Can Out-of-Order Instruction Execution in Multiprocessors be made Sequentially Consistent?

Lisa Higham and Jalal Kawash

Department of Computer Science
The University of Calgary, Canada
higham@cpsc.ucalgary.ca

Department of Computer Science
American University of Sharjah, UAE
jkawash@aus.edu

Abstract. We investigate all possible combinations of re-ordering of read and write instructions and their effects on the correctness of programs that are designed for sequential consistency. With certain combinations of re-orderings, any program that accesses shared memory through only reads and writes and that is correct assuming sequential consistency, can be transformed to a new program that does not use any explicit synchronization, and that remains correct in spite of the instruction re-ordering. With other combinations of re-ordering, such transformations do not exist, without resorting to explicit synchronization.

1 Introduction

Designers of concurrent algorithms typically assume sequential consistency, a consistency model that is formalized by Lamport [11]. Sequential consistency requires that memory operations of all processors appear to be "executed in some sequential order, and the operations of each processor appear in this sequence in the order specified by its program" (program order). Sequential consistency is intuitive, but disallows many possible hardware and software optimizations.

Adve and Gharachorloo [1] identify several optimization techniques that cause instructions to be re-ordered so that they appear to execute out of program order. This is called *instruction re-ordering*. Write buffers with read bypasses, overlapping writes, non-blocking reads, and optimizing compilers can lead to all forms of instruction re-ordering. They also cite many commercial multiprocessors that utilize instruction re-ordering, such as the AlphaServer 8200/8400, Cray T3D/T3E, and SparcCenter 1000/2000 (See Figure 1). Other examples include the Java Virtual Machine (JVM), IBM PowerPC, Intel Itanium, and .Net. Instruction re-ordering aims at improving the system's performance but it relaxes sequential consistency, making the job of programming multiprocessors even harder.

Multiprocessor machines that incorporate instruction re-ordering are also equipped with more powerful instructions than reads and writes, such as read-modify-write and memory barrier instructions. These synchronization primitives

Architecture	write-read re-ordering	write-write re-ordering	read-write re-ordering	read-read re-ordering
IBM 370 [1]	√			
SPARC TSO [14, 7]	√			√ [10]
SPARC PSO [14, 7]	√	√	√ [10]	√ [10]
SPARC RMO [14, 5]	√	√	√	√
IBM PowerPC [2]	√	√	√	√
DEC Alpha [3, 5]	√	√	√	√
JVM [12, 6]	√	√	√	√
Intel Itanium[9]	√	√	√	√
.Net [13]	√	√	√	√

Fig. 1. Examples of some commercial systems that utilize instruction re-ordering

can be used to enforce orderings on instructions that otherwise might be re-ordered causing incorrect computation. Using these powerful instructions, however, is expensive; excessive use can result in inefficient implementations, possibly defeating the purpose of instruction re-ordering altogether.

Other related studies (see the full version of the paper for a bibliography [8]) provide programming strategies for high performance multiprocessors most of which rely on the wise usage of synchronization.

2 Summary of Results

We assume that multiprocessors are *coherent* [4], requiring execution order to maintain program order of instructions applied to the same memory location. If a read of one memory location precedes in program order a write to a different memory location and this read appears after this write in execution order, this is called *read-write re-ordering*. Reordering types *write-read, write-write, and read-read* are defined similarly. Call a shared memory multiprocessor program whose shared memory consists of only atomic locations (that is, variables that support only read and write instructions) a *(read/write) multi-program*. The fundamental question guiding this work is:

> Under what conditions is there a general transformation that transforms any read/write multi-program that is correct under sequential consistency to another read/write multi-program that is still correct in spite of possible instruction re-ordering?

Such a transformation is called a *read/write transformation* and constitutes inserting only additional read and write operations to a given read/write multi-program, which solves some problem $\mathcal{P}$ under sequential consistency, but without altering its original semantics. Hence, the transformed multi-program is also a read/write multi-program. The purpose of these additions is to restore program order and maintain sequential consistency in spite of instruction re-ordering.

single type	two combinations	three combinations
read-read $\surd$	read-read, write-write $\surd$	read-read, write-write, write-read $\times$
write-write $\surd$	read-read, read-write $\times$	read-read, write-write, read-write $\times$
read-write $\surd$	read-read, write-read $\times$	read-read, write-read, read-write $\times$
write-read $\surd$	read-write, write-read $\surd$	write-write, read-write, write-read $\surd$
	write-write, write-read $\surd$	
	write-write, read-write $\surd$	

Fig. 2. Summary of results

Since the semantics of the original program are maintained, the transformed program still solves problem $\mathcal{P}$.

The results of this investigation are summarized in Figure 2. The possibilities (represented by $\surd$) in Figure 2 indicate the existence of a general read/write transformation for any sequentially consistent program to a program that is still correct in spite of the indicated instruction re-ordering combination.

The impossibilities of Figure 2 (represented by $\times$) indicate that there is no *general* read/write transformation for the indicated combinations of instruction re-ordering. That is, any read/write transformation fails to transform at least one multi-program that is known to be correct for sequential consistency. Such general transformations for the indicated combinations of instruction re-orderings must augment the specified program with explicit synchronization operations.

More precisely, let $\mathcal{A}$ be an arbitrary read/write multi-program that solves a problem $\mathcal{P}$, under sequential consistency. The results of our research are:

1. For any combination of re-ordering types that excludes read-read re-ordering, there exists a read/write transformation, which transforms $\mathcal{A}$ to a read/write program $\mathcal{A}'$ that solves $\mathcal{P}$ in spite of the re-ordering. The transformation is general; it is correct for any read/write multi-program under any combination of read-write, write-read, and write-write re-orderings.
2. The exclusion of the read-read re-ordering is sufficient but not necessary. For any combination of read-read and write-write re-ordering only, such a read/write transformation still exists.
3. If both read-read and read-write (or both read-read and write-read) re-ordering combinations are possible, there is no general read/write transformation. Any correct general transformation must use stronger operations than reads and writes, such as read-modify-write and memory barrier instructions, for at least some programs.

3 Conclusion

The transformations we used are simple and general; they can be applied to any read/write multi-program that is correct for sequential consistency. They are also optimal for general transformations — these that apply to *any* multi-program that is correct for sequential consistency. However, optimality for general transformations does not necessarily imply optimality for individual multi-program

instances. When given a *fixed* instance, it may be possible to apply further optimizations that exploit information from the given multi-program and the problem it solves. Such information (from both programs and problems) is unavailable to general transformers.

Our results imply that the IBM PowerPC, DEC Alpha, JVM, and SPARC TSO, PSO, and RMO (Figure 1) require the use of explicit synchronization in order to solve certain problems. Hence, one of our future research directions is to augment the target program with memory barrier instructions and to minimize the number of such instructions.

References

1. S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.
2. F. Corella, J. Stone, and C. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report RC18638, IBM, 1994.
3. C. C. Corportaion. *The Alpha Architecture Handbook.* Compaq Computer Corporation, 1998. Order number: EC-QD2KC-TE.
4. M. Frigo. The weakest reasonable memory model. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, 1998.
5. L. Higham, L. Jackson, and J. Kawash. Specifying memory consistency of write buffer multiprocessors. Technical Report 2004-758-23, Department of Computer Science, The University of Calgary, August 2004. Submitted for publication.
6. L. Higham and J. Kawash. Java: Memory consistency and process coordination (extended abstract). In *Proc. 12th Int'l Symp. on Distributed Computing, Lecture Notes in Computer Science volume 1499*, pages 201–215, September 1998.
7. L. Higham and J. Kawash. Memory consistency and process coordination for SPARC multiprocessors. In *Proc. of the 7th Int'l Conf. on High Performance Computing, Lecture Notes in Computer Science volume 1970*, pages 355–366, December 2000.
8. L. Higham and J. Kawash. Impact of instruction re-ordering on the correctness of shared-memory programs. Technical Report 2005/794/25, Department of Computer Science, The University of Calgary, July 2005.
9. Intel Corporation. Intel Itanium architecture software developers manual, volumes 1-3. 2002.
10. J. Kawash. *Limitations and Capabilities of Weak Memory Consistency Systems.* Ph.D. dissertation, Department of Computer Science, The University of Calgary, January 2000.
11. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.
12. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.
13. A. D. Robison. Memory consistency and .Net. *Dr. Dobb's Journal*, pages 46–50, April 2003.
14. D. Weaver and T. Germond, editors. *The SPARC Architecture Manual version 9.* Prentice-Hall, 1994.