CA
USA
hans.boehm@hp.com

SYNONYMS

Memory consistency models, memory ordering

DEFINITION

In the context of shared-memory systems, the memory model specifies the values that a shared-memory read in a program may return. It is part of the interface between a shared-memory program and any hardware or software that may transform that program – it specifies both the possible behaviors of the memory accesses for the programmer and constrains the legal transformations and executions for the implementer.

For high-level language programs, the memory model describes the behavior of accesses to shared variables or heap objects, while for machine code, the memory model describes the behavior of hardware instructions that access shared memory. Without an unambiguous memory model, it is not possible to reason about the correctness of a shared-memory program, compiler, dynamic optimizer, or hardware.

DISCUSSION

Sequential Consistency

In a single thread program, a read is expected to return the value of the last write to the same address, where last is uniquely defined by the program text or program order. This allows the programmer to view the memory accesses as occurring one at a time (atomically) in program order. It also allows the hardware or compiler to aggressively reorder memory accesses as long as program order is preserved between a write and other accesses to the same address as the write.

A natural extension for shared-memory programs is the sequential consistency memory model [Lam79] which offers simple interleaving semantics. With sequential consistency, all memory accesses appear to be performed in a single total order preserving the program order of each thread. Each read access to a memory location yields the value stored by the last preceding write access to the same memory location. Although sequential consistency appears to be an appealing programming model, it suffers from two deficiencies:

- It can be difficult or inefficient to implement. Hardware often makes memory accesses visible to other threads out of order, for example by buffering stores. Compilers often do the same, for example, by moving an access to a loop-invariant variable out of the loop. Unlike with single-threaded programs, these optimizations can be observed by the program, and may thus violate sequential consistency for multithreaded programs.
- The definition of sequential consistency is based on the interleaving of individual memory accesses, and is thus only meaningful if the programmer understands how memory is accessed; e.g., a byte vs. a word at a time. This is sensible at the machine architecture level, but may not be realistic at the programming language level.

Relaxed Memory Models

Recognizing the limitations of sequential consistency, more relaxed or weak memory models have been proposed. Until the late 1990's, most of this work was in the context of hardware instruction set architectures and implementations. These implementation-centric models are mostly driven by specific optimizations desirable for hardware. The most common class relaxes the program order requirement of sequential consistency. For example, the SPARC Total Store Order (TSO) [WG94] and the x86 models [Cor10b] allow reordering a write followed by a read (to a different location) in program order. The IBM PowerPC memory model [Cor10a] allows reordering both reads and writes (to different locations). Such models additionally provide various forms of fence instructions to enable programmers to explicitly impose program orders not guaranteed by default. Other optimizations exploited by such models include subtle relaxations of the memory model to allow writes from multiple threads to become visible in inconsistent orders to different observer threads, as well as the ability to make memory references visible to other threads out of order even if a data- or control-dependence exists between the references. Although most of the work on this genre of implementation-centric models is in the context of hardware, some programming environments such as OpenMP 3.0 [The08] also express their memory models in terms of fence instructions to ensure explicit program ordering.

These relaxed models provide performance benefits over sequential consistency (through both hardware and compiler optimizations), but are inappropriate as programming interfaces. They often require description of subtle interactions and/or are not well-matched to software requirements, making them too complex to reason and/or sub-optimal for performance.

The Data-Race-Free Memory Models

The data-race-free models [AH90, Adv93] (also referred to as properly labeled models [Gha95]) take a programmer-centric approach. They observe that good programming practice requires programs to be well-synchronized or data-race-free. In such programs, concurrent non-read-only accesses to the same variable are either explicitly marked as synchronization or are separated by explicit synchronization. This ensures that no thread can ever observe another thread between synchronization points, allowing the implementation to aggressively transform code between synchronization points while still maintaining the appearance of sequential consistency.

The data-race-free models formally define the notion of data-race-free programs and guarantee sequential consistency only for those programs. For full implementation flexibility, they do not provide any guarantees for programs that contain data races. This approach allows nearly standard optimizations, while providing a simple model for programmers and was the approach adopted by the Ada programming language [Uni83] and the POSIX standard [IT01] (albeit informally).

State-of-the-art in 2010

In the last decade, there has been much work to standardize high level language programming models, specifically in the context of Java and C++. There is now convergence towards data-race-free as the model of choice. For all practical purposes, the Java memory model [MPA05, PtJEG09] and the upcoming C and C++ standards [C++10, BA08] provide the data-race-free model, but with two unfortunate caveats.

First, Java's safety and security properties require that some semantics need to be provided for all programs, including programs with data races. It has been extraordinarily difficult to develop reasonable semantics for racy programs, while still preserving full implementation flexibility. The current formal specification of the Java memory model is therefore quite complex and has an unresolved bug.

Second, although data-race-free maps well to current implementation-centric hardware models for most purposes, there are useful programming idioms where available hardware level fences are too heavyweight for language-level synchronization mechanisms. For this reason, Java and C++ provide low-level synchronization constructs that provide potentially better performance on some current hardware, but at the cost of significant complexity to the programming model.

Although most language specifications are converging to the data-race-free model, current implementations often lag. In particular, it is still quite common for C and C++ compilers to effectively introduce copies of an otherwise unmodified variable to itself, for example by overwriting adjacent structure fields when a small field is modified [Boe05]. These can introduce visible data races, and are hence incompatible with the model.

In summary, although data-race-free provides the best tradeoff between performance and ease-of-use today, it falls short for safe languages and current implementations.

Further Reading

A more complete overview and retrospective on memory models is provided in [AB10]. It also surveys some possible approaches for overcoming the deficiencies of current data-race-free models. A tutorial on hardware-oriented relaxed memory models can be found in [AG96]. A description of the x86 hardware memory model, together with a discussion of some of the issues arising in precisely defining such models, is given in [SSO$^+$10].

References

[AB10] Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, August 2010.

[Adv93] Sarita Vikram Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1993.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AH90] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *Proc. 17th Intl. Symp. Computer Architecture*, pages 2–14, 1990.

[BA08] Hans-J. Boehm and Sarita Adve. Foundations of the C++ concurrency memory model. In *Proc. Conf. on Programming Language Design and Implementation*, pages 68–78, 2008.

[Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. In *Proc. Conf. on Programming Language Design and Implementation*, 2005.

[C++10] C++ Standards Committee, Pete Becker, ed. Programming Languages - C++ (final committee draft). C++ standards committee paper WG21/N3092=J16/10-0082, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3092.pdf`, March 2010.

[Cor10a] IBM Corp. Power ISA Version 2.06 Revision B. `http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf`, 2010.

[Cor10b] Intel Corp. Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3A: System Programming Guide, Part1. `http://www.intel.com/products/processor/manuals/`, 2010.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for Shared Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[IT01] IEEE and The Open Group. *IEEE Standard 1003.1-2001*. 2001.

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

[MPA05] Jeremy Manson, William Pugh, and Sarita Adve. The Java memory model. In *Proc. Symp. on Principles of Programming Languages*, 2005.

[PtJEG09] William Pugh and the JSR 133 Expert Group. The Java memory model. `http://www.cs.umd.edu/~pugh/java/memoryModel/` and referenced pages, July 2009.

[SSO$^+$10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[The08] The OpenMP ARB. OpenMP application programming interface: Version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

[Uni83] United States Department of Defense. *Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A-1983 Standard 1003.1-2001*, 1983. Springer.

[WG94] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual*. Prentice Hall, 1994. Version 9.