Weakness is a virtue

Position paper proposed to EC^2 2013

Jade Alglave

University College London

Abstract. The number of interleavings that a concurrent program can have is typically given as the key difficulty in automatic analysis of concurrent software. Weak memory, as implemented by modern multiprocessors such as Intel x86, IBM Power and ARM, is generally believed to make this problem even harder. On the contrary, we believe that by embracing rather than fleeing from weak memory, we can obtain efficient verification techniques.

1 Introduction

Automatic verification of concurrent programs represents a challenge, whether it aims at proving the full correctness of a program (*e.g.* a program sorting a list actually sorts the list), or at checking specific properties (*e.g.* the program is free of data races) short of full correctness. Full correctness is rarely proved without the help of the user. On the contrary, checking some specific properties can be done in an automated manner. For sequential programs impressive practical results in this direction have been obtained by marrying verification and static program analysis, exemplified by the application of SLAM [BBC$^+$] to industrial device drivers and Astree [CCF$^+$05] to Airbus code.

It seems to us that concurrent verification still struggle to scale. The very few existing tools for concurrency verify programs in the hundreds of lines of code, but hardly any will verify a thousand lines [DKW08]. We propose here an hypothesis (backed up by initial experimental evidence in a bounded model-checking setting [AKT13]) that we believe could enhance the scalability of automatic tools checking that a concurrent program does not violate certain safety-critical properties of interest.

2 Background

To check properties, we need to define an execution model describing the behaviour of a program. Formal methods traditionally resort to Lamport's *Sequential Consistency* (SC) [Lam79], where *"the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."*.

Thus on SC, an execution is an *interleaving* of the instructions of the program, *i.e.* a *total order*. Choosing SC as execution model poses at least two problems.

SC and Weak Memory First, the large number of interleavings modelling the executions of a program makes their enumeration intractable. *Context bounded* methods [QR05,MQ05,LR09] can reduce the number of interleavings to consider, but are unsound in general. On the Fibonacci example of [Bey12], the Poirot tool, which implements the method of [LR09], gives wrong results due to the bound (see [AKT13]). *Partial order reduction* [Pel93,God96,FG05] reduces the interleavings in a sound way, but suffers from limited scalability. The ESBMC

tool [CF11], which implements this method, needs 13.8s to analyse Fibonacci, whereas it takes 0.3s to analyse a sequential version.

We believe that the intractability of interleavings already hints at rethinking if not SC, at least its formulation: we do not need total orders in the definition of SC. Multiprocessors (*e.g.*, Intel x86 or IBM Power) take us further, by forcing us to reconsider SC altogether.

Indeed, for performance reasons, the chips implement *weak memory models*, featuring for example *instruction reordering* or *store buffering* (appearing on x86), or *store atomicity relaxation* (a particularity of Power and ARM). These chips allow more behaviours than SC, which has a dramatic effect on programmers, most of whom learned to program under the assumption of SC.

These observations meet the one made by Hoare in [Hoa72], that attributing an interleaving semantics to parallel programs is problematic, in particular due to the following difficulties:

- *"implementing the interleaving on genuinely parallel hardware"*, which we interpret as multiprocessors featuring weak memory models;
- *"designing programs to control the fantastic number of combinations involved in an arbitrary interleaving"*, which echoes the aforementioned difficulties in current model-checking methods.

We aim at verifying concurrent programs. Naturally, the semantics modelling how programs behave influences the verification process. Models roughly fall into two classes: *operational* and *axiomatic*.

Operational and Axiomatic Models On the one hand, in model-checking and program analysis, models commonly adopt an operational style, where an execution is valid if it is produced by a machine switching between tasks. This style models executions via interleavings, with transitions accessing the memory (as on SC), and others accessing buffers or queues, implementing the features of the hardware. This builds on interleaving notions, hence inherits the limitations of SC interleaving based verification methods, *e.g.* the *"severely limited scalability"*, as [LNP+12] puts it. For example, [ABP11] (restricted to Sun Total Store Order, TSO) bounds the number of context switches.

On the other hand, several hardware vendors' specifications are in terms of partial orders [spa94,alp02]. Following this lead, we previously defined a large class of memory models (including x86, Power and ARM) [AMSS10,AMSS12,Alg12]. An execution is defined via partial orders over memory accesses, *e.g.* the program order in a thread, or the communication through memory.

3 Hypothesis

The study of these models led us to the following conclusion. It seems to us that the common opinion within the model-checking community is that we should stick to SC, and more crucially to its operational definition, because it is easier to reason with. On the contrary, we believe that by embracing rather than fleeing from weak memory, we can obtain efficient verification techniques, if we adopt the right models. To demonstrate this contention, we would like to show that:

Weakness is a virtue.

If the weakness leads to performant chips, why would it not lead to performant verification? Programming and analysis could fit the models, rather than try to control or harness them.

In other terms, *verifying a program againt weak memory models should be no harder than against SC*. Certain programs should even be easier to verify under weak memory: if a program exploits the weakness of the model (*e.g.* lock-free idioms), it fits closely the model it was designed for. Thus, if the analysis techniques also fit the models, the verification process might get altogether easier.

More generally, whether considering SC or weak memory, *partial orders can help verification to scale, by avoiding to enumerate interleavings*. Preliminary experiments provide initial evidence for our hypothesis: we were able to verify, in a bounded model-checking setting, a queue mechanism in Apache HTTP server software, which represents 28900 lines of code, in $2.8s$, when it takes $1.7s$ to analyse a particular interleaving of the same program [AKT13].

Partial orders filtered by axiomatic models seem an ideal setting, as they are constraint-based specifications to exploit the efficiency of constraint solvers, *e.g.* SAT or SMT, as done in [BAM07,SW10,SW11,AKT13] for programs with bounded loops. Note that this is radically different from partial order reduction, which considers total orders as primary, and uses partial orders merely as an optimisation tool, to reduce the number of total orders to examine.

4 Perspectives

Partial order models (also known as *independence* or *true concurrency models*) traditionally step away from interleavings, as done *e.g.* in [Win82,Pra82,Pra86,BAF94]. Yet, as Hayman and Winskel note in [HW08]: *"It is surprising that, to our knowledge, there has been no comprehensive study of the semantics of programming languages inside an independence model."*.

Indeed, such models have almost never been used to define a semantics for a language, except communicating processes languages, *e.g.* CCS [Win82] and CSP [Bro02], and programs with while loops in [HW08]. Mathematical properties of such models have been studied in detail, but in isolation from their use in the interpretation of programming languages.

This suggests the following topics of research:

- the invention of semantics of real-world assembly and imperative concurrent programing languages based on partial order models;
- the invention of new abstractions of these models enabling scalable automatic verification.

Our hypothesis concerning independence models for verification extends naturally from memory architectures to distributed systems. Many people have observed that the modern memory architectures can be thought of as distributed systems, and the weakness in shared-memory systems is certainly reminiscent of weak consistency in databases. It will be particularly important to develop formal foundations which underpin these informal similarities and observations. An interesting step in this direction has been taken in recent work by Burckhardt, Gotsman, and Yang, which generalises models such as ours [Alg12] to show that independence models are useful to describe properties of distributed systems (see [BGY]).

References

[ABP11] M. Atig, A. Bouajjani, and G. Parlato. Getting Rid of Store-Buffers in the Analysis of Weak Memory Models. In *CAV*, 2011.

[AKT13] J. Alglave, D. Kroening, and M. Tautschnig. Partial Orders for Efficient BMC of Concurrent Software. In *CAV*, 2013.

[Alg12] J. Alglave. A Formal Hierarchy of Weak Memory Models. In *FMSD*, 2012.

[alp02] Alpha Architecture Reference Manual, Fourth Edition, 2002.

[AMSS10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models. In *CAV*, 2010.

[AMSS12] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models (Extended Version). In *FMSD*, 2012.

[BAF94] Y. Ben-Asher and E. Farchi. Using True Concurrency to Model Execution of Parallel Programs. In *IJPP*, 1994.

[BAM07] S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.

[BBC+] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenbergand C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys 2006*.

[Bey12] D. Beyer. Competition on software verification - (SV-COMP). In *TACAS*, 2012.

[Bro02] S. Brookes. Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes. In *CONCUR*, 2002.

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min, D. Monniaux, and X. Rival. The astre analyzer. In *ESOP*, 2005.

[CF11] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *ICSE*, 2011.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. In *TCAD*, 2008.

[FG05] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model-Checking Software. In *POPL*, 2005.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, 1996.

[Hoa72] T. Hoare. Towards a Theory of Parallel Programming. In *Operating Systems Techniques*, 1972.

[HW08] J. Hayman and G. Winskel. Independence and Concurrent Separation Logic. In *LMCS*, 2008.

[Lam79] L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 1979.

[LNP+12] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, 2012.

[LR09] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *FMSD*, 2009.

[MQ05] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, 2005.

[Pel93] D. Peled. All from one, one for all. In *CAV*, 1993.

[Pra82] V. Pratt. On the composition of processes. In *POPL*, 1982.

[Pra86] V. Pratt. Modeling Concurrency with Partial Orders. In *International Journal of Parallel Programming*, 1986.

[QR05] S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. In *TACAS*, 2005.

[spa94] Sparc Architecture Manual Version 9, 1994.

[SW10] N. Sinha and C. Wang. Staged Concurrent Program Analysis. In *FSE*, 2010.

[SW11] N. Sinha and C. Wang. On Interference Abstractions. In *POPL*, 2011.

[Win82] G. Winskel. Event Structure Semantics for CCS and Related Languages. In *ICALP*, 1982.