

Multi-Core Memory Models and Concurrency Theory

Edited by

Hans J. Boehm¹, Ursula Goltz², Holger Hermanns³, and
Peter Sewell⁴

1 Hewlett Packard Labs - Palo Alto, US, Hans.Boehm@hp.com

2 TU Braunschweig, DE, goltz@ips.cs.tu-bs.de

3 Universität des Saarlandes, DE, hermanns@cs.uni-sb.de

4 University of Cambridge, GB, peter.sewell@cl.cam.ac.uk

Abstract

This report documents the programme and the outcomes of Dagstuhl Seminar 11011 “Multi-Core Memory Models and Concurrency Theory”.

Seminar 03.–01. January, 2011 – www.dagstuhl.de/11011

1998 ACM Subject Classification C.1.2 [Multiple Data Stream Architectures (Multiprocessors)]: Parallel processors; D.1.3 [Concurrent Programming]: Parallel programming; F.3.1 [Specifying and Verifying and Reasoning about Programs]

Keywords and phrases Relaxed Memory Models, Concurrency Theory, Multi-Core, Semantics, Parallel Programming, Cache Coherence

Digital Object Identifier 10.4230/DagRep.1.1.1

Edited in cooperation with Christian Eisentraut and Malte Lochau

1 Executive Summary

Hans J. Boehm (Hewlett Packard Labs - Palo Alto, US)


Ursula Goltz (TU Braunschweig, DE)

Holger Hermanns (Universität des Saarlandes, DE)

Peter Sewell (University of Cambridge, GB)

Christian Eisentraut (Universität des Saarlandes, DE)

Malte Lochau (TU Braunschweig, DE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Hans J. Boehm, Ursula Goltz, Holger Hermanns, Peter Sewell, Christian Eisentraut, and Malte Lochau

The current and future trend to multi-core and many-core computing systems suggests that within the next decade, concurrent multi-threaded programming will continue to replace sequential programming as the standard programming paradigm. However, concurrency and modern computer architecture do not go together easily:

- Current programming language memory models are still incomplete. Mainstream languages such as Java increasingly promise sequential consistency for data-race-free programs. However, how data races can be handled in a way that supports reasonable performance, security, and debugability, is currently completely unknown.
- Hardware specifications are so informal that it is very hard to know whether we have a correct implementation of the language specs (if we knew how to specify those fully). It is not clear that existing ISAs, which have a long history, are a good match for the language



Except where otherwise noted, content of this report is licensed under a Creative Commons BY-NC-ND 3.0 Unported license

Multi-Core Memory Models and Concurrency Theory – 11011, *Dagstuhl Reports*, Vol. 1, Issue 1, pp. 1–26

Editors: Hans J. Boehm, Ursula Goltz, Holger Hermanns, and Peter Sewell



DAGSTUHL
REPORTS

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

semantics we are developing. There is an argument that this misalignment continues to encourage languages to support complex low-level constructs.

- The concurrent algorithms that are now being developed, and which are key to exploiting multiprocessors (via high-performance operating systems, hypervisor kernels, and concurrency libraries), are very subtle, so informal reasoning cannot give high confidence in their correctness.
- While there is a rich literature on concurrency theory, and extensive prior work on software verification for concurrency software (including process calculi, model-checking, temporal logics, rely-guarantee reasoning, and separation logic), almost all of it assumes a global notion of time, unrealistic though that is for these systems and algorithms.

Concurrency theory has a long tradition in investigating the foundational principles of concurrent computation, devoted to parallelism and causality of operations. It has created a wealth of models and notions readily applicable in this semantic challenge. Recent developments in the research communities of programming languages and concurrency theory, respectively, indeed show a growing trend towards cross-fertilization.

This seminar has fostered cross-fertilization of different expertises that will help to develop novel practical and, at the same time, theoretically sound paradigms for multi-core programming. It brought together concurrency theorists, memory model experts, computer systems architects, compiler experts, and formal verification researchers. The aim of the seminar was to address in particular:

1. Survey of problem domain: state of the practice in multi-core-programming and state of the art in memory consistency models.
2. Application of concurrency theory approaches to reveal underlying concepts of parallelism, reordering, causality, and consistency.
3. Cross-fertilization across formal approaches to memory consistency models and semantics of multithreaded programming.
4. Attack points for formal analysis and computer aided programmer support.

Many of the questions that stood at the outset of this seminar have not been conclusively answered, thus yielding many potentials for further investigation. However, what makes this seminar uniquely successful is that it initiated a vivid exchange between a multitude of different scientific and industrial communities. During the seminar, it became clear that the current and future challenges of multi-core programming and memory models design in software and hardware can only be solved if the communities continue to exchange ideas and will learn from each other.

Schedule

Monday

Evening Session

19:00 Welcome

19:30-20:30 5 min intro talks

Tuesday

Morning session

9:00 HANS BOEHM – *Memory Models for Threads in Mainstream Programming Languages*

10:40 Coffee Break

11:00 more 5 min intro talks

11:30 SCOTT OWENS – *x86-TSO*

12:15 Lunch

14:00 rest of 5 min intro talks

15:00 MILO MARTIN – *InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors*

15:40 Coffee Break

16:30 DOUG LEA – *Some Weak Idioms*

17:15 MARK HILL – *Calvin: Deterministic or Not? Free Will to Choose*

17:30 VIJAY SARASWAT – *RAO Memory Model*

Wednesday

Morning session (chair: Rob van Glabbeek)

9:00 GERARD BOUDOL – *Why is True Concurrency Theory (Possibly) Relevant to the Study of Relaxed Memory Models*

9:45 SARITA ADVE – *Rethinking Parallel Languages and Hardware for Disciplined Parallelism*

10:30 Coffee Break

11:15 HEIKE WEHRHEIM – *Verifying Linearisability of a Lazy Concurrent Set*

11:35 MAGED MICHAEL – *Memory Ordering Tradeoffs*

11:55 ALEXEY GOTSMAN – *Modular Verification of Preemptive OS Kernels*

12:15 Lunch

14:00 Hike

Afternoon session (chair: Hans Boehm)

15:30 Coffee Break

16:00 SUSMIT SARKAR – *Understanding POWER Multiprocessors*

16:45 MADAN MUSUVATHI – *A Case for a SC-Preserving Compiler*

17:05 LUIS CEZE – *A Case for Concurrency Exceptions*

18:00 Dinner

Evening session

19:30 ARVIND – *Commit-Reconcile and Fences (CRF): A Memory Model for Compiler Writers and Architects*

20:15 BRANDON LUCIA – *Detecting, Avoiding and Understanding Errors in Concurrent Programs*

Thursday

Morning session (chair: Doug Lea)

- 9:00** MARK BATTY – *Mathematizing C++ Concurrency*
9:45 JAMES RIELY – *Generative Operational Semantics for Relaxed Memory Models*
10:10 ANDREAS LOCHBIHLER – *A Unified Machine-Checked Model for Multithreaded Java*
10:30 Coffee Break
11:15 STEPHAN DIESTELHORST – *Extending the AMD64 Memory Model with an Advanced Synchronization Facility*
11:55 ERIK HAGERSTEN – *Selling Your Memory Model Ideas to HW Architects*
12:15 Lunch

Afternoon session

- 14:00** JAROSLAV SEVCIK – *Validity of Program Transformations (in the Java Memory Model)*
14:45 MICHAEL KUPERSTEIN – *Partial Coherence Abstractions*
15:05 LAURA EFFINGER-DEAN – *Simpler Reasoning About Variables in Multithreaded Programs*
15:30 Coffee Break
16:15 PAUL MCKENNEY – *Multi-Core Memory Models and Concurrency Theory: A View from the Linux Community*
16:40 JADE ALGLAVE – *Fences in Weak Memory Models*
17:00 SELA MADOR-HAIM – *Generating Litmus Tests for Contrasting Memory Consistency Models*
17:20 SEBASTIAN BURCKHARDT – *Concurrent Revisions: A Strong Alternative to SC*
18:00 Dinner

Evening session

- 19:30** SAMUEL MIDKIFF – *Compilers and Memory Models – Selected Topics*

Friday

Morning session

- 9:00** LISA HIGHAM – *Specifying Memory Consistency Models for Write-Buffer Multiprocessors and Proving Them Correct*
9:20 GUSTAVO PETRI – *Speculation for Relaxed Memory: Using “True Concurrency”*
9:40 SIBYLLE FROESCHLE – *True-Concurrency: Foundational Concepts*
10:00 RICHARD BORNAT AND MIKE DODDS – *Program Logics: We have come a Long Way, Baby*
10:20 Coffee Break
11:00 Open Problem session
12:15 Lunch

2 Table of Contents

Executive Summary

<i>Hans J. Boehm, Ursula Goltz, Holger Hermanns, Peter Sewell, Christian Eisentraut, and Malte Lochau</i>	1
---	---

Overview of Talks


Rethinking Parallel Languages and Hardware for Disciplined Parallelism <i>Sarita Adve</i>	7
Fences in Weak Memory Models <i>Jade Alglave</i>	8
Commit-Reconcile and Fences (CRF): A Memory Model for Compiler Writers and Architects <i>Arvind</i>	8
Mathematizing C++ Concurrency <i>Mark Batty</i>	9
Memory Models for Threads in Mainstream Programming Languages <i>Hans J. Boehm</i>	9
Program Logics: We have come a Long Way, Baby <i>Richard Bornat and Mike Dodds</i>	9
Why is True Concurrency Theory (Possibly) Relevant to the Study of Relaxed Memory Models <i>Gerard Boudol</i>	10
Concurrent Revisions: A Strong Alternative to SC <i>Sebastian Burckhardt</i>	10
A Case for Concurrency Exceptions <i>Luis Ceze</i>	11
Extending the AMD64 Memory Model with an Advanced Synchronization Facility <i>Stephan Diestelhorst</i>	11
Simpler Reasoning About Variables in Multithreaded Programs <i>Laura Effinger-Dean</i>	12
True-Concurrency: Foundational Concepts <i>Sibylle Froeschle</i>	13
Modular Verification of Preemptive OS Kernels <i>Alexey Gotsman</i>	13
Selling Your Memory Model Ideas to HW Architects <i>Erik Hagersten</i>	13
Specifying Memory Consistency Models for Write-Buffer Multiprocessors and Proving Them Correct <i>Lisa Higham</i>	14
Calvin: Deterministic or Not? Free Will to Choose <i>Mark D. Hill</i>	14

Partial Coherence Abstractions	
<i>Michael Kuperstein</i>	15
Some Weak Idioms	
<i>Doug Lea</i>	15
A Unified Machine-Checked Model for Multithreaded Java	
<i>Andreas Lochbihler</i>	15
Detecting, Avoiding and Understanding Errors in Concurrent Programs	
<i>Brandon M. Lucia</i>	16
Generating Litmus Tests for Contrasting Memory Consistency Models	
<i>Sela Mador-Haim</i>	17
InvisiFence: Performance-Transparent Memory Ordering in Conventional Multipro- cessors	
<i>Milo M. K. Martin</i>	17
Multi-Core Memory Models and Concurrency Theory: A View from the Linux Community	
<i>Paul McKenney</i>	18
Memory Ordering Tradeoffs	
<i>Maged Michael</i>	18
Compilers and Memory Models – Selected Topics	
<i>Samuel Midkiff</i>	19
A Case for a SC-Preserving Compiler	
<i>Madan Musuvathi</i>	19
x86-TSO	
<i>Scott Owens</i>	19
Speculation for Relaxed Memory: Using “True Concurrency”	
<i>Gustavo Petri</i>	20
Generative Operational Semantics for Relaxed Memory Models	
<i>James Riely</i>	20
RAO Memory Model	
<i>Vijay A. Saraswat</i>	21
Understanding POWER Multiprocessors	
<i>Susmit Sarkar</i>	21
Validity of Program Transformations (in the Java Memory Model)	
<i>Jaroslav Sevcik</i>	21
Verifying Linearisability of a Lazy Concurrent Set	
<i>Heike Wehrheim</i>	22
State of the Art and Open Problems	22
Participants	26

3 Overview of Talks

3.1 Rethinking Parallel Languages and Hardware for Disciplined Parallelism

Sarita Adve (University of Illinois - Urbana, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Sarita Adve

Main reference DeNovo: Rethinking Hardware for Disciplined Parallelism, Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Bocchino, Sarita V. Adve, and Vikram V. Adve, Second USENIX Workshop on Hot Topics in Parallelism (HotPar), 2010.

URL <http://denovo.cs.illinois.edu/Pubs/10-hotpar-denovo.pdf>


For parallelism to become tractable for mass programmers, shared memory programming languages and environments must evolve to enforce disciplined practices that ban "wild shared-memory behaviors;" e.g., unstructured parallelism, arbitrary data races, and ubiquitous non-determinism. This software evolution is also a rare opportunity for hardware designers to rethink hardware from the ground up to exploit opportunities exposed by such disciplined software models. Such a co-designed effort is more likely to achieve many-core scalability and large gains in power/performance than a software-oblivious hardware evolution. In this talk, we first briefly overview the Deterministic Parallel Java (DPJ) language that provides a disciplined shared-memory programming model. We then discuss DeNovo, a hardware architecture motivated by languages such as DPJ. We show how a disciplined parallel programming model can greatly simplify the cache coherence protocol and memory consistency model, while enabling a more efficient communication and cache architecture.

Our programming model ensures data-race-freedom, so DeNovo does not need to deal with protocol races and transient states, making it simpler and more extensible than software-oblivious protocols. Our programming model makes shared-memory side effects explicit, so each core keeps its cache coherent, eliminating the need for sharer-lists in a directory and associated invalidation traffic. To evaluate simplicity, we verify a base version of DeNovo with model checking and compare it with an otherwise equivalent state-of-the-art MESI protocol. The DeNovo version entails 25X fewer reachable states and takes 30X less time to verify. To evaluate extensibility, we add two sophisticated performance-enhancing optimizations to DeNovo: flexible bulk transfers and cache-to-cache direct data transfers. These add significant races and new protocol states with MESI, but not with DeNovo. With a simple extension, DeNovo's storage overhead breaks even with efficient MESI versions at about 30 core systems, but unlike MESI, it maintains scalability beyond that point. The net result is a system that seamlessly integrates message passing-like interactions within a shared memory programming model with improved design complexity, up to 67% reduction in memory stall time, and up to 70% reduction in network traffic.

This talk covers joint work with Rob Bocchino, Nicholas P. Carter, Byn Choi, Ching-Tsun Chou, Nima Honarmand, Rakesh Komuravelli, Robert Smolinski, Hyojin Sung, Vikram S. Adve, Tatiana Shpeisman, Marc Snir, and Adam Welc.

3.2 Fences in Weak Memory Models

Jade Alglave (University of Oxford, GB)


License  Creative Commons BY-NC-ND 3.0 Unported license
© Jade Alglave

Joint work of Luc Maranget, Susmit Sarkar and Peter Sewell

We present a class of relaxed memory models, defined in Coq, parameterised by the chosen permitted local reorderings of reads and writes, and the visibility of inter- and intra-processor communications through memory (e.g. store atomicity relaxation). We prove results on the required behaviour and placement of memory fences to restore a given model (such as Sequential Consistency) from a weaker one. Based on this class of models we develop a tool, *diy*, that systematically and automatically generates and runs litmus tests to determine properties of processor implementations. We detail the results of our experiments on Power and the model we base on them. This work identified a rare implementation error in Power 5 memory barriers (for which IBM is providing a workaround); our results also suggest that Power 6 does not suffer from this problem.

3.3 Commit-Reconcile and Fences (CRF): A Memory Model for Compiler Writers and Architects


Arvind (MIT, Cambridge, MA, USA)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Arvind

The rise of multicores is forcing us to readdress one of the fundamental problems of multi-threaded programming, namely memory models. Sequential Consistency (SC), in spite of its simple definition, has been found wanting because of exorbitant performance overheads in hardware implementations as well as opacity in reasoning at a high-level. Several weaker memory models have been proposed, all of which are motivated primarily by implementation concerns – in fact, the semantics of these memory models are given by the implementation rather than by any abstract rigorous method and understood only by experts. In view of this difficulty, many people think that we should just stick to Sequential Consistency as the interface between hardware and software. The use of various synchronization primitives and programming libraries has already made SC irrelevant for application programmers. There is no reason for compilers and architectures to adhere to SC if the higher-layers do not require it. We think that the Commit-Reconcile and Fences (CRF) memory model which was proposed earlier has the desirable properties for our purpose: its definition is given in terms of algebraic rules and it can directly model all the behaviors permitted by other relaxed memory models; CRF can be implemented in a variety of ways by the underlying machine; whether it be an in-order machine preserving sequential consistency or an aggressively out-of-order machine. So the high level programs can be ported across machines with different memory models. The talk will describe CRF and show how it could be used to give a meaningful semantics to a memory model for Java.

3.4 Mathematizing C++ Concurrency

Mark Batty (University of Cambridge, UK)


License  Creative Commons BY-NC-ND 3.0 Unported license
© Mark Batty

The next versions of C++ (C++0x) and of C (C1X) will have new concurrency features, aiming to support high-performance code with well-defined semantics. Unfortunately, as we near the end of the long standardization process, not all has been well. Unsurprisingly, the prose specification style of the draft standards is poorly suited to describe the complex design of the relaxed memory model, and in fact there have been significant problems.

I will discuss work on formalization of the memory model, what was broken, and some resulting improvements to the C++0x draft standard. In addition I will present a tool, Cppmem, for graphically exploring the semantics of small concurrent C++0x programs, and describe a proof of the correctness of a compilation strategy targeting x86-TSO.

3.5 Memory Models for Threads in Mainstream Programming Languages

Hans J. Boehm (HP Labs - Palo Alto, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Hans J. Boehm

Although multithreaded programming languages are common, there has been a surprising amount of confusion surrounding the basic meaning of shared variables. There is finally a growing consensus, both that programming languages should by default guarantee an interleaving-based semantics for data-race-free programs, and on what that should mean. We discuss the motivation for, and both implementation and user consequences of, this guarantee.

Unfortunately, it is also increasingly clear that such a guarantee, though useful, is insufficient for languages intended to support sand-boxed execution of untrusted code. The existing solution in Java is only partially satisfactory. The solution to this problem is far less clear.

3.6 Program Logics: We have come a Long Way, Baby

Richard Bornat (Middlesex University, GB) and Mike Dodds (Cambridge University, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Richard Bornat and Mike Dodds

My brief, given at the last minute, was to persuade the audience that program proof has come a long way since Hoare logic was promoted in the 1970s. We were (and are) woefully unprepared to give a fair survey of world work in the area: we are simple program provers. The talk ended for an apology for that deficiency, and this abstract must start with one.

Hoare logic was not a solution to the ‘Software Engineering Problem’. Loops were (necessarily) hard; there was no effective treatment of procedures; arrays were difficult to

deal with, and there was no treatment of pointers. Nevertheless it was the basis of successful program verification work in the 80s and 90s, notably in B, Spark, JML and others.

Reynolds and O’Hearn, in 2000, built on work of Burstall’s (separated lists can be dealt with separately) to make separation logic, an extension of Hoare logic which deals with access and assignment to the heap, with *local reasoning*. An effective treatment of pointers led, quite quickly, to a treatment of concurrency (O’Hearn’s CSL).

Separation logic has had an impact on program analysis and may (because of local reasoning) be able to make an impact on the problem, much discussed in the workshop, of detecting data races in Java programs. At present the Abductor tool (DiStefano) can analyse the whole Linux kernel in 20 minutes, finding space leaks in list-manipulating procedures; Windows device drivers can be handled in seconds, finding safety bugs (Yang, Cook).

In concurrency there are separation logic hand proofs of several intricate algorithms, and work on automatic proof of linearisability.

In summary, there is much useful work which can be exploited. We also mentioned an important negative point: ownership types don’t work. Ownership, as various proofs of simple algorithms demonstrates, is dynamic not static.

3.7 Why is True Concurrency Theory (Possibly) Relevant to the Study of Relaxed Memory Models


Gerard Boudol (INRIA Sophia Antipolis, FR)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Gerard Boudol

I briefly introduce "true concurrency" theory, and in particular the formalism of labelled transition systems with independence which, I think, are appropriate to provide abstract formalization of relaxed memory models.

3.8 Concurrent Revisions: A Strong Alternative to SC

Sebastian Burckhardt (Microsoft Research - Redmond, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Sebastian Burckhardt


Main reference Sebastian Burckhardt, Alexandro Baldassin, Daan Leijen, “*Concurrent programming with revisions and isolation types*”, pp. 691–707, OOPSLA’10, ACM, 2010.

URL <http://dx.doi.org/10.1145/1869459.1869515>

Although considered a strong memory model, sequential consistency (SC) is in fact still weaker than desirable in the sense that the semantics is based on nondeterministic interleavings. In contrast, Concurrent Revisions, a programming model for shared-memory concurrency/parallelism, is deterministic: the semantics of shared memory accesses are based on deterministic replication & conflict resolution rather than nondeterministic arbitration as in SC.

3.9 A Case for Concurrency Exceptions


Luis Ceze (University of Washington, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Luis Ceze

We argue in this talk that concurrency errors should be treated as exceptions, i.e., have fail-stop behavior and precise semantics. We propose an exception model based on conflict of synchronization-free regions, which precisely detects a broad class of data-races. We show that our exceptions provide enough guarantees to simplify high-level programming language semantics and debugging, but are significantly cheaper to enforce than traditional data-race detection. We also propose a new exception that enforces disciplined shared-memory communication in a code-centric manner. To make the performance cost of enforcement negligible, we propose architecture support for accurately detecting and precisely delivering concurrency exceptions.

3.10 Extending the AMD64 Memory Model with an Advanced Synchronization Facility

Stephan Diestelhorst (AMD - Dornbach, DE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Stephan Diestelhorst

Joint work of Diestelhorst, Stephan; Christie, David; Hohmuth, Michael; Pohlack, Martin;

Main reference Jae-Woong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, Dan Grossman: *ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory*. MICRO 2010: 39-50

URL <http://dx.doi.org/10.1109/MICRO.2010.40>

Current general-purpose, high-performance microprocessor cores aggressively reorder instructions to increase throughput. With the wide availability of shared memory multi-core systems, architects have to pay attention to the architecturally visible memory semantics provided by these systems.

In our talk, we will look at how current AMD microprocessors provide strong memory semantics while still allowing aggressive reordering on the microarchitectural level. We will also highlight the functionality of the HyperTransport (tm) interconnect that connects processors and provides cache coherence and contributes significantly to the overall memory semantics.

On top of this foundation, we introduce AMD's Advanced Synchronization Facility (ASF), an experimental AMD64 ISA extension. ASF provides applications with speculative regions, akin to transactions in transactional memory, which allow atomic read-modify-write constructs on multiple independent memory locations. In the course of our presentation of ASF, we highlight various design decisions and illustrate why and how extensions to existing complex microarchitectures need to be simple.


References

- 1 Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, Dan Grossman. *ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory*. Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43), 2010
- 2 Felber, P.; Riviere, E.; Moreira, W.M.; Harmanci, D.; Marlier, P.; Diestelhorst, S.; Hohmuth, M.; Pohlack, M.; Cristal, A.; Hur, I.; Unsal, O.S.; Stenström, P.; Dragojevic, A.;

- Guerraoui, R.; Kapalka, M.; Gramoli, V.; Drepper, U.; Tomic, S.; Afek, Y.; Korland, G.; Shavit, N.; Fetzer, C.; Nowack, M.; Riegel, T. *The VELOX Transactional Memory Stack*. IEEE Micro Special Issue on European Multicore Processing Projects, 2010
- 3 Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack. *Sane Semantics of Best-effort Hardware Transactional Memory*. 2nd Workshop on the Theory of Transactional Memory (WTTM'10), 2010
 - 4 Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, Etienne Riviere *Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack*. EuroSys, 2010
 - 5 Jaewoong Chung, David Christie, Martin Pohlack, Stephan Diestelhorst, Michael Hohmuth, Luke Yen. *Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support*. 5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'10), 2010
 - 6 Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, Dave Christie, Jae-Woong Chung, Luke Yen. *Implementing AMD's Advanced Synchronization Facility in an Out-of-order x86 Core*. 5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'10), 2010
 - 7 Conway, P.; Hughes, B. *The AMD Opteron Northbridge Architecture*. Micro, IEEE , vol.27, no.2, pp.10-21, March-April 2007
 - 8 Conway, P.; Kalyanasundharam, N.; Donley, G.; Lepak, K.; Hughes, B. *Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor*. Micro, IEEE , vol.30, no.2, pp.16-29, March-April 2010
 - 9 AMD "Advanced Synchronization Facility" Proposal.
<http://developer.amd.com/cpu/asf/Pages/default.aspx>

3.11 Simpler Reasoning About Variables in Multithreaded Programs

Laura Effinger-Dean (University of Washington - Seattle, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Laura Effinger-Dean


Joint work of Effinger-Dean, Laura; Boehm, Hans-J.; Chakrabarti, Dhruva; Joisha, Pramod

We present a simple characterization of code regions for which we can prove that a given shared variable cannot be modified by other threads in the system.

Following the new C++0x standard, we assume data-race-freedom. Our "interference-free regions" expand on simple synchronization-free regions to include limited patterns of synchronization operations. In particular, our observations demonstrate that it is legal to eliminate a redundant variable access across a lock or unlock operation. By coalescing the interference-free regions for multiple accesses to the same variable, we can extend this characterization to complex control flow.

3.12 True-Concurrency: Foundational Concepts

Sibylle Froeschle (Universität Oldenburg, DE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Sibylle Froeschle


In this talk I will give a short introduction to models and foundational concepts in concurrency theory with an emphasis on true-concurrency.

In particular, I will mention the computational dichotomy of true-concurrency: truly-concurrent versions of concepts such as behavioural equivalences can be computationally hard but for classes with good structural properties they are often efficiently decidable.

I will speculate on some connections to the challenges of multi-core memory models.

3.13 Modular Verification of Preemptive OS Kernels

Alexey Gotsman (IMDEA Software - Madrid, ES)


License  Creative Commons BY-NC-ND 3.0 Unported license
© Alexey Gotsman

Most major OS kernels today run on multiprocessor systems and are preemptive: it is possible for a process running in the kernel mode to get descheduled. Existing modular techniques for verifying concurrent code are not directly applicable in this setting: they rely on scheduling being implemented correctly, and in a preemptive kernel, the correctness of the scheduler is interdependent with the correctness of the code it schedules. This interdependency is even stronger in mainstream kernels, such as Linux, FreeBSD or XNU, where the scheduler and processes interact in complex ways.

We propose the first logic that is able to decompose the verification of preemptive multiprocessor kernel code into verifying the scheduler and the rest of the kernel separately, even in the presence of complex interdependencies between the two components. This is achieved by establishing a novel form of refinement between an operational semantics of the real machine and an axiomatic semantics of OS processes, where the latter assumes an abstract machine with each process executing on a separate virtual CPU. The refinement is local in the sense that the logic focuses only on the relevant state of the kernel while verifying the scheduler. Our logic soundly inherits proof rules from concurrent separation logic to verify OS processes thread-modularly. We illustrate its power by verifying an example scheduler, modelled on the one from Linux 2.6.11.

3.14 Selling Your Memory Model Ideas to HW Architects

Erik Hagersten (University of Uppsala, SE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Erik Hagersten


First, I will discuss how architect tells good ideas from bad ideas by weighing tradeoffs between complexity (implying schedule slip), performance and power consumptions. Of the 1000s of the published architecture papers promising 10% performance improvement each year, only a fraction can possibly have impact since the performance improvement per year

due to architectural improvement is roughly 20% and not 10 000%. This is often due to the overwhelming complexity of the proposals.

Second, I will ask the formal method's community for favors. 1) Do not define how to implement the memory models: I will show examples of correct implementations for coherence and sequential consistency that violate such definitions. 2) When verifying a protocol, consider all transactions, not just the LD/ST/Atomics: They tend to only represent 10-20% of all transactions in a protocol. 3) Do consider all interacting protocols in the system, not just one at a time. 4) Finally, I urge protocol verification to not only verify coherence and memory models. Just as important (and often harder) properties to verify include livelock and deadlock properties.

3.15 Specifying Memory Consistency Models for Write-Buffer Multiprocessors and Proving Them Correct

Lisa Higham (University of Calgary, CA)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Lisa Higham

Joint work of Higham, Lisa; Kawash, Jalal; Jackson, LillAnne

Main reference Lisa Higham, LillAnne Jackson, Jalal Kawash: Specifying memory consistency of write buffer multiprocessors. pp. 2-42, ACM Trans. Comput. Syst. 25(1), 2007.

URL <http://doi.acm.org/10.1145/1189736.1189737>


Multiprocessor architectures employ various hardware components such as multiple busses, write-buffers, and replicated memory to enhance the efficiency.

As a consequence, the computations that can arise satisfy guarantees substantially weaker than sequential consistency. Given a specific multi-processor architecture, our goals are: 1) to develop a simple, precise predicate that specifies exactly what computations can arise from a multiprocessing program that executes on that architecture; and 2) prove that the predicate is correct.

This talk illustrates our techniques for achieving this goal through a case study of shared memory architectures that use write buffers.

3.16 Calvin: Deterministic or Not? Free Will to Choose





Mark D. Hill (University of Wisconsin - Madison, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Mark D. Hill

Calvin implements optional determinism with Total Store Order (TSO) compatibility for about a 20% performance loss relative to a conventional system. This talk is a short adaptation for non-architects of an upcoming HPCA 2011 paper

3.17 Partial Coherence Abstractions

Michael Kuperstein (Technion - Haifa, IL)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Michael Kuperstein





Joint work of Kuperstein, Michael; Vechev, Martin; Yahav, Eran;

We present an approach for automatic verification of concurrent programs running under relaxed memory models. Verification under relaxed memory models is a hard problem. Given a finite state program and a safety specification, verifying that the program satisfies the specification under a sufficiently relaxed memory model is undecidable. For somewhat stronger memory models, the problem is decidable but has non-primitive recursive complexity. In this paper, we focus on models that have store-buffer based semantics, e.g. SPARC TSO and PSO.

We use abstract interpretation to provide an effective verification procedure for programs running under this type of models. Our main contribution is a family of novel partial-coherence abstractions, specialized for relaxed memory models, which partially preserve information required for memory coherence and consistency.

3.18 Some Weak Idioms

Doug Lea (SUNY - Oswego, US)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Doug Lea

The design and implementation of core libraries and runtime systems entail several idiomatic non-sequentially-consistent constructions. This talk illustrates some such techniques encountered in publishing and transferring objects and messages.

3.19 A Unified Machine-Checked Model for Multithreaded Java

Andreas Lochbihler (KIT - Karlsruhe Institute of Technology, DE)

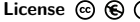
License     Creative Commons BY-NC-ND 3.0 Unported license
© Andreas Lochbihler

I present a machine-checked formalisation of the Java memory model (JMM) and connect it to an operational semantics for source and bytecode. It extends previous formalisations by dynamic memory allocations, thread spawns and joins, infinite executions, the wait-notify mechanism and thread interruption. I proved that the model provides the Java data race freedom (DRF) guarantee.

I instantiated the JMM with JinjaThreads, a large subset of Java (bytecode), thereby providing the missing link between operational semantics on statement and instruction level and the JMM. To discharge the assumptions of the DRF proof, I constructed sequentially consistent executions of source and bytecode.

3.20 Detecting, Avoiding and Understanding Errors in Concurrent Programs

Brandon M. Lucia (University of Washington - Seattle, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Brandon M. Lucia

Joint work of Lucia, Brandon M.; Ceze, Luis;

Main reference Brandon, L.; Ceze, L.; “Finding concurrency bugs with context-aware communication graphs.”
Proc. of 42nd Int’l Symp. on Microarchitecture, pp. 553–563. Dec. 2009.

URL <http://dx.doi.org/10.1145/1669112.1669181>

Programmers are the weak link in the chain of concurrent software development.

People make mistakes that lead to subtle concurrency errors that are difficult to find and fix. These errors degrade the reliability of software, and can lead to costly system failures. The movement of accessible concurrent programming to the mainstream is dependent on a solution to the problems posed by concurrency errors.





In this talk I will describe two different approaches to this problem.

First, I will discuss Bugaboo, a technique for automatically isolating patterns of inter-thread communication that lead to buggy program behavior. The key contribution of this work is the introduction of context-aware communication graphs, a new graphical data-flow abstraction for program execution. These graphs encode communication between instructions, and an abstract view of the order of communication events. By collecting context-aware communication graphs from many different execution, we can use statistical reasoning to identify communication events most likely related to a failure. Our reasoning is simple: communication that occurs often in buggy executions, and rarely or never in correct executions is likely related to the failure. We develop supervised and unsupervised approaches to error identification. We also describe hardware extensions that enable graph collection with negligible performance overhead.

Second, I will discuss ColorSafe and Atom-Aid, two similar techniques for avoiding atomicity violation bugs. Atomicity errors are the result of incorrect program synchronization: a programmer should have prevented a particular buggy interleaving of operations from different threads, but failed to do so. Prior work has shown that the manifestation of atomicity violations is characterized by an unserializable interleaving of memory accesses. We leverage serializability analysis to find atomicity bugs. Each thread maintains a history of locally performed memory accesses and a history of memory accesses performed by other threads. Periodically, at the end of fixed length execution epochs, threads analyze the serializability of recent memory access interleavings to find likely atomicity bugs. Upon finding a likely violation, our system uses dynamic atomic regions (viz. transactions) to prevent interleaving, thereby avoiding the buggy program behavior. In addition we describe a technique for generalizing serializability analysis to simultaneously consider multiple memory locations instead of single locations only. We develop simple hardware support to make these techniques efficient.

3.21 Generating Litmus Tests for Contrasting Memory Consistency Models

Sela Mador-Haim (University of Pennsylvania, US)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Sela Mador-Haim

Well-defined memory consistency models are necessary for writing correct parallel software. Developing and understanding formal specifications of hardware memory models is a challenge due to the subtle differences in allowed reorderings and different specification styles. To facilitate exploration of memory model specifications, we have developed a technique for systematically comparing hardware memory models specified using both operational and axiomatic styles. Given two specifications, our approach generates all possible multi-threaded programs up to a specified bound, and for each such program, checks if one of the models can lead to an observable behavior not possible in the other model. When the models differs, the tool finds a minimal “litmus test” program that demonstrates the difference. A number of optimizations reduce the number of programs that need to be examined. Our prototype implementation has successfully compared both axiomatic and operational specifications of six different hardware memory models. We describe two case studies: (1) development of a non-store atomic variant of an existing memory model, which illustrates the use of the tool while developing a new memory model, and (2) identification of a subtle specification mistake in a recently published axiomatic specification of TSO.

3.22 InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors

Milo M. K. Martin (University of Pennsylvania, US)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Milo M. K. Martin

Joint work of Blundell, Colin; Martin, Milo M.K.; Wenisch, Thomas F.;

Main reference Blundell, C.; Martin, M.M.K.; Wenisch, T.F.; InvisiFence: performance-transparent memory ordering in conventional multiprocessors, Proc. of 36th Int’l Symp. on Comp. Architecture (ISCA’09).

URL <http://dx.doi.org/10.1145/1555754.1555785>


A multiprocessor’s memory consistency model imposes ordering constraints among loads, stores, atomic operations, and memory fences. Even for consistency models that relax ordering among loads and stores, ordering constraints still induce significant performance penalties due to atomic operations and memory ordering fences. Several prior proposals reduce the performance penalty of strongly ordered models using post-retirement speculation, but these designs either (1) maintain speculative state at a per-store granularity, causing storage requirements to grow proportionally to speculation depth, or (2) employ distributed global commit arbitration using unconventional chunk-based invalidation mechanisms.

In this paper we propose INVISIFENCE, an approach for implementing memory ordering based on post-retirement speculation that avoids these concerns. INVISIFENCE leverages minimalistic mechanisms for post-retirement speculation proposed in other contexts to (1) track speculative state efficiently at block-granularity with dedicated storage requirements independent of speculation depth, (2) provide fast commit by avoiding explicit commit arbitration, and (3) operate under a conventional invalidation-based cache coherence protocol. INVISIFENCE supports both modes of operation found in prior work: speculating only when

necessary to minimize the risk of rollback-inducing violations or speculating continuously to decouple consistency enforcement from the processor core. Overall, INVISIFENCE requires approximately one kilobyte of additional state to transform a conventional multiprocessor into one that provides performance-transparent memory ordering, fences, and atomic operations.

3.23 Multi-Core Memory Models and Concurrency Theory: A View from the Linux Community

Paul McKenney (IBM - Beaverton, US)

License  Creative Commons BY-NC-ND 3.0 Unported license

© Paul McKenney

Main reference McKenney, Paul E., “Is Parallel Programming Hard, And, If So, What Can You Do About It?”, kernel.org, Corvallis, OR, USA 2011.

URL <http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html>


This talk presents multi-core memory models and concurrency theory from the viewpoint of a practitioner with 20 years experience producing production-quality shared-memory parallel code and with 10 years with the Linux kernel community. This experience has led to the following conclusions: (1) although abstraction is valuable, intimate knowledge of underlying software and hardware layers is equally valuable, (2) although general techniques are valuable, specialized techniques resulting in excellent performance, scalability, real-time response, and energy efficiency are equally valuable, (3) although concurrent-software validation techniques are valuable, techniques drawn from hardware validation may well prove more valuable, and (4) different levels of abstraction required different programming paradigms.

That said, within the Linux kernel community, organizational mechanisms are at least as important as specific techniques and tools. These organizational mechanisms include the maintainership hierarchy with its focus on quality assurance, an informal apprenticeship/-mentoring model, a strong tradition of design and code review, and aggressive pursuit of modularity and simplicity.

These mechanisms have the important side effect of focusing attention on techniques that are known to work well in a given situation, which allows ordinary practitioners to successfully produce production-quality parallel designs and code.

3.24 Memory Ordering Tradeoffs

Maged Michael (IBM TJ Watson Research Center - Yorktown Heights, US)

License  Creative Commons BY-NC-ND 3.0 Unported license

© Maged Michael

Joint work of Attiya, Hagit; Guerraoui, Rachid; Hendler, Danny; Kuznetsov, Petr; Michael, Maged; Vechev, Martin;

Main reference Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, Martin T. Vechev: Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, 487-498





URL <http://doi.acm.org/10.1145/1926385.1926442>

It is often the case that in designing concurrent algorithms that it appears that avoiding both atomic operations and store-load ordering is difficult. It is shown that it is impossible such patterns in algorithms for methods that are strongly non-commutative. The SNC condition can be avoided by specification relaxations, such as limiting concurrency, limiting the API,

relaxing determinism, or relaxing the requirement of linearizability. This result opens the door for trade-offs between synchronization overhead and specification.

3.25 Compilers and Memory Models – Selected Topics





Samuel P. Midkiff (Purdue University, US)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Samuel Midkiff

This talk discusses three important topics related to memory models. First, we argue that static compiler analyses are insufficient, and will always be insufficient, to certify programs as race-free. Thus, memory models for general purpose languages that depend on programs being race-free for correctness will almost certainly require some runtime checking. Second, we present some results showing the effectiveness of a tool that helps programmers to identify transaction regions, and thus help the programmer in the creation of race free programs. We also argue that analyses such as this one, lock assignment and other transformations handling programs with pointers benefit from the use of semantic information about standard library code. Semantic information we used included information about destructive data structure operations, commutativity of operations on the data structure, and whether the data structure is thread-safe. These properties follow directly from the specification of the code, and thus are trivially easy to specify for routines that implement the specification. Finally, we discuss the benefit of hardware that provides isolation for code regions in threads. We show that allowing the compiler to set and know regions that are excited in isolations, as is done in the BulkSC architecture, allows a sequentially consistent memory model to be implemented with an increase in performance.

3.26 A Case for a SC-Preserving Compiler





Madan Musuvathi (Microsoft Corp. - Redmond, US)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Madan Musuvathi

In this talk, I will argue against relaxing memory models in the compiler. I will demonstrate that a SC-preserving compiler, one which guarantees that every SC behavior of the binary is a SC behavior of the source program, is feasible with acceptable performance overhead. Time permitting, I will also demonstrate how static and dynamic analyses can further reduce this overhead.

3.27 x86-TSO

Scott Owens (University of Cambridge, GB)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Scott Owens


Exploiting the multiprocessors that have recently become ubiquitous requires high-performance and reliable concurrent systems code, for concurrent data structures, operating

system kernels, synchronisation libraries, compilers, and so on. However, concurrent programming, which is always challenging, is made much more so by two problems. First, real multiprocessors typically do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion.

In this talk we focus on x86 processors. We review several recent Intel and AMD specifications, showing that all contain serious ambiguities, some are arguably too weak to program above, and some are simply unsound with respect to actual hardware. We present a new x86-TSO programmer’s model that, to the best of our knowledge, suffers from none of these problems. It is mathematically precise (rigorously defined in HOL4) but can be presented as an intuitive abstract machine which should be widely accessible to working programmers. We illustrate how this can be used to reason about the correctness of a Linux spinlock implementation and describe a general theory of data-race-freedom for x86-TSO. This should put x86 multiprocessor system building on a more solid foundation; it should also provide a basis for future work on verification of such systems.

3.28 Speculation for Relaxed Memory: Using “True Concurrency”


Gustavo Petri (INRIA Sophia Antipolis, FR)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Gustavo Petri

We present a semantic framework to describe speculative computations of parallel programs. An important ingredient of our framework is the definition of *valid* speculation, for which we use standard true concurrency techniques. Interestingly, the effects of speculations are similar to those observed in relaxed memory models. We briefly show how to instantiate the TSO, PSO and RMO memory models of Sparc using this framework.

3.29 Generative Operational Semantics for Relaxed Memory Models

James Riely (DePaul University - Chicago, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© James Riely

Joint work of Jagadeesan, Radha; Pitcher, Corin; Riely, James;
Main reference Jagadeesan, R.; Pitcher, C.; Riely, J.; “Generative Operational Semantics for Relaxed Memory Models”, ESOP 2010, pp. 307–326.
URL http://dx.doi.org/10.1007/978-3-642-11957-6_17

The specification of the Java Memory Model (JMM) is phrased in terms of acceptors of execution sequences rather than the standard generative view of operational semantics. This creates a mismatch with language-based techniques, such as simulation arguments and proofs of type safety.





We describe a semantics for the JMM using standard programming language techniques that captures its full expressivity. For data-race-free programs, our model coincides with the

JMM. For lockless programs, our model is more expressive than the JMM. The stratification properties required to avoid causality cycles are derived, rather than mandated in the style of the JMM.

The JMM is arguably non-canonical in its treatment of the interaction of data races and locks as it fails to validate roach-motel reorderings and various peephole optimizations. Our model differs from the JMM in these cases. We develop a theory of simulation and use it to validate the legality of the above optimizations in any program context.

3.30 RAO Memory Model





Vijay A. Saraswat (IBM TJ Watson Research Center - Hawthorne, US)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Vijay A. Saraswat

Talk describes essence of the RAO Model presented in PPOPP 2007.

3.31 Understanding POWER Multiprocessors





Susmit Sarkar (University of Cambridge, GB)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Susmit Sarkar

IBM POWER multiprocessors have a very relaxed memory model (ARM is similar), including instances in which programmers can observe non-atomicity of stores, register shadowing, and speculative executions past branches. I will describe joint (ongoing) work with Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams on our extensive testing of Power G5, 5, 6, and 7, producing some perhaps-surprising results, and on an abstract-machine semantics that explains these results.

3.32 Validity of Program Transformations (in the Java Memory Model)


Jaroslav Sevcik (University of Cambridge, GB)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Jaroslav Sevcik

In this talk, I will give an overview of validity of compiler transformations in the Java Memory Model, Sequential Consistency and the DRF guarantee. I will also explain several interesting examples of transformations that are problematic for the Java Memory Model. Finally, I will show that there are program transformations that are valid under SC but not under TSO or DRF.

3.33 Verifying Linearisability of a Lazy Concurrent Set

Heike Wehrheim (Universität Paderborn, DE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Heike Wehrheim

Joint work of Derrick, John; Schellhorn, Gerhard; Wehrheim, Heike;

Linearisability is the key correctness criterion for concurrent implementations of data structures shared by multiple processes.

In the seminar we presented a proof of linearisability of the *lazy* implementation of a set due to Heller et al. The lazy set presents one of the most challenging issues in verifying linearisability: a linearisation point of an operation (here, *contains*) set by a process other than the one executing it. For this we have developed a proof strategy based on refinement which uses thread local simulation conditions (the proof obligations talk about at most two processes at a time) and the technique of potential linearisation points. The former allows us to locally prove linearisability for arbitrary numbers of processes, the latter permits disposing with backward reasoning. The operation *contains* may get several potential linearisation points (including some set by e.g. the *remove* operation of another process), only the last one is a valid one. As *contains* is not modifying the abstract data structure (the set), this type of reasoning is sound for proving linearisability.

All proofs (including the one showing soundness of our proof obligations) have been mechanically carried out using the interactive prover KIV.

References

- 1 J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 33, no. 1 (2011).
- 2 J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanizing a correctness proof for a lock-free concurrent stack. In *FMOODS 2008*, volume 5051 of *LNCS*, pages 78–95. Springer, 2008.

4 State of the Art and Open Problems

The field of relaxed memory models, unusually, cuts across many areas of Computer Science: Computer Architecture, Programming Languages, Compilation, Concurrent Algorithms, Concurrency Theory and Semantics, Hardware Verification, and Software and Algorithm Verification. Real progress in the development and understanding of memory models in a multi-core/shared-memory world can only be achieved by taking a holistic view. In this concluding note we summarise some discussion at the workshop of the state of the art and open problems in the area. This is at best an outline and idiosyncratic summary, far from complete.

Models for Mainstream Hardware

A basic question is that of establishing usable and rigorous models for today’s mainstream multiprocessors, which include ARM, Itanium, Power, Sparc, x86, and IBM zSeries. The state of the art varies for each of these: Sparc TSO has long had a precise model; for x86, Power and ARM we heard in the workshop about recent work that covers common-case programming, though further work is needed to cover the full architecture in each case;

and for Itanium the vendor specification is reasonably precise, and there has been various academic work, though we are not aware of extensive empirical testing.

One might also consider more ‘exotic’ hardware: Tiler, the Intel SCC, Cray machines (influencing OpenMP), and various GPUs.

Some other hardware architectures are of less current interest: Alpha is no longer made, though a few machines remain and it has influenced the Linux kernel memory barriers; Sparc RMO and PSO are not used for Sparc systems; PA-RISC is no longer made (but said to be SC).

Simple Hardware Deltas

A line of discussion throughout the workshop concerned ‘simple’ proposals for hardware developments, i.e., those which a hardware vendor might reasonably incorporate into near-term revisions of their architecture. For example, there was discussion of: adding explicit synchronisation variables to an ISA; support for race detection, using the above (perhaps not so ‘simple’); faster store fences; and lightweight (limited) transaction memory support.

Radical Hardware Deltas

More long-term questions include: determining how much one gains from a relaxed hardware model (difficult to establish as a vast amount of hardware and software has been co-designed in the current ecosystems); considering different APIs for hardware coherence mechanisms, exposing more of the hardware (between coherent shared memory and message passing); and considering how scalable coherent memory can be made.

Hardware Verification

Multiprocessor relaxed memory model behaviour is an emergent property of an entire multiprocessor design, involving both core behaviour (especially speculation) and memory communication fabrics. This leads us to ask how one can *prove* that a realistic microarchitecture does implement an architectural memory model, whether better memory model testing can be introduced into the hardware design process, and, more generally, whether it would be useful to work routinely with the formal ‘abstract microarchitecture’ models that one would use for such verification.

Models for Mainstream Programming Languages

Despite very extensive work on programming language memory models, it is arguable that none are yet fully satisfactory.

The semantically simplest option is SC, but providing SC requires one to limit compiler optimisation and to make use of hardware synchronisation primitives (barriers etc.). Folklore suggests that, at least on high-performance compilers, the cost of those is substantial, but there were interesting preliminary results to the contrary presented at the workshop.

A more efficiently implementable alternative is “DRF and catch-fire”: guaranteeing SC behaviour for programs that are race-free (in some sense) in an SC semantics, but providing no guarantees at all for other programs. This permits a wide range of optimisations (the verification of those was discussed at the meeting), and arguably matches programmer intuition for most code except low-level concurrent algorithms, but raises development problems: pragmatically, given that almost all software is buggy, how can race-freedom be ensured?

That brings us to “DRF and abort” models, which are similar except that one has a guarantee that any such race (or, in some cases, any race that matters) will force termination or raise an exception; there was discussion of how this can be implemented.

Statically, one might instead have type systems that exclude races altogether, or even that establish determinacy, but that are flexible enough for a wide range of code, and in general one might establish better concurrent programming models, perhaps with explicit ownership transfer and explicit treatment of data and computation locality.

DRF and catch-fire is at the heart of the C++0x/C1x memory model, a formal development of which was presented at the meeting; an open technical question here is how one should allow close-to-the-hardware non-SC synchronization, and yet still forbid causal cycles and thin-air reads.

The Java Memory Model was also discussed. Java aims to provide SC behaviour for race-free code, but additionally (and in contrast to C++/C) to guarantee some basic security properties for arbitrary code. The current JMM is flawed in the sense that it prohibits certain compiler optimisations that implementations do in fact perform; how best to admit those optimisations while still ensuring the security properties is an open question.

Semantics, Verification, and Reasoning

Basic Semantic Questions

For any well-motivated relaxed memory model, we should ask some basic questions: how to characterise observational congruences; whether we can make effective use of noninterleaving semantics; what the expressiveness of various sublanguages is; and the decidability and complexity of various problems.

Concurrent Algorithm Verification

For algorithm verification, several directions are being pursued: interactive proof (directly above a semantics), program logics, automatic methods, and linearisability proofs. There is a great need for effective compositional methods in this domain, especially to show that libraries using weakly ordered atomic operations can encapsulate that weakness, giving SC behaviour as far as any client can observe.

Compiler Verification

We heard in the meeting of first steps in verifying complete compilers, e.g., from a C-like language with TSO semantics to x86 with its TSO-based semantics, and of verifying compilation of executions from C++0x to x86. There is much more to be done, on verifying optimisations w.r.t. particular models, targetting the weaker and more subtle hardware models, and extending these results to full-scale languages.

Dynamic Verification

There are many properties that are difficult to prove but useful to check, especially if this can be done with reasonable performance: better dynamic race detection, atomicity, and more general concurrency error detection.

Radical models

Finally, when one looks beyond the largest conventional coherent shared-memory machine that one can reasonably build, one can simultaneously ask about the (co)design of hardware, programming model and language, compilation techniques, algorithms, and semantics.

Participants

- Sarita Adve
University of Illinois - Urbana, US
- Jade Alglave
University of Oxford, GB
- Arvind
MIT - Cambridge, US
- Mark Batty
University of Cambridge, GB
- Hans J. Boehm
HP Labs - Palo Alto, US
- Peter Boehm
University of Oxford, GB
- Richard Bornat
Middlesex University, GB
- Ahmed Bouajjani
LIAFA - Université Paris VII, FR
- Gérard Boudol
INRIA Sophia Antipolis, FR
- Sebastian Burckhardt
Microsoft Research - Redmond, US
- Luis Ceze
University of Washington, US
- Stephan Diestelhorst
AMD - Dornbach, DE
- Mike Dodds
University of Cambridge, GB
- Laura Effinger-Dean
University of Washington - Seattle, US
- Christian Eisentraut
Universität des Saarlandes, DE
- Sibylle Fröschle
Universität Oldenburg, DE
- Ursula Goltz
TU Braunschweig, DE
- Alexey Gotsman
IMDEA Software - Madrid, ES
- Richard Grisenthwaite
ARM Ltd. - Cambridge, GB
- Erik Hagersten
University of Uppsala, SE
- Holger Hermanns
Universität des Saarlandes, DE
- Lisa Higham
University of Calgary, CA
- Mark D. Hill
University of Wisconsin - Madison, US
- Marieke Huisman
University of Twente, NL
- Bengt Jonsson
University of Uppsala, SE
- Joost-Pieter Katoen
RWTH Aachen, DE
- Michael Kuperstein
Technion - Haifa, IL
- Doug Lea
SUNY - Oswego, US
- Malte Lochau
TU Braunschweig, DE
- Andreas Lochbihler
KIT - Karlsruhe Institute of Technology, DE
- Michele Loreti
University of Firenze, IT
- Brandon M. Lucia
University of Washington - Seattle, US
- Gerald Lüttgen
Universität Bamberg, DE
- Sela Mador-Haim
University of Pennsylvania, US
- Luc Maranget
INRIA - Le Chesnay, FR
- Milo M. K. Martin
University of Pennsylvania, US
- Paul McKenney
IBM - Beaverton, US
- Michael Mendler
Universität Bamberg, DE
- Maged Michael
IBM TJ Watson Research Center - Yorktown Heights, US
- Samuel P. Midkiff
Purdue University, US
- Madan Musuvathi
Microsoft Corp. - Redmond, US
- Uwe Nestmann
TU Berlin, DE
- Scott Owens
University of Cambridge, GB
- Gustavo Petri
INRIA Sophia Antipolis, FR
- James Riely
DePaul University - Chicago, US
- Vijay A. Saraswat
IBM TJ Watson Research Center - Hawthorne, US
- Susmit Sarkar
University of Cambridge, GB
- Jens-Wolfhard Schicke
TU Braunschweig, DE
- Jaroslav Sevcik
University of Cambridge, GB
- Peter Sewell
University of Cambridge, GB
- Rob van Glabbeek
NICTA - Kensington, AU
- Heike Wehrheim
Universität Paderborn, DE
- Derek Williams
IBM Research Lab. - Austin, US
- Sascha Zickenrott
Universität des Saarlandes, DE

