Multiprocessor Architectures Don't Really Exist (But They Should)

(Invited Talk)

Peter Sewell
Computer Laboratory
University of Cambridge
Cambridge, UK
http://www.cl.cam.ac.uk/~pes20/weakmemory/

***Abstract*—Multiprocessors and high-level concurrent languages generally provide only relaxed (non-sequentially-consistent) memory models, to permit performance optimisations. One has to understand these models to program reliable concurrent systems but, despite work in this area over many years, the specifications of real-world multiprocessors and languages are typically ambiguous and incomplete informal-prose documents, cannot be used for testing hardware or software, sometimes give guarantees that are too weak to be useful, and are sometimes simply unsound. Such informal prose is a very poor medium for loose specifications.**

This talk will review various problems with some current specifications, for x86 (Intel 64/IA32 and AMD64), and Power and ARM processors, and for the Java and C++ languages, and describe ongoing work to produce rigorously defined specifications for some of these.

Processor vendors document *architectures*, including for example Intel 64 and IA-32 [1], AMD64 [2], ARMv7 [3], and Power 2.06 [4]. These are the key interface between hardware and software, describing to programmers (of compilers, operating systems, and other low-level code) what behaviour they can depend upon from the processors. Architectures are subject to conflicting requirements; they must at least:

1) reveal enough processor behaviour for effective programming;
2) be sound with respect to a range of previous specific processor implementations;
3) accessible to programmers and hardware designers, and not overwhelmingly complex (as a full processor design would be);
4) not reveal sensitive IP; and
5) not unduly constrain future processor design.

They are, therefore, necessarily *loose specifications*, describing a range of possible programmer-visible behaviour rather than specifying a single (completely deterministic) behaviour for each program. They are typically expressed in informal prose, often with pseudocode descriptions of the behaviour of each instruction.

In some cases this looseness is relatively straightforward, especially for sequential code. For example, an architecture may specify that the value of some flag is left undefined by a particular machine instruction. Even this has the well-known potential to cause difficulties: suppose some processor implementations in fact leave the flag set, and some code (perhaps inadvertently) depends on that fact. Programs are validated by running them on specific implementations, so that dependency may remain a lurking bug, to be discovered only if a later processor implementation has different behaviour.

When we turn to the behaviour of *multiprocessors* the situation is much worse. Multiprocessor implementations typically involve a range of optimisations (store buffers, out-of-order execution, speculation, optimised cache protocols, etc.) which are not observable by single-threaded programs but which give rise to *relaxed* or *weakly consistent* memory behaviour of multi-threaded code. For example, in a simple implementation with store buffers, two threads that each store to a location and then read from the other location can, in the same execution, read from the initial state; an outcome that would be impossible above an intuitive sequentially consistent shared memory. The relaxed-memory behaviour that a state-of-the-art processor implementation exhibits may be much more subtle, perhaps involving notions of data and control dependencies, different kinds of barrier, various atomic operations, and so on. Multiprocessors effectively reveal (via such behaviour) more about their implementation choices than uniprocessors do, and so to accommodate changes in these choices their architectural specifications must be correspondingly looser.

However, in practice these specifications are often so flawed as to be useless. In our previous work on x86 memory models [5], [6] we detail problems with a number of Intel and AMD specifications. In some cases they are simply too ambiguous to be understood; in some cases they are too loose, making guarantees that are much weaker than the actual implementations and that do not suffice for reasonable programming; and in some cases they are simply wrong, guaranteeing that some behaviour, that real processors actually exhibit, cannot happen.

There may be many reasons for this, but a key technical point is that informal natural-language prose is a *very* poor medium in which to express subtle loose specifications. It is almost inevitably ambiguous, and prose specifications cannot be directly used in verification — for testing that processor implementations conform to the architecture, or for testing that software runs correctly above the architecture, not merely above some particular implementation, or of mechanised proof of either kind of result. It is in this sense that multiprocessor

architectures do not really exist: they are not well-defined artefacts that can be used.

We argue instead (as others have in the past) that architectures, and particularly their more intricate concurrency aspects, should be rigorously defined. If it were not for the need for looseness, one could just have golden executable models, in some well-defined deterministic programming language. But as it is, one must express architectures in other forms, e.g. by giving nondeterministic abstract machines or axiomatic descriptions. In our previous work we defined a memory model for x86 processors, x86-TSO [6], using both of those styles and proving equivalence between them.

To the best of our knowledge, it meets the criteria above. It is similar to the relatively well-understood SPARC TSO model and so it should be feasible to program above it. In the (limited) black-box testing we have been able to carry out, it appears to be sound with respect to current implementations. The abstract-machine presentation of the model should be widely accessible, to programmers, hardware architects, and programming language designers and implementers. Obviously we cannot comment on the future intentions of vendors, but some recent vendor specifications explicitly rule out a key example of weaker behaviour, and x86-TSO appears to be consistent with current folk wisdom. In addition:

6) it is completely unambiguous and mathematically precise, expressed in the logic of the HOL4 mechanised proof assistant [7];
7) it is testable: we have a tool that given a litmus-test program can compute the set of all the executions that the model allows (currently this tool is hand-written from the model, but ideally it would be derived from the statements of the model); and
8) it is integrated with an equally precise semantics for (a fragment of) the instruction set, replacing the conventional pseudocode by precise definitions of the sequential behaviour of instructions. This can be used directly for testing [5]. (Similar rigorous specifications for the sequential behaviour of ARM instructions have been produced by Fox [8].)

It must be emphasised that this mathematical precision does not come at the price of any over-specification: a precise specification is by no means necessarily a tight specification.

To conclude, for us to have any hope of building reliable systems, as multiprocessors become ever more dominant, it is essential that the architectural interfaces between hardware and software become well-defined and comprehensible to both sides, so that it can be tested (or even proved) that hardware conforms to its architectural specification, and that software runs correctly above an architecture. This will be all the more true as new proposals are made, e.g. for transactional memory, and as low-level concurrent programming is no longer confined (if it ever was) to a handful of operating system lock implementers.

The ambiguity of informal prose is a very poor way of delimiting the intended range of behaviour of a loose specification; it should be replaced by simple, but rigorous, mathematical techniques.

ACKNOWLEDGMENTS

This is based on joint work with Mark Batty, Anthony Fox, Magnus Myreen, Scott Owens, Susmit Sarkar, Jaroslav Ševčík, and Tom Ridge, of the University of Cambridge; and Jade Alglave, Thomas Braibant, Luc Maranget, and Francesco Zappa Nardelli, of INRIA. EPSRC funding, in the form of grant EP/F036345, is gratefully acknowledged.

REFERENCES

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual (5 vols)*. Intel Corporation, Mar. 2009, rev. 30.
[2] *AMD64 Architecture Programmer's Manual*. Advanced Micro Devices, Sep. 2007, (3 vols).
[3] ARM, *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)*, April 2008.
[4] *Power ISA™ Version 2.06*. IBM, 2009.
[5] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave, "The semantics of x86-CC multiprocessor machine code," in *Proc. POPL 2009*, Jan. 2009.
[6] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-TSO," in *Proc. TPHOLs, LNCS 5674*, 2009, pp. 391–407.
[7] "The HOL 4 system," http://hol.sourceforge.net/.
[8] A. C. J. Fox, "Formal specification and verification of ARM6." in *Proc. TPHOLs, LNCS 2758*, 2003, pp. 25–40.