Static Analyses over Weak Memory

Vincent Nimal

Balliol College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2014

Abstract

Writing concurrent programs with shared memory is often not trivial. Correctly synchronising the threads and handling the non-determinism of executions require a good understanding of the interleaving semantics. Yet, interleavings are not sufficient to model correctly the executions of modern, multicore processors. These executions follow rules that are weaker than those observed by the interleavings, often leading to reorderings in the sequence of updates and readings from memory; the executions are subject to a *weaker memory consistency.* Reorderings can produce executions that would not be observable with interleavings, and these possible executions also depend on the architecture that the processors implement. It is therefore necessary to locate and understand these reorderings in the context of a program running, or to prevent them in an automated way.

In this dissertation, we aim to automate the reasoning behind weak memory consistency and perform transformations over the code so that developers need not to consider all the specifics of the processors when writing concurrent programs. *We claim that we can do automatic static analysis for axiomatically-defined weak memory models.* The method that we designed also allows re-use of automated verification tools like model checkers or abstract interpreters that were not designed for weak memory consistency, by modification of the input programs.

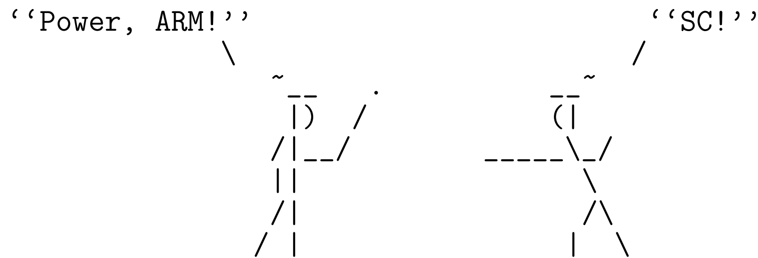
We define an abstraction in detail that allows us to reason statically about *weak memory models* over programs. We locate the parts of the code where the semantics could be affected by the weak memory consistency. We then provide a method to explicitly reveal the resulting reorderings so that usual verification techniques can handle the program semantics under a weaker memory consistency. We finally provide a technique that synthesises synchronisations so that the program would behave as if only interleavings were allowed. We finally test these approaches on artificial and real software. We justify our choice of an axiomatic model with the

scalability of the approach and the runtime performance of the programs modified by our method.

Research Hypothesis

We can do automatic static analysis for axiomatically-defined weak memory models.

We support this claim with the analysis of two instances: instrumenting programs so that their weak memory behaviours are revealed explicitly, and synthesising synchronisations within the program so that (some or all) the weak memory behaviours are prevented. In each of these instances, we analyse the impact of these transformations on the program in terms of performance, the scalability of the techniques with respect to large programs and the precision of the static analyses.



Other Contributions

We list the other contributions of this thesis below.

1. Program analyses: (Chap. 2 Sec. 2.2)
   (a) A class of analyses sound for weak memory models (with Jade Alglave and Michael Tautschnig)
   (b) A set of examples showing the unsoundness of some analyses
2. Static analyses: (Chap. 3)
   (a) An abstraction of the input program that captures all the executions valid in Alglave's framework under weak memory consistency (i.e., that is sound)
   (b) In App. B, we suggest ideas towards a proof of soundness of this abstraction, assuming three rules that a semantics generating Alglave's event structures would need to respect
   (c) A set of engineering strategies to ensure that the approach is feasible in practice (with the avoidance of several exponential explosions)
   (d) An analysis that detects statically potential weak memory behaviours, parametrised by the architecture of the processor considered
   (e) A description of the choices made for the actual implementation of the tools, and their justifications
   (f) A worst-case analysis of this program analysis
   (g) A short survey of other search techniques that could have been alternatively implemented in the tools to enumerate the critical cycles
   (h) A comparison of the different semantics that could have been used to handle the weak memory consistency
3. Safety verification: (Chap. 4)

(a) A technique for instrumenting statically concurrent C programs so that weak memory behaviours would be revealed by interpreting the instrumented program under sequential consistency
(b) An implementation of this technique in the CProver framework: the tool goto-instrument –mm
(c) An experimentation of the instrumentation with 5 model-checkers assuming sequential consistency and a comparison with 4 model-checkers handling natively some weak memory models (with Michael Tautschnig)
(d) A validation of this implementation against 500+ Litmus tests (with Michael Tautschnig)
(e) Some experiments over classic synchronisation programs and larger software
(f) A strategy to select better instrumentation placements/choices
(g) A large resulting database of short concurrent programs that can be used for benchmarking

4. Synchronisation synthesis: (Chap. 5)
   (a) Four integer linear programming (ILP) encodings optimising the placements and types of memory synchronisation in a program w.r.t. weak memory
   (b) A fully automated program transformation combining the critical cycle detection with the ILP encodings, implemented in the tool musketeer
   (c) A set of empirical strategies to prevent compiler hazards, assuming simple gcc-like compiling style
   (d) A comparison of the different encodings
   (e) A comparison to other existing techniques, both analytical and experimental
   (f) An extensive experimentation over a set of 700+ Debian executables (with Daniel Poetzl)

Acknowledgements

Gratefully, I am looking back on four years of very intense research under the supervision of Daniel Kroening, Jade Alglave and Joel Ouaknine, and in inspiring exchanges with Michael Tautschnig, Daniel Poetzl and Saurabh Joshi. We experimented a lot, we coded a lot, and we also learned a lot.

Daniel Kroening was an inspiring supervisor, offering lots of new ideas to explore in our research at every meeting. I could unfortunately only explore a few of these branches—which means that there is still a large tree of weak memory problems to solve. Jade Alglave followed my research progress very closely and I appreciate a lot the stimulating exchanges and the time she invested in our research. Joel Ouaknine supported me with precious advice, especially concerning my future career planning. Beyond my supervisors, Michael Tautschnig has been of great support when I first dived into the CProver codebase maintained by Daniel Kroening—and helped me to gain trust into my own research.

I am very thankful to Paul McKenney and Luke Ong, who accepted to examine my thesis. Their comments during my DPhil viva were invaluable and the exchanges we had were very motivating.

During my DPhil, I did an internship in the RSE group of Microsoft Research Bangalore in India under the excellent supervision of Akash Lal. I learned a lot in the three months with the group and felt very well supported and encouraged there.

The support of great colleagues and friends in the group were very precious for the completion of my DPhil. I would like to thank Ganesh

Narayanaswamy and Dario Cattaruzza for having been very supportive friends all these years, Daniel Poetzl for the numerous times I came to his office with some uncertainties about my own research, Ruben Martins and Cristina David for their great friendship, Saurabh Joshi for the numerous discussions about fence insertion strategies.

Many thanks to Lihao Liang for our long discussions at night before papers' deadlines, Vijay D'Silva for having introduced me to abstract interpretation—and to lots of verification concepts—, Leopold Haller and Fazilat Nassiri for all the excellent conversations in Oxford and Berkeley, Tyler Sorensen for having so nicely welcomed me for my first time in the USA and for the invaluable discussions we had regarding memory fences, Kareem Khazem for his visit in Oxford and his questions about goto-instrument, Pascal Kesseli for the pleasant discussions we had in our office.

I am grateful to Alex Horn for having dived so deep in the theoretical part of my thesis, Matt Lewis for having been so calm when I was stressed by the multiple submission deadlines, Peter Schrammel, César Rodríguez and Nassim Seghir for good conversations, Björn Wachter for our technical discussions at lunch, Martin Brain for his passion in verification and his inspiring thoughts, Subodh Sharma for explaining me GPU threadings, Ashutosh Natraj and Samuel Bucheli for the interesting discussions about UAV safety, Liana Hadarean, Elizabeth Polgreen, David Landsberg, Rajdeep Mukherjee, Hongyi Chen, Daniel Neville and Marcello Sousa for the good discussions over a coffee.

I would also like to thank Alexander Kaiser, Alastair Donaldson, Thomas Wahl, Nannan He, Ajitha Rajan and Vojtěch Forejt for their collegial support. Julie Sheppard, Pamela Farries and Elizabeth Polgreen assisted me a lot with the complex administrative system of the University. I am very grateful to them.

It was always a great pleasure to meet in Oxford, Paris and London Théodore Bluche, Deran Onay, Lilit Darbinian, Yiannos Stathopoulos, Narayan Kamath, Swaroop Rath, Boris Dadachev, Larisa Han, Maria Boghiu, Karim Lara Ayub, and all the other friends from Masters and

College. Ben Sadler, Matthew Lim, Lino Ferreira and Kate Ham were always offering excellent advice and good friendship at St. Ebbe's.

I would also like to thank friends from overseas. Abdallah Saffidine as an old friend was always up for fruitful pre-deadlines discussions. A great thanks to all the friends from MSR India, Abhayendra Singh, Richard Wang, Srikar Tati, Pranav Ramkrishnan, Kate Sydenham, Prathmesh Prabhu, Rahul Sharma, Ekin Akkus and all the others for having made this internship in India such a remarkable time.

The conferences, visits and email exchanges were occasions to share ideas about this thesis. I would like to thank Ganesh Gopalakrishnan and Zvonimir Rakamaric for their invitation to present my work in Utah, Alex Linden, Roland Meyer and Egor Derevenetc for invaluable discussions about weak memory models, Geoffrey Winn for welcoming us in IBM Hursley, Carsten Fuhs for all the paper proofreadings he did for us, and Gérard Berthelot for his courses in concurrency, model-checking and Petri nets when I was an undergraduate student.

My initial dissertation required intense proofreading. Many thanks to Elizabeth Polgreen for having read the dissertation from the title page to the last appendix. Jade Alglave proofread in detail my technical work, that I very much appreciate. Many thanks to Alex Horn, Cristina David and Élise Nimal who proofread full chapters of this dissertation.

During these four years, I received the constant, unconditional and strong support of my parents, Jean-Claude Nimal and Marie-José Nimal, my sister Élise Nimal, my grand-parents Jean Nimal and Josette Nimal, and my larger family. Jean, my grand-father, saw the start of my DPhil but did not see this final dissertation. I would have loved to present it to him and discuss this research with him in Bordeaux. This dissertation is dedicated to him.

Finally, I would like to thank Nadine Ruprecht and her family. Nadine supported me from the first day we met—the first day of my DPhil programme, at the induction day in the Examination Schools—and accom-

panied me across my DPhil these four years. This dissertation would not exist without her precious encouragements.

Vincent Pierre Jean Nimal

En mémoire de mon grand-père, Jean Nimal, qui a accompagné mes premiers pas en sciences

Contents

1 Introduction **1**

1.1 Handling Modern Processors' Concurrency 2

1.1.1 Weak memory consistency 4

1.1.2 Approaches to semantic evaluation of programs 5

1.2 A motivating example 6

1.3 Thesis 9

1.4 Organisation of the dissertation 9

2 Weak Memory and Program Analyses **12**

2.1 Memory Models and Safety Verification 16

2.1.1 Accesses to shared memory 16

2.1.2 Weak memory models 19

2.1.3 A framework to decide valid executions 22

2.1.4 Impact over the program semantics 31

2.2 Memory Models and Program Analyses 33

2.2.1 Analyses expressed with Data Flow Equations 33

2.2.2 Abstract Event Structures as Traces 36

2.2.3 Soundness of Analyses on Weak Memory Models 38

2.2.4 Other approaches to program analyses for weak memory . . . 42

2.3 Rephrasing Trace Properties in terms of Static Critical Cycles 43

2.3.1 Valid executions and critical cycles 43

2.3.2 Trace properties for programs and program analyses 44

2.3.3 Relating executions and critical cycles 51

2.4 Summary 54

3 An Abstraction for Static Analyses over Weak Memory **58**

3.1 Static over-approximation of the control-flow graph 61

3.1.1 Semantics and Abstraction 64

3.1.2 Constructing AEGs 65
3.1.3 Strategy for loops and recursions 72
3.1.4 Engineering improvements regarding the analysis 77
3.2 Search strategies in the abstract event graph 80
3.2.1 Enumerating the critical cycles in an abstract event graph . . 80
3.2.2 Analysis of the complexity in worst case 83
3.3 Novelty of our contribution w.r.t. the related work 86
3.3.1 Semantics for the weak memory related analyses 86
3.3.2 Survey of techniques for enumerating the critical cycles 87
3.4 Summary 90

4 Instrumentation over Weak Memory 91
4.1 Instrumentation of event structures 95
4.1.1 Abstract machine 95
4.1.2 Illustration using examples 100
4.1.3 Instrumentation 102
4.2 Instrumentation of C programs 104
4.2.1 Delaying an access to shared memory 104
4.2.2 Weighted selection of unsafe pairs 109
4.2.3 Boundedness of the buffers and read set 109
4.2.4 Trade-off between detection and instrumentation precisions . . 114
4.3 Experiments 116
4.3.1 Validation 116
4.3.2 A case study: worker synchronisation in PostgreSQL 120
4.4 Novelty of our contribution w.r.t. the related work 122
4.5 Summary 123

5 Synchronisation Synthesis over Weak Memory 124
5.1 Introduction to optimised fence synthesis 127
5.2 Synthesis 129
5.2.1 Cost function of the ILP 130
5.2.2 Constraints in the ILP 131
5.3 Insertion of fences and dependencies 133
5.4 Experiments and Impact 134
5.4.1 Experiments and benchmarks 136
5.4.2 Impact of inferred fences on runtime 138
5.4.3 A case study: Memcached 140

5.5 Alternative encodings 141
5.6 Optimality and control flows 147
5.7 Novelty of our contribution w.r.t. the related work 148
5.7.1 Semantics for the weak memory related analyses 148
5.7.2 Comparison with trencher 150
5.8 Summary 151

6 Future Work **153**
6.1 Future work in program analyses 153
6.2 Future work in critical cycle detection 153
6.3 Future work in program instrumentation 155
6.4 Future work in fence synthesis 158

7 Conclusion **160**

Bibliography **163**

Glossary **174**

A A short survey of sound and unsound domains **179**
A.1 Numerical abstractions 179
A.2 Pointers analyses 180
A.3 (Other) Classes of Analyses 182

B Ideas towards soundness arguments **184**
B.1 Assumptions regarding $S^?$, the semantics from C to event structures . 184
B.2 Soundness ideas for AEG construction 185
B.3 Assumption for loop-yielded memory events 188
B.4 Soundness ideas for static instrumentation 188

C Transformations of a few classic programs **191**
C.1 Store buffering **(sb)** 191
C.2 Load delaying **(lb)** 191
C.3 Message passing **(mp)** 191
C.4 Independent reads independent writes **(iriw+dps)** 191

D Basic combinatoric details **196**
D.1 Duplication in presence of nested loops 196
D.2 Static analysis of pointers to function 197

E Data, tools and experiments reproducibility . . . 199
E.1 goto-instrument –mm . . . 200
E.1.1 Setting the experimental environment . . . 200
E.1.2 A short tutorial . . . 201
E.1.3 How to reproduce the experimental data . . . 202
E.1.4 How to install the new version of the tool . . . 202
E.1.5 A short note regarding the license . . . 202
E.2 musketeer . . . 203
E.2.1 Setting the experimental environment . . . 203
E.2.2 A short tutorial . . . 203
E.2.3 How to reproduce the experimental data . . . 203
E.2.4 How to install the new version of the tool . . . 203
E.2.5 A note regarding the license . . . 204
E.3 Third-party tools . . . 204

List of Figures

1.1 Two threads interfering and their interlieavings 2
1.2 A multi-threaded C program implementing a store-buffering (**sb**). . . 3
1.3 A set of executions of the program in Fig. 1.2 valid under x86 4
1.4 A short program with pointers and assignments. 6
1.5 A C implementation of Peterson's mutual exclusion algorithm [Pet81]. 7
1.6 Memory states for a schedule of Peterson's mutex algorithm 8
1.7 Design of the analyses and mapping between chapters and modules. . 11

2.1 Abstraction lattice for weak memory. 17
2.2 Some steps from C source to the actual code execution 18
2.3 Simplified steps between C source and the actual code execution . . . 19
2.4 SC, state-based SC and event-based SC 21
2.5 A C program and a corresponding event structure. 23
2.6 A C program and some corresponding event structures 25
2.7 An event structure and two possible executions. 26
2.8 Examples of executions incompatible with an event structure. 27
2.9 An address dependency. 28
2.10 **(sb)** with and without valid executions. 29
2.11 Relations maintained by architectures. 30
2.12 Two event structures and executions. 31
2.13 **(sb)** program and its event structure. 37
2.14 A program implementing **(mp)** and its outcomes per architecture. . . 45
2.15 A program implementing **(sb)** and its outcomes per architecture. . . 46
2.16 A program implementing **(sb+fences)** 47
2.17 A program implementing ($\tilde{\mathbf{sb}}_{TSO}$) and its outcomes per architecture. 47
2.18 Interpretations for **(mp)** and **(sb)** 48
2.19 Concurrent program with pointers affected by weak memory reorderings. 50
2.20 A non-robust program and a robust program 52

2.21 Two transformed programs . . . 54
2.22 Summary of the surveyed program analyses . . . 56
2.23 Table summarising some weak memory properties . . . 57

3.1 Instructions of goto-programs and their semantics. . . . 62
3.2 A C program and its goto-program below. . . . 63
3.3 The AEG of Fig. 3.2 and two executions corresponding to it. . . . 64
3.4 From C programs to AEGs and valid executions . . . 65
3.5 Connected cycles . . . 66
3.6 AEG construction for assignment. . . . 68
3.7 AEG construction for function call. . . . 69
3.8 AEG construction for guarded statements. . . . 69
3.9 AEG construction for forward jump. . . . 69
3.10 AEG construction for backward jump. . . . 70
3.11 AEG construction for atomic. . . . 70
3.12 cmp construction for start_thread. . . . 71
3.13 cmp construction for start_thread and join. . . . 72
3.14 An implementation of Dekker's mutual exclusion algorithm [Dij65]. . . 74
3.15 Three AEGs for Dekker's mutex algorithm . . . 75
3.16 Construction of po_s for mutual recursive functions. . . . 76
3.17 Dynamic cycles. . . . 77
3.18 Function pointers used in dynamically allocated array. . . . 78
3.19 (Parametric) AEGs of $(\mathbf{sb}^n)$ and $(\mathbf{ww}^n_m)$. . . . 79
3.20 Complete directed graph (clique) with 4 vertices. . . . 82
3.21 Abstract worst case of AEGs. . . . 84
3.22 Complexity trade-off in cycle detection techniques . . . 88
3.23 Example of transitive closure computation . . . 89

4.1 Representation of a weakly consistent system with two threads . . . 96
4.2 **(iriw+dps)** implemented in C with POSIX threads . . . 97
4.3 Litmus test for **(iriw+dps)** on Power and ARM . . . 98
4.4 **(sb)** on TSO with the abstract machine . . . 101
4.5 **(iriw+dps)** on Power with the abstract machine . . . 102
4.6 Choices for instrumenting **(sb)** for TSO. . . . 103
4.7 Choices for instrumenting **(iriw+dps)** for Power. . . . 104
4.8 Instrumented code for **(iriw+dps)** . . . 106
4.9 Integer linear programming problem to choose the pairs to instrument 109

4.10 A variant of **(lb)** with several reads to the same shared variable . . . 111
4.11 A variant of **(sb)**, **(sb^n+fence)** 112
4.12 A variant of **(sb^n+fence)** (Fig. 4.11) where we read from the n-buffer. 113
4.13 A variant of store-buffering with an unbounded loop. 114
4.14 Trade-off between detection and instrumentation 115
4.15 Tools used for the experiments of Sec. 4.3. 117
4.16 All tools on all litmus tests and models. 118
4.17 Instrumentation experiments with cbmc 119
4.18 Token passing in `pgsql.c`. 120
4.19 **(lb)** and **(mp)** idioms detected in `pgsql.c`. 122

5.1 ppo and fences per architecture 127
5.2 Example of resolution with between 129
5.3 ILP for inferring fence placements. 130
5.4 Message Passing from Fig. 2.14 with its corresponding event structure. 133
5.5 Choices for placing a fence. 133
5.6 `isb.c` and `isb-O3.s`. 134
5.7 `addr.c` and `addr-O3.s`. 135
5.8 All tools on the CLASSIC series for TSO 135
5.9 All tools on the FAST series for TSO 136
5.10 musketeer on selected benchmarks in DEBIAN series for TSO and Power 137
5.11 Overheads for the different fencing strategies 139
5.12 Confidence intervals for data structure experiments. 140
5.13 Runtime overheads due to inserted fences in memcached for each strategy 140
5.14 Confidence intervals for data structure experiments 141
5.15 A 2-cycle example with useless po_s edges 142
5.16 Comparison of the alternative encodings in terms of variables. 143
5.17 Illustrations of the different encodings for the ILP construction. . . . 145
5.18 Number of variables per encoding and case. 146
5.19 Computational cost for retrieving the variables per encoding and case. 146
5.20 Parametric example **(mp^n)** 147
5.21 Example that requires empty events placed at the branchings. 148
5.22 Fence synthesis tools 149
5.23 Cycles sharing the edge (a, b) 151

6.1 Power expressed with Herding cat language [AMT13] 156
6.2 Critical cycle frequency analysis to tune the ILP formula. 159

B.1 Ideas to prove soundness 190

C.1 Implementation and instrumentation of **(sb)** 192
C.2 Implementation and instrumentation of **(lb)** 193
C.3 Implementation and instrumentation of **(mp)** 194
C.4 Implementation and instrumentation of **(iriw+dps)** 195

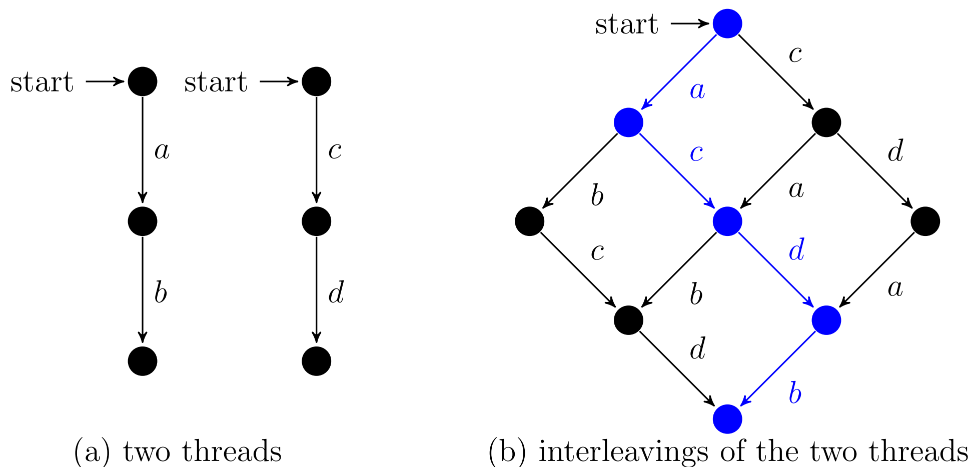
Chapter 1

Introduction

A concurrent program is often thought of as a program calling a concurrency library that abstracts all the low-level concurrency. A program written with POSIX threads and spawning two threads could expect to see these threads being run on different cores during one runtime and on a single core during another. Scheduling decisions and low-level concurrency would be hidden to the users. Yet, the effects of some concurrency optimisations performed by the processor(s) can propagate from the hardware execution to the library layer, and even affect eventually the semantics of the original program. The processor can for instance allow the reodering of accesses to shared memory, in some cases that we will explain in the next chapter. If the program's correctness relies on a specific order of accesses to the shared memory, running such a program with, e.g., POSIX threads, might violate the correctness of the program. These reorderings are specific to *weak memory consistent* processors, which constitute the majority of the usual multi-core processors. There exist however different memory-consistency models, and one would need to be architecture-specific when writing a concurrent program.

One option is to remove all the dataraces from the concurrent programs. It is a valid approach; creating a datarace-free program, however, is not always trivial, and there are good evidences that a significant part of the industrial code was engineered with dataraces, e.g., for performance reasons.

In this dissertation, we will assume that we analyse concurrent programs with shared memory and potential dataraces. The code is assumed to be written in C, without specific annotations, and it could even have been written with a concurrency model in mind that may not correspond to the actual one ruling the hardware/software of the targeted platform—for instance the interleavings.



(a) two threads (b) interleavings of the two threads

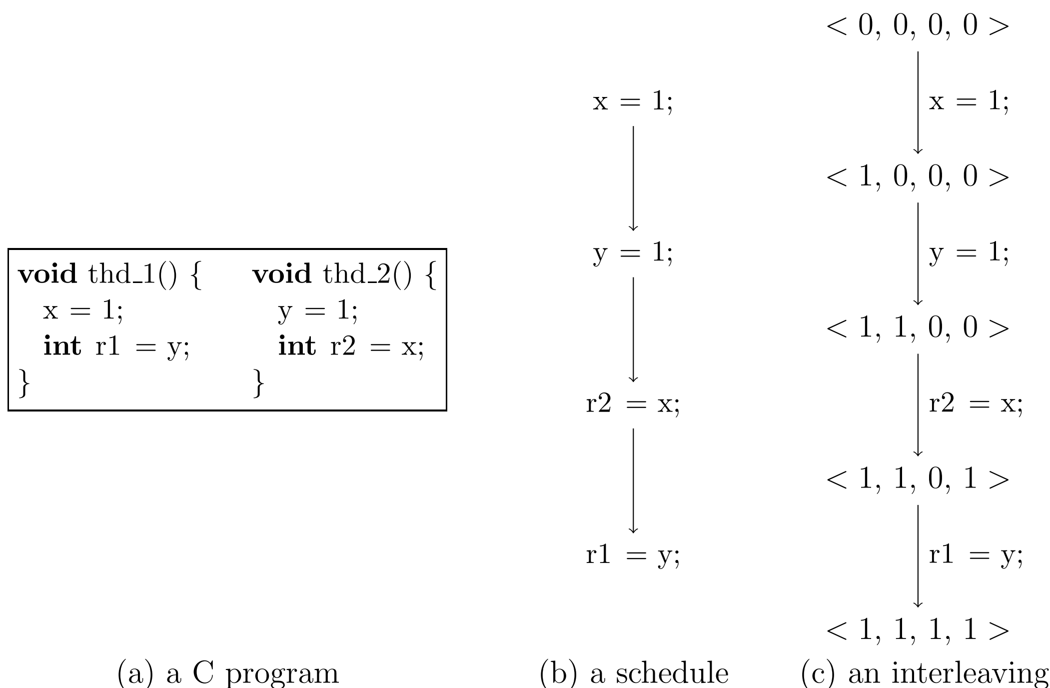
Figure 1.1: Two interfering threads on the left-hand side, and the set of possible interleavings on the right-hand side.

We will develop, in the context of software verification, some new techniques—some of them re-using existing analysers—in order to verify and fix concurrent C programs in an automated manner. These techniques will rely on different models to capture the effects of the hardware running the input programs. As we aim at developing static analyses assisting program development, and it is common in this domain, we will accept a degree of imprecision. Our analyses are based on an axiomatically-defined weak memory model. We will argue that using such a model rather than an operational model was an important decision for building analyses scaling to large-size[1] programs in our context.

1.1 Handling Modern Processors' Concurrency

Interleaving A concurrent program is a program where the order of execution of two parts (or more) of the code—usually two functions called *threads*—is not explicitly given. The order during a program execution is dynamically decided by a *scheduler*. If the *thread order*—that is, the order in which the instructions of a given thread—is imposed, an execution decided by the scheduler will be an *interleaving* of the interpretations of each instruction. In Fig. 1.1(a), we have two threads with the instructions a then b on the left thread, and c and d on the right thread. The

[1] By "*large*", we mean a C program with between 1000 and 10000 lines of code. The complexities of our analyses actually depend largely on the potential concurrent communications rather than the proper number of lines of code, so these numbers are provided to convey an idea of the analysis range, based on our observations, and not as actual references. There are examples within this range that are too thread-intensive to be analysed in a reasonable time by our analyses.

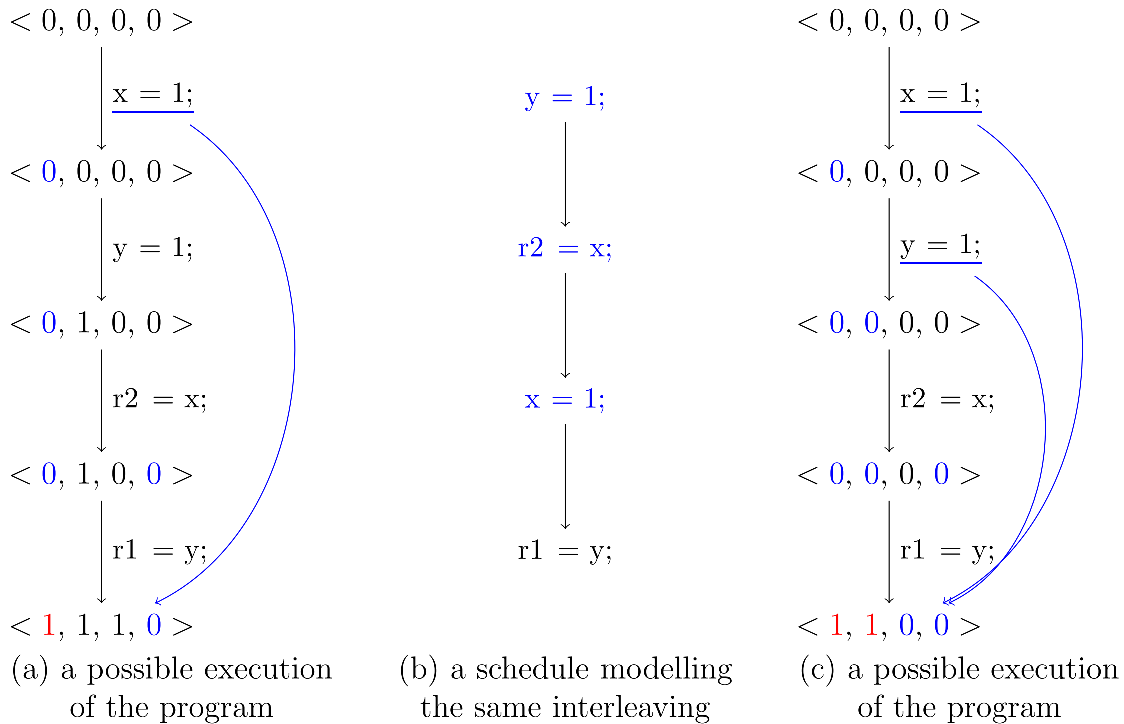


(a) a C program (b) a schedule (c) an interleaving

Figure 1.2: A multi-threaded C program implementing a store-buffering (**sb**).

vertices can be understood as the memory state of the machine before and after the instructions. This, however, subsumes that the instructions are interpreted atomically, which is not necessarily the case for processors, as we will see further in the dissertation. Fig. 1.1(b) represents all the possible interleavings of this program with two threads. Any path starting from the top of the automaton and reaching the last state is an interleaving. The blue path describes for instance the (set of) execution(s) where the scheduler decides to execute first a, then c, followed by d, and finally b.

Program execution We defined in the previous paragraph the order of interpretation of the instructions of a multi-threaded program. Evaluating an instruction can modify the values stored in registers and memory. We will say that evaluating an instruction can allow us to transition from one memory state to another. *A program execution* can be understood to be a sequence of memory states, resulting from the interpretation of a sequence of instructions. This dissertation focusses on memory and how the memory is modified, and not on the order under which a program is interpreted. Out-of-order executions, due to *pipelining* for instance, will not be covered. We will define precisely the scope of our analyses in Chap. 2. Since there is no ambiguity, we will always refer to a sequence of memory states obtained from the evaluation of a possible interleaving as an interleaving itself.



(a) a possible execution of the program (b) a schedule modelling the same interleaving (c) a possible execution of the program

Figure 1.3: A set of executions of the program in Fig. 1.2(a) valid under x86 for the schedule in Fig. 1.2(b).

In the excerpt of the multi-threaded C program in Fig. 1.2(a), `thd_1` and `thd_2` share two variables, x and y, and have two local variables. Fig. 1.2(b) is a possible schedule of the instructions, and Fig. 1.2(c) displays a transition system representing the memory states, here modelled as tuples <`x`, `y`, `r1`, `r2`>, during a possible program execution with schedule Fig. 1.2(b).

1.1.1 Weak memory consistency

On modern multicore processors, a program execution is not necessarily an interleaving, like e.g. in Fig. 1.2(c). The instruction might for example not be interpreted atomically, i.e., the next instruction would already be executed before the write to the shared memory of the first instruction would have affected it. Processors implementing a *weak memory consistency*[2] can contain mechanisms that postpone the moment at which a value is written to or read from the shared memory. These mechanisms are often modelled as a set of write buffers and read queues—we will discuss this in more detail in Chap. 4, where we insert similar mechanisms but at the software level, directly inside the source. In Fig. 1.2(c), the write of x = 1; might actually take

[2]Also called *relaxed memory consistency* in the literature.

longer to reach the memory, resulting in the execution displayed in Fig. 1.3(a). The write underlined (in blue) is delayed and updates (in red) the shared memory only after the other instructions. The values in blue are those which changed in the states compared to the execution without reorderings in Fig. 1.2(c).The memory states are changed, but could still be obtained by running the schedule described in Fig. 1.3(b). We will say in the next chapter that this execution is still *sequentially consistent*. Yet, in Fig. 1.3(c), we have an example of a program execution, observable with a x86 processor, that cannot be modelled by an interleaving: the two writes underlined in blue are postponed at the end of the execution, and the registers both read 0 from memory—despite having both the writes appearing before the reads in the two threads.

1.1.2 Approaches to semantic evaluation of programs

In this dissertation, we will rely on two sorts of semantic evaluation of a program. The semantics can be expressed *operationally*, with a "step-by-step" explanation of how each instruction will affect the memory, caches and registers, or with an *axiomatic* model, which explains the executions at a higher level but cannot necessarily map trivially to a "step-by-step" explanation.

We can illustrate these two approaches with a simple pointer analysis. Let us suppose that we want to check, in the program in Fig. 1.4(i), that `q` points to `x` after instruction (e). We can use a maps-to analysis, which computes step-by-step the environment (i.e., the memory representation) after each instruction. We can define it *operationally* with three rules:

$$\frac{}{< \sigma > x = \phi < \sigma[x \mapsto \text{eval}(\phi)] >} \qquad \frac{}{< \sigma > *x = \phi < \sigma[\sigma(x) \mapsto \text{eval}(\phi)] >}$$

$$\frac{< \sigma_1 > S < \sigma_2 > \quad < \sigma_2 > T < \sigma_3 >}{< \sigma_1 > S; T < \sigma_3 >}$$

In these rules, σ is the propagated environment, that is, the values held by all the variables; ϕ an expression and $A[b \mapsto c]$ means "replace the existing entry (resp. entries) for b in the function (resp. relation) A by c", i.e., $A := (A \setminus (\{b\} \times \text{Im}(A))) \cup \{b \mapsto c\}$, where $\text{Im}(A)$ is the set of images of A. Equivalently, we can define it denotationally with the transformers $\tau[x = \phi](\sigma) = \sigma[x \mapsto \text{eval}(\phi)]$, $\tau[*x = \phi](\sigma) = \sigma[\sigma(x) \mapsto \text{eval}(\phi)]$ and $\tau[END](\sigma) = \sigma$. This program analysis computes iteratively the environments at each program line, as detailed in the Fig. 1.4(ii). After (e), we can read that q points to x. However, we have kept track of some values that were unnecessary.

(i) A short program with pointers.

(a) x = 1;
(b) p = &x;
(c) ∗p = ∗p+1;
(d) q = p;
(e) ∗q = ∗q+1;

(ii) operational computation

	x	p	q
(0)	0	0	0
(a)	1	0	0
(b)	1	&x	0
(c)	2	&x	0
(d)	2	&x	&x
(e)	3	&x	&x

(iii) axiomatic computation

- $(e)\ q \rightarrow x$?
  $\exists (d) \leq (e), \exists (b) \leq (d), \exists p$ such that $(b)\ p \rightarrow x$ and (d) p = q;
- $(b)\ p \rightarrow x$?
  (b) p = &x;

Figure 1.4: A short program with pointers and assignments.

We can alternatively define axiomatically an analysis based on a ternary relation $(.)\,.\rightarrow .$ defined as follows:

$$(i)\,p \rightarrow x \triangleq (i)\,\mathrm{p = \&x;} \vee (\exists j \leq i, \exists k \leq j, \exists q, (k)\,q \rightarrow x \wedge (j)\,\mathrm{q = p;}).$$

This analysis is demonstrated in Fig. 1.4(iii). We first replace $(e)q \rightarrow x$ by its definition, and try to solve $(e)\,\mathrm{q = \&x;} \vee (\exists j \leq e, \exists k \leq j, \exists r, (k)\,r \rightarrow x \wedge (j)\,\mathrm{r\ = q;})$. We can use $j = d$, $k = b$, and there is indeed $r = p$ such that (d)p = q;), and $(b)p \rightarrow x$, since we have (b)p = &x; in the code. Its computation focusses on the relevant part of the information. It does not, however, give the whole trace in the program leading to `q` points to `&x`.

Both of these approaches have advantages and disavantages. For the problems related to weak memory addressed with static analyses, we will argue in Sec. 2.3 and Chap. 5 that the axiomatic approach is more convenient for designing such methods. In Chap. 4, we will combine our static analysis with an instrumentation of the input source with operational features like buffers and queues, and show experimentally that this combination performs better than a straightforward operational semantics.

We will describe in Chap. 3 how we exploit the framework of Alglave [Alg10] to ensure the soundness of our approaches for C programs.

1.2 A motivating example

Running a program written for interleavings on a processor with weak memory consistency can result in unexpected behaviours. Some of these behaviours can alter properties of the program. A short yet remarkable example is the concurrent algorithm of Peterson [Pet81] which ensures that two sections (the *critical sections*)

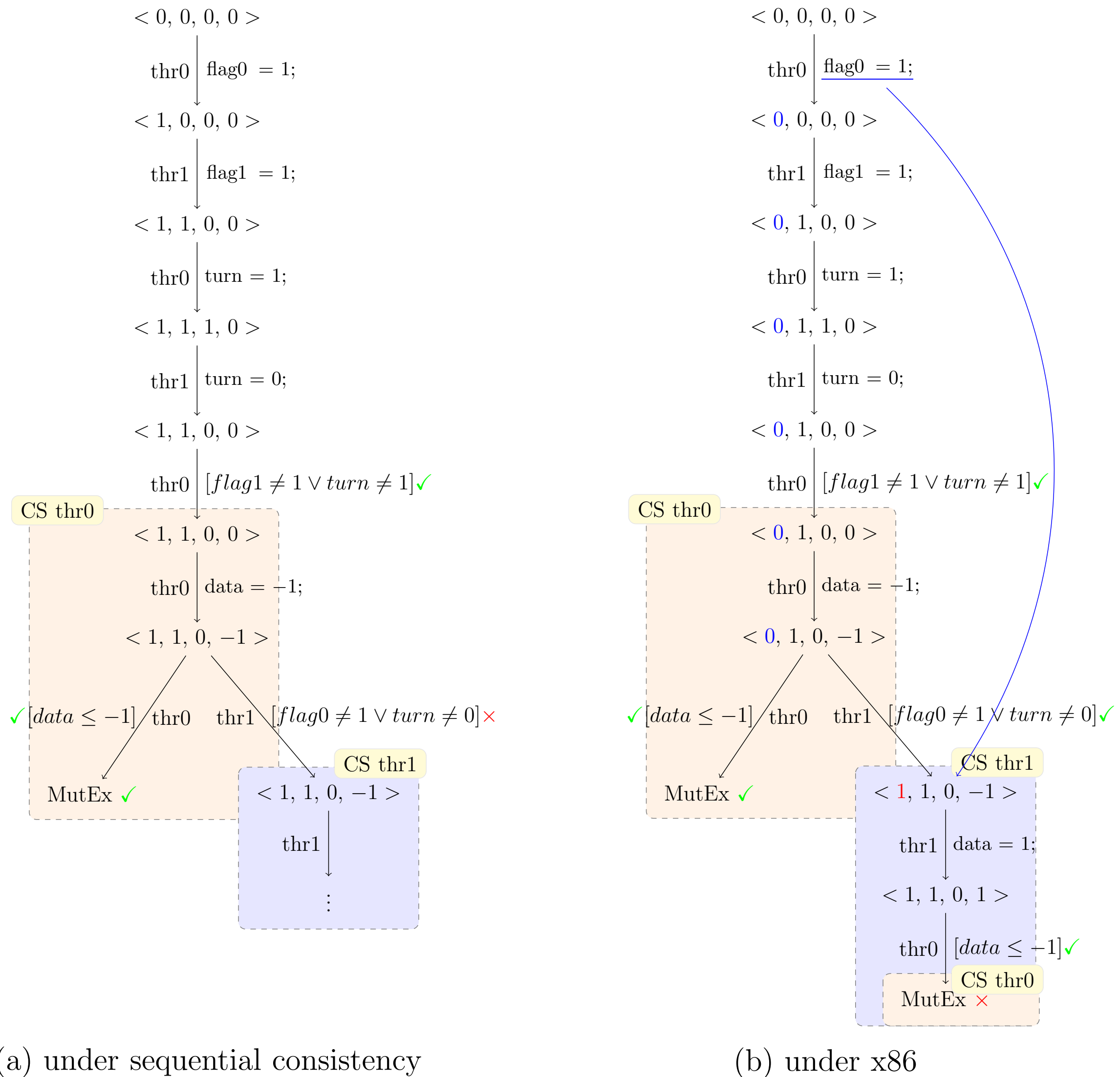
```
volatile int turn; // ID of the current thread running
volatile int flag0 = 0, flag1 = 0; // Boolean flags
int data = 0; // variable to test mutual exclusion

void* thr0(void* arg) {
  flag0 = 1;
  turn = 1;
  while (flag1==1 && turn==1);
  // begin: critical section
  data = −1;
  assert (data<=−1);
  // end: critical section
  flag0 = 0;
}

void* thr1(void * arg) {
  flag1 = 1;
  turn = 0;
  while (flag0==1 && turn==0);
  // begin: critical section
  data = 1;
  assert (data>=1);
  // end: critical section
  flag1 = 0;
}
```

Figure 1.5: A C implementation of Peterson's mutual exclusion algorithm [Pet81].

remain *mutually exclusive*. It means that there is at most one thread running at any time in the critical section. When running the algorithm with interleavings only, the critical sections are indeed mutually exclusive. However, if we implement the algorithm e.g. in C as shown in Fig. 1.5, this property no longer holds. A possible reason for this is non-atomicity of some writes. Below we explain in detail what happens in the synchronisation phase.

In Fig. 1.6(a), we display one specific schedule of threads and the memory states before and after each instruction. Each memory state represents the values of flag0, flag1, turn and data, respectively. We first run thr0: thr0 sets flag0, meaning that it wants to enter the critical section. thr1 wakes up and also manifests its interest for entering its critical section by setting its flag flag1. thr0 then passes the hand to thr1 by assigning 1 to turn and waits in the blocking *while*-loop. thr1 passes the control to thr0 by assigning 0 to turn, and goes into the waiting loop. At this point, thr0 sees that flag1 holds 0, and breaks from the loop, since the guard $[flag1 \neq 1 \vee turn \neq 1]$ is evaluated to **true**. It gets into the critical section and assigns -1 to data. thr1 cannot enter its critical section yet, since the guard $[flag0 \neq 1 \vee turn \neq 0]$ is still evaluated to **false**; it remains in its blocking loop. Thus the assertion in thr0, which checks that the data was not modified in the critical section by another thread that would also be in critical section, holds.

Now, if we run the same schedule on a machine that implements x86, when thr0 sets its (shared) flag flag0 to 1, this might not be immediately visible to thr1—the write might be buffered by thr0. thr0 enters the critical section as before, and assigns -1 to data, but the write buffer has not been flushed yet, meaning that thr1 still



(a) under sequential consistency (b) under x86

Figure 1.6: Memory states for a given thread schedule of Peterson's mutual exclusion algorithm.

believes that flag0 holds 0. The guard $[flag0 \neq 1 \vee turn \neq 0]$ is thus **true**, and thr1 can exit the blocking while to enter its critical section—*violating the mutual exclusion property of the algorithm.* It then writes 1 to data, and when thr0 checks its assertion $[data \leq -1]$, the assertion is violated.

This behaviour could not be detected by an analyser that assumes interleavings as models of execution—which is the case for most of the model-checkers. In Chap. 4, we will introduce a technique combining static analysis and program transformation to allow existing analysers for concurrent programs to capture these unexpected behaviours.

Once we have identified the issue, we can place some synchronisations to prevent

behaviours like those in Fig. 1.6(b) that violate the mutual exclusion property. By inserting x86's `mfence` right before each blocking while in Fig. 1.5, the buffered write in Fig. 1.6(b)—whose time in buffer is represented by the blue arrow—must be flushed prior to entering the critical section of thr0. The execution violating mutual exclusion is thus no longer permitted. Placing such fences requires a good understanding of the algorithm, the architecture on which the program will run and the synchronisation primitives provided by the architecture. We will automate the placement of synchronisations in Chap. 5.

1.3 Thesis

We can do automatic static analysis for axiomatically-defined weak memory models. We support this claim with the analysis of two instances: instrumenting programs so that their weak memory behaviours would be revealed explicitly, and synthesising synchronisations within the program so that (some or all) the weak memory behaviours would be prevented. In each of these instances, we analyse the impact of these transformations on the program in terms of performance, the scalability of the techniques with respect to large programs and the precision of the static analyses.

1.4 Organisation of the dissertation

The dissertation is organised as follows: in Chap. 2, we explain the relevant part of Alglave's thesis [Alg10] that we use for our analyses. We also describe our previous work on program analyses that are sound for weak memory models. This shows in particular that points-to analyses sound for concurrent programs are also sound under weak memory consistency. We need this result to analyse C programs. In Chap. 3, we introduce an abstraction computed out of the control flow graph of a C program that captures all the potential relations—as defined in Chap. 2—that might appear during an execution of the program on a given architecture. In Chap. 4, we introduce an instrumentation technique that first makes use of the static analysis detecting weak memory behaviour that we developed in Chap. 3 in order to reduce the locations in the code that we augment with buffers and queues to simulate weak memory behaviours for analysers (in an operational style). In Chap. 5, we address the dual problem, which consists of forbidding some weak behaviours for a given architecture. In the same chapter, we fully rely on the static analysis we developed in Chap. 3, and use integer linear programs to optimise the placement of synchronisations. In Chap. 6 we

present some potential extensions of our work for each of the chapters, and conclude this journey amongst the analyses for weak memory models in Chap. 7.

The functional diagram in Fig. 1.7 summarises the main components of our methods, and graphically describes how the chapters interact with each other. At the beginning of each chapter, we will highlight the components addressed in the chapter. Our methods rely on some previously developed techniques: the generation of a goto-program—an internal representation described in Chap. 3—from a C program (`goto-cc`, which is the work of Christoph Wintersteiger and Daniel Kroening), the analysis of a goto-program or a C program with a model checker or abstract interpreter, and the generation of C program from an existing goto-program (`goto-instrument --dump-c`, developed by Daniel Kroening and Michael Tautschnig). The fence insertion script was written by Daniel Poetzl to allow a good integration of our methods in the experiments on Debian packages.

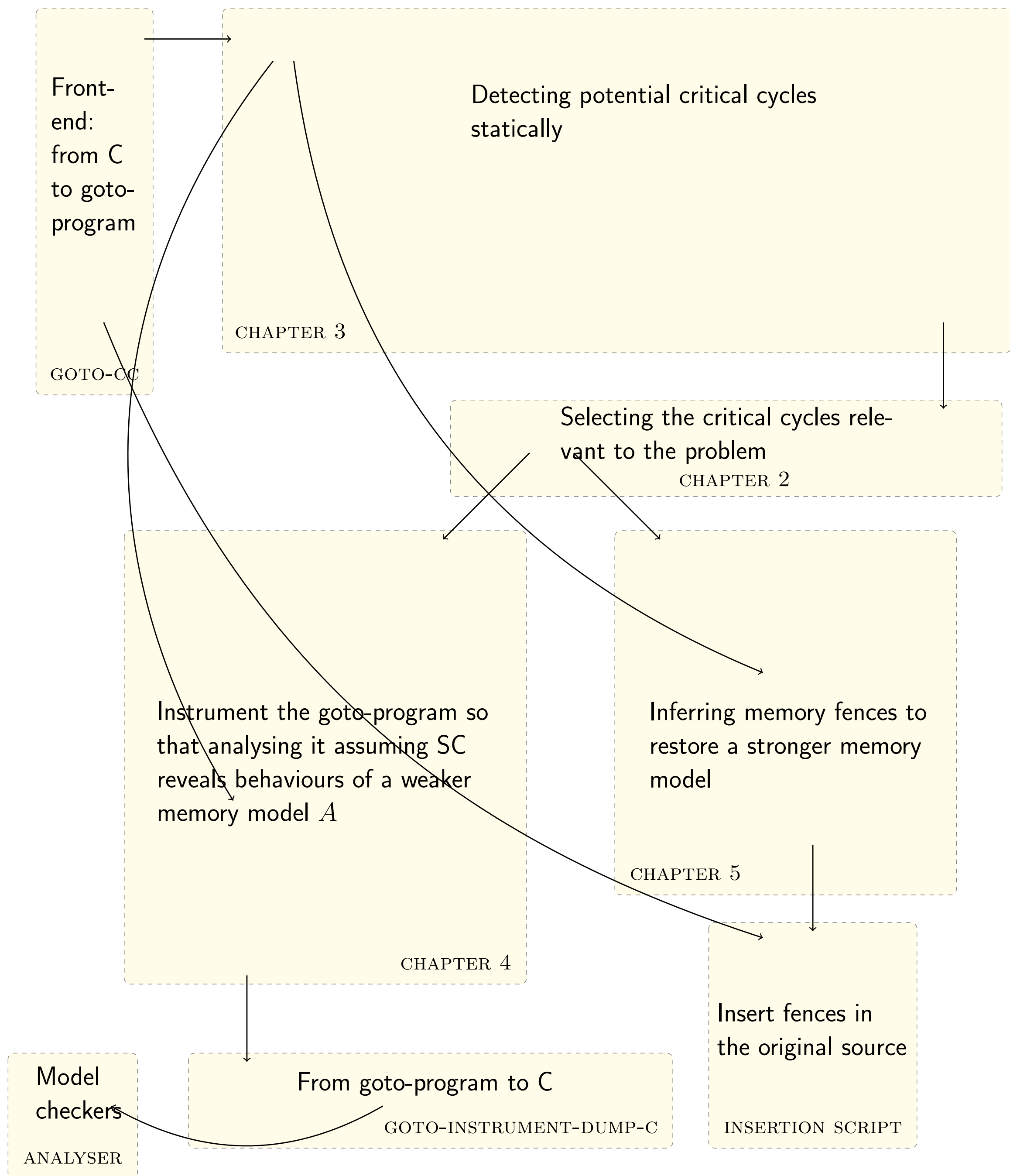
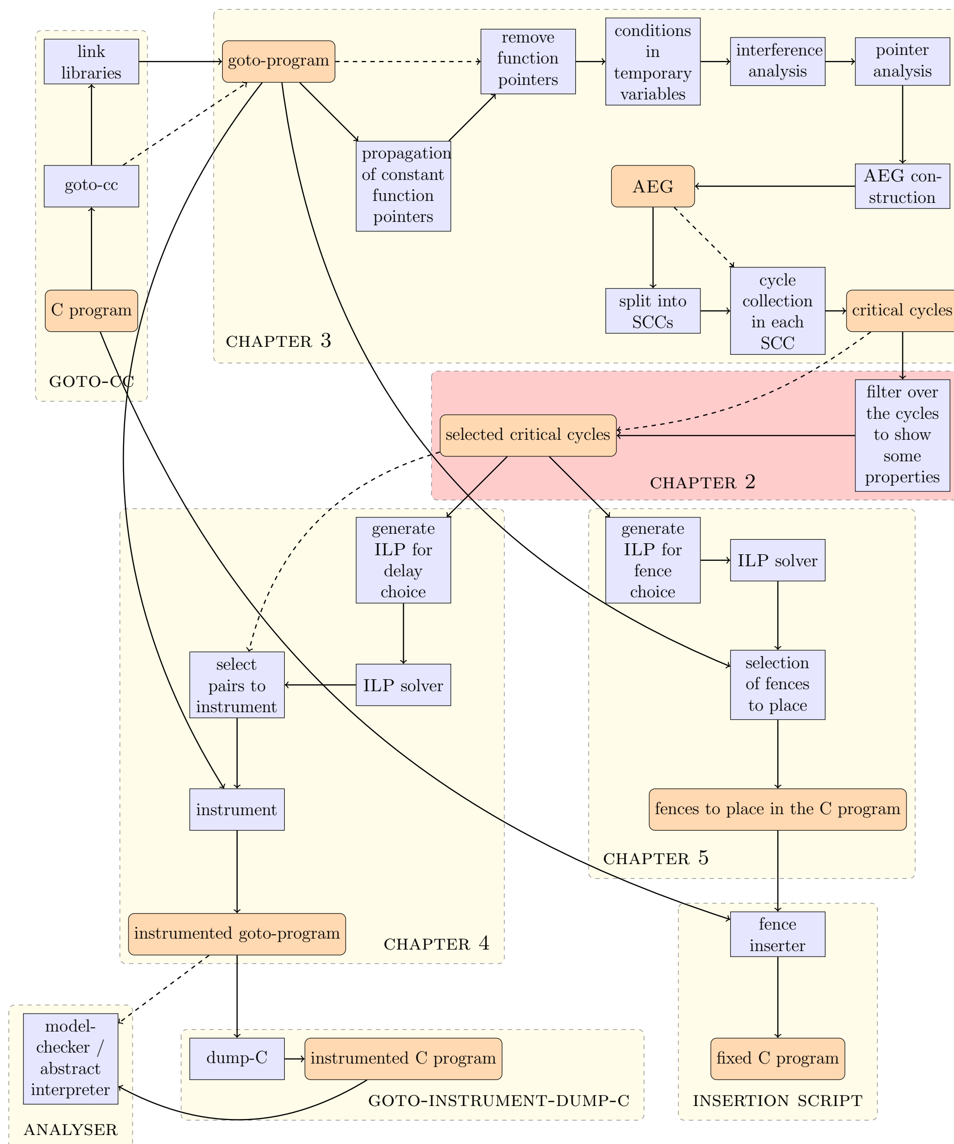


Figure 1.7: Design of the analyses and mapping between chapters and modules.

Chapter 2

Weak Memory and Program Analyses



In this chapter, we explain how weak memory consistency can, informally speaking, "*impact the semantics of concurrent C programs*". We explain what "*impact*" means in the context. Besides type lengths and memory alignments, C programs may be thought as fairly machine-independent. However, even though compilers implement (most of) the C standards, this is not the case. The meaning of the program written in C should nevertheless be conserved down to the machine code, and then in any execution of this code. This is why the semantics of the C program must take into account the processor architecture specificities, including the *memory consistency*. By "*impacting the semantics*", we mean in this dissertation that the C program can exhibit behaviours that depend on the memory consistency—hence increasing the space of possible states that the program would describe.

Weak memory consistency is a property enforced at the level of the processor. We will not attempt to explain how this is implemented in the actual processors. We will rather rely on the framework of Alglave [Alg10] to describe formally the weak memory consistency whose effects could be observed in the examples of Chap. 1. We will give for some of the unintuitive behaviours an intuition based on the use of queues and buffers. These intuitions will actually be put into work in Chap. 4, which adds to an input program some queues and buffers so that weak memory behaviours can be explicitly revealed in the programs, even under sequential consistency.

Weak memory behaviours affect the executions, and depend on *processor architectures*. By default, compilers like gcc or msvc do not make any assumption regarding the memory consistency implemented by the processor. The memory consistency must be taken into account by the C developer. Since 2014, these compilers now implement C11 and C++11 standards [c1111, cpp11], that provide a certain level of guarantee concerning the memory consistency for the developers. The new paradigms developed in these newer language specifications are, however, more complex than the traditional interleavings of instructions—for which an execution is an interleaving of instructions in the threads. Their combinations, together with well-established synchronisation algorithms like Lamport's and Szymanski's mutual exclusion algorithms [Lam87, Szy88] or newly-written parallel code, would also increase the amount of effort it is necessary to dedicate to the development of safe concurrent code. We will not postulate on whether to use *release-acquire* semantics[1] or other concurrent semantics rather than interleavings and automated restoration of memory consistency

[1] `http://msdn.microsoft.com/en-gb/library/windows/hardware/ff540496\%28v=vs.85\%29.aspx` gives an introduction to this semantics.

at compile-time via memory fences[2]. We would only argue that most of the existing code is written with either an architecture in mind (x86 for instance) or actual interleavings, and that the automated restoration of memory consistency will provide an important re-usability of the existing code.

Program analyses are essential techniques that collect information like pointer addresses or variable use in programs. This information is used e.g. for performing optimisations at compilation time, transforming or simplifying programs for model-checkers, or directly finding some specific, unexpected behaviours of programs in order to fix them. Yet, almost all the program analyses that target concurrent programs do rely on interleavings—they do not assume weak memory consistency. As a result, as we will observe in Sec. 2.2, some of these program analyses might miss some behaviours due to the memory consistency, and thus become unsound. The unsoundness can then propagate to the whole compilation/verification chain to which the analysis belongs.

In this chapter, we first introduce the background for weak memory, and explain in detail the behaviours that can affect the semantics of concurrent programs. We identify the impact of these effects and limit the scope of our studies to the additional non-determinism added by the hardware. We will not consider reorderings operated by the compilers in the context of optimisations, even though some of these might be captured by our models. Compiler-enforced semantics such as release-acquire semantics in C++11 or interaction between different memory models are also beyond the scope of this thesis.

The second section of this chapter describes our earlier work on program analyses. We observe that some of these analyses are sufficiently over-approximative to capture the effects induced by weak memory consistency. Some other analyses, however, are not sound with this respect. For these unsound analyses, we provide a way of restoring this soundness, at the cost of a loss of precision.

In the third section, we enumerate a few properties for analyses and programs w.r.t. weak memory models. We explain how Shasha and Snir's critical cycles [SS88] relate to valid executions, how their static counterparts can over-approximate them, and how to express the aforementioned properties in terms of static critical cycle.

We will argue in Chap. 3 and Sec. 2.3 that soundness for weak memory over program analyses is a notion that is too coarse to be of direct use for most applications.

[2] `http://preshing.com/20120913/acquire-and-release-semantics/` provides a nice correspondence between acquire-release semantics and memory fences.

This work nevertheless constitutes a necessary background to some analyses that we apply in Chap. 4 and Chap. 5, in particular the pointer analysis.

2.1 Memory Models and Safety Verification

In this section, we explain the concept of weak memory consistency. We introduce Alglave's framework for deciding valid executions of a program running on a range of modern multicore processors, and how the weak memory consistency can affect the semantics of programs. Programs will be represented here as *event structures*, which is a structure where loops have been unrolled, calls resolved, memory addresses resolved and expressions evaluated. Only the scheduling of the events remains unsolved. In the lattice of program and execution abstractions, presented in Fig. 2.1, this structure lies almost on top, meaning that the reasoning we will apply will be fairly faithful to the intended (potential) executions, but far from the actual, static program written in C. We will explain how to relate these structures to actual C programs in Chap. 3.

The content of this section is not new, and only covers the part of weak memory models relevant to this dissertation. However, we try to complete the explanation with an intuitive link between the objects manipulated in Alglave's framework and the usual program and execution concepts. Readers interested in learning more about how these memory models were designed can read the original paper from Shasha and Snir [SS88], the work of Sarkar et al. for x86 [SSN+09] and Power [SSA+11], and the thesis of Alglave [Alg10] for the model we use. The development of these models also involved extensive testing against the hardware, reported in e.g. [AMSS11]. A good summary of the different memory consistencies—used for multicore processors and more generic distributed systems—can be found in [Tan95, Sec. 6.3].

2.1.1 Accesses to shared memory

In concurrent programs with shared memory, communications between threads can be achieved by writing and reading to a common space, the *shared memory*. In the context of multicore systems, and even more with distributed systems and distributed memory in general, these operations and communications might not be *atomic*. They could indeed be decomposed into several steps, like fetching the instruction, retrieving the address in the memory, computing the new value, sending it and eventually having it stored in memory. There is no guarantee that the final step of an instruction would be performed before the first step of the next instruction; several instructions might be interpreted before the first one succeeds in updating the memory; the evaluations

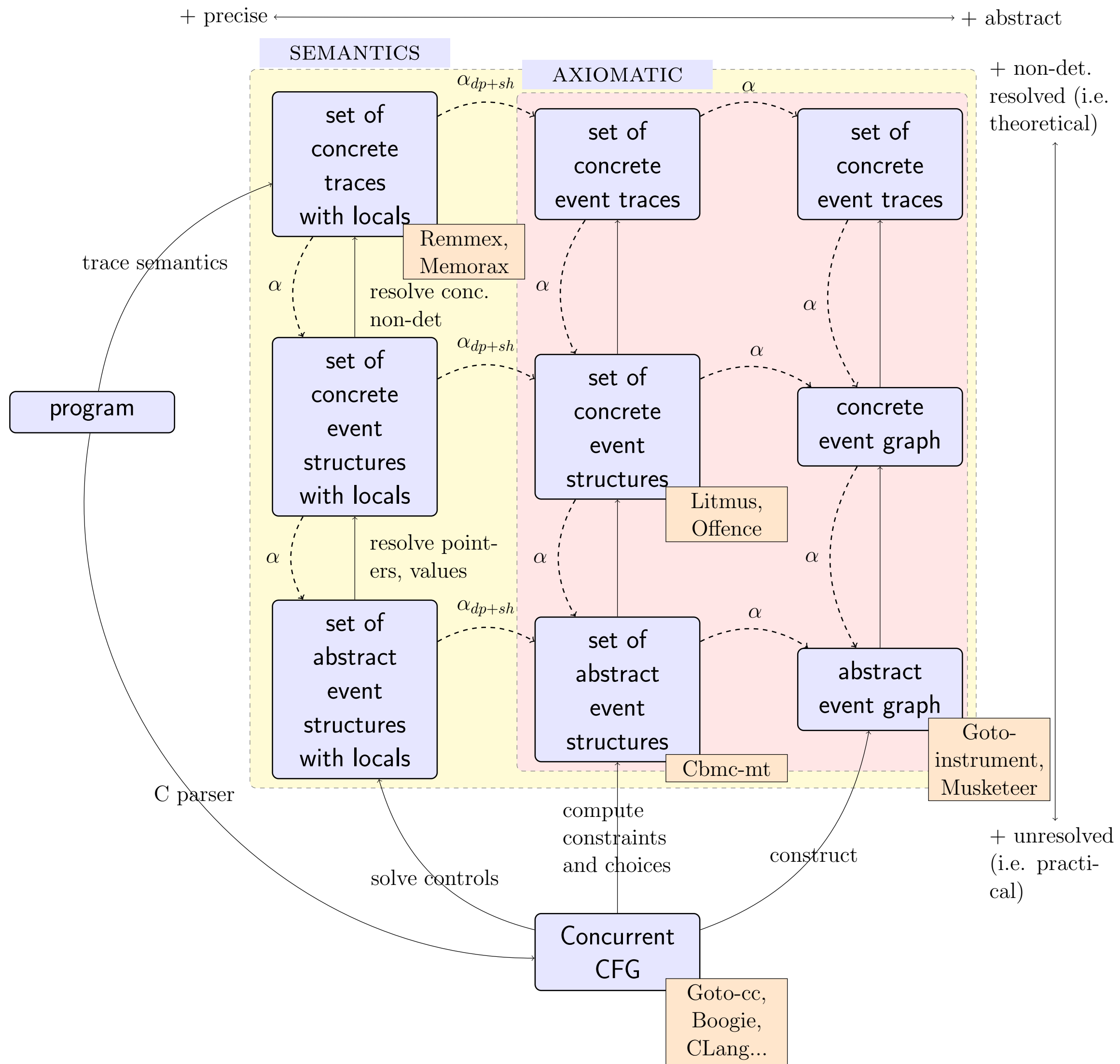


Figure 2.1: Abstraction lattice for weak memory.

of two consecutive instructions might finally be reordered, or these instructions could also be rewritten in a different order. These four scenarios all result in out-of-order executions. These relaxations originate from different layers—we borrow here the enumeration of [DMD13, p. 51]:

- In the software layer,

  (S1) compiler-level optimisations;

  (S2) execution of nested signal handlers.

- In the hardware layer,

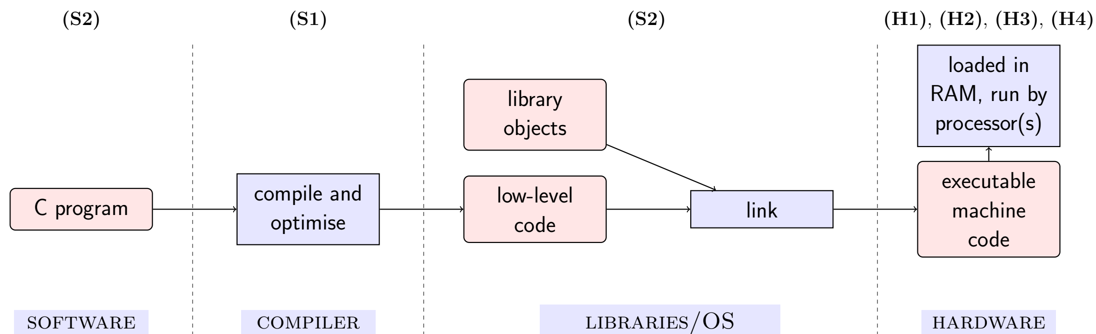


Figure 2.2: Some usual operations happening between the C source and the actual code execution.

- **(H1)** shared memory multi-processors;
- **(H2)** weak memory ordering;
- **(H3)** pipelining and superscalar architectures;
- **(H4)** out-of-order instruction scheduling.

We place these relaxations in some usual steps involved in the execution of a program in Fig. 2.2. We will, however, only address relaxations that are directly affecting concurrency, and we will assume a simplified execution process, as depicted in Fig. 2.3. We now justify our choice of relaxations.

As we mentioned in the introduction, C compilers do not guarantee memory consistency—thus **(H1)** and **(H2)** are also visible at C level. **(H1)** and **(H2)** can be captured by observing the order in which the shared memory is effectively updated or read. When the shared memory is updated, we will say that a *write event* has been issued. If the processor read from the shared memory in order to assign a value to a local register, a *read event* has been issued. Alglave's framework reasons about the orders in which these memory accesses can be issued to describe the validity of an execution with respect to **(H1)** and **(H2)**. **(S2)**, **(H3)** and **(H4)** are not specific to concurrent programs, and are beyond the scope of this dissertation. There are some arguments that would suggest that a fragment of **(S1)** is also captured by our models—the compiler does indeed move in the code accesses to shared variables, unless the developers made use of assembly inlinings or compiler-specific keywords like **volatile**. However, we will not provide any guarantees with respect to **(S1)**, as it remains an open problem. The arrival of new paradigms with C11 makes this problem even harder[3]. When we deal with compiler optimisations, e.g. for dependency

[3]In particular, as noted Paul McKenney, accesses to shared memory that are not marked *atomic* in C11 should result in an undefined behaviour, which is not modelled in our analyses

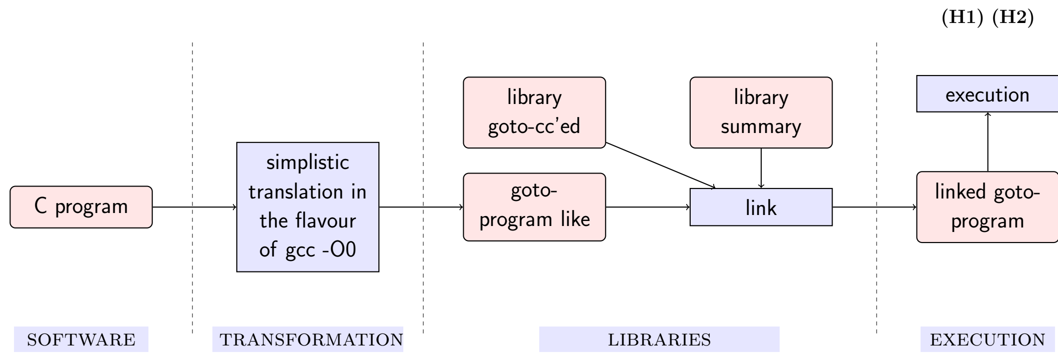


Figure 2.3: Some simplifications that we assume between the C source and the actual code execution.

insertion in Chap. 5, our arguments for **(S1)** will only be supported by empirical observations.

The choice of the concurrency libraries is another potential source of unexpected behaviours. We might indeed decide to consider the standards, and model the functions accordingly, or consider a specific implementation of the library, that may or may not contain fences depending on the platform.

2.1.2 Weak memory models

Weak memory models are models that impose orders between accesses to shared memory, for a given program execution. They can be expressed operationally, often together with the rest of the semantics of the program, in which case the order is constructed step-by-step. This is also known as operational semantics. Models can also be expressed axiomatically, and the constraints over the accesses to shared memory are ensured by a set of relations between them. The axiomatic models are more rarely combined with a whole semantics—which will justify in Chap. 4 our choice to use an axiomatic semantics for a static analysis to narrow down the places of interest in the code, and an operational semantics to insert instrumentations in these places in the code.

Machines and architectures In this dissertation, we will always assume that our programs run on a *machine*. We do not refer to Turing machines—except in Sec. 4.1.1. We refer to a system composed of a (or several) processing unit(s) with registers, a memory divided into the local parts, local to the threads, and the shared one. Write buffers and read queues specific to processing unit(s) can also appear between the processing unit(s) and the memory. Fig. 4.1 in Chap. 4 provides an

intuitive representation of such a system for two CPUs. In practice, these machines represent a (or several) processor(s) and its (their) memory.

Depending on the presence of mechanisms that are abstracted by the buffers and queues in the processor, an execution of a program under a given machine will respect the memory consistency implemented by the processor. We will say that the processor implements an *architecture*. Examples of architectures are x86, IBM's Power and ARM.

Strict consistency Under *strict consistency* [Tan95, p. 315], the memory events are strictly ordered, that is, "*any read to a memory location x returns the value stored by the most recent write operation to x*", as written in [Tan95, p. 315]. Informally, it means that two consecutive instructions in a same thread cannot yield two events happening in the reverse order [Tan95, p. 332]. The "most recent write" subsumes the existence of an absolute timeline. Reasoning in terms of timelines is, however, not the preferred strategy adopted by the concurrency community, since executions that are actually different might be drawn in a probably uncountable set of possible timelines. The models usually considered in this context rely on logical relations between events that abstract away the timelines[4].

Sequential consistency Lamport defined in [Lam79] *Sequential consistency* (abbreviated SC) as "*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*". In other words, this corresponds to the standard, syntactic enumeration of all the interleavings of the threads. The "as if" of the definition might, however, leave some freedom in the interpretation of what a sequentially consistent execution can possibly be, and we find in the literature [AM11, BDM13, AKNP14] sequential consistent executions sometimes defined as executions whose results can be modelled by an interleaving of the threads. In this dissertation, we will assume SC as the "strict" sequential consistency, i.e., an execution on a SC architecture is an interleaving. We gather the other interpretations into distinct weaker forms of sequential consistency that we explain below. In our static analysis, we will target these weaker forms of SC.

[4]As noted by Paul McKenney, this is only true in our case because we overapproximate fixed overheads for individual instructions like memory fences. This approach, however, does not capture cache-miss overheads, for example.

```
void thd_1() {          void thd_2() {
  x = 1;  /*(1)*/         z = 2;
  r1 = y; /*(2)*/       }
}
```

(a) A reordering not permitted under SC

```
void thd_1() {      void thd_2() {
  x = 1;              y = 1;
  int r1 = y;         int r2 = x;
}                   }
```

(b) A reordering not permitted under "event-based" SC

Figure 2.4: Counter-examples to show non-equality of SC, "state-based SC" and "event-based SC".

Weaker forms of sequential consistency Sequential consistency does not authorise any reorderings of events. Yet some reorderings might not affect the semantics of the program—and leaving the processor(s) more freedom is more likely to improve performance at runtime. There exist several ways of relaxing SC whilst keeping the intended meaning of the program. Although they are not specifically named in the literature [AM11, SNM+12, BDM13, AKNP14, DM14], they result from two properties that we will explain in detail in Chap. 2.3: *state-based robustness*, as defined e.g. in [DM14], and *event-based robustness*, or "*classic*" robustness. The first one ensures that no memory states unreachable under SC can be reached under a state-based robust program. The second one guarantees that no reorderings of events (reads from or updates to the shared memory) could potentially affect the semantics of the program, regardless of the current memory state. The state-based SC is strictly more precise than the event-based SC, which is itself strictly more precise than the traditional SC.

We illustrate these strict inclusions in Fig. 2.4. In Fig. 2.4(a), an execution of the program under event-based or stated-based weaker SC would permit the reorderings of events (1) and (2), whereas SC would not allow it, since this would not be an interleaving. The figure in Fig. 2.4(b) depicts a *store-buffering*: the accesses to the shared memory `x` and `y` on each thread can be reordered, leading to the case where the local variables `r1` and `r2` would both hold 0 at the end of the execution. This configuration, as we will explain further in this section, is not possible under SC, neither stated-based or event-based SC. However, if we replace the 1 by 0, any reordering that would happen during an execution of the program would not affect the memory state, since no action is performed. The execution of such a program under state-based SC would allow reorderings. An execution of the same program under event-based SC would not allow any reorderings, as the events do impact the shared memory—even though they leave it unmodified.

In this dissertation, we will, however, target the event-based SC, since the state-based SC requires the computation of the whole transition system (between reachable

states of memory), which is undecidable in general and computationally expensive in practice. The event-based SC retains some relaxations that are benefit to the performance at a lower cost. In the rest of the dissertation, unless explicitly stated, we will always consider this form of sequential consistency.

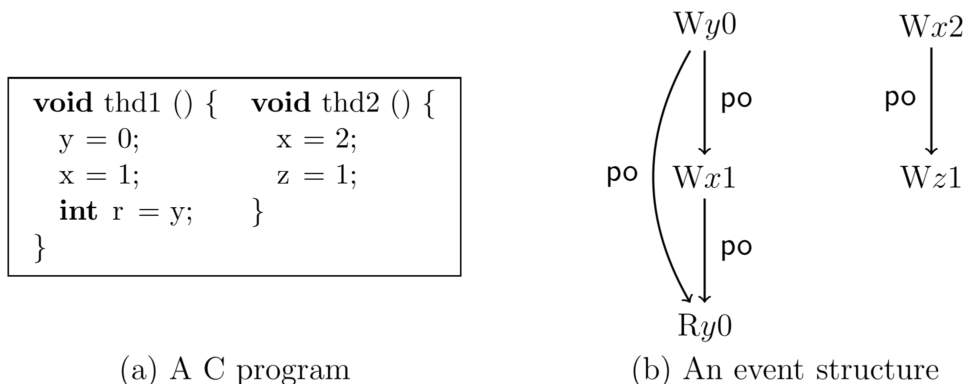
Weaker consistency Numerous multi-core processors implement memory consistencies that are weaker than sequential consistency. They indeed authorise some executions whose effects cannot be reproduced with any interleaving of the program. The framework that we present in the next subsection covers the next architectures: x86/TSO, PSO, RMO, Power and ARM. The original framework also covers Alpha, but we did not adapt our techniques to this architecture as it is now considered obsolete.

Among the architectures, there is a hierarchy that classifies the architectures that are allowing more or less behaviours than the others [Alg10, p. 53]. In this dissertation, we will say that an architecture A is *weaker* that an architecture B if given a program, A would allow more different executions than B. We will express this more formally in Sec. 2.3.

2.1.3 A framework to decide valid executions

We present in this subsection the framework of Alglave [Alg10], that allows us to decide, for x86/TSO, PSO, RMO, Power and ARM, if an execution is valid or not. We will not explain how to test these models against the hardware or how to prove that they relate (or not) to the specifications provided by processor manufacturers [spa94, ppc09]. Readers who would like to learn more about these are invited to read [Alg10].

Memory event Memory events, as we defined before, are accesses to shared memory. This can be writing a value v to or reading a value v' from a location in the shared memory with address x. We will write them respectively $\mathrm{W}xv$ and $\mathrm{R}xv'$. Intuitively, a write event represents the moment at which the location in memory is updated; the read event represents the moment when the read from the memory location is performed. Given an assignment, for example MOV x, 1 in x86, that assigns 1 into x, $\mathrm{W}x1$ corresponds to the end of the effect of the evaluation of this instruction, when the memory is indeed updated.



(a) A C program (b) An event structure

Figure 2.5: A C program and a corresponding event structure.

Event structure In Alglave's framework, we manipulate a structure relating memory events called *event structure*. This structure is not to be confused with Winskel's event structures [NPW79], that are composed of a partial order over a set of events called *causal dependency* and a relation between them called *conflict relation*. The former imposes an order between the events, whereas the latter intuitively means that two events in relation cannot be performed together. These structures are usually used for concurrent semantics assuming SC (e.g. in [Win82]).

The event structure used in Alglave's framework is composed of a finite set of memory events $\mathbb{E}$ and a transitive binary relation over these events po in $\wp(\mathbb{E} \times \mathbb{E})$. The purpose of this structure is to represent the order in which the memory events appear in the definition of a thread, in the description of a given program. In Fig. 2.5, the event structure of the C program written on the left-hand side is drawn on the right-hand side—assuming that thd1 and thd2 functions will be called as (interfering) threads.

The existence of an event structure subsumes a certain number of constraints, imposed by that input program, that the executions should respect. The framework does not specify a specific language or semantics to derive these event structures[5]. C-like languages handled by this framework would be subject to the following constraints:

(C1) the instructions themselves: the processor cannot skip some instructions—unless told so—or execute some unwritten instructions;

(C2) the control flow graph (CFG), that is formed from all the conditional and unconditional jumps inside a function (including loops), should be statically resolved;

[5] This will force us to assume three axioms regarding the semantics in our soundness proof for goto-programs in Chap. B.

(C3) the functions called should be statically resolved;

(C4) the threads running should be statically determined;

(C5) all the expressions should be evaluated, including the values read from and written to memory (including the addresses of memory locations).

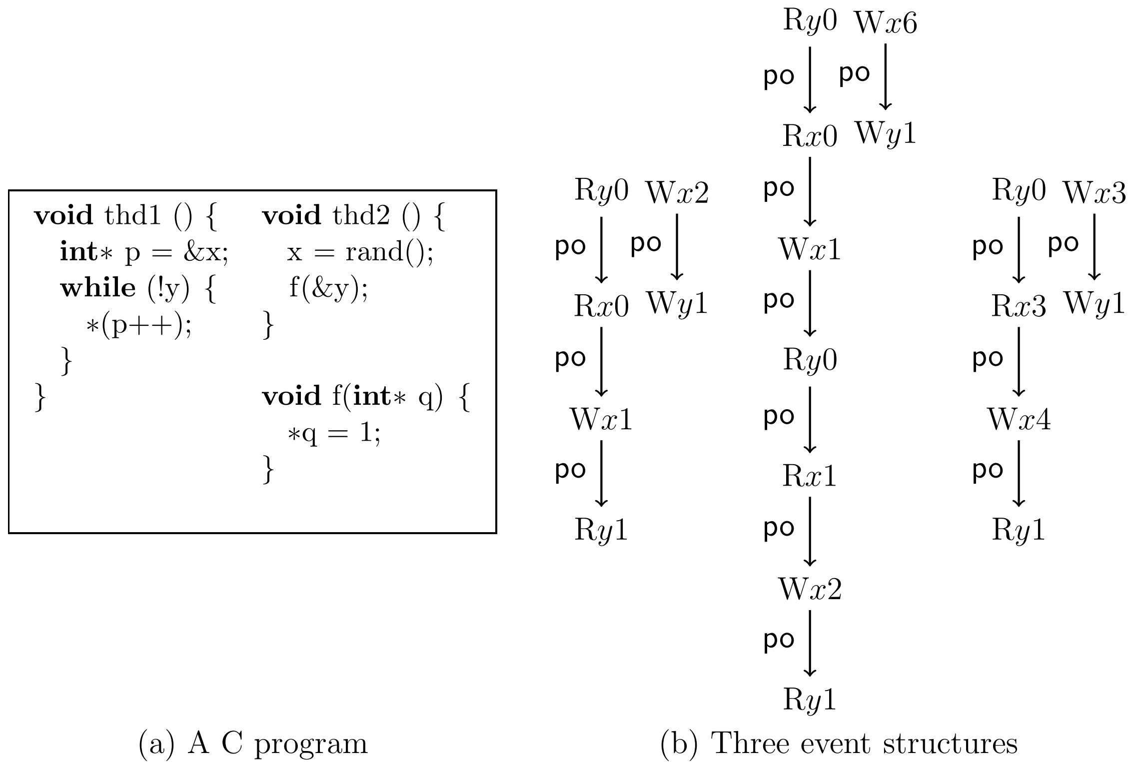
An event structure can be constructed under the constraints **(C1)** to **(C5)** as follows: given a program P, an event structure relates events that are yielded from the instructions of P **(C1)**. The jumps are resolved and the functions are treated as if they were inlined **(C2)**, **(C3)**. The threads interfering—that is, the threads that could potentially run at the same time, be interleaved and access some common memory locations—are known **(C4)**. The values read and written, and in particular all the addresses (pointers, pointers to functions and label values in case of use of the gcc-specific && operator[6]) are resolved **(C5)**.

A program can thus have several corresponding event structures, and possibly an infinity of them (in case of e.g. unbounded loops). In Fig. 2.6, we have a C program on the left-hand side with, in thread thd1, a loop reading from the shared variable y, and the incrementation of the shared variable x via the pointer p. In thread thd2 a pseudo-random value is assigned to x, then function f is called and writes 1 to y. If thd1 and thd2 are interfering, thd2 might never be executed before thd1. y is never set to 1, and the loop is unbounded. There is thus infinitely many possible event structures.

Note that in [AKT13], Alglave et al. introduced, in the context of bounded model checking, *symbolic event structures*, that unwinds loops and leaves values and addresses unresolved. This structure thus only requires determining the events from the instructions **(C1)** and the threads interferring **(C4)**. These do not handle unbounded loops.

Execution In Alglave's framework, an *execution* compatible with an event structure is a set of communications inside and between the threads. They connect writes and reads to the shared variables. These executions, relative to an event structure, must not be confused with the usual executions of a program, that we will call *program execution* in the following. We say that an execution (relative to an event structure) is *compatible* with an event structure when the relations composing the execution relate the memory events provided by the event structure. Three relations between events implement these communications:

[6] `https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html`



(a) A C program

(b) Three event structures

Figure 2.6: A C program and some corresponding event structures (transitive po-edges not represented).

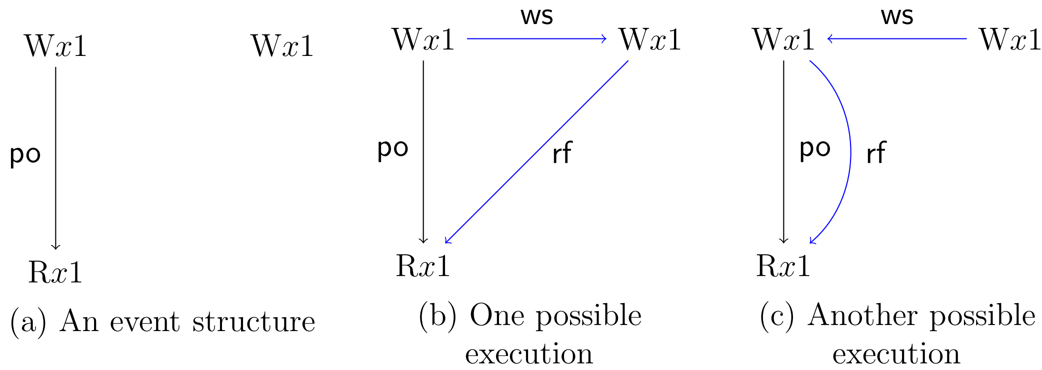
(rf) *read-from,* written rf, which is a relation between write events and reads events targeting the same shared memory location. Formally, $\mathsf{rf} \subseteq \{(w, r) \in \mathrm{W} \times \mathrm{R} \mid \mathrm{location}(w) = \mathrm{location}(r)\}$. Intuitively, a read event reads from the memory the value that was written by the write event.

(ws) *write-serialisation*, written ws, that orders all the writes to a same memory location—$\mathsf{ws} \subseteq \{(w1, w2) \in \mathrm{W}\times\mathrm{W} \mid \mathrm{location}(w1) = \mathrm{location}(w2)\}$. Informally, this means that writes to a given memory location must be totally ordered.

(fr) *from-read*, written fr, is a relation constructed out of rf and ws: $\mathsf{fr} \subseteq \{(r, w) \in \mathrm{R} \times \mathrm{W} \mid \exists w' \in \mathrm{W},\ (w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{ws}\}$. Intuitively, $\mathrm{R}xv$ is in fr-relation with $\mathrm{W}xv'$ if the write event writes a value v' to the memory whereas the read event reads a value v from a write event that happened before $\mathrm{W}xv'$.

Note that several executions can be compatible with a given event structure. The executions in Fig. 2.7(b) and (c) are compatible with the event structure in Fig. 2.7(a).

Similarly to a program execution that may or may not exist given a program, an execution compatible with an event structure may or may not exist, based on



(a) An event structure

(b) One possible execution

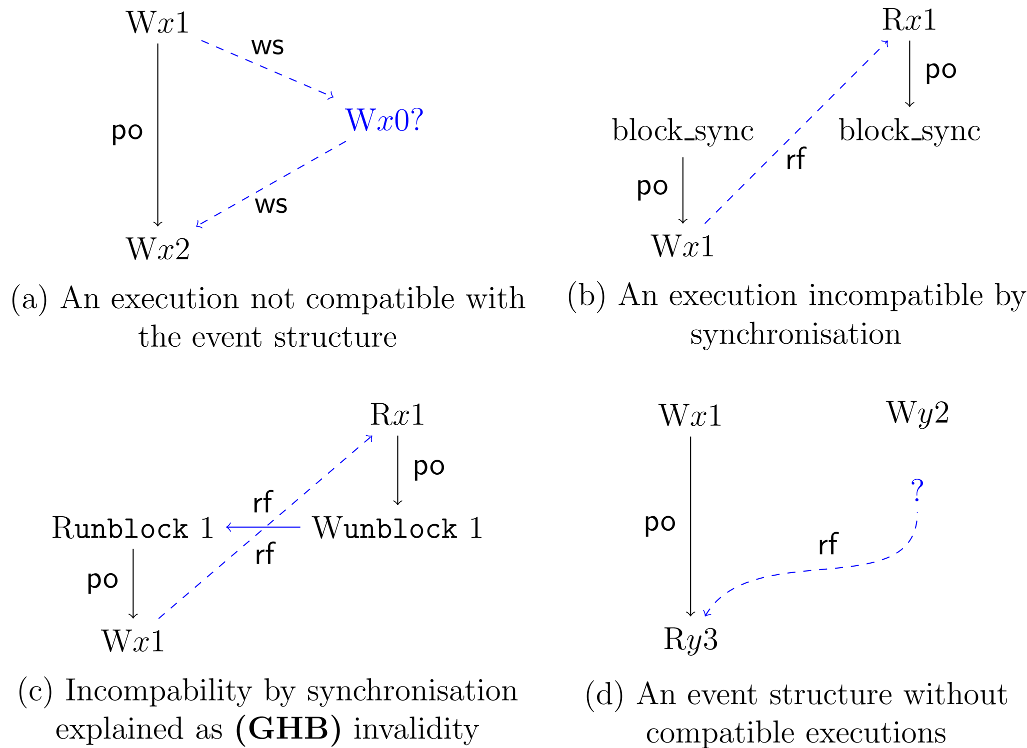
(c) Another possible execution

Figure 2.7: An event structure and two possible executions.

the specifications provided by the event structure. An execution relating events not existing in the event structure—writing e.g. a value that does not appear in the event structure—cannot exist as a valid execution. In Fig. 2.8(a), the execution is not compatible with the event structure in Fig. 2.7(a)—however, another event structure might correspond to the program computation that would yield such communications.

In this paragraph, we consider the event structures and executions in Fig. 2.8(b-d), which are not related to the event structure of Fig. 2.7. Some communications in an execution might also be unable to take place in a program due to synchronisations between threads. In Fig. 2.8(b), the execution is not compatible with the event structure because of the presence of thread synchronisation: two blocking synchronisations prevent the thread to write first to x than read from it. This invalidity by synchronisation can also be explained with the *execution validity* as we will define it in the paragraph "**valid executions**". In Fig. 2.8(c), we assume that the blocking synchronisations are implemented with a simple shared variable `unblock` and a waiting while, without fences. To ensure that thread 1 gets access to the write, the while must read 1 into `unblock`. The write which places 1 in `unblock` is in thread 2, after the read of the previous write. If both `po` edges are enforced in the given architecture, this execution is invalid. If not, the blocking synchronisation would be unsafe for weak memory and not guaranteeing synchronisation. The execution in the example of Fig. 2.8(b) would then be valid—it would just be an abstraction of the one in Fig. 2.8(c).

If an event structure has no compatible execution, it means that there is no program execution that corresponds to this event structure—since the thread communications required by the event structure could not exist. In this case, at least one of the conditions **(C1)** to **(C5)** must have been violated. In Fig. 2.8(d), the event structure cannot correspond to any program execution of the program since there are



(a) An execution not compatible with the event structure

(b) An execution incompatible by synchronisation

(c) Incompability by synchronisation explained as **(GHB)** invalidity

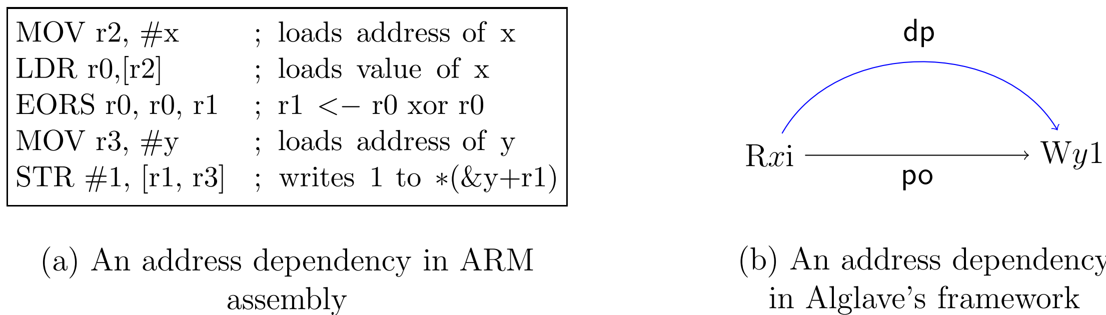
(d) An event structure without compatible executions

Figure 2.8: Examples of executions incompatible with an event structure.

no executions compatible with the event structure. This is due to values that cannot be read, since these are never written—it thus violates **(C5)**.

Dependencies Event structures (and executions) only relate shared memory events—they abstract the use of local registers or variables. Processors, however, take into account some interactions between different accesses to shared variables within a thread. These interactions, called *data-*, *address-* and *control-dependencies*, indicate that the two accesses to shared memory in dependency must be executed in this order. In Fig. 2.9, the two writes in the assembly code on the left are in address dependency, since the value of the first shared variable is required to compute the address of the second access to shared memory. Note that the XOR always computes 0, and, in this specific case, the address is always the same. This is a trick applied by assembler developers to impose an artificial dependency. We will see in Chap. 5 that inserting such a dependency in C is more challenging due to compiler optimisation, even at `-O0`.

A dependency relation (written `dp`) between the events is added to the event structures to keep track of the pairs of memory events in dependency.



(a) An address dependency in ARM assembly

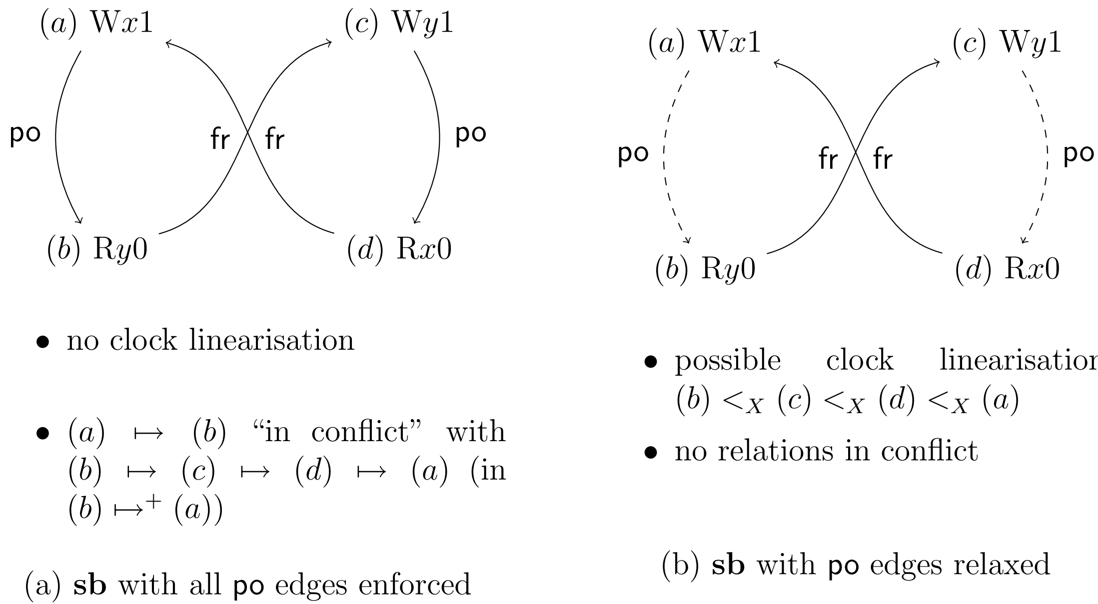
(b) An address dependency in Alglave's framework

Figure 2.9: An address dependency.

Relations maintained by architectures Depending on the architecture under which the event structure is considered, the relations that we defined earlier can be *enforced* or *relaxed*. A relation *enforced* between two memory events means that it forms an order that cannot be contradicted by another relation (or certain combinations of these relations). For instance, if $\mathrm{W}x1 \xrightarrow{\mathsf{po}} \mathrm{R}x1$ is enforced, there cannot be another pair $\mathrm{R}x1 \rightarrow \mathrm{W}x1$ in one of the relations that would be maintained. Intuitively, it means that a program execution that is valid under the architecture—that is, a program execution that could be observed if we were running the programon a processor implementing this architecture—can correspond to a linearisation $<_X$ (a total order) over the memory events. This order, that can be a *clock order*[7] as defined by Alglave et al. in [AKT13], cannot be contradicted by any relation enforced. In Fig. 2.10(a), the execution of the event structure of **sb** under an architecture maintaining its pairs $(a) \mapsto (b)$ and $(c) \mapsto (d)$ is not valid, as $(a) \mapsto (b)$ conflicts with $(b) \mapsto (c) \mapsto (d) \mapsto (a)$. No clock linearisation non-conflicting with the relations is possible: the execution is not valid under the architecture.

A relation is *relaxed* if the clock order can contradict this relation. If a pair of memory events is related by a relaxed relation, we will say that the pair can be *reordered* in the clock order. For brevity and since there is no ambiguity, we will often say that the pair can simply be reordered, or alternatively that the (order of the) pair is *not maintained* by the architecture. In Fig. 2.10(b), the execution of the event structure of **sb** under an architecture relaxing its pairs $(a) \mapsto (b)$ and $(c) \mapsto (d)$ is valid, since there is no conflict between the relations. $(b) <_X (c) <_X (d) <_X (a)$ is a possible clock linearisation. In Chap. 4, the reordered pair will be re-characterised with the help of delay and flush operations.

[7]Alglave et al. use a clock order to determine the existence or non-existence of a valid path in the context of bounded model checking [AKT13]. In case of violated assertion, the clock order appears in the counter-example trace returned by cbmc.



(a) **sb** with all po edges enforced

(b) **sb** with po edges relaxed

Figure 2.10: **(sb)** with and without valid executions.

Based on processors specifications and empirical tests against the hardware, [Alg10] characterised the relations that are relaxed and enforced for the architecture x86/TSO, PSO, RMO, Power and ARM, amongst others. We present these relations in Fig. 2.11. po that is enforced is called preserved program order (written ppo). rf that is enforced is called global[8] read-from (written grf).

Valid executions Given an event structure $E = (\mathbb{E}, \mathsf{po})$ and an execution $X = (\mathsf{rf}, \mathsf{ws})$ compatible with this event structure, we say that the execution is valid under an architecture A if this execution does not lead to conflicting orders among the relations over memory events that are maintained by the architecture. Conflicting orders can be characterised as a cycle in the union of relations. This is the basis of the characterisation of a valid execution in the framework. If we write po-loc as the restriction of po to same locations, an execution is valid if

(uniproc) there is no cycle in $\mathsf{po\text{-}loc} \cup \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{ws}$;

(global-happen-before) there is no cycle in $\mathsf{ppo} \cup \mathsf{grf} \cup \mathsf{fr} \cup \mathsf{ws}$; (written **(GHB)**)

(thin-air) there is no cycle in $\mathsf{dp} \cup \mathsf{rf}$.

More formally, we write:

[8] Since enforcing a read-from communication means that we make it visible to all the threads.

	program order				communications		
Architecture	poWR	poWW	poRW	poRR	rf	ws	fr
SC	✓	✓	✓	✓	✓	✓	✓
x86/TSO		✓	✓	✓	✓	✓	✓
PSO			✓	✓	✓	✓	✓
RMO					✓	✓	✓
Power						✓	✓
ARM						✓	✓

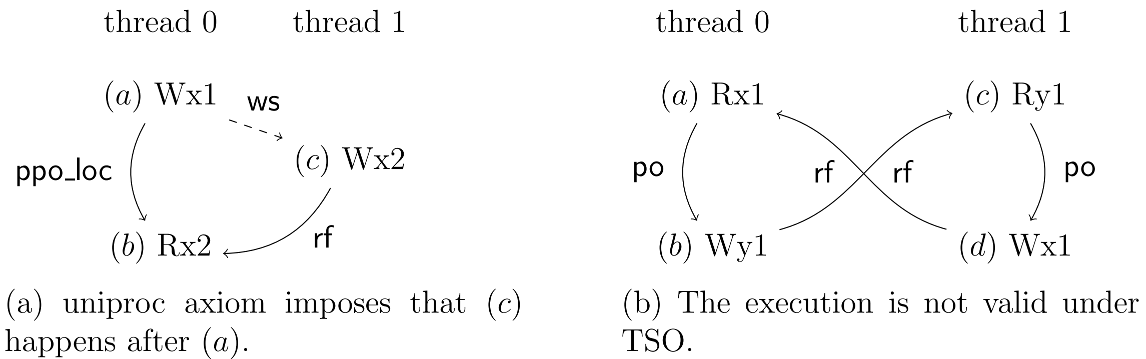
Figure 2.11: Relations maintained by architectures.

Definition 1 (Validity)**.** *The execution X is valid w.r.t. the event structure E under the architecture A (written $A.valid(E, X)$), if $acyclic(\textsf{po-loc} \cup \textsf{rf} \cup \textsf{fr} \cup \textsf{ws}) \wedge acyclic(\textsf{ppo} \cup \textsf{grf} \cup \textsf{fr} \cup \textsf{ws}) \wedge acyclic(\textsf{dp} \cup \textsf{rf})$.*

We now explain the intuitions behind each of these three constraints. The intuition behind **(uniproc)** is that, if a thread updates a memory location and read a different value from this same location, then the external update responsible for this new value in memory must happen after the first write. In Fig. 2.12(a), uniproc axiom imposes that the event (a) and (b) are ordered by write-serialisation ws in these event structure and execution. A direct consequence is that any value taken by a specific memory location can be obtained by interpreting an interleaving for this very memory location. We used this property in Chap. 2 to show that non-relational program analysis were sound for weak memory models.

(GHB) describes a constraint over the *global-happen-before* order, that we introduced in Fig. 2.10. In Fig. 2.12(b), the execution described is impossible under x86: intuitively, (c) has to happen before (b), which should happen before (a) since the order between R and W is maintained under this architecture. (a) reads the value written by (d), thus should happen before. And (c) should happen before (d), which forms a cycle of consequences. No sequence of events can satisfy this constraint.

(thin-air) is a constraint to prevent values from coming "*out of thin air*". In Fig. 2.12(b), if the pairs of events (a) and (b), and (c) and (d) are in dependency, it means that the value read and treated in (a) (respectively (c)) is used for the write in (b) (respectively (d)). Yet, the value written by the write in (b) (respectively (d)) is needed by the read in (c) (respectively (a)) in order to compute data that will be used by the write of (d) (respectively (b)). This circular value dependency could raise any number, which is not a behaviour permitted on the architecture that the framework supports.



(a) uniproc axiom imposes that (c) happens after (a).

(b) The execution is not valid under TSO.

Figure 2.12: Two event structures and executions.

[AMT13] introduced a more generic way of expressing architectures, which allowed in particular to write a thinner specification of the Power memory consistency. We discuss in Chap. 6 how to extend our techniques and tools to exploit these specifications.

An important concept in subsequent chapters is the notion of a *critical cycle*. A critical cycle is a cycle in **(GHB)** under an architecture B that is no longer a cycle under a weaker architecture A [SS88]. In Fig. 2.10, $(a) \mapsto (b) \mapsto (c) \mapsto (d) \mapsto (a)$ is a cycle in **GHB** under the architecture of Fig. 2.10(a) and does not form a cycle under the architecture of Fig. 2.10(b): it thus forms a critical cycle for these two architectures.

2.1.4 Impact over the program semantics

We wrote in the introduction of the chapter that weak memory consistency could affect the semantics of a concurrent program. We explained that the specific behaviours permitted on modern multicore processors could indeed be counter-intuitive—especially if one assumes sequential consistency when programming. These behaviours can affect any place in the code that relies on a specific order of accesses to shared memory and involves at least two distinct memory locations—otherwise, the **(uniproc)** axiom guarantees that any value taken by one single shared variable at any time can be captured by an interleaving of the program threads. These cases appear in particular in algorithms and techniques implementing some thread synchronisations. This is why we will experiment our techniques and tools first on some classic mutual exclusion algorithms in Chap. 4 and Chap. 5.

In verification for weak memory, we find mainly two approaches in the literature:

(full) either detecting (and fixing) any behaviours that is not sequentially consistent (e.g. [AKNT13, BDM13]);

(prop) or checking that some specifications are not violated by some new weak memory behaviours—with for instance the help of assertions in the code [AKT13], error states in the transition system [LW11] or temporal logic properties [DMD13].

(full) does not require prior knowledge of places in the code that might be sensitive to weak memory. However, the techniques addressing this problem will impact all the semantics of the input program, and perhaps affect some optimisations, restrict the degree of concurrency of the program and finally impact its runtime performance. Tools addressing **(prop)** will be more precise and target only the parts of the code that may affect—through weak memory behaviours—the final state of the assertion or property. Yet, this requires existing specifications of the input programs, for SC or the base architecture considered.

(full) can be addressed by **(prop)** by computing all the possible states under sequential consistency at every program point, and insert after each program point an assertion checking that only SC states are reachable there. With properties, we can use the same strategy with a property specifying the possible variable environments at every program point. This solution is not feasible in general, due to the exponential number of traces and variable environments to consider. Addressing **(prop)** via **(full)** is also possible in some cases, by restricting the scope of search to the places that may impact assertions. Depending on the precision of this restriction, this may or may not strongly worsen the whole analysis efficiency.

In this dissertation, we assume that no additional information was provided by the developer in the code. We will thus mostly address **(full)**. Our instrumentation technique in Chap. 4 addresses **(full)**, but is actually used afterwards in combination with a model checker to solve **(prop)**. In our fence synthesis technique, we also address **(full)**, but suggests a method to limit it to (an over-approximation of) **(prop)**.

Most of our benchmark programs call external libraries—including concurrency libraries like POSIX threads. When analysing a program doing such, we must decide if the program should be considered in isolation—and assuming that the libraries are correct with respect to weak memory—or if we consider the whole system environment. In our experiments, we assume that libraries external to the programs or packages are correctly coded. We did not, however, implement an advanced synchronisation analysis, meaning that we will not rely on synchronisation libraries and will be capable of finding bugs that may involve an incorrect implementation of some libraries.

2.2 Memory Models and Program Analyses

In the previous section, we introduced some weak memory models that can apply to software verification. Prior to building static analyses in Chap. 3 and 5, and to safety verification in Chap. 4 that make use of these models, we briefly comment the combination of weak memory consistency and *program analyses*. As we will see in Chap. 3 and 4, some of our work will also rely on the soundness of program analyses for weak memory models. We indeed apply pointer analyses and dependency analyses prior to the proper static analysis.

This section is adapted from the work developed in [AKL+11], co-written with Jade Alglave, Daniel Kroening, John Lugton and Michael Tautschnig. Although soundness of pointer analysis for concurrent programs is essential for our static analysis developed in Chap. 3, the dissertation can be understood without this section.

2.2.1 Analyses expressed with Data Flow Equations

Analyses for sequential programs A common way of describing certain types of program analyses is to express the result as the solution of a set of data flow equations. These (possibly recursive) equations relate the results of the analysis before and/or after each instruction of the input program, with respect to a representation of the program flow (usually a control flow graph—CFG—or a set of traces in it). The equations can be written as a collection of *transfer functions*, also called *transformers*, $(f_l)_{l \in Stmt}$ over a complete lattice $< L, \sqsubseteq, \sqcup, \sqcap >$. To compute the solution to the whole program, there are two commonly used strategies. The first approach is the *meet-over-paths* solution (written **(mop)**). It computes the least upper bound of the analysis applied to all the traces valid in the program [NNH99]:

$$Analysis_{\mathbf{(mop)}}(\mathcal{P}) = \bigsqcup_{<i_0,\ldots,i_n> \in Traces_{\mathcal{P}}} f_{i_n} \circ \ldots \circ f_{i_0}(init).$$

This method may be easily undecidable [NNH99] due to the extraction of traces or infinite traces—an unbounded loop in the CFG forms a good example. The *maximal fixed point solution* (written **(mfp)**), which solves a set of equations related by the control flow graph of the program, is often preferred, as it overapproximates the intersections in the control flow graph with the least upper bound[9].

$$Analysis_{\mathbf{(mfp)}}(l) = \begin{cases} f_l(init), \text{ if } l \in Init_{\mathcal{P}} \\ f_l(\bigsqcup \{Analysis(l') | (l', l) \in Flow_{\mathcal{P}}\}) \text{ otherwise} \end{cases}$$

[9]Other expressions of the **(mfp)** are possible. We assume here that the lines at which information was computed are part of the domain of analysis.

$$Analysis_{\mathbf{(mfp)}}(P) = \bigsqcup_{l} Analysis_{\mathbf{(mfp)}}(l)$$

Because **(mfp)** over-approximates **(mop)**, we will always observe [NNH99]:

$$Analysis(\mathcal{P})_{\mathbf{(mop)}} \sqsubseteq Analysis(\mathcal{P})_{\mathbf{(mfp)}} \tag{2.1}$$

Analyses for concurrent programs The semantics of concurrent programs is more intricate than sequential programs semantics, since even under SC, one might need to consider a factorial number of traces. Similarly to the sequential case, we can solve a program analysis for concurrent programs predominantly by using two methods [Min11]: computing the join of all the traces—often called *trace semantics* or *interleaving semantics* **(trace)**—or overapproximating the trace semantics by computing least upper bounds between possibly interfering instructions—we call the latter *inference semantics* **(inter)**. Again, we will always observe that:

$$Analysis(\mathcal{P})_{\mathbf{(trace)}} \sqsubseteq Analysis(\mathcal{P})_{\mathbf{(inter)}} \tag{2.2}$$

Abstraction Most of the analyses do not work on the whole semantics of the program, but rather on an abstraction of it. The *abstract interpretation* framework [CC76] is often used to express the relationship[10] between the semantics of the analysis and the actual semantics of the program. An abstract interpretation, for a program $\mathcal{P}$ written in a programming language $\mathcal{L}$, and targeting an abstract domain $\mathcal{A}$, is an analysis that computes a result in $\mathcal{A}$ with certain guarantees. The result is captured from an abstract semantics $\mathcal{F}_{\mathcal{A}}$ in $\mathbb{P}_{\mathcal{L}} \to \mathcal{A}$, which abstracts the concrete (collecting) semantics $\mathcal{F}_{\mathcal{D}}$ in $\mathbb{P}_{\mathcal{L}} \to \mathcal{D}'$, where $\mathcal{D}$ is the semantic domain (in the context of software verification, the set of memory environments), $\mathbb{P}_{\mathcal{L}}$ the set of programs accepted by the language $\mathcal{L}$ and $\mathcal{D}' = \wp(\mathcal{D})$ [Cou96] to collect all the possible behaviours—multiple because of varying inputs, scheduling-dependent results or non-determinism. The main idea behind this semantics transformation is to simplify the domain of the result $< \wp(\mathcal{D}), \subseteq, \cup, \cap >$, where $\subseteq$, $\cup$ and $\cap$ are respectively the common inclusion relation, union and intersection for sets. The hope is indeed to work in a simpler lattice $< \mathcal{A}, \sqsubseteq, \sqcup, \sqcap >$ where fixed point computations will terminate[11]

[10] In detail, the two semantics must form a Galois connection.

[11] Possibly with the additional use of a widening operator ∇, as *e.g.* in [Min01].

For example, in the octagon abstraction described in [Min01], $\mathcal{L}$ is a simple imperative language, with global variables, assignments, arithmetic operators and *if*-statements. The concrete domain D is the sets of values that the variables can hold at every line of the program, *i.e.*, $\mathcal{D} = Loc \rightarrow (Var \rightarrow Val)$, where Loc, Var and Val are respectively the sets of program locations, variables and values. The collecting semantics is defined by $\mathcal{D}' = Loc \rightarrow \wp(Var \rightarrow Val)$. These sets of values are overapproximated by octagons in the abstract domain $\mathcal{A} = Loc \rightarrow Oct^N$, where Oct is the set of octagons and N the number of octagons we use to represent the variables[12]. Each octagon relates the values of two variables with bounded constraints of the form $ax + by \leq c$, where x and y are the values of two variables, a and b two integers in $\{-1, 0, 1\}$ and c in $\mathbb{Q}$. The concretisation of a set of octagons lies in $\wp(Var \rightarrow Val)$.

In the section, an *abstraction* will refer to a map α from $\mathcal{D}'$ to $\mathcal{A}$, and a *concretisation* will represent a map γ from $\mathcal{A}$ to $\mathcal{D}'$.

Soundness of a program analysis In formal verification, one of the most important properties an analysis can satisfy is the *soundness*. Informally, a program analysis is sound if it does not miss any result. It might however bring some false positive—spurious results—unless the analysis is proved to be *complete*. Soundness can be defined as a relation $\sigma \subseteq \mathcal{D}' \times \mathcal{D}'$ which satisfies $\forall \mathcal{P} \in \mathbb{P}_{\mathcal{L}}$, $\sigma\left(\mathcal{F}(\mathcal{P}), \gamma\left(\mathcal{F}'(\mathcal{P})\right)\right)$[13]. In most cases, the soundness relation can be defined as $\sigma = \subseteq$, *i.e.*, the overapproximation. This definition is used in particular for *may forward* analysis [NNH99], and it is the one we will adopt in Sec. 2.2.3 and Chap. 2.3.

Soundness in concurrency We have two ways of describing the concurrency of the program: trace semantics ((**trace**)), and interference semantics ((**inter**)). Using Equation 2.2.1, if the analysis is sound as a **(trace)** solution, then it is also sound as an **(inter)** solution. Indeed, $\forall \mathcal{P} \in \mathbb{P}_{\mathcal{L}}$, $\mathcal{F}_{\mathcal{D}}(\mathcal{P}) \subseteq Analysis(\mathrm{P})_{\mathbf{trace}} \subseteq Analysis(\mathrm{P})_{\mathbf{inter}}$. The converse is not true.

[12] Usually, $N = \binom{\#Var}{2}$, and all the combinations of two variables are represented. ConcurInterproc (*http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi*), which uses the APRON library [Jea09] does so—even though it represents a set of constraints, without delimiting the octagons. We can however overapproximate results by considering a limited number of octagons, that is, a limited number of relations among variables.

[13] We use here a concretisation function γ to ensure that the results can be compared in the same domain $\mathcal{D}'$. In [Cou96], the function γ is explicitly contained in the definition of $\mathcal{F}'$, whereas in [D'S10], based on Cousot and Cousot's definition [CC92, p. 516], the relation is defined on $\mathcal{D}' \times \mathcal{A}$, and γ would implicitly appear in the choice of the relation to consider.

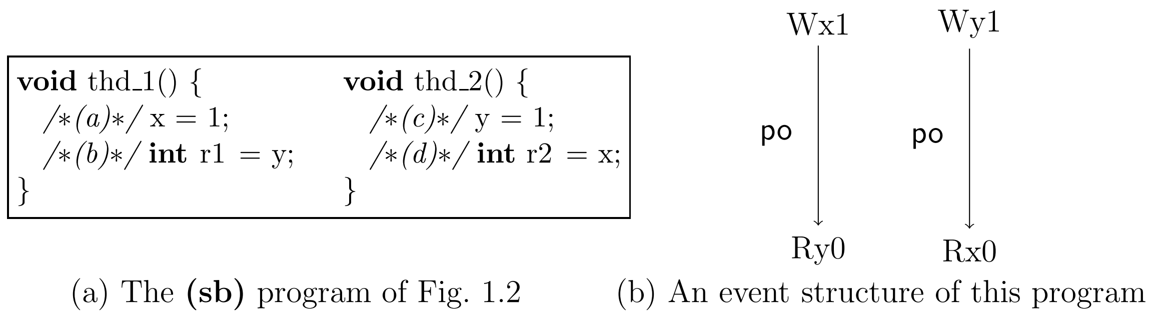
Relational and non-relational abstractions As explained in e.g. [JGR05], abstract domains can have various degrees of precision in representing the concrete elements and their relations. Relationality is certainly the most common property that distinguishes these domains. *Non-relational* domains, usually expressed with $A \to \wp(B)$, where A and B are sets, cannot express relations between several elements of A. For example, with all the $a_i \in A$ and all the $b_i \in B$, let us say that we want to encode: "if $a_2 \mapsto b_2$ then $a_1 \mapsto b_1$", with a_1 possibly holding the values b_1 or b_3, and a_2 the values b_2 or b_4. The only way to cover these pairs in $A \to \wp(B)$ is to use a function $\{a_1 \mapsto \{b_1, b_3\}, a_2 \mapsto \{b_2, b_4\}\}$. This function, however, is an overapproximation of what we wanted, since we can also have $a_1 \mapsto b_3$ and $a_2 \mapsto b_2$, that are not satisfying the implication we wanted to encode. A possible way to express this relation between elements of A is to consider the functions in the *relational* domain $\wp(A \to B)$: we could then have $\{\{a_1 \mapsto b_1, a_2 \mapsto b_2\}, \{a_1 \mapsto b_1, a_2 \mapsto b_4\}, \{a_1 \mapsto b_3, a_2 \mapsto b_4\}\}$, which is precisely the semantics of the implication. Every function in $A \to \wp(B)$ can be naturally expressed in $\wp(A \to B)$: this means that relational domains, $\wp(A \to B)$, are strictly more expressive than non-relational ones, $A \to \wp(B)$.

Note that in the following, we will more often write these domains with $\wp(A \times B)$ for $A \to \wp(B)$, for better readability. These domains are isomorphic. We will also write $\wp\wp(A \times B)$ for $\wp(A \to \wp(B))$.

Flow-sensitive and flow-insensitive analyses A flow-insensitive analysis is an analysis that always returns the same result for an input program, regardless of how the instructions are ordered in it [NNH99, p. 101]. In the context of concurrent programming, an analysis of this class would capture the solution of all the possible orders of instructions. Thus flow-insensitive analyses are SC-sound by definition. They are also more approximative than flow-sensitive analyses.

2.2.2 Abstract Event Structures as Traces

Because the soundness of **(trace)** implies **(inter)**, and since weak memory models as a set of constraints are more convenient to use with traces, we will decompose the program into traces and analyse them as if we were computing **(trace)**. As we explained in Sec. 2.1.3, event structures describe programs in terms of their trace semantics. They do not encode flow controls, addresses and values. To derive event structures from a description of the program we proceed in two steps. The set of all paths at program level first translates to a set of *abstract event structures*, where



(a) The **(sb)** program of Fig. 1.2 (b) An event structure of this program

Figure 2.13: **(sb)** program and its event structure.

events take the form of a direction and two variables. We resolve controls and addresses but leave the values unresolved. This allows us to translate, e.g., a store (a) $x \leftarrow \sigma$ to the (abstract) event ((a)) $\mathrm{W}x\sigma$.

We write $\mathcal{E}$ for the set of all abstract event structures, $\mathbb{A}$ for the set of all addresses, and $\mathbb{V}$ for the set of all values. We define the type $\mathcal{R}$ of *results* (or valuations) as $\mathcal{R} \triangleq \wp(\mathbb{A} \times \mathbb{V})$, i.e., a result is set of pairs (x, v) where x is an address and v a value.

Each abstract event structure induces multiple *concrete event structures* under a given set of initial valuations of variables. That is, an abstract event ((a)) $\mathrm{W}x\sigma$ with $R = \{(\sigma, 0), (\sigma, 1)\}$ translates to concrete events ((a)) $\mathrm{W}x0$ and ((a)) $\mathrm{W}x1$. We recall that a concrete event structure does not imply that there actually exists a program execution on an architecture A with the same values, but for each program execution there is a concrete event structure, as we explained in Sec. 2.1.3. The set of all sets of concrete event structures is denoted by $\mathcal{E}_{\mathrm{conc}}$. We use the mapping $\mathrm{conc} : \mathcal{E} \to \mathcal{R} \to \mathcal{E}_{\mathrm{conc}}$ to translate abstract to concrete event structures.

For this study of program analyses and weak memory models, we distinguish abstract from concrete event structures as follows: program analyses will be applied to abstract event structures, but reasoning about actual values will be performed in concrete event structures.

Example of an analysis under weak memory consistency We revisit the example given in Fig. 1.2. In Fig. 2.13, we construct an event structure of the program. We now analyse this event structure. We first perform an *interval* analysis [CC76] (or *box* analysis) to determine the possible values of local registers `r1` and `r2`. We compute an interval of values for each variable. The *join* operation results in the smallest interval that contains both intervals that are to be merged. We consider all possible interleavings of statements of the two threads and compute the join over all these

traces. For instance, for the traces $(a);(b);(c);(d)$ and $(c);(d);(a);(b)$ we obtain the intervals $[0,0]\times[1,1]$ and $[1,1]\times[0,0]$, respectively. The join $[1,1]\times[0,0]\sqcup[0,0]\times[1,1]$ yields the box $[0,1]\times[0,1]$, already including the result that can be derived from the other interleavings, i.e., $[1,1]\times[1,1]$. More interestingly, this overapproximation also includes the point $(0,0)$, which is the additional value that one can observe on a weak memory model.

2.2.3 Soundness of Analyses on Weak Memory Models

We define an *analysis* $[\![\cdot]\!]$ as a mapping between abstract event structures and initial valuations to sets of pairs (i,r) where i is a program location (of type $\mathbb{L}$) and r is a result as defined in the preceding section. We make explicit[14] the initial state of values of type $\mathcal{R}$, commonly being the empty set or the set of all possible values:

$$[\![\cdot]\!] \in \mathcal{E} \rightarrow \mathcal{R} \rightarrow \wp(\mathbb{L} \times \mathcal{R})$$

Note that our definition captures *relational* analyses. Indeed the type of the result, namely $\wp(\mathbb{L}\times\mathcal{R})$, can be rewritten as $\mathbb{L}\rightarrow\wp(\mathbb{A}\rightarrow\wp(\mathbb{V}))$.

Returning to the above example of an abstract event (and therefore a singleton event structure) (a) $\mathrm{W}x\sigma$ with initial valuations $R=\{(\sigma,0),(\sigma,1)\}$ we apply an analysis tracking the relation between x and σ as follows:

$$[\![((\mathrm{a}))\,\mathrm{W}x\sigma]\!](\{(\sigma,0),(\sigma,1)\}) = \{((a),\{(x,0),(\sigma,0)\}),((a),\{(x,1),(\sigma,1)\})\}$$

Definition of soundness in the context of weak memory Rinard and Rugina define in [RR99, A.3] an analysis to be *sound*

> *"[...] if it is at least as conservative as the result obtained by using the standard pointer analysis algorithm for sequential programs on all the interleavings of the legal executions."*

A *legal execution* corresponds to the execution of one thread. Thus their work assumes SC as the execution model. We generalise their idea to weak memory models. Given an architecture A, we write $\mathrm{values}_A(E,R)$ for the set of values that executions witnesses X can yield on A, where X is an execution witness associated to a concrete event structure obtained from concretising E with initial valuations R.

[14] Starting with a specific value is referred to as *seeding* in the literature, for instance in [RLL07].

We recall that we write $<_X$ for a linearisation order on program locations induced by an execution X, as we used in Sec. 2.1.3. Intuitively, it corresponds to the order in which the memory events of X hit the memory.

We write $\text{last}(r, i, x)$ when the location of x is less than (or unrelated to) (i) in $<_X$, and x is one of the last elements in the relation r, *i.e.* there is no element x' such that $(x, x') \in r$. If a given write $w = (i)Wxv$ is the last element in $\text{ws}(X)$ at location (i), then the value v is the *current value of x at location* (i). For example in Fig. 2.12, the current value of x at line (c) (resp. (a)) is 1 (resp. 2), for the last write to x at line (c) (resp. (a)) is the write $(c)Wx2$ (resp. $(a)Wx1$).

Thus, we define $\text{values}_A(E, R)$ as the set of possible *environments* at each location (an environment mapping each address to its current value):

$$\begin{aligned}\text{values}_A(E, R) \;\triangleq\; & \{(i, r) \mid \exists X.\, A.\text{valid}(\text{conc}(E, R), X) \wedge \forall x, v.\, (x, v) \in r \implies \\ & \exists w.\, ((\text{last}(\text{ws}(X), i, w) \vee \text{last}(\text{po}(X), i, w)) \wedge \text{addr}(w) = x \wedge \text{val}(w) = v)\}\end{aligned}$$

For example in Fig. 2.13, $\text{values}_{SC}(E, R)$ contains, for program location (a), the result $((a), \{(\mathtt{r1}, 0), (\mathtt{r2}, 0), (x, 1), (y, 0)\})$.

Formally, we define soundness of over-approximating analyses for a weak architecture A as follows:

Definition 2. *An analysis $[\![\cdot]\!]$ is A-*sound *iff the result of $[\![\cdot]\!]$ on an abstract event structure E with initial values R describes a state space at least as large as that of* $\text{values}_A(E, R)$*:*

$$\mathsf{sound}_A([\![\cdot]\!]) \;\triangleq\; \forall E, R.\, \text{values}_A(E, R) \preceq [\![E]\!](R)$$

with $\preceq$ the location-wise environment set inclusion, that is, $U \preceq V$ iff $\forall (i, r) \in U.\, \exists r'.(i, r') \in V \wedge r' \subseteq r$.

For instance, we have $\{((a), \{(\mathtt{r1}, 0), (\mathtt{r2}, 0), (x, 1), (y, 0)\})\} \preceq \{((a), \{(\mathtt{r1}, 0), (\mathtt{r2}, 0)\})$. This means that we consider an analysis result to be A-sound if it is at least as conservative as taking all the values yielded by all valid executions on A.

Note that under-approximating analyses for SC architectures are also under-approximating for all weak memory models, since for all weak architectures A, trivially $\text{values}_{\text{SC}}(E, R) \preceq \text{values}_A(E, R)$. We therefore focus the presentation on showing soundness of over-approximating analyses.

A-soundness for Non-relational Analyses We now define a particular class of program analyses by restricting the signature of the output of the analysis. We only consider analyses $\langle\!\langle\cdot\rangle\!\rangle$ that map abstract event structures to pairs (i, r) where i is a program location and r a result, with the additional constraint that r *is a singleton*:

$$\langle\!\langle\cdot\rangle\!\rangle \in \mathcal{E} \rightarrow \mathcal{R} \rightarrow \wp\,(\mathbb{L} \times (\mathbb{A} \times \mathbb{V}))$$

In practice, this means that we apply an abstraction to our general type of analyses to obtain *non-relational* ones. Indeed the restricted result type of an analysis $\langle\!\langle\cdot\rangle\!\rangle$ applied to an event structure E can be rewritten as $\mathbb{L} \rightarrow (\mathbb{A} \rightarrow \wp\,(\mathbb{V}))$.

The type of values_A has to be restricted similarly. We write $\widehat{\text{values}_A}(E, R)$ to indicate this, i.e. $\widehat{\text{values}_A}(E, R)$ is of type $\mathbb{L} \rightarrow (\mathbb{A} \rightarrow \wp\,(\mathbb{V}))$. For example in Fig. 2.13, $\widehat{\text{values}_{\text{SC}}}(E, R)$ contains $\{((a), (\texttt{r1}, 0)), ((a), (\texttt{r2}, 0)), ((a), (x, 1)), ((a), (y, 0))\}$.

We want to determine when a given analysis, although designed with SC in mind, is sound for a weak architecture A. For example, for the program given in Fig. 1.2, we have $\widehat{\text{values}_{\text{x86}}}(E, R) = \widehat{\text{values}_{\text{SC}}}(E, R)$ (with R mapping all variables to 0). Hence in this case, an analysis that computes at least $\widehat{\text{values}_{\text{SC}}}(E, R)$ is also sound for x86, since it also computes all the values that this specific program can yield on an x86 machine. We show that *any analysis* $\langle\!\langle\cdot\rangle\!\rangle$ with (1) matching signature and (2) that is SC-sound as defined above satisfies this requirement. This means that collecting the values produced by the SC executions (i.e. $\widehat{\text{values}_{\text{SC}}}(E, R)$) suffices to obtain the values yielded by a weaker model A. This property is guaranteed by the **(uniproc)** check, since **(uniproc)** means that SC holds per location. To prove this claim, we first show the inclusion of value sets.

Lemma 1. $\forall E, R.\, \widehat{\text{values}_A}(E, R) \subseteq \widehat{\text{values}_{SC}}(E, R)$

Details of the proof can be found in [AKL⁺11]. The argument used for this proof is the one of **(uniproc)**, which ensures that any value obtained during an execution valid in the weak memory framework can be obtained by an interleaving.

The lemma is sufficient to show our main theorem, which states that for a non-relational analysis $\langle\!\langle\cdot\rangle\!\rangle$, its SC-soundness (i.e., $\forall E, R.\, \widehat{\text{values}_{\text{SC}}}(E, R) \subseteq \langle\!\langle E\rangle\!\rangle(R)$) entails its A-soundness on any architecture A. That is to say, we show that a non-relational analysis, though defined with SC in mind, is sound on a weaker architecture A when this analysis collects at least all the values yielded by all the executions valid on SC:

Theorem 1. $\forall \langle\!\langle\cdot\rangle\!\rangle \in \mathcal{E} \rightarrow \mathcal{R} \rightarrow \wp\,(\mathbb{L} \times (\mathbb{A} \times \mathbb{V}))\,.\, \mathsf{sound}_{SC}(\langle\!\langle\cdot\rangle\!\rangle) \implies \mathsf{sound}_A(\langle\!\langle\cdot\rangle\!\rangle)$

The result is obtained by reasoning over traces **(trace)**. Traces are the most precise, yet not necessarily computable, representation for program executions. Hence our results are *independent of (1) programming language specifics* such as locks or dynamic synchronization primitives and hold for all other program representations, such as (concurrent) control flow graphs (MFP) or Petri nets, since they are overapproximations of the sets of traces; *(2) analysis specifics* such as fixed point iteration strategies and sources of imprecision[15].

Proving Soundness of Analyses over Programs Thm. 1 gives sufficient conditions for an analysis over event structures to be A-sound. We explain here how this result transfers to programs.

Given an analysis designed for a specific language $\mathcal{L}$, we need to relate the set $\mathbb{P}_{\mathcal{L}}$ of all the programs which can be written in this language to the event structures. We introduce $S^{\text{aes}} \in \mathbb{P}_{\mathcal{L}} \rightarrow \wp(\mathcal{E} \times \mathcal{R})$, which maps a program $\mathcal{P}$ to corresponding abstract event structures and initial values, w.r.t. the semantics of the language $\mathcal{L}$. Each event created by S^{aes} is labelled by the program counter of the statement in $\mathcal{P}$ it is derived from.

To express the soundness of a program analysis $[\![\cdot]\!]_{\mathcal{L}}$, we require $\text{values}_A(E, R)$ and $[\![\mathcal{P}]\!]_{\mathcal{L}}$ to have the same type $\wp(\mathbb{L} \times \mathcal{R})$, with $\mathbb{L}$ being the program counters of statements in program $\mathcal{P}$. As above, we define $\widetilde{\text{values}}_A(\mathcal{P})$ as the values yielded by executions of $\mathcal{P}$ on A, i.e. $\widetilde{\text{values}}_A(\mathcal{P}) \triangleq \bigcup_{(E,R) \in S^{\text{aes}}(\mathcal{P})} \text{values}_A(E, R)$. Hence the A-soundness of a program analysis is merely a lifting of the A-soundness of the corresponding event structure analysis:

$$\widetilde{\mathsf{sound}}_A([\![\cdot]\!]_{\mathcal{L}}) \triangleq \forall \mathcal{P}.\, \widetilde{\text{values}}_A(\mathcal{P}) \preceq [\![\mathcal{P}]\!]_{\mathcal{L}}$$

Therefore, the A-soundness of SC-sound non-relational program analyses holds as a corollary of Thm. 1:

Corollary 1. $\forall [\![\cdot]\!]_{\mathcal{L}} \in \mathbb{P}_{\mathcal{L}} \rightarrow \wp(\mathbb{L} \times (\mathbb{A} \times \mathbb{V}))\,.\, \widetilde{\mathsf{sound}}_{SC}([\![\cdot]\!]_{\mathcal{L}}) \implies \widetilde{\mathsf{sound}}_A([\![\cdot]\!]_{\mathcal{L}})$

We surveyed some domains commonly used in the program analysis literature in App. A. We report the analyses sound or not proven sound in Fig. 2.22. We emphasise that pointer analyses implemented as *points-to*—that is, with a representation of the memory—are sound for weak memory models. We will use this result in the construction of the abstraction in Chap. 3. Soundness w.r.t. weak memory models is not necessarily true in the case of alias analyses, that can be relational.

[15]This imprecision might however cover the missing states of an unsound analysis by **(trace)**.

2.2.4 Other approaches to program analyses for weak memory

In case of unsound analyses, we suggested in [AKL^{+}11] a repairing strategy, more precise than the Cartesian abstraction, that follows the idea developed by Rinard and Rugina [RR99] for pointer analysis targeting concurrent programs. We apply the analysis to each thread in isolation, then analyse them again using the results previously computed on the other interfering threads. We iterate until reaching a fixed point.

Some papers have proposed different approaches to the problem of program analyses for memory models weaker than SC, but existing proofs are usually tailored for a specific analysis. Chugh *et al.* and De *et al.* suggested the removal of all the dataraces[16] in programs [CVJL08, DDN11], followed by the use of an SC-sound analysis for datarace-free programs. Some concurrent programs might, however, strongly rely on these dataraces, the conversion to datarace-free program might not be trivial and the synchronisations added to the program could impact negatively on its runtime performance. Therefore we did not assume this property.

Another strategy consists of inserting memory barriers in the program to restore its sequential consistency, and run some SC-sound analyses on it [BAM, AM11, AKNP14]. We will discuss this transformation in detail in Chap. 5.

Miné proposed in [Min11] to abstract the trace semantics for concurrent programs by an interference semantics. He introduces some program transformations on traces in order to model a weaker memory consistency, and proved that these transformations preserve the semantics of the program. However, the model that these transformations encode might not match the weak memory models we consider.

[Fer08] (implicitly) used a similar interference semantics, but adapted to Happen-Before rules, to perform analysis on Happen-Before memory model (including Java Memory Model).

[Yan04] proposes a generic framework that, given a memory model expressed as constraints and a language semantics, computes the result of a static analysis by constraint solving. This is, however, a theoretical framework, not designed to be efficient. It is quite unlikely that program analyses expressed as dataflow equations can be computed by this means.

[16] Under C11, this should certainly be restricted to the only variables non-declared atomic, as pointed Paul McKenney.

2.3 Rephrasing Trace Properties in terms of Static Critical Cycles

In this section, we recall the concepts necessary to define the validity of an execution on an architecture. The properties that we investigate in Chap. 2, Chap. 4 and Chap. 5—namely the robustness of a program, the weak memory instrumentation and the soundness of a program analysis—are all defined w.r.t. *traces* of an input program. We call a trace a sequence of memory updates resulting from the interpretation of a single path in the CFG of the program. More specifically, we instantiate these traces as the valid executions that we introduced earlier, relying on the model of Alglave [Alg10]. However, traces (and executions) are inconvenient for expressing properties statically over a program. Indeed, enumerating all the potential traces (or executions) that the program could yield is not practically feasible in general. We will instead rely on the existence of static critical cycles. The (dynamic) critical cycles, introduced by Shasha and Snir [SS88] and adapted to contemporary memory models by Alglave and Maranget [AM11], are relations amongst accesses to shared memory that characterise situations that would correspond to a non-SC execution. We will explain how critical cycles relate to valid executions, how their static counterparts can over-approximate them, and how to express the aforementioned properties in terms of static critical cycle.

The content of this section is not new. The definitions of the executions and critical cycles are borrowed from [SS88] and [Alg10]. The notion of *robustness* for traces was introduced by Alglave et al. in [AM11] (under the name of *stability*) and Bouajjani et al. in [BMM11]. We however express these properties in terms of static critical cycles rather than dynamic valid executions.

2.3.1 Valid executions and critical cycles

The characterisations of execution validity under an architecture depend mainly on acyclicity. An invalid execution (for an event structure) as the one presented in Fig. 2.12(b) thus contains a cycle in the order. This cycle is a *critical cycle* [SS88]. Shasha and Snir established in [SS88, Sec. 3] the equivalence between the existence of a critical cycle in an execution and the invalidity of the execution. This equivalence was proved for the framework we use in [Alg10, p. 153] .

Property 1 (Critical cycle and execution validity)**.** *Given an architecture A and an event structure, an execution is invalid iff there exists a critical cycle in the orders.*

Example. In Fig. 2.12(b), we consider the architecture TSO, meaning that the execution is invalid if there is a (critical) cycle in $\mathsf{ppo} \cup \mathsf{rf} \cup \mathsf{ws} \cup \mathsf{fr}$*, where* ppo *are all the* po *except write-read. There is indeed a critical cycle:* (a, b, c, d)*. Under Power,* ppo *does not contain read-write, meaning that there is no critical cycle in this execution. The execution would be valid.*

Shasha and Snir also suggest searching cycles inside a graph summarising all the communications between the threads and the program orders in [SS88, Sec. 4]. This graph would capture all the executions of the program. Yet, this graph is dynamic: to ensure that Prop. 1 always holds, the graph must capture all the values, have all the memory addresses resolved, all the flow controls solved. An unbounded execution corresponds to an infinite path in the graph, so this graph would also need to be infinite in case of unbounded loops.

Computing this dynamic graph or all the dynamic executions is inaccessible to a technique that would interpret statically the code of a program, due to the combinatoric explosion of trace enumeration. Exploring all the executions of the program is not a valid option. We computed an over-approximation of the graph, the AEG, in Chap. 3, which captures all the dynamic, critical cycles without enumerating executions. We indeed showed that each dynamic, critical cycle was covered by a static, critical cycle in the AEG. Under this abstraction, we relaxed the equivalence of Prop. 1 into a simple implication: if there exists an invalid execution under a given architecture, then it is captured by one static, critical cycle in the AEG. Conversely, if there is no static, critical cycle in the AEG, then there is no execution that cannot be captured by an interleaving.

We can also consider executions of a program that would be valid under an architecture A but not under a stronger architecture B. In this case, there are static, critical cycles in the AEG of the program that exist under architecture B and not under architecture A. This is an implication, so if there are such cycles, it means that there *might* exist executions of this program that are valid under A but not under B. This implication is however sufficient for us, as we target soundness, that is, not missing any executions.

2.3.2 Trace properties for programs and program analyses

We define a couple of properties that we will exploit in the next chapters. We will first express these properties on the basis of executions allowed on some ar-

```
void thd_1() {        void thd_2() {
  x = 1;                int r1 = y;
  y = 1;                int r2 = x;
}                     }
```

Outcomes for r1 and r2:

SC, TSO	PSO, ..., $Power/ARM$
(0,0), (0,1), (1,1)	(0,0), (0,1), (1,1), (1,0)

Figure 2.14: A program implementing **(mp)** and its outcomes per architecture.

chitectures. Given a program P and an architecture A, $\mathsf{traces}_A(P)$ is the set of all the executions of P run on A. E.g., $\mathsf{traces}_{SC}(P)$ is the set of all the interleavings of P. In Alglave's framework, we would define $\mathsf{traces}_A(P)$ as $\mathsf{traces}_A(P) \triangleq \{\tau \mid \exists \mathcal{E} \in \text{event_structures}(P), A.\text{valid}(\mathcal{E}, \tau)\}$, where event_structures$(P)$ is the set of event structrures that could be extracted from P (as introduced in the discussion about semantics in Chap. 3).

Properties for programs

Robustness [BMM11] (also called *stability* in [AM11]) is a property qualifying a program with respect to a memory consistency. Given an architecture A, a program P will be robust on A if running it on A will not produce any new behaviours as if it were run under SC.

Definition 3 (Robustness)**.** *Given an architecture A, given a program P, $\mathsf{robust}_A(P) \triangleq \mathsf{traces}_A(P) \subseteq \mathsf{traces}_{SC}(P)$.*

Example. ***(mp)*** *(e.g. in Fig. 2.14) is robust under TSO. It is not robust under Power, since there is $\tau \in \mathsf{traces}_{Power}(MP)$ which ends up with $r_1 \equiv 1 \wedge r_2 \equiv 0$, whereas none of those in $\mathsf{traces}_{TSO}(MP)$ can.*

In practice, a program robust for A is a program that can be understood under sequential consistency for any architecture $A_1 \geq A$.

The definition can be generalised to the robustness of a program P on an architecture A against an architecture B (we will assume that $A < B$). It means that running the program P on A will not cause new behaviours that would not be captured by a run on B.

More formally, we define parametric robustness as follows.

Definition 4 (Parametric robustness)**.** *Given two architectures A and B such that $A < B$, given a program P, $\mathsf{robust}_{A/B}(P) \triangleq \mathsf{traces}_A(P) \subseteq \mathsf{traces}_B(P)$.*

Example. ***(sb)*** *(e.g. in Fig. 2.15) is robust for Power against TSO. Indeed, the only additional traces that can appear compared to SC are contained in TSO.*

In practice, a program robust for A against B can be directly ported from B to A without adding synchronisation and still be correct (provided it is correct for B). A direct property is that parametric robustness is implied by the former definition of robustness. This definition is weaker.

Property 2 (Parametric robustness is weaker)**.** *Given a weak memory architecture* B*,* $\mathsf{robust}_A(P) \Rightarrow \mathsf{robust}_{A/B}(P)$*.*

Proof. $B < SC$*, meaning that* $\mathsf{traces}_{SC}(P) \subseteq \mathsf{traces}_B(P)$*.* □

Example. ***(sb+fences)*** *(e.g. in Fig. 2.16) is robust for Power, this robust for Power against TSO.*

When P is also known to be robust for B, robustness for A against B implies the robustness for A. This explains the following equivalence relating robustness in the sense of respectively Def. 3 and Def. 4.

Property 3 (Robustness combination)**.** $\mathsf{robust}_{A/B}(P) \wedge \mathsf{robust}_B(P) \equiv \mathsf{robust}_A(P)$*.*

Proof. *Prop. 2 for* $(\Leftarrow)$*,* $\subseteq$ *transitivity for* $(\Rightarrow)$*.* □

Example. ***(sb+fences)*** *(e.g. in Fig. 2.16) is robust for Power against TSO, and* ***(sb+fences)*** *is robust for TSO. Therefore, it is robust for Power.*

The weaker robustness definition will help us defining the problem of porting a program from one weak architecture to another. Restoring robustness will be the subject of Chap. 5.

In Chap. 4, we will develop a program transformation for making explicit the weak memory consistency. It means that, given a weakly consistent architecture A, the program transformed for this architecture A can be interpreted under a stronger architecture (like SC) and still reveal the same behaviours as if it were executed under A. We define this transformation formally below.

```
void thd_1() {        void thd_2() {
  x = 1;                y = 1;
  int r1 = y;           int r2 = x;
}                     }
```

Outcomes for r1 and r2:

SC	*TSO, ..., Power/ARM*
(1,0), (0,1), (1,1)	(0,0), (0,1), (1,0), (1,1)

Figure 2.15: A program implementing **(sb)** and its outcomes per architecture.

```
void thd_1() {            void thd_2() {
  x = 1;                    y = 1;
  asm("sync");              asm("sync");
  int r1 = y;               int r2 = x;
}                         }
```

Outcomes for r1 and r2:

SC, ..., *Power/ARM*
(0,1), (1,0), (1,1)

Figure 2.16: A program implementing **(sb+fences)** and its outcomes per architecture.

```
void thd_1() {                    void thd_2() {
  if(*)                             if(*)
    x = 1;                            y = 1;
  else                              else
    buffer[x].put(1);                 buffer[y].put(1);
  int r1;                           int r2;
  if(*)                             if(*)
    r1 = y;                           r2 = x;
  else                              else
    r1 = buffer[y].take();            r2 = buffer[x].take();
}                                 }
```

Outcomes for r1 and r2:

SC, ..., *Power/ARM*
(0,0), (1,0), (0,1), (1,1)

Figure 2.17: A program implementing $(\tilde{\mathbf{sb}}_{TSO})$ and its outcomes per architecture.

Definition 5 (Transformation). *Given two architectures A and B such that $A < B$, given a program P, a transformation from B to A produces an instrumented program P' such that $\mathsf{trans}_{B \to A}(P, P') \triangleq \mathsf{traces}_A(P) \subseteq \mathsf{traces}_B(P')$.*

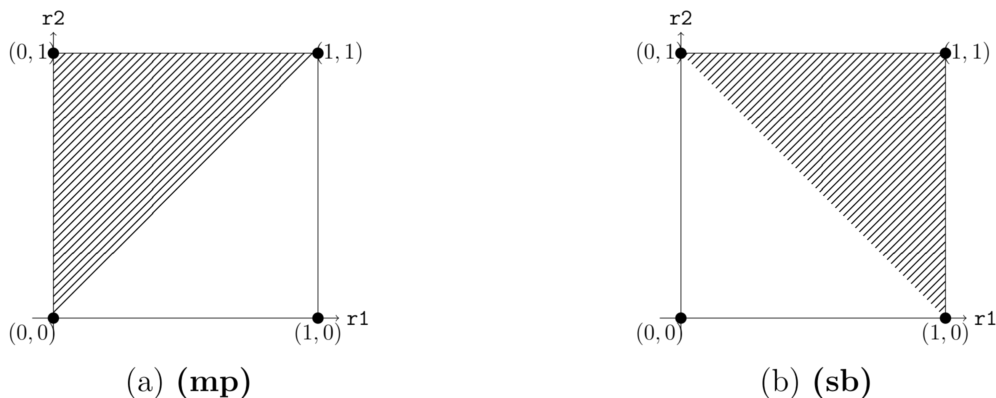
Example. *$(\tilde{\mathbf{sb}}_{TSO})$ (e.g. in Fig. 2.17) generates all the traces under SC that **(sb)** (e.g. in Fig. 2.15) would under TSO.*

A trivial property is that if the instrumented program is the same as the original program, then the original program is robust. Conversely, if a program is robust, its transformed program will be identical.

Property 4 (Robustness by transformation). *$\mathsf{trans}_{B \to A}(P, P) \equiv \mathsf{robust}_{A/B}(P)$.*

Example. *We have $\mathsf{trans}_{SC \to TSO}(MP, MP)$, since **(mp)** (e.g. in Fig. 2.14) is robust for TSO.*

In practice, it does not mean that a robust program would not be transformed by our transformer. It means that it would be unnecessary. Due to over-approximations, it can happen that the P' we get is different from P, even though P is robust. The converse however still applies: if the transformer does not modify P, then since the



(a) **(mp)** (b) **(sb)**

Figure 2.18: Interpretation of r_1 and r_2 in respectively **(mp)** (e.g. from Fig. 2.14) and **(sb)** (e.g. from Fig. 2.15) after the last line under the octagon domain [Min01].

approach is sound, it means that P is robust.

Properties for program analyses

In Sec. 2.2, we investigated the soundness of program analyses for weak memory semantics. We first define the soundness of a program analysis f at the level of a program P, assuming the soundness of f for SC, i.e., $\mathsf{sound}_{SC}(f)$ (and in particular $\mathsf{sound}_{SC}(f, P)$). We recall that we write $f(\mathsf{traces}_A(P))$ for $\bigsqcup_{t \in \mathsf{traces}_A(P)} f(t)$.

Definition 6 (Soundness of program analysis w.r.t. a program)**.** *Given an architecture A, a program analysis f is sound for A w.r.t. a program P if $\mathsf{sound}_A(f, P) \triangleq \mathsf{sound}_{SC}(f, P) \Rightarrow f(\mathsf{traces}_A(P)) \subseteq f(\mathsf{traces}_{SC}(P))$.*

*Example. The abstract interpretation of **(mp)** (e.g. in Fig. 2.18) under the octagon abstraction is sound for **(mp)** under TSO. It is not sound for **(sb)** under TSO (e.g. in Fig. 2.18).*

Definition 7 (Soundness of program analysis)**.** *A program analysis is sound for an architecture A if $\mathsf{sound}_A(f) \triangleq \forall P. \mathsf{sound}_A(f, P)$.*

Example. A points-to analysis such as the one described in [RR99] is sound for Power, as it captures all the addresses the pointers could point to despite the weak memory consistency, thanks to the abstraction (see Sec. 2.2).

Note that, by these definitions, we mostly work on the meet-over-path version of the

program analyses. If the meet-over-path is sound, so is the maximal-fixed-point one. It might be the case that, for some specific analyses, the meet-over-path solution would be unsound, but the maximal-fixed-point would be sound thanks to the over-approximation of the meet operator. We leave this question as future work.

We extend the soundness definition with the basis architecture as parameter. A program analysis f is sound for an architecture A against an architecture B if, if f is known to be sound for B, then all the results for a program P under A would be contained in the results for P under B.

Definition 8 (Parametric soundness of an analysis w.r.t. a program)**.** *Given two architectures A and B such that $A < B$, a program analysis f is sound for A against B w.r.t. a program P if $\mathsf{sound}_{A/B}(f, P) \triangleq \mathsf{sound}_B(f, P) \Rightarrow f(\mathsf{traces}_A(P)) \subseteq f(\mathsf{traces}_B(P))$.*

*Example. The octagon abstraction is not sound for TSO: $(r_1, r_2) = (0, 0)$ would be missing when analysing **(sb)** (Fig. 2.15). If however the program is known to be correct despite the case where $(r_1, r_2) = (0, 0)$—i.e., if the analysis is actually sound for the program under TSO—then it is also sound for Power and ARM, as they would not produce any new values.*

Definition 9 (Parametric soundness of an analysis)**.** *A program analysis is sound for an architecture A against a base architecture B if $\mathsf{sound}_{A/B}(f) \triangleq \forall P.\mathsf{sound}_{A/B}(f, P)$.*

An immediate property is that soundness for A against B implies soundness for A.

Property 5 (Soundness combination)**.** $\mathsf{sound}_{A/B}(f) \wedge \mathsf{sound}_B(f) \Rightarrow \mathsf{sound}_A(f)$.

Proof. Given a program P, we combine the two hypotheses to get $f(\mathsf{traces}_A(P)) \subseteq f(\mathsf{traces}_B(P))$. We want to prove $\mathsf{sound}_A(f, P)$, that is, $\mathsf{sound}_{SC}(f, P) \Rightarrow f(\mathsf{traces}_A(P)) \subseteq f(\mathsf{traces}_{SC}(P))$. By combining $\mathsf{sound}_B(f, P)$ again with $\mathsf{sound}_{SC}(f, P)$, we get $f(\mathsf{traces}_B(P)) \subseteq f(\mathsf{traces}_{SC}(P))$. Using $\subseteq$ transitivity, we prove the claim. □

Example. The interval analysis is sound for TSO, as we observed in Sec. 2.2. It is also sound for Power against TSO, as intervals form a non-relational domain. This program analysis is therefore sound for Power.

An interesting property is that if P is a program robust for A against B, f is monotonic and sound for B (at least for the program P), then f is sound for A w.r.t. P.

int a, b, c;	**void** thd_1() {	**void** thd_2() {
int∗ p = &c;	p = &a;	q = &b;
int∗ q = &c;	++∗q;	++∗p;
	}	}

Outcomes for a, b and c:

SC	TSO, ..., $Power/ARM$
(a,b,c): (1,0,1), (0,1,1), (1,1,0)	(a,b,c): (1,0,1), (0,1,1), (1,1,0), (0,0,2)

Figure 2.19: Concurrent program with pointers affected by weak memory reorderings.

Property 6 (Parametric robustness and soundness)**.** *$\textsf{robust}_{A/B}(P) \wedge monotonic(f) \Rightarrow \textsf{sound}_{A/B}(f, P)$.*

Proof. We use f monotonicity and the robustness to get $f(\textsf{traces}_A(P)) \subseteq f(\textsf{traces}_B(P))$. We then get directly the results. □

Example. Let us consider the points-to analysis of [RR99] and the program with pointers in Fig. 2.19. We know that this program is robust for RMO against TSO, since the only reorderings that can happen would be captured by TSO. We also know that the points-to analysis is monotonic. Therefore, the points-to analysis is sound for RMO against TSO for this pointer program. And since the points-to analysis is sound for TSO, it is sound for RMO for this program.

In particular, if we have $\textsf{robust}_A(P)$ and f monotonic, we get $\textsf{sound}_A(f, P)$ with the same proof. In other words, if an analysis is monotonic and a program is robust against A, then the analysis of this program will trivially be valid for A as well.

Property 7 (Robustness and soundness)**.** *$\textsf{robust}_A(P) \wedge monotonic(f) \Rightarrow \textsf{sound}_A(f, P)$.*

Proof. Identical to the previous proof. □

As we showed in Sec. 2.2, a program analysis can be unsound (e.g. under the octagon domain abstraction). We provided a strategy to restore the soundness. A program analysis is indeed repaired if it satisfies the next definition.

Definition 10 (Repair)**.** *Given two architectures A and B, a program P and a program analysis f, f' is the repaired program analysis in $\textsf{repair}_{B \to A}(f, f', P) \triangleq f(\textsf{traces}_A(P)) \subseteq f'(\textsf{traces}_B(P))$.*

Example. The repaired version of the analysis under the octagon domain contains all the values that could be obtained after an execution under one of the architecture covered by the framework.

A trivial property is that if we know the program analysis to be sound for B and if repairing it for A against B does not modify the analysis, then it is trivially sound for A against B. A good example is the concurrent pointer analysis that inspired the repair technique: it is sound for SC, repairing it to work on any architecture covered by Alglave's framework does not modify the analysis, thus it is sound for these architectures.

Property 8 (Repair and soundness)**.** *Given two architectures A and B such that $A < B$ and a program analysis f, $(\mathsf{sound}_B(f) \Rightarrow \mathsf{repair}_{B\rightarrow A}(f, f)) \equiv \mathsf{sound}_{A/B}(f)$.*

2.3.3 Relating executions and critical cycles

Executions are not convenient to manipulate statically. Their use can lead to some trivial combinatorial explosion in the analysis algorithms. As we mentioned before, we prefer to rely on the existence of critical cycles. A critical cycle contains all the information relevant to the impact of a reordering due to weak memory over the program execution. Constructing an over-approximation from the code is also reasonably direct, as we observed in Chap. 3. This over-approximation however weakens Prop. 1, which implies that there might be spurious static, critical cycles, i.e. cycles that would not correspond to a real dynamic cycle. The unreachability of these critical cycles in practice and the abstractions of the values, addresses and loops can explain this imprecision.

We now re-characterise the previous problems, robust and trans, using critical cycles rather than allowed executions. We also determine which static, critical cycles to consider after the search we did in Chap. 3. Some of the collected cycles might indeed be irrelevant to parametric properties assuming a basis architecture.

Robustness for an architecture A against a stronger architecture B consists of ensuring that all the executions of A can be reproduced in B. In terms of critical cycles, it means that all the cycles that we would find in the program under B should also exist under A. Otherwise, A would allow an execution[17] forbidden under B.

[17] In [BDM13], this execution is called trace τ.

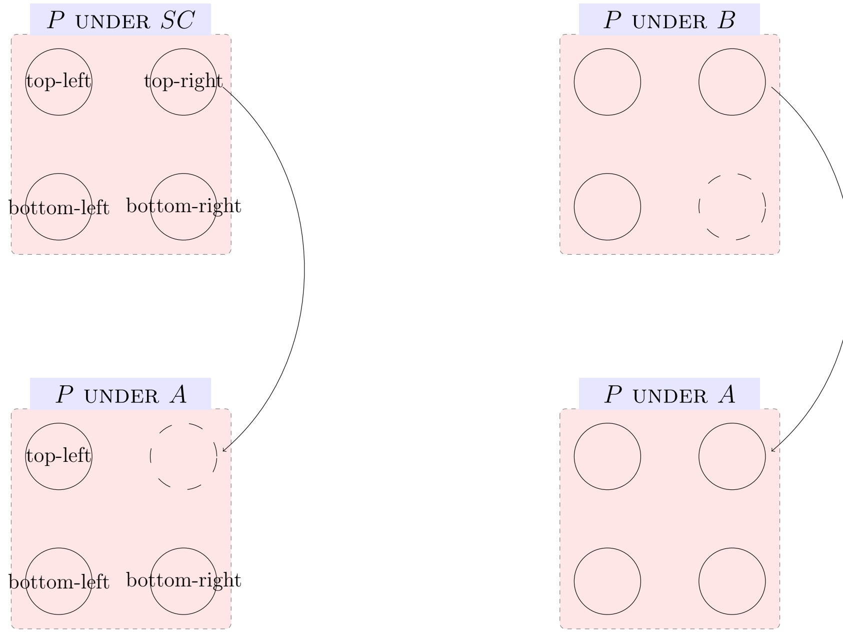


Figure 2.20: Robustness: a program which is not robust for A on the left; a program which is robust for A against B on the right.

Property 9 (Relating cycles and robustness)**.** *Given two architectures A and B such that $A < B$ and a program P, $\mathsf{ccycles}_B(P) \subseteq \mathsf{ccycles}_A(P) \Rightarrow \mathsf{robust}_{A/B}(P)$.*

Proof. *By contradiction: for any $\tau \in \mathsf{traces}_A(P)$, if $\tau \notin \mathsf{traces}_B(P)$, there is a critical cycle for B that does not exist in A. Yet, by hypothesis, $\mathsf{ccycles}_B(P) \subseteq \mathsf{ccycles}_A(P)$. Hence the robustness.* □

Example. *In Fig. 2.20, we represent a concurrent program P by a box. The continuous cycles in each of these boxes represent the static, critical cycles existing in the program for the architectured considered (in the label above the box). The dashed cycles are those that do not exist under the given architecture. The program P on the left is not robust since there is a critical cycle under SC that does not appear under A. It means that an execution possible on A (say TSO) is not representable as an interleaving, On the right hand side, the program P is robust for A against B. It also means that if the program is robust for B, then it is robust for A by Prop. 3.*

When implementing a robustness algorithm, it means that we need to look for the cycles safe in B, unsafe under A. The following problem defines this search.

Problem 1 (Find cycles)**.** *Find the critical cycles in $\mathsf{ccycles}_B(P) \backslash \mathsf{ccycles}_A(P)$.*

Example. In Fig. 2.20, $\mathsf{ccycles}_{SC}(P)\backslash\mathsf{ccycles}_A(P) = \{top\text{-}right\}$ *on the left hand side, which is indeed a critical cycle to address if we want to fix the robustness. On the right hand side, we have* $\mathsf{ccycles}_{SC}(P)\backslash\mathsf{ccycles}_A(P) = \emptyset$*. The program is indeed robust.*

Transformation of a program P from an architecture B to an architecture A requires that all the executions of P valid under A are captured when running the instrumented program P' under B. In terms of cycles, it means that any critical cycle appearing in the interpretations of P' under B must appear in the interpretations of P under A. Indeed, if there is a cycle in the interpretations of P' under B that does not appear in the interpretions of P under A, it means that there exists an execution of P valid under A which is not reproduced by P' under B.

Property 10 (Relating cycles and transformation)**.** *Given two architectures A and B such that $A < B$, a program P and a transformed program P',* $\mathsf{ccycles}_B(P') \subseteq \mathsf{ccycles}_A(P) \Rightarrow \mathsf{trans}_{B\rightarrow A}(P, P')$.

Proof.(informal) We use the same argument used in the proof of Prop. 9, except that we need to assume that if $\tau \notin \mathsf{traces}_B(P')$*, it is due to a reordering (and thus there exists a cycle) and not to the fact that* $P \neq P'$*. The latter is discarded by the nature of the transformation.* □

In practice, for a transformation, we want to compute the transformed program P', but we do not have access to it. We know the critical cycles of P under B and A respectively. We do a case analysis:

- $\mathsf{ccycles}_A(P)\backslash\mathsf{ccycles}_B(P)$, i.e. the critical cycles in A and not in B, should not exist since we assume that B is stronger than A. Otherwise, an execution could be invalid on A but valid on B.

- $\mathsf{ccycles}_A(P) \cap \mathsf{ccycles}_B(P)$, i.e., the cycles common to A and B. Because we want the cycles of P' under B to be in P interpreted under A, we do not need to change these cycles in P intrepreted under B.

- $\mathsf{ccycles}_B(P)\backslash\mathsf{ccycles}_A(P)$, i.e. the cycles in B that do not appear under A. These cycles need to be restored in P', so we modify them in P.

To compute the transformation, we thus need to solve the Prob. 1, that is, we have to look for $\mathsf{ccycles}_B(P)\backslash\mathsf{ccycles}_A(P)$. In Fig. 2.21, $\mathsf{ccycles}_{SC}(P)\backslash\mathsf{ccycles}_A(P) = \{\text{top-left}, \text{bottom-left}\}$ in the left program. These are indeed the cycles that need to

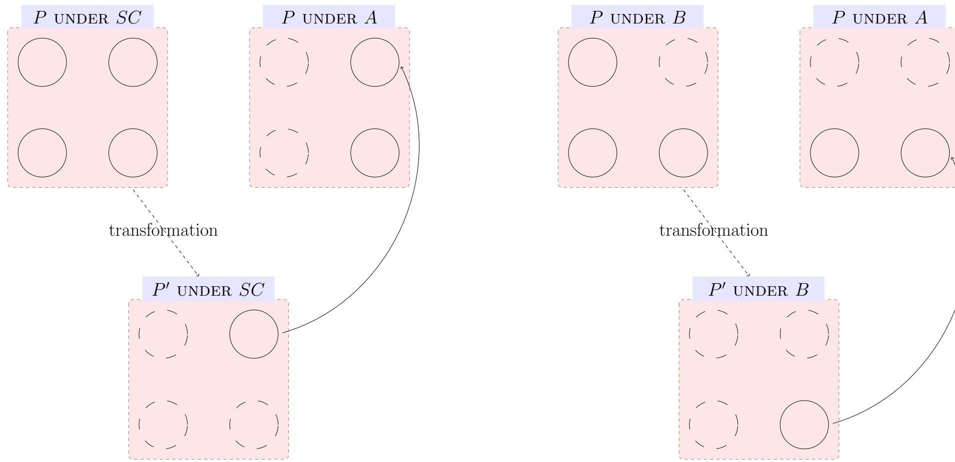


Figure 2.21: Transformation: a program transformation to A from SC on the left; a program transformation to A from B on the right.

be broken in P' so that P' under SC does not contain cycles which would not exist in P under A. Similarily, $\mathsf{ccycles}_B(P)\backslash\mathsf{ccycles}_A(P) = \{\text{top-left}\}$, which is the cycle to break in P'. That is, the additional executions to authorise so that all the executions of P under A are simulated by P' in B. We explain the transformation in detail in Chap. 4.

We finally relate the soundness of program analyses under weak memory consistency to critical cycles. We rely on the correspondance between robustness and critical cycles that we established in Prop. 9.

Property 11 (Relating cycles and soundness)**.** *Given a program analysis f, two architectures A and B such that $A < B$ and a program P, we have $\mathsf{ccycles}_B(P) \subseteq \mathsf{ccycles}_A(P) \wedge monotonic(f) \Rightarrow \mathsf{sound}_{A/B}(f, P)$.*

Proof. *We use Prop. 9, which gives $\mathsf{traces}_A(P) \subseteq \mathsf{traces}_B(P)$. f is monotonic, thus $f(\mathsf{traces}_A(P)) \subseteq f(\mathsf{traces}_B(P))$. We then get $\mathsf{sound}_{A/B}(f, P)$.* □

2.4 Summary

In this introductory chapter, we presented the background for weak memory that we will exploit for the next chapters. We explained the effects of weak memory consistency on the semantics of a concurrent program. We specified the limits of our studies, focussing on the reorderings due to weak memory consistency implemented by the processor. We introduced event structures and explained how they relate to

program source, program executions and executions. We presented Alglave's framework, which decides the validity of an execution for a given event structure under a given architecture.

We then turned to program analyses and their soundness for concurrent programs, and specifically with respect to weak memory models. We established that analyses working on non-relational domain were sound for weak memory, based on the **(uniproc)** axiom and some trace reasoning. We finally briefly surveyed a couple of analyses and their respective domains. The table in Fig. 2.22 summarises these analyses and domains. In particular, the points-to analysis as defined in [RR99] is sound w.r.t. weak memory models. We can apply it safely prior to any static analysis. We will do this in the next chapter, Chap. 3.

We finally enumerated properties for weak memory based on traces. We then reformulated the trace properties in terms of static critical cycles, and emphasised that looking for these cycles does not require keeping or constructing a precise historic of the execution. This abstraction of the historic is the key of our approach, and allows us to use the abstract event graph that we defined in Chap. 3 to detect static, critical cycles and perform some instrumentation or fence inference in respectively Chap. 4 and Chap. 5. We provide in Fig. 2.23 a table summarising the properties that we studied for weak memory consistency, their characterisations in terms of traces and critical cycles, and their applications in this dissertation.

Analysis	Reference	SC-sound	A-sound	SC-sound $\Rightarrow$ A-sound
box abstract interpretation	[CC76]+[Jea09]	✓	✓	✓
octagon abstract interpretation	[Min01]+[Jea09]	✓		
logahedron abstract interpretation	[HK09]+[Jea09]	✓		
TVPI abstract interpretation	[SKH02]+[Jea09]	✓		
polyhedron abstract interpretation	[CH78]+[Jea09]	✓		
points-to analysis + interleavings	[NNH99]	✓	✓	✓
iterative points-to analysis	[RR99]	✓	✓	✓
partition refinement points-to analysis	[Ste96]	✓	✓	✓
alias analysis + interleavings	[LR92]	✓	unkn.	unkn.
alias abstract interpretation + interleavings	[Deu92]	✓	unkn.	unkn.
bit vector analysis	[KD94]+[FK10] or [KSV96]			✓

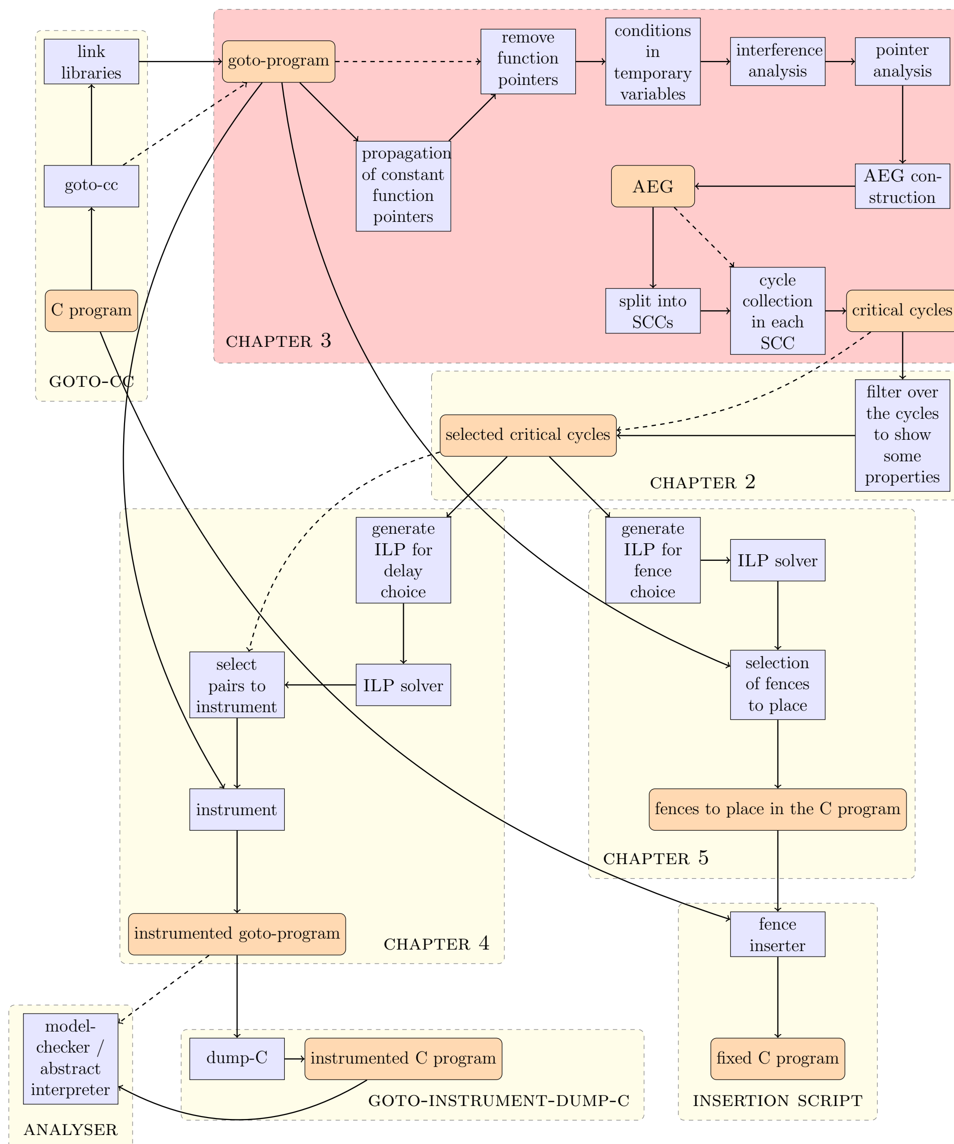
Figure 2.22: Summary of the surveyed program analyses. [a]+[b] in the references means that we need to combine the work in [a] and in [b] to obtain the analysis described in the entry.

property	denotation	trace expression	cycle characterisation	application
robustness	$\mathsf{robust}_A(P)$	$\mathsf{traces}_A(P) \subseteq \mathsf{traces}_{SC}(P)$	$\mathsf{ccycles}_{SC}(P) \subseteq \mathsf{ccycles}_A(P)$	fence insertion (Chap. 5)
parametric robustness	$\mathsf{robust}_{A/B}(P)$	$\mathsf{traces}_A(P) \subseteq \mathsf{traces}_B(P)$	$\mathsf{ccycles}_B(P) \subseteq \mathsf{ccycles}_A(P)$	fence insertion (Chap. 5)
transformation	$\mathsf{trans}_{B\rightarrow A}(P, P')$	$\mathsf{traces}_A(P) \subseteq \mathsf{traces}_B(P')$	$\mathsf{ccycles}_B(P') \subseteq \mathsf{ccycles}_A(P)$	instrumentation (Chap. 4)
soundness	$\mathsf{sound}_A(f)$	$\mathsf{sound}_{SC}(f, P) \Rightarrow f(\mathsf{traces}_A(P)) \subseteq f(\mathsf{traces}_{SC}(P))$	$\mathsf{ccycles}_{SC}(P) \subseteq \mathsf{ccycles}_A(P)$	program analysis (Chap. 2)
parametric soundness	$\mathsf{sound}_{A/B}(f)$	$\mathsf{sound}_B(f, P) \Rightarrow f(\mathsf{traces}_A(P)) \subseteq f(\mathsf{traces}_B(P))$	$\mathsf{ccycles}_B(P) \subseteq \mathsf{ccycles}_A(P)$	program analysis (Chap. 2)
repair	$\mathsf{repair}_{B\rightarrow A}(f, f', P)$	$f(\mathsf{traces}_A(P)) \subseteq f'(\mathsf{traces}_B(P))$	via Prop. 8	program analysis (Chap. 2)

Figure 2.23: Table summarising some properties w.r.t. weak memory models, their expressions and their applications.

Chapter 3

An Abstraction for Static Analyses over Weak Memory



In Chap. 2, we observed that some program analyses could not handle weak memory consistent semantics in a sound way. We studied and extracted a method from a pointer analysis for concurrent programs [RR99] that restores the soundness for weak memory. The proofs rely on some generic properties of the weak memory models, namely *uniproc* and *weak-uniproc* [AKL+11]. However, they require breaking relations between variables, meaning that a precise, sound program analysis might need to be over-approximated in order to ensure soundness for weak memory.

Program analyses can be seen as abstractions of the actual, operational semantics of a program run under an architecture. For instance, a trace-by-trace pointer analysis can be seen as a projection of the actual operational semantics of the program onto the pointer addresses and assignments to pointers. Instead of ensuring that program analyses are sound for weak memory given any program, one can prove that some specific programs are *robust* on an architecture A against a given architecture B. That is, all the executions observed when running the program on an architecture A can be explained by a trace of the program with the semantics of B. For instance, if we know that a program P is robust against SC when run on TSO, it means that all the executions of P on a x86 processor can be explained by an interleaving.

A direct consequence of the robustness is that, if a program is robust for A against B, we know that, if we run a program analysis sound for B, it will also be sound for A for this very program. This property, however, is strong and does not apply to a large range of analyses. Another practical application is the program written for B can be ported to A without caring about the specificities of the concurrency semantics of A.

Reorderings and critical cycles

The *reorderings* are at the heart of the concurrency semantics for weak memory. Enforcing robustness thus requires reasoning over these objects. A pair of accesses to the shared memory is said to be *reordered* when, at the time of evaluation of the second access, it is not guaranteed that the first access and its consequences were made visible to the other threads.

A reordering of two accesses is not expected under SC; it does not mean that it is necessarily impacting the semantics of the program. Actually, processors implement weak memory consistency for performance reasons. The reorderings improve the execution time and, in some of the cases (e.g. in datarace-free programs), do not impact at all the semantics of the program. Preventing all these reorderings would thus be counter-productive as it would slow down the execution (and the mechanisms used for preventing them, the fences, are expensive instructions, as we will discuss in

Chap. 5). The "improved SC" (defined in Chap. 2), that most of the techniques try to restore, does not prevent all the reorderings—it just banishes those whose effects cannot be modelled with interleavings.

Reorderings that may impact the correctness of a program are those involved in *critical cycles*, introduced by Shasha and Snir [SS88]. These objects are relating some partial orders amongst shared memory accesses between interfering threads. They explain why a given execution is observable on one architecture and not on another. The critical cycles are the objects that we manipulate in the rest of this chapter to define, check and fix robustness and program transformations for simulating a weaker architecture. Finding all the critical cycles in a program for an architecture allows us to address these two questions.

Plan of the chapter

In this chapter, we introduce a static graph, namely the *abstract event graph* (AEG), that abstracts the accesses to shared memory and the communications between them. We describe how to construct this graph following the control-flow graph (CFG) of the input program, and we show that it captures all the executions modelled by Alglave's framework. We discuss some engineering strategies that played an important role in our implementation so that the technique would scale to programs of reasonable size[1]. We then explain how to collect the static, critical cycles that lie in the AEG, and estimate the worst case scenario (and AEG) for our search in the graph. We conclude this chapter with a discussion on the semantics applied in several static analyses taking into account weak memory consistency, and some alternative strategies—and their limits—for enumerating critical cycles.

3.1 Static over-approximation of the control-flow graph

We want to find all the critical cycles that are relevant to the problem we want to address. Given a program, we detect these cycles statically. This allows us to avoid enumeration of all the traces, which would be an obstacle to scalability. It also gives us all the critical cycles at once. In Chap. 5 this allows us, for example, to find all

[1]As we mentioned in Chap. 1, it is hard to give a measure of the programs that can be analysed, since it mostly depends on the degree of concurrency in it—with, e.g., the number of communications between threads. Our average experiments considered programs between 1000 and 10000 lines of code.

instruction	semantics	relevant to WM
GOTO	branching (potentially with guard)	yes
ASSUME	assumption	yes[3]
ASSERT	assertion	yes[3]
SKIP	skip	no
START_THREAD	spawns an asynchronous thread	yes
END_THREAD	end of the thread[4]	yes
LOCATION	equivalent to skip	no
END_FUNCTION	end of the function	yes
ATOMIC_BEGIN	atomic section	yes
ATOMIC_END	end of atomic section	yes
RETURN	return of a function	yes
ASSIGN	assignment	yes
DECL	declaration of a local variable	no
DEAD	marks the end-of-live of a local variable	no
FUNCTION_CALL	call to a function	yes
THROW	throws an exception	no
CATCH	catches an exception	no
OTHER	other instruction that could not be modelled with those above	yes[5]

Figure 3.1: Instructions of goto-programs and their semantics.

the fences preventing two cycles lying in two executions in one single step, instead of examining the two executions separately.

To analyse a C program, e.g. at the top of Fig. 3.2, we convert it with the help of goto-cc into a *goto-program* (bottom of Fig. 3.2), the internal representation of the CProver framework[2]. This internal representation comprises 18 instructions, allowing us to define over it[3] the semantics constructing the AEG. We list them in Fig. 3.1.

The C program in Fig. 3.2 features two threads that can interfere. The first thread writes the argument "input" into x, then randomly writes 1 in y or reads z then writes 1 into x. The second thread successively reads[4] y, z and x. In the corresponding goto-program (below in Fig. 3.2), the **if-else** structure has been turned into a read of the guard of the **if** followed by a goto construction.

[2]`http://www.cprover.org/goto-cc`

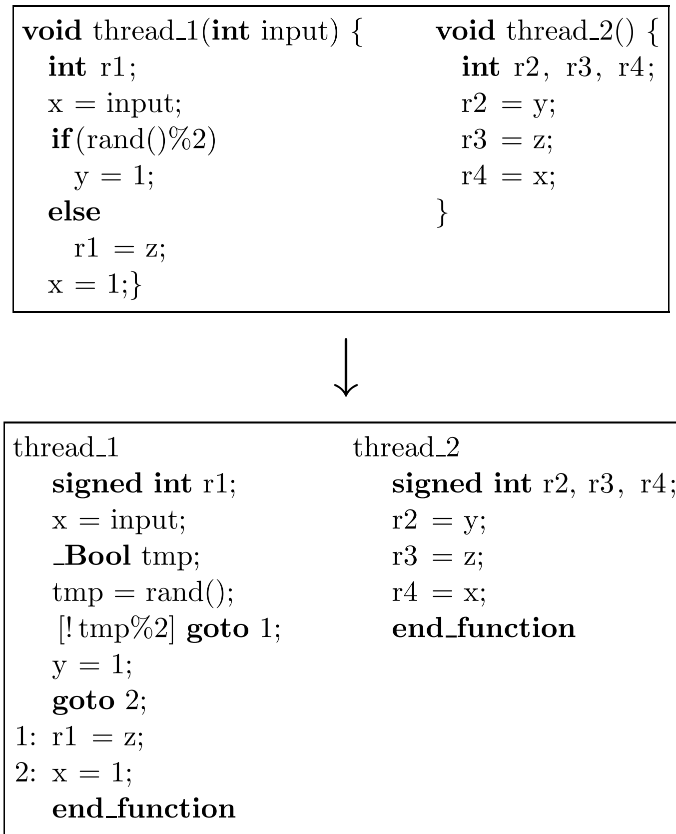
[3]Borrowed from the code of `src/goto-programs/goto_program_template.h`.

[3]Assume and assert are only relevant if we evaluate their conditions in the analysis—which we do not do for the AEG.

[4]This instruction is equivalent to end_of_function.

[5]This instruction is actually used in CProver for modelling fences or compare-and-swap.

[4]Note that the accesses to z in this program will always return the initial value. This is intended: in Fig. 3.3, we will construct an abstraction of the relations between the events. Having an "unrelated" z simplifies the graph.



```
void thread_1(int input) {         void thread_2() {
  int r1;                            int r2, r3, r4;
  x = input;                         r2 = y;
  if(rand()%2)                       r3 = z;
    y = 1;                           r4 = x;
  else                             }
    r1 = z;
  x = 1;}
```

↓

```
thread_1                        thread_2
    signed int r1;                  signed int r2, r3, r4;
    x = input;                      r2 = y;
    _Bool tmp;                      r3 = z;
    tmp = rand();                   r4 = x;
    [!tmp%2] goto 1;                end_function
    y = 1;
    goto 2;
1:  r1 = z;
2:  x = 1;
    end_function
```

Figure 3.2: A C program and its goto-program below.

Given a goto-program, e.g. at the bottom of Fig. 3.2, we then compute an *abstract event graph* (AEG) from the accesses to shared memory (as displayed in Fig. 3.3(a)), e.g. in Fig. 3.3(b). We will explain how to explore the AEG to find the potential critical cycles in the next section.

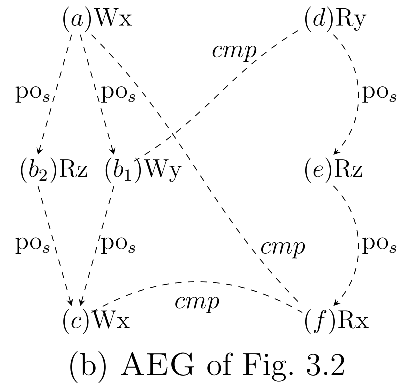
We now explain how to build an AEG from a goto-program. In Fig. 3.3(b), the events a, b_1, b_2 and c (respectively d, e and f) correspond to thread_1 (respectively thread_2) in Fig. 3.2. We only consider accesses to shared variables, and ignore the local variables.

An AEG represents all the executions of a program. Fig. 3.3(c) and (d) give two executions associated with the same AEG, displayed in Fig. 3.3(b).

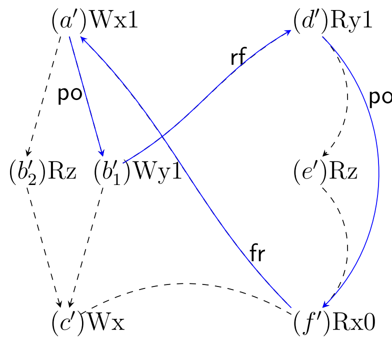
In particular, in an AEG the events do not have values, whereas the executions have concrete values. Also, an AEG merely indicates if two accesses to the same variable could form a data race (see the *cmp* relation in Fig. 3.3(a), which is a symmetric relation), whereas an execution has oriented relations (e.g. which is the write that a read takes its value from, see e.g. the rf arrow in Fig. 3.3(b) and (c)).

```
thread_1                     thread_2
    signed int r1;               signed int r2,
    x = input;                     r3, r4;
    _Bool tmp;                   r2 = y;
    tmp = rand();                r3 = z;
    [!tmp%2] goto 1;             r4 = x;
    y = 1;                       end_function
    goto 2;
1:  r1 = z;
2:  x = 1;
    end_function
```

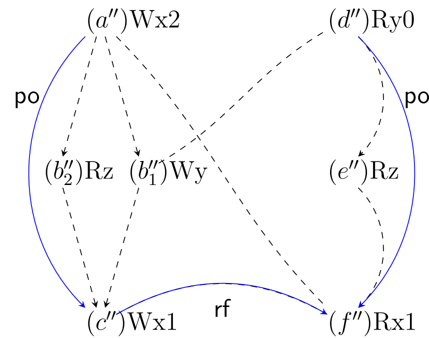
(a) accesses to shared memory in Fig. 3.2



(b) AEG of Fig. 3.2



(c) an execution with a critical cycle



(d) an execution without critical cycle

Figure 3.3: The AEG of Fig. 3.2 and two executions corresponding to it.

3.1.1 Semantics and Abstraction

Alglave in [Alg10] works directly from event structures, which can be seen as partly resolved programs. Addresses, values, control flows, threads and loops must indeed be resolved—only concurrency scheduling remains unresolved in the event structures. In this chapter, we introduce a structure, namely the AEG, which abstracts all the executions valid for a program under a given architecture. We construct this AEG directly from the goto-program, which summarises a significant segment of the C semantics, but we use Alglave's model to show that it indeed captures all the possible executions. As depicted in Fig. 3.4, we want to show that α_{AEG} is a sound abstraction, i.e., that the resulting AEG captures all the executions. We do not have $S^?$, the semantics which formally generates the event structures out of goto-programs or C programs. Its intuition is, however, very clear: we resolve the addresses, controls, values and loops, and leave the communications between threads unresolved (we compare this staged evaluation to other strategies inside a lattice of abstractions for weak memory in Sec. 3.3 and in particular in Fig. 2.1). In order to show that the AEG captures all the executions, we therefore assume a set of three

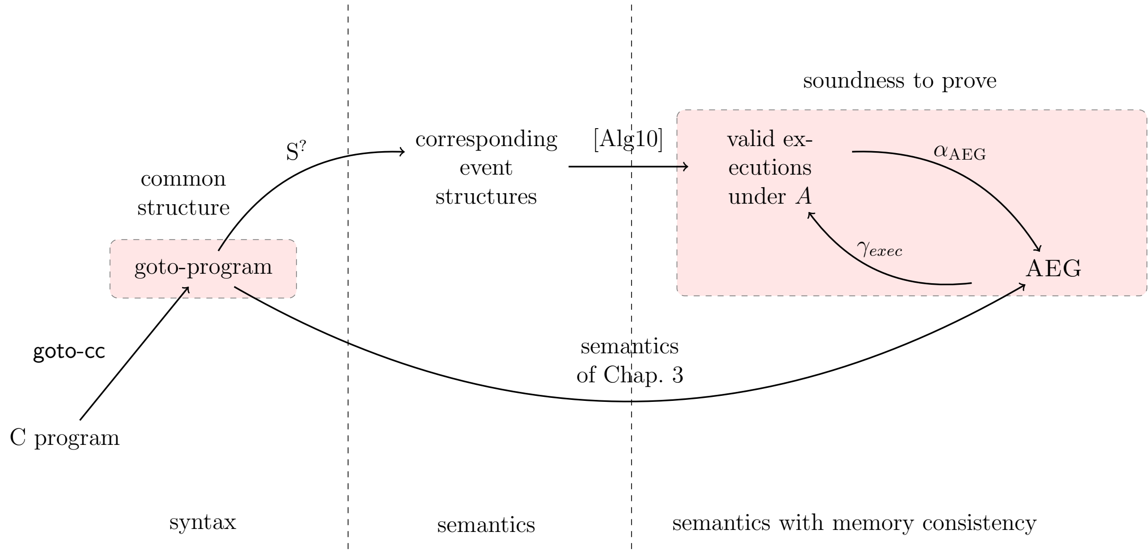


Figure 3.4: From C programs to AEGs and valid executions

rules that the semantics $\mathrm{S}^?$ connecting C to event structures should meet, in App. B.

3.1.2 Constructing AEGs

Given a goto-program, we build an AEG $\triangleq (\mathbb{E}_s, \mathrm{po}_s, cmp)$ as follows (e.g. the one in Fig. 3.3(b)): the *abstract events* $\mathbb{E}_s$, the *static program order* po_s and the *competing pairs cmp* using the transformers of Fig. 3.6 to Fig. 3.13.

An abstract event represents the set of events with same program point, memory location and direction (write or read) during an execution. In Fig. 3.3(b), (a)Wx abstracts the concrete events (a')Wx1 and (a'')Wx2 in the two executions depicted in Fig. 3.3(c) and (d). We will not try to evaluate the values read or written by these accesses, since our objective is to finally reason over static, critical cycles regardless of the executions that may support them. If we wanted to, we would need to have a (a priori operational) trace semantics that determines the values taken all along the execution. This semantics would, however, need to handle weak memory, as some values might depend on previously existing weak memory behaviours. In Fig. 3.5 the store-buffering relaxation SB2 can only happen if the store-buffering relaxation SB1 is captured. Otherwise, we unsoundly omit it. The use of a trace semantics in addition to the axiomatic semantics would also be fairly inefficient. We will discuss this last point in Sec. 3.3 and compare our approach to existing trace-based approaches.

```
#include <pthread.h>

volatile int x = 0, y = 0;

void* thread_0 (void* arg) {
  x = 1;
  *((int*) arg) = y;
  return 0;
}

void* thread_1 (void* arg) {
  y = 1;
  *((int*) arg) = x;
  return 0;
}

int main() {
  pthread_t pt;
  int r1 = 0, r2 = 0;

  /* SB1 */
  pthread_create(&pt, 0, thread_0, (void*) &r1);
  thread_1( (void*) &r2);
  pthread_join(pt, 0);

  if(r1==0 && r2==0) {
    /* SB2 */
    pthread_create(&pt, 0, thread_0, (void*) &r1);
    thread_1( (void*) &r2);
    pthread_join(pt, 0);
  }

  return 0;
}
```

Figure 3.5: A weak memory behaviour (SB2) only observable if another weak memory behaviour (SB1) is observed.

The static program order po_s statically represents the program order po. It abstracts all the (dynamic) po edges that connect two events in program order and that cannot be decomposed as a succession of po edges in this execution. We write po_s^+ (resp. po_s^*) for the transitive (resp. reflexive transitive) closure of this relation.

In Fig. 3.3(c), $(a')\text{Wx1}\xrightarrow{\mathsf{po}}(b1')\text{Wy1}$ is a po edge which is abstracted by the $(a)\text{Wx}\xrightarrow{\text{po}_s}(b1)\text{Wy}$ in the AEG in Fig. 3.2(b). $(d')\text{Ry1}\xrightarrow{\mathsf{po}}(f')\text{Rx0}$ in Fig. 3.3(c) cannot be directly abstracted by a po_s, because in this execution, $(d')\text{Ry1}\xrightarrow{\mathsf{po}}(f')\text{Rx0}$ can be decomposed into $(d')\text{Ry1}\xrightarrow{\mathsf{po}}(e')\text{Rz0}$ and $(e')\text{Rz0}\xrightarrow{\mathsf{po}}(f')\text{Rx0}$. It, however, appears in po_s^+, as $(d')\text{Ry1}\xrightarrow{\mathsf{po}}(e')\text{Rz0}$ is in $(d)\text{Ry}\xrightarrow{\text{po}_s}(e)\text{Rz}$ and $(e')\text{Rz0}\xrightarrow{\mathsf{po}}(f')\text{Rx0}$ is in $(e)\text{Rz}\xrightarrow{\text{po}_s}(f)\text{Rx}$.

The competing pairs *cmp* over-approximate the external communications $\mathsf{wse}\cup\mathsf{rfe}\cup\mathsf{fre}$ needed to define a (dynamic) execution. In Fig. 3.3(b), the two *cmp* edges (a, f) and (b_1, d) abstract in particular the fre and rfe in Fig. 3.3(c). We do not need to represent internal communications, already covered by po_s^+.

This construction is similar to the usual first steps of static data race detections (see [KSKZ09, Sec. 5]), where shared variables involved in write-read or write-write communications between threads are collected. As further work, we could reduce the set of competing pairs using a higher-level synchronisation analysis, as in e.g.,

[SFW+05]: if we assume the correctness of locks, for example, some threads might never interfere, and communication between these is thus spurious.

To build the AEG we define a semantics of goto-programs in terms of abstract events, static program order and competing pairs. We write $\tau[i]$ to represent the semantics of a goto-instruction i. Other notations, e.g., follow(f) or body(f), are explained in Sec. 3.1.2.

We do not compute the values of our variables, and thus do not interpret the expressions. This explains why `assert` and `assume` instructions are a priori irrelevant to us for the transformation. They might, however, be useful to discard spurious cycles and alternative problems, as we will discuss in Chap. 5. In Fig. 3.3(b), (a)Wx represents the assignment "x = input" on thread 1 in Fig. 3.2 (since "input" is a local variable). This abstracts the values that "input" could hold, e.g. 1 (see (a')Wx1 in Fig. 3.3(c) or 2 (see (a'')Wx2 in Fig. 3.3(d)). Prior to building the AEG, we copy expressions of conditions or function arguments into local variables. Thus all the work over shared variables is handled in the assignment case.

We now present the construction of the AEG starting with the intra-thread instructions (e.g. assignments, function calls), creating po_s edges, then the thread constructor, creating *cmp* edges.

Building po_s

Assignments `lhs=rhs` We decompose this statement into sets of abstract events: the reads from shared variables in `rhs` and `lhs`, denoted by evts(`rhs`) and evts(`lhs`), and the writes to the potential target objects trg(`lhs`) (determined from `lhs` as explained below). We assume that in conditionals, variables are evaluated from left to right; the order of evaluations of the variables for the other expressions depends on the compiler implementation.

In our implementation of the AEG constructor, we chose to ignore critical cycles that would involve two reads from the same expression—that is, critical cycles that are compiler-implementation dependent. We connect all the reads of `rhs` and all the reads of `lhs` except trg(`lhs`) to the incoming po_s. We then connect each of them to the potential target writes to trg(`lhs`), as depicted in Fig. 3.6.

If we still want to consider the compiler-implementation dependent critical cycles, we can transform the program to have at most one read to the shared memory per assignment. Another strategy can consist of constructing all the possible po_s orders between the reads. To avoid the factorial number of orders to construct, we can fix

an arbitrary order for the reads and construct a po_s backedge from the last read to the first read[5]. Because our cycle detection can follow the transitive closure of po_s (po_s^+)—i.e., it can "jump" over some events—and will not need more than two events per thread in a cycle, any combination involved in a cycle will be captured. We can also apply the unwinding strategy described later in Sec. 3.1.3.

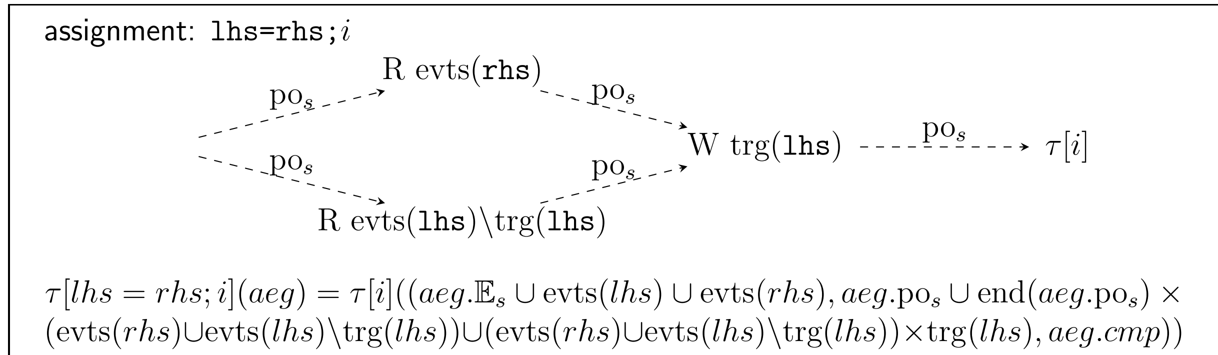


Figure 3.6: AEG construction for assignment.

This is the set of objects (in the C sense [c1111]) that could be written to, according to our pointer analysis. They are either fields of structures, structures, arrays (independent of their offsets) or variables. If we have e.g. `*(&t+y+r)=z+3` (where `t`, `y` and `z` are shared), `t` is our trg variable, and we obtain $(R\texttt{y}, W\texttt{t}) \in \text{po}_s$ and $(R\texttt{z}, W\texttt{t}) \in \text{po}_s$.

We also maintain a map from local to shared variables, to record the dependencies between abstract events. For instance, if we have `int r1=x; int r2=r1; *(&y+r2)=1` we know that the Rx from the rhs of the first instruction is in dependency with `r2` (via the use of `r1`). Moreover, the Wy issued by the last statement also depends on `r2`.

Procedure calls f() We build the po_s corresponding to the function's body (written body(f)) once as in Fig. 3.7. We then replace the call to a function f() by its body. This ensures a better precision, in the sense that a function can be fenced in a given context and unfenced in another. We discuss how we handle (possibly mutually) recursive calls in Sec. 3.1.3.

Guarded statement We do not keep track of the environments (i.e., the values of the variables at each program point). Thus we cannot evaluate the expression of the guard of a statement. Hence, we abstract the guard and make this statement non-deterministically reachable, by adding a second po_s edge by-passing the statement, as depicted in Fig. 3.8.

[5]This solution was originally suggested by Daniel Poetzl.

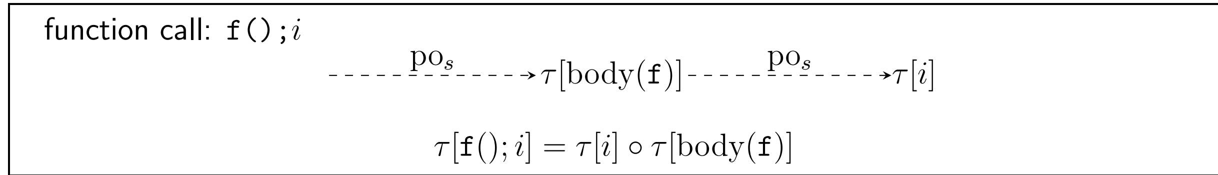


Figure 3.7: AEG construction for function call.

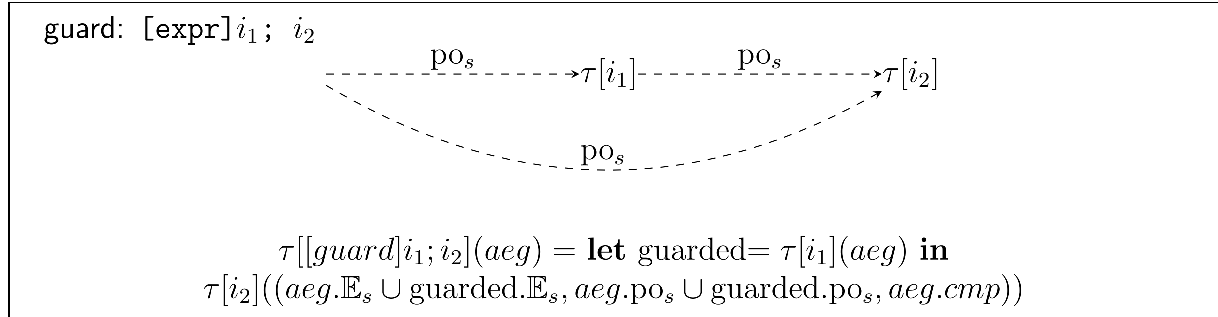


Figure 3.8: AEG construction for guarded statements.

Forward jump to a label L We connect the previous abstract events to the next abstract events that we generate from the program point L. In Fig. 3.9, we write follow(L) for the sequence of statements following the label L.

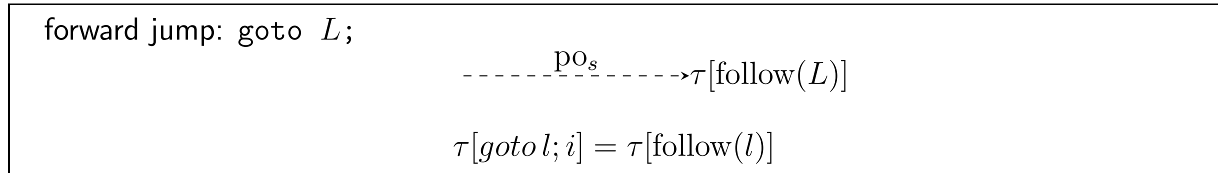


Figure 3.9: AEG construction for forward jump.

Backward jump (unbounded loops), i.e., a jump to a label already visited. We connect the last abstract event of the copy to the first abstract event of the original body with a po_s edge, as shown in Fig. 3.10.

Atomic sections are special goto-instructions, for modelling idealised atomic sections without having to rely on the correctness of their implementation. These are used in many theoretical concurrency and verification works. For example, we used them for copying data to atomic structures, e.g., the implementation of the Chase-Lev queue [CL05], or for implementing compare-and-swaps, e.g., the implementation of Michael and Scott's queue [MS96]. There is no consensus on the semantics of atomic sections in the context of weak memory models—some papers like [AKT13] authorise e.g. reorderings of events on the same threads across an atomic begin or end statement, whereas the experiments of e.g., [LNP+12] would suggest an atomic begin

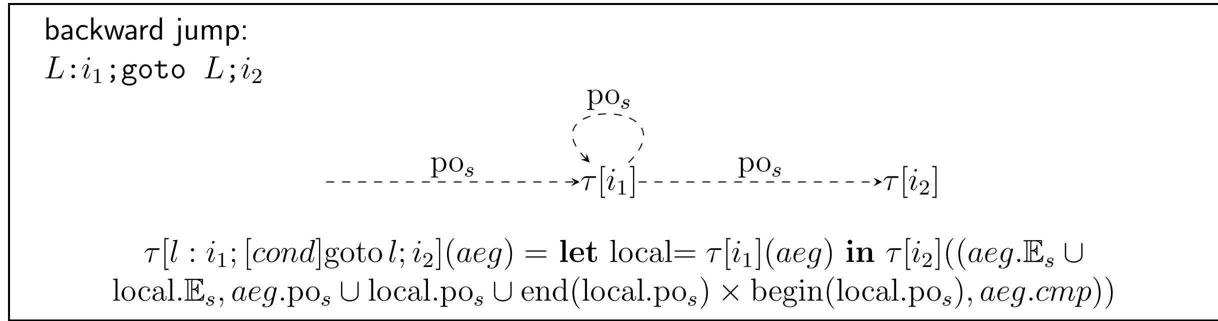


Figure 3.10: AEG construction for backward jump.

or end statement that would prevent any reordering of events across. We decide to assume that reorderings cannot happen across an atomic section, thus we place two full fences f right after the beginning of the section and just before the end of section, as e.g. in Fig. 3.11.

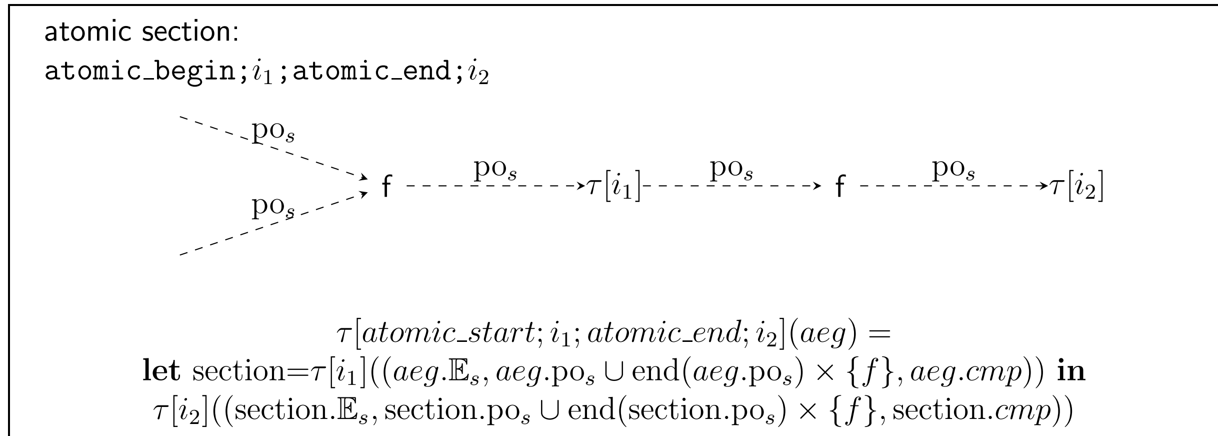


Figure 3.11: AEG construction for atomic.

Construction of cmp

During the construction of po_s, we also compute the competing pairs that abstract external communications, i.e., between two distinct threads during an execution. For each write w to a memory location x, we augment the *cmp* relation with pairs made of this write and any write to x from an interfering thread: this abstracts the coherence wse.

Similarly, we augment *cmp* with pairs made of w and any read of x from an interfering thread. Symmetrically, for each read r of y, we add pairs made of r and any write to y from an interfering thread to *cmp*. This abstracts the from-read fre and read-from rfe. The equation in Fig. 3.12 formalises the computation of these

competing pairs after encountering a new_thread instruction. We write in the equation $A \otimes B$ for $\text{sym}(\text{writes}(A) \times \text{writes}(B) \cup \text{writes}(A) \times \text{reads}(B) \cup \text{reads}(A) \times \text{writes}(B))$, with $\text{sym}(R) \triangleq R \cup \{(x, y) \mid (y, x) \in R\}$ and $\bar{\emptyset} = (\emptyset, \emptyset, \emptyset)$.

When we construct the AEG for a new_thread instruction, we do not connect the events preceding the creation of the thread to the events yielded by the new thread by po_s. This is not necessary since all the events of this new thread will be totally ordered with the events preceding the creation of this thread. The direct consequence is that it is impossible to form a cycle involving one event e preceding the creation of the thread and one event e' of this thread: any order attempting to connect back to e would indeed violate the imposed order between e and e'.

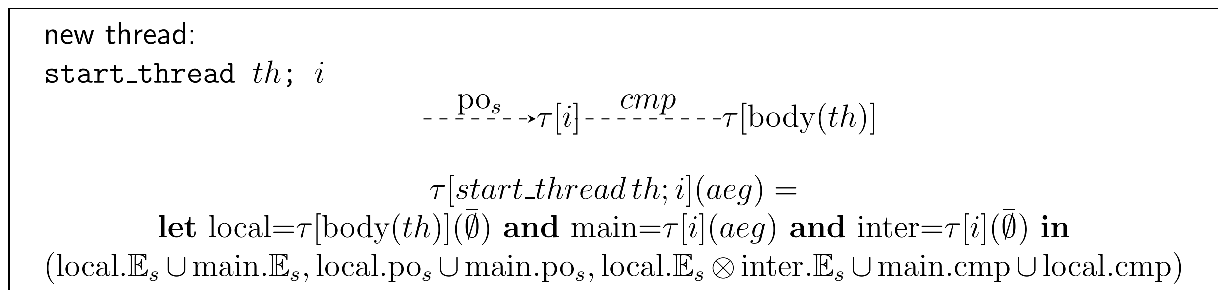


Figure 3.12: cmp construction for start_thread.

This strategy for computing the interferences between threads is sound, provided the pointers to functions in the calls could be correctly resolved. It is however an over-approximation: the formulation of Fig. 3.12 ignores the possibility that the function spawning a new thread might wait at some program point until the thread terminates—for example with the function pthread_join with the POSIX threads. Our analysis would ignore the join and suppose that this thread might interfere with threads that could be spawned after the join. If we can statically determine a join of the spawned thread, we can limit the interferences between threads by computing only *cmp* between the creation and join of this thread, as formulated in Fig. 3.13.

Other high-level synchronisation constructions such as locks and mutexes can also limit the interferences between threads. Applying a synchronisation analysis prior to our construction would also reduce the amount of over-approximation of thread interference. We discuss this point in Fig. 6 as a partial solution to reduce the approximation and improve the scalability of the approach.

We describe in detail in Sec. B.2 in App. B some ideas towards a formal argument of the soundness of this construction.

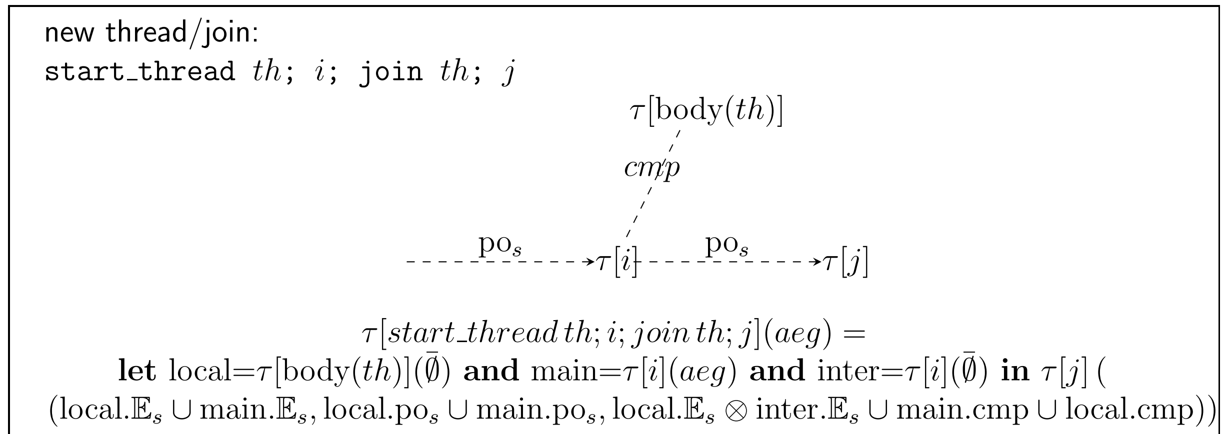


Figure 3.13: cmp construction for start_thread and join.

3.1.3 Strategy for loops and recursions

(Unbounded) Loops and arrays

We explain how to deal with loops statically. If we build our AEG directly following the CFG, with a po_s back-edge connecting the end of the body to its entry, we already handle most of the cases. The only missing cases are those where two events are issued by the same instructions—we will address them afterwards.

Recall from Chap. 2 that in a critical cycle (2.i) there are two events per thread, and (2.ii) two events on the same thread target two different locations[6]. Let us analyse the cases.

The first case is an iteration i of this loop on which a critical cycle connects two events (a_i) and (b_i). The critical cycle will be trivially captured by its static counterpart that abstracts in particular these events with abstract events (a) and (b).

Now, for a given execution, if a critical cycle connects the event (a_i) of an iteration i to the event (b_j) of a later iteration j (i.e., $i \leq j$), then these events are abstracted respectively by (a) and (b) in the AEG. As we do not evaluate the expressions, we abstracted the loop guard and any local variable that would vary across the iterations. Thus, all the iterations can be statically captured by one abstract representation of the body of the loop. Then, thanks to the po_s back-edge and the transitivity of our cycle search, any critical cycle involving (a_i) and (b_j) is abstracted by a static critical cycle relating (a) and (b), even though (b) might be before (a) in the body of the loop.

[6]Litmus tests involving more events per thread contain actually several critical cycles. In this static detection, we treat these cycles separately. In the instrumentation of Chap. 4, however, the instrumentations of several distinct cycles can compose to answer cases where a write buffer of size N is required, for example. We explain such cases in Sec. 4.2.3.

The only case that is not handled by this approach is when (a_i) and (b_j) are abstracted by the same abstract event, say (c). As the variables addressed by the events on the same thread of a cycle need to be different, this case can only occur when (a_i) and (b_j) are accessing an array or a pointer whose index or offset depends on the iteration. We do not evaluate these offsets or indices, which implies that two accesses to two distinct array positions might be abstracted by the same abstract event (c).

In order to detect such critical cycles, we copy the body of the loop and do not add a po_s back-edge. Hence, a static critical cycle will connect (c) in the first instance of the body and (c) in the second instance of the body to abstract the critical cycle involving (a_i) and (b_j). The back-edge is no longer necessary, as the abstract events reachable through this back-edge are replicated in the second body. Thus, all the previous cases are also covered.

We have implemented the duplication of the loop bodies only for loops that contain accesses to arrays. In case of nested loops, we ensure that we duplicate each of the sub-bodies only once in order to avoid an exponential explosion. Let us suppose that we have a loop l_1 of $\#l_1$ events, that contains a loop l_2 of $\#l_2$ events, containing itself a loop l_3 and so on until a nth loop l_n. The duplication of this loop with nested duplications could yield up to $2\,(2^n - 1)\max_{\Delta\#}\{l_1, ..., l_n\}$, where $\max_{\Delta\#}$ is the maximum difference of cardinalities of successive loops, whereas duplicating each of the loops in isolation once generates a maximum of $\frac{n(n+3)}{2}\max_{\Delta\#}\{l_1, ..., l_n\}$ events[7]. This approach is sufficient owing to the maximum of two events per thread in a critical cycle and the transitivity of `po`.

In Fig. 3.15, we have several AEGs for the thread 1 an implementation of Dekker's mutual exclusion algorithm [Dij65] in Fig. 3.14. In Fig. 3.15(a), the AEG is the one we would obtain if we had constructed it with the transformers of Sec. 3.1.2, with 8 events and some po_s back-edges. "(2) Rfg2" is for instance the read from `flag2` in the first while-loop of thread 1 in the implementation in Fig. 3.14.

In Fig. 3.15(b), we applied the duplication of loops' events and get 34 events. The event "(2) Rfg2" is duplicated 8 times due to nested loops: "(2_1) Rfg2" to "(2_8) Rfg2".

If we reduce the duplications to only those that are necessary, as we discussed in the previous paragraph, we obtain the AEG in Fig. 3.15(c), which contains 23 events. "(2) Rfg2" is only duplicated 4 times.

[7] Details about how to compute these numbers can be found in App. D.

```
int flag1 = 0; int flag2 = 0; int turn = 0;

void* thr1(void* arg) {
  while(1) {
    flag1 = 1;
    while(flag2 >= 1) {
      if(turn != 0) {
        flag1 = 0;
        while(turn != 0) {};
        flag1 = 1;
      }
    }
    // begin of critical section
    // end of critical section
    turn = 1;
    flag1 = 0;
  }
}

void* thr2(void * arg) {
  while(1) {
    flag2 = 1;
    while (flag1 >= 1) {
      if (turn != 1) {
        flag2 = 0;
        while (turn != 1) {};
        flag2 = 1;
      }
    }
    // begin of critical section
    // end of critical section
    turn = 1;
    flag2 = 0;
  }
}
```

Figure 3.14: An implementation of Dekker's mutual exclusion algorithm [Dij65].

(Mutually) Recursive functions

In the case of recursive calls, we can connect the abstract events of the function so that any cycle appearing in a sequence of dynamic calls can appear in the abstract event graph as a cycle. The strategy is similar to the method used for unbounded loops. In this construction, we call *last events of the body* of a function the events that are preceding any `return` in the function. The *first events of the body* of a function are the first events of the subgraph built out of the function.

1. The function calls are first "unfolded twice": we explore the body of the recursive function with a depth of 2, as if we were inlining them twice;

2. Inside the body, we construct the abstract event graph as before;

3. At the recursive callsite, we connect by po_s the last abstract events of the first function body before the call to the first abstract events of the second body, and the last abstract events of the second body to the first abstract events after the call in the first body. This models the simplest dynamic call case, in which an event of the caller and an event of the callee could both belong to a critical cycle;

4. Then we connect by po_s the last abstract events before the recursive call in the second body to the first abstract events in the first body, and we connect

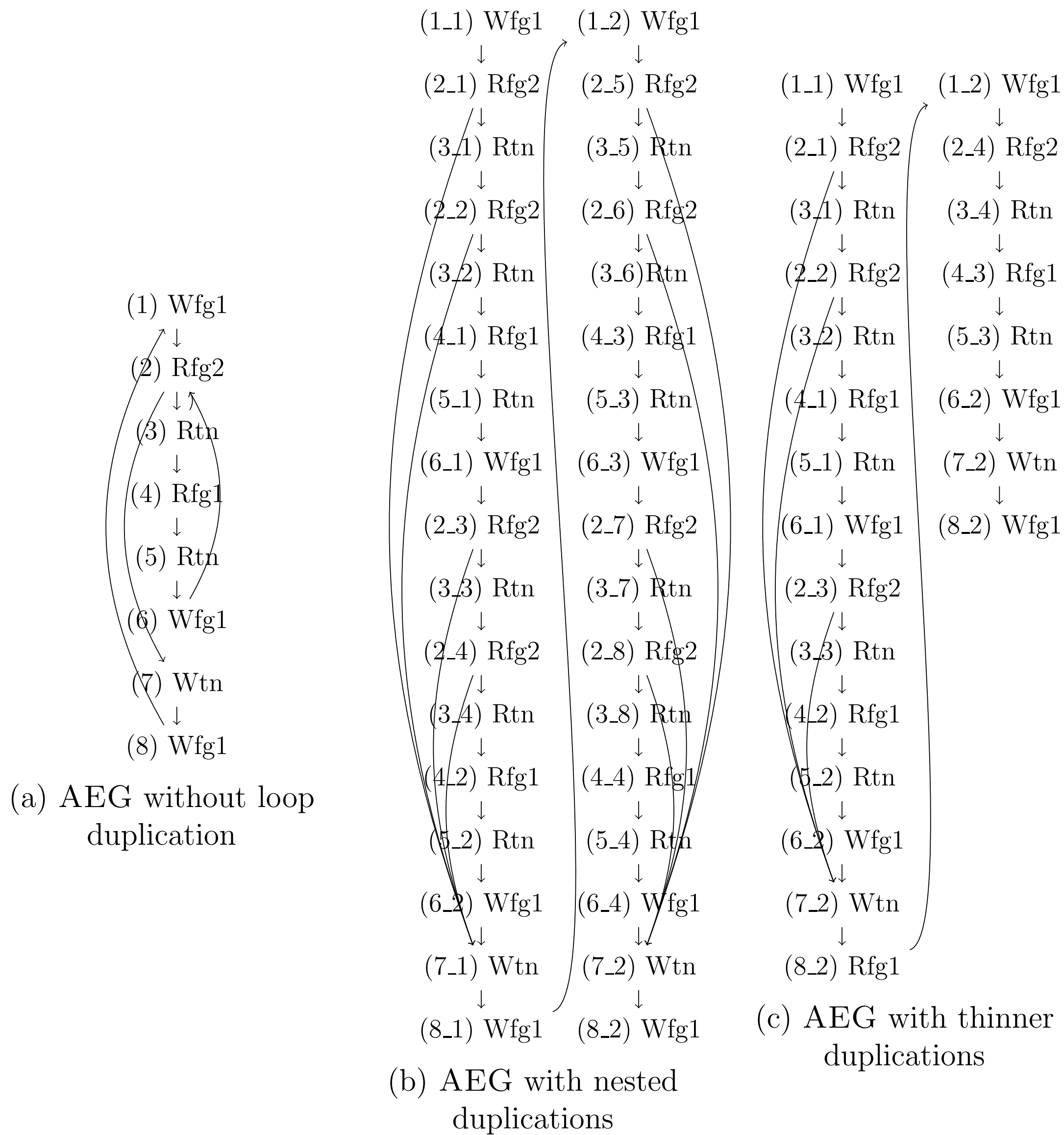


Figure 3.15: Three different sub-AEGs of the first thread for the implementation of Dekker's mutual exclusion algorithm in Fig. 3.14.

the last abstract events of the first body to the first abstract events after the recursive call in the second body. This models the recursive calls, in which one might encounter a critical cycle involving twice the same static abstract event.

We construct the *cmp* during this process as before. Any cycle occurring for a sequence of dynamic function calls is a cycle in this subgraph. To extend this to mutually recursive functions, we take the *period* in the sequence of recursive calls to the first function, that is, the smallest sequence of calls that is repeated, then duplicate it and apply the same transformations, as depicted in Fig. 3.16. In Fig. 3.16, each of the vertical line is the body of a function. The circles are abstract events, the squares are call-sites, the dashed lines represent a potential function call. The coloured edges are the po_s edges we construct: in green, the edges built in step 2; in red, the edges added in step 3; in blue, the edges described in step 4. Note that the absence of a green edge over a recursive callsite would mean that this function would trivially not terminate if the callsite is reachable.

Note that this method has not been implemented yet in the tools goto-instrument and musketeer

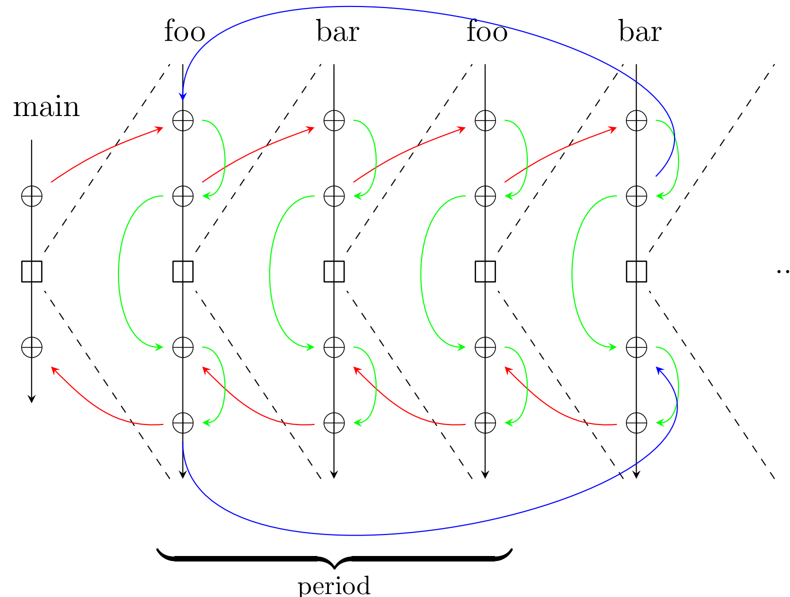
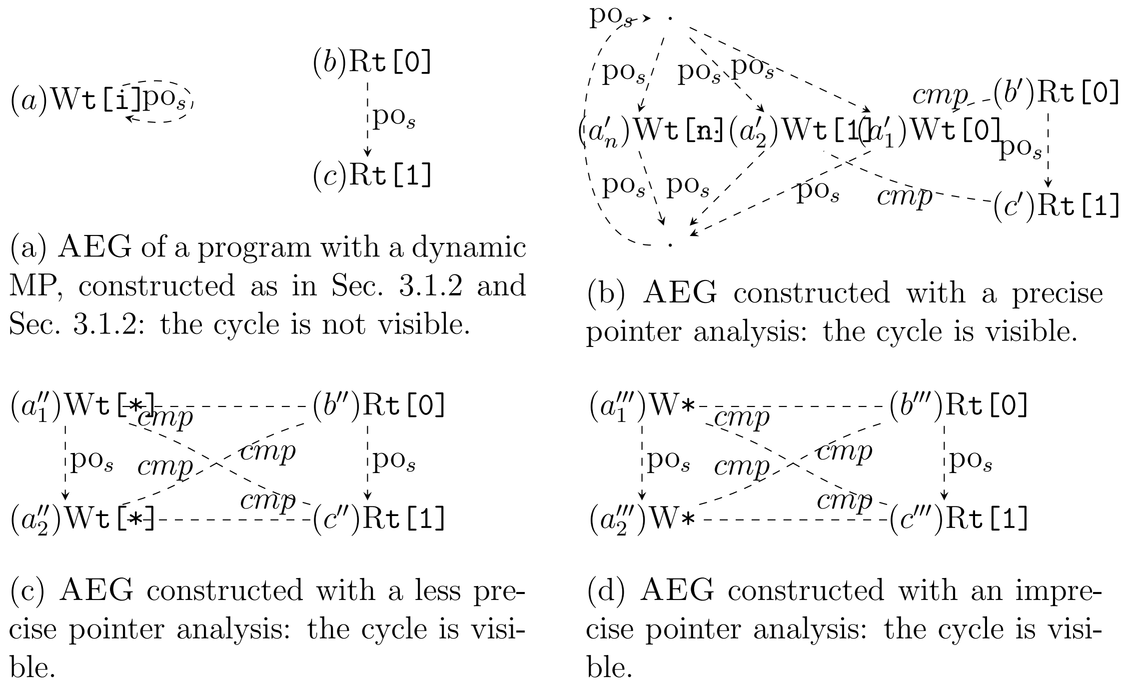


Figure 3.16: Construction of po_s for mutual recursive functions.



(a) AEG of a program with a dynamic MP, constructed as in Sec. 3.1.2 and Sec. 3.1.2: the cycle is not visible.

(b) AEG constructed with a precise pointer analysis: the cycle is visible.

(c) AEG constructed with a less precise pointer analysis: the cycle is visible.

(d) AEG constructed with an imprecise pointer analysis: the cycle is visible.

Figure 3.17: Dynamic cycles.

3.1.4 Engineering improvements regarding the analysis

Interplay between cycles and pointer analyses

We want to find all critical cycles in the static AEG. Yet, some cycles might reveal themselves only dynamically, e.g., across several iterations of a same loop. In Fig. 3.17(a), the AEG on top comes from a two-threads program, where the first thread loops and writes to a shared array `t` and the second thread reads from this array.

This program could exhibit a *message passing* pattern, where two writes of the loop could be reordered. If we build our AEG as in Sec. 3.1.2, we only build one abstract event for all writes of the loop—which is unsufficient if we want to detect a critical cycle involving two writes of this loop, on two iterations that might not even be consecutive. How we address this challenge depends on the precision of our pointer analysis. Recall from Chap. 2 that in a critical cycle (2.i) there are two events per thread, and (2.ii) two events on the same thread target two different locations.

If we have a precise pointer analysis, we insert as many abstract events as necessary for the objects pointed to, as in Fig. 3.17(b). Otherwise, we underspecify the abstract event: in Fig. 3.17(c), we abstract Wt[i] by Wt[*]. We then copy the body of the loop twice, hence the two Wt[*] in po_s. They abstract any two writes to distinct places in `t` that could occur across the iterations, as required by (2.i) and (2.ii) above.

```
#include <stdlib.h>

int x; // == 0, since global

void f() { ++x; }

void g() { x*=2; }

int main()
{
  // declare
  void (**p) ();

  // allocate
  p = malloc(2*sizeof(void (*) (void)));

  // init
  p[0] = f;
  p[1] = g;

  // call
  unsigned i;
  for(i = 0; i<2; ++i) p[i]();

  return x;
}
```

Figure 3.18: Function pointers used in dynamically allocated array.

If the analysis cannot determine the location of an access, we insert an abstract event accessing any shared variable, as in the bottom of Fig. 3.17(d). This event can communicate with any variable accessed in other threads.

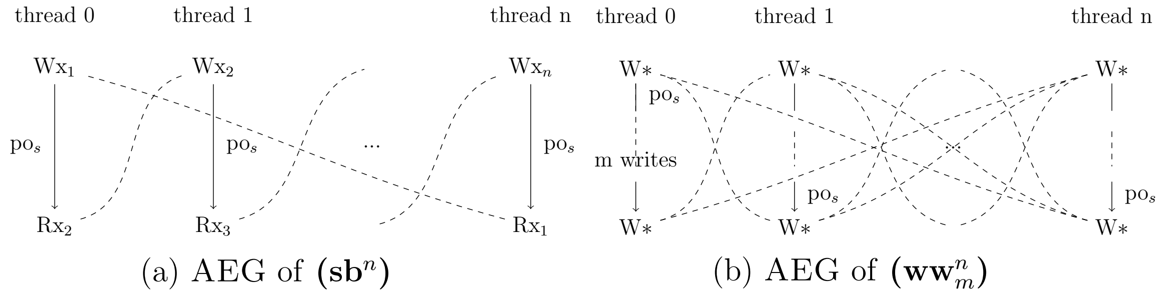
A note on pointers to functions

Goto-programs, like C programs, have pointers to functions. Determining the addresses of those pointers is relatively straightforward for dynamic analysis; it is not the case for static analysis. We must capture all the possible functions those pointers would refer to, and there are three challenging cases that one needs to consider in the static settings:

1. the function pointers passed as arguments in a function call;
2. the function pointers called from the same static call-site but calling different functions depending on the context (iteration, context of the caller);
3. the function pointers stored in dynamically allocated arrays.

The first item is critical for our analysis, since thread libraries like POSIX thread require passing the function executed by the new thread as a pointer to function. Creating a new thread executing the function my_function could, for instance, be achieved with pthread_create(&pthread, &attributes, my_function, &arguments).

Item 2 is connected to the first one: if we form a wrapper around pthread_create, for example, and call this wrapper from two different places with two different functions passed as arguments, we should not replace the pointer to function by one of these in



(a) AEG of $(\mathbf{sb}^n)$ (b) AEG of $(\mathbf{ww}^n_m)$

Figure 3.19: (Parametric) AEGs of $(\mathbf{sb}^n)$ and $(\mathbf{ww}^n_m)$.

the body of the wrapper, since it would instantiate the whole wrapper for only one of the calls.

Item 3 results from the idea of placing function pointers inside structures or arrays, but in dynamically allocated arrays. The program in Fig. 3.18 is an instance of this practice. We do not address this last item, as it requires a very precise pointer analysis applied prior to our analysis that would be able to capture all these dynamic allocations. Placing function pointers in dynamically allocated arrays is nevertheless not a recommended practice in C.

The default strategy initially available in the CProver codebase consists of replacing the call to function pointers by a non-deterministic choice between all the functions whose addresses are used at some point in the code and whose prototypes are compatible with the prototype of the pointer. This strategy solves item 1, since it does not require knowing precisely which function was actually pointed, and item 2, as the non-deterministic choice covers all the potential calls.

In the context of our analysis, this over-approximation can strongly impact the size of the graph, particularly the number of cmp. Indeed, pthread_create takes as argument a function of the generic type **void**∗ (∗) (**void**∗). With the default strategy, any function whose address is taken (and thus any function called for a new thread) can be non-deterministically reached from this call. In practice, if we consider $(\mathbf{sb}^n)$ in Fig. 3.19(a), inlining the functions precisely would yield n cmp connections in the AEG, whereas if we apply the default strategy, we would end up with $2n^3$ cmp. For $(\mathbf{ww}^n_m)$, in Fig. 3.19(b), we should have $o\left(\frac{m^2n^2}{2}\right)$ cmp with a precise strategy whereas removing the pointers with the default strategy would produce $o\left(\frac{m^2n^4}{2}\right)$ cmp connections[8]. Besides the number of spurious cmp connections and the number of spurious nodes in the graph, it also means that threads that should not interfere could appear as interfering in that case.

[8]Details of the computations can be found in App. D.

We implemented two alternative strategies. The first one consists of directly replacing the calls to pthread_create by the function called itself with the marker __CPROVER_ASYNC_0:, which notifies to goto-cc that this function starts in a new, asynchronous thread. To address item 1 correctly, we need to inline[9] all the potential wrappers around the pthread_creates. To address item 2, we need to replicate the instantiated wrappers so that each call with a different function does not call the previous function. This can be hard in the case of dynamic calls. This approach can be combined with the default strategy for the function calls not involving pthread_create, but it requires a certain amount of manual modification of the code in order to be sound.

The second approach, completely automated, performs a propagation of constant function pointers following the call graph. It automatically instantiates the functions called with function pointers passed as arguments. This soundly resolves most of the cases. The default approach can be applied afterwards to ensure that we also handle the non-constant function pointers calls.

3.2 Search strategies in the abstract event graph

In the previous section, we computed the abstract event graph that over-approximates the partial orders relating the possible executions of the program. Dynamic, critical cycles can be found in this structure by looking for their static counterparts, that is, the static critical cycles. In this section, we justify our choice of strategy for critical cycle detection and we do a worst case analysis.

3.2.1 Enumerating the critical cycles in an abstract event graph

Prior to looking for the static, critical cycles in the AEG of a program, we need to set the weak memory property we are trying to compute or restore. Instrumenting a program for weak memory—that is, modifying the code so that weak memory behaviours would be revealed when running the instrumented program under an SC architecture—can be performed incrementally. We may consider one critical cycle at a time, instrument, then address the next one and continue. Restoring SC in an optimal way is, however, not easy to perform incrementally. In this section, in order to embrace most of the problems related to static analyses for weak memory, we assume that we collect all the static, critical cycles from the AEG at once.

[9] Using for instance the C keyword inline.

We enumerate the critical cycles through an explicit search. We implemented a variant of Tarjan's search algorithm [Tar73] adapted to AEG. In addition to the checks for critical cycles rather than simple directed cycles, we also alternate the exploration between po_s and cmp edges, allow transitive jump for po_s and backtrack earlier when some conditions necessary for critical cycles are not holding—we detail those conditions in this section. We justify the choice of explicit search in a short survey of how to enumerate cycles in a graph in Sec. 3.3.

Finding strongly connected components in a graph is linear (complexity in $O(E+V)$ with Tarjan's algorithm [Tar72], where E is the number of edges and V the vertices). Finding whether there is a cycle is also straightforward—this decision procedure lies in the P complexity-class. Enumerating all the (elementary) directed cycles, however, is more difficult. This problem is #P-complete [Val79, 14., p. 417]. The complexity of a search (explicit or not) for directed cycles inside a graph is necessarily greater than the number of cycles. For a complete graph of v vertices, i.e., a graph whose vertices are all bi-connected (for example Fig. 3.20), the number of directed cycles [Joh75] is

$$\sum_{i=1}^{v-1} \binom{v}{v-i+1}(v-i)!, \quad \text{or}^{10} \quad \sum_{j=2}^{v} \binom{v}{j}(j-1)!,$$

which is growing faster than exponential of v ($\sum_{i=0}^{v} \binom{v}{i}$). Using a similar reasoning as in [Cha68], we sum the number of directed cycles per size. Given that the graph is complete, we know that any set of j vertices hosts directed cycles of size j; there are $\binom{v}{j}$ of them. We want to find the directed cycles in each of these sets of vertices. Since, again, all of these vertices are (bi-)connected, each permutation of two vertices in the sequence of vertices of this set gives a new directed cycle; the number of directed cycles going through all these vertices is then the number of permutations $v!$ in the list of vertices. We know, however, that each directed cycle can be represented with v rotations of the sequence of vertices (by shifting each of the vertices on the left/right), therefore we have $v!/v = (v-1)!$ directed cycles in this set. This implies that there are $\sum_{j=2}^{v} \binom{v}{j}(j-1)!$ directed cycles in the complete graph.

Note that, for a given length l, we have $\binom{v}{l}(l-1)!$ directed cycles of size l, which is asymptotically equivalent to v^l/l.

Because of the exponential complexity of any directed cycles enumeration in the generic case, there was almost no research aiming at improving the (exponential) search in 1970 [Tar73]. In 1973 Tarjan proposed a new search in the graph which is

[10] Using invariant $j + i - 1 \equiv v$.

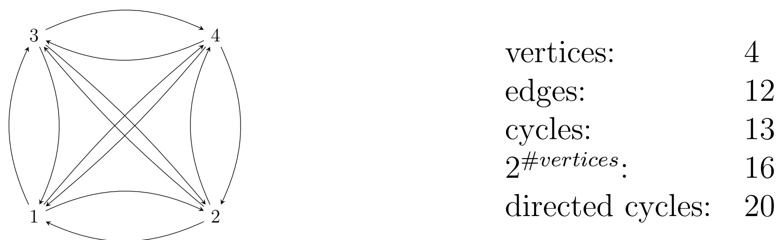


Figure 3.20: Complete directed graph (clique) with 4 vertices.

polynomial in the numbers of edges, vertices and cycles. In practice, this algorithm is useful if we do not have to handle a sup-exponential number of directed cycles—which, as we see in the next section, is our case for abstract event graphs.

Explicit search-based algorithms with backtrack

Paton's, Tiernan's, Weinblatt's and Syslo's algorithms [Pat69, Tie70, Wei72, Sys73], introduced between 1969 and 1973, are DFS with improved backtracking. These algorithms can however be exponential in the number of vertices and in the number of edges [Tar73, LT82]: $O(\exp(E+V)\times(C+1))$. Tarjan's algorithm [Tar73] reduces the time complexity to a polynomial one, namely $O(E\times V\times(C+1))$ in 1973. The idea is to explore and maintain a set of visited states – no need to remember all the directed cycles met so far. Johnson's algorithm [Joh75] improves Tarjan's search and ends up with a complexity of $O((E+V)\times(C+1))$ in 1975. The Szwarcfiter-Lauer algorithm [SL76] finally reduces that time complexity to $O(V+E\times(C+1))$ in 1976. Later, the research mostly focussed on preprocessing *via* graph transformations and efficiency improvements [LT82] rather than complexity improvement.

In our approach, we implemented a variant of Tarjan's algorithm. We backtrack when a critical cycle cannot be formed given the current path— even though it may form a directed cycle in the graph. Most of the properties of critical cycles—like thin-air or the cumulativity with some lightweight fences—can only be checked once we have formed the directed cycle in the search, since they require all the events. The following properties can be applied on-the-fly and help reduce the exploration space:

- there are at most two accesses per thread in a critical cycle [Alg10],
- there are at most two writes and one read per variable in a critical cycle [Alg10],
- a cycle cannot contain only accesses to the same variable (uniproc),

- if a full fence was met, no reordering passing through can happen in the current thread,
- a relevant critical cycle must contain at least one pair unsafe w.r.t. the targeted architecture.

Encoding of the abstract event graph

The graph itself can be encoded with the help of an adjacency matrix, a symbolic representation or adjacency lists (pointers). The techniques surveyed in Sec. 3.3 are not always as easy to implement with some of these structures. We implemented the graph as adjacency lists, for its compact representation and to facilitate the explicit search.

Directed cycles can have edges in po_s^+, which is not represented in the AEG. Either we compute po_s^+ in the graph and add edges (using matrices representation and Warshall algorithm [War62], this operation is cubic in the number of vertices), or we take into account this transitivity in our exploration (thus increasing the cost of this exploration). Exploring a graph where po_s^+ transitions have been added is semantically equivalent to exploring the original graph but allowing some transitive "jump" in the exploration of po_s. The exploration space is the same; practically, instead of backtracking directly, the "algorithm with jumps" will try to skip the last vertex. In the transitive graph, there would be an edge coming from this vertex. The normal algorithm would then backtrack and try this edge, in a similar fashion. In practice, encoding without transitivity has the advantage that a fence in the input program does not need to be replicated in the AEG as many times as we have transitive po_s edges passing through.

3.2.2 Analysis of the complexity in worst case

Actually, we will not have to enumerate an exponential number of directed cycles as we should in the generic case, thanks to the po_s relation, which allows us to provide a thinner expression of the graph and complexities associated with it. We "separate" the v vertices into the number of events per thread, m, and the number of threads, n. We will usually assume in the following that $m >> n$ and compute complexities asymptotic for m.

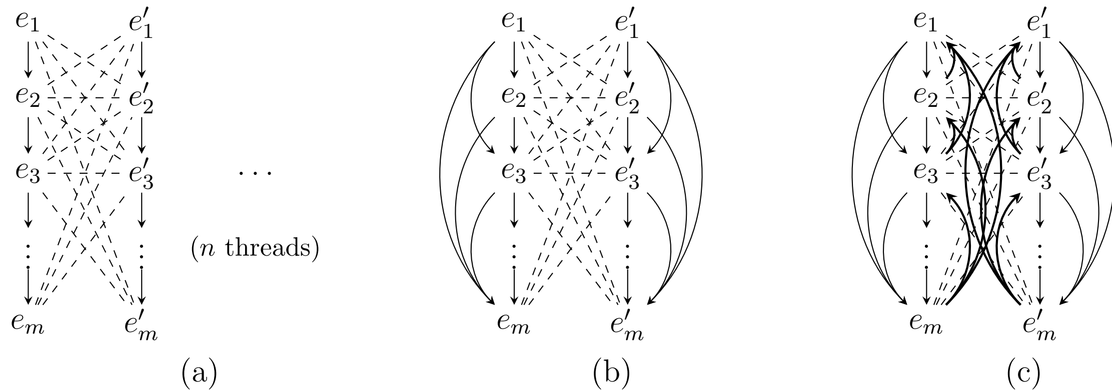


Figure 3.21: Abstract worst case of AEGs.

Worst abstract event graph

To compute the analysis' complexity in the worst case, we need an AEG which is likely to be the most expensive to explore for our analysis. We will consider three examples to bring progressively the transitivity in po_s and the back-edge in po_s (which encodes loops from the code). We will not consider cases with diamonds in po_s (if-then-else), as they do not provide any particular difficulty.

The first case (Fig 3.21.a) is a set of n threads, all with m events. We do not assume a particular architecture. We over-approximate the communications between events in different threads by connecting each event to all the other events in the other threads. For this case, we assume that there is no transitivity for po_s. The second case is the same, but with transitivity (Fig. 3.21.b). The third case includes po_s back-edges in the threads (Fig. 3.21(c)).

Number of directed cycles in the graph

We estimate an upper-bound of the number of directed cycles regardless of the architecture. In the graph 3.21(a), that is, without po_s back-edge and transitivity, given two threads, we can pick one of the $m-1$ po_s edges of the first thread, and link it with any of the $m-1$ po_s edges of the second thread, leading to $(m-1)^2$ possible directed cycles. Now, for n threads, we first pick a combination of i threads, with i between 2 and n, then pick one of the $m-1$ edges in the first thread, and connect it with one of the $m-1$ edges from the second thread, and the same for the next threads. The number of directed cycles in this graph is then

$$\sum_{i=2}^{n} \binom{n}{i} (m-1)^i,$$

which equals $m^n - n \times m + n - 1$ using Newton binomial formula. The number[11]is then a $o(m^n)$.

With transitivity in graph 3.21(b), after having picked the threads, instead of picking one edge, that is, one concrete po_s, we take two events in the thread, and connect them to two events of a second thread and so on. We then have

$$\sum_{i=2}^{n} \binom{n}{i} \binom{m}{2}^{i},$$

or $\sum_{i=2}^{n} \binom{n}{i} (\frac{m(m-1)}{2})^i$, which gives $(\frac{m(m-1)}{2} + 1)^n - n \times m^2/2 + n \times m/2 - 1$ directed cycles. The number is in $o((m^2/2)^n)$.

With back-edge, in graph 3.21(c), we need to distinguish the positive and negative edges connected two events. We then take the number of arrangements of two events in this thread, and try to connect them with the other threads in the same way. We end up with

$$\sum_{i=2}^{n} \binom{n}{i} (A_m^2)^i,$$

leading to $\sum_{i=2}^{n} \binom{n}{i} (m(m-1))^i$, then $(m(m-1)+1)^n - n \times m^2 + n \times m - 1$ directed cycles. The number is a $o(m^{2n})$.

Tarjan's and Szwarcfiter-Lauer's algorithms complexities for this graph

Tarjan's algorithm time complexity is in $O(V \times E \times (C+1))$, where E is the number of edges, V the number of vertices and C the number of directed cycles to find. Szwarcfiter-Lauer's procedure is in $O(V + E \times (C+1))$. C is actually the total number of directed cycles in the graph, regardless of the size or the alternation of cmp and po_s^+.

We now estimate the time and space complexities of the two algorithms in the context of the worst case abstract event graph.

For the non-transitive graph, we have $E = n \times (m-1) + m^n$, $V = n \times m$ and $C = O(m^n)$. The time complexities would be respectively $O(n \times m^{2n+1})$ and $O(m^{2n})$.

With transitivity, we get $E = n\binom{m}{2} + m^n$, $V = n \times m$ and $C = O(m^{2n}/2^n)$. The time complexities are in $O(n \times m^{3n+1}/2^n)$ and $O((m^{3n}/2)^n)$ respectively.

With transitivity and *po*-loops, we get $E = n \times A_m^2 + m^n$, $V = n \times m$ and $C = O(m^{2n})$. The time complexities are then in $O(n \times m^{3n+1}/2^n)$ and $O(m^{3n}/2^n)$ respectively.

[11] This number could be predicted with the $v^{l(v)}/l$ directed cycles of length $l(v)$ in a complete graph computed earlier, as we have $v = m \times n$ and $l(v) = n$, inducing m^n.

The space-complexity of these two algorithms is in $O(V + E + S)$, where S is the sum of the length of the elementary directed cycles. Here, $S = 4 \times C$, so we have a space-complexity in $O(m^{2n}/2^{n-2})$.

3.3 Novelty of our contribution w.r.t. the related work

3.3.1 Semantics for the weak memory related analyses

We survey the semantics used by weak memory analyses in the literature. As we mentioned previously, these analyses use either operational or axiomatic models. The former tend to be used in safety verification, whereas the latter appear to be more adapted to static analyses. We summarise in Fig. 2.1 these semantics and attempt to classify them inside a lattice of abstractions. The semantics closest to the source, at the bottom of the lattice, have been only partially evaluated, and are therefore still very close to the original program and its CFG. The semantics on top of the lattice work on execution traces, which require a complete evaluation of the program and would in effect be more theoretical. The semantics on the left-hand side rely on traces, the semantics on the right-hand side abstract them into less precise structures. Working with the AEG allows us to be close to the source, thus to the actual code and offer an analysis which can be used practically in a straightforward manner. It, however, lies on the right-hand side since it abstracts the executions, meaning that we might detect some spurious executions.

The work of Shasha and Snir [SS88] is a foundation for the field of weak memory consistency. Most of the work cited below inherits their notions of *delay* and *critical cycle*. A delay is a pair of instructions in a thread that can be reordered by the underlying architecture. A critical cycle essentially represents a minimal violation of SC.

Operational models Linden and Wolper [LW13] explore all executions (using what they call *automata acceleration*) to simulate the reorderings occuring under TSO and PSO. Abdulla et al. [AAC$^+$13] couple predicate abstraction for TSO with a counterexample-guided strategy. They check if an error state is reachable. Kuperstein et al. [KVY10] explore all executions for TSO, PSO and a subset of RMO, and along the way build constraints encoding reorderings leading to error states. The same authors [KVY11] improve this exploration under TSO and PSO using an abstract interpretation they call *partial coherence abstraction*, relaxing the order in the write

buffers after a certain bound, thus reducing the state space to explore. Bouajjani et al. [BDM13] build on an operational model of TSO. They look for *minimum violations* (viz. critical cycles) by enumerating *attackers* (viz. delays).

All the approaches above focus on TSO and its siblings PSO and RMO, whereas we also handle the significantly weaker Power.

Axiomatic models Krishnamurthy et al. [KY96] apply Shasha and Snir's method to *single program multiple data* systems, that is, for programs calling n times the same thread. Their abstraction is similar to ours, except that they do not handle pointers and unbounded loops. They also do not handle thread creation: all the threads must be instantiated at the same time. Lee and Padua [LP01], Fang et al. [FLM03], Sura et al. [SFW+05] and Alglave and Maranget [AM11] propose fence synthesis techniques for weak memory that reason over an axiomatic models. We will explain them in detail in Chap. 5.

3.3.2 Survey of techniques for enumerating the critical cycles

Algorithms and encoding for critical cycle enumeration We briefly survey in this section the techniques for directed cycles enumeration and their costs. A deeper analysis of some of these techniques can be found in [LT82, BvL87] for the explicit search and matrix-based techniques, and in [HKSV97] for the symbolic directed cycles enumeration.

The graph in Fig. 3.22 gives an insight of the trade-off in these techniques between space and time complexities. Note that some of the techniques are quite dependent from the number of directed cycles. This explains why their complexity varies. We denote these variations in the graph with bars. In worst case, with a sup-exponential number of directed cycles, the complexities correspond to the crosses at the end of each bar.

Decision procedures and graph alteration Finding the directed cycles in a graph can be achieved in several ways. One possible method is to perform a depth-first search (DFS), e.g., as in [BJG08, p. 26], in the graph and check whether we encounter a vertex which has already been visited. If so, we extract this directed cycle and remove it from the graph. We iterate this process until we visit the whole graph without any directed cycle or if the graph is empty. This technique might however not reveal all the directed cycles: removing one edge of a directed cycle could prevent us from detecting a sup-directed cycle. Another option is to try to

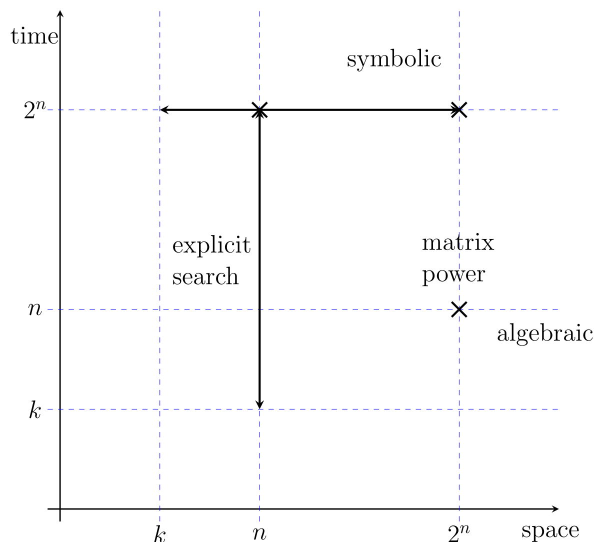


Figure 3.22: Space and time complexities trade-off for the directed cycles detection techniques.

construct the topological sort of this graph, as in [Knu68]. If it is possible, there is no directed cycle in the graph. If not, we extract the directed cycle and remove it from the graph, then iterate. Similarly, we can apply any decision procedure finding a cycle, removing it from the graph and re-iterating until it finds no cycle. We do not consider all those methods as we want to detect and enumerate *all* the directed cycles in the graph, without missing any of them.

Explicit search-based algorithms in search-space A first possibility is to construct the search-space and to explore it explicitly. This idea, proposed in 1968 by [Cha68] and inspired from [PM64], requires encoding the whole search space, leading to a certain space-explosion and a strong infeasibility in practice.

Explicit search-based algorithms with backtrack We discussed in Sec. 3.2 the explicit search-based algorithms with backtracking. The most efficient algorithms are those from Tarjan [Tar73], in $O(E \times V \times (C+1))$, Johnson [Joh75], in $O((E+V) \times (C+1))$, and Szwarcfiter and Lauer, in $O(V + E \times (C+1))$.

Matrix-power based directed cycles enumeration Another option is to compute the transitive closure of the graph with the power of the adjacency matrix. These techniques, proposed between 1966–1968 by [Pon66, Yau68, Kam67, Dan68], with intermediate tests to remove non-simple paths for [Dan68], are keeping track of each of the possible explored paths. All the paths are explored and stored in parallel, with

lots of redundancies, leading to a potential space explosion. In Fig. 3.23, we compute the directed cycles of the 4 vertices graph by multiplying the adjacency matrix. At iteration i, all the paths in the matrix are of size $i+1$. The directed cycles can be collected on the diagonal. In the figure, we represent iteration 2. We find the directed cycles 143, 241, 314, 431 and 412. Note that they are all rotations of 143 and 412. This representation generates lots of redundancies.

Figure 3.23: Example of transitive closure computation. The intermediate matrix on the right, after one iteration, shows the (simple) path of size 3 between each vertex. The directed cycles of length 3 are on the diagonale.

Algebraic directed cycles generation In [SS79], Srimani and Sengupta propose to detect all the directed cycles in an algebraic way. They compute reachability equations for each vertex, that is, the v equations (as in equation 3.1) for each vertex v_i which relates this vertex to all the other vertices reachable within k steps, encoding paths into vectors X_{i_k} and solving these equations by successive substitutions. Circuits are the paths which relate v_i to itself. The fixpoint is reached after v iterations.

$$v_i = X_{i_1} v_{i_1} \sum_{m=1}^{l_1} v_{i_1,m} + \ldots + X_{i_k} v_{i+k} \sum_{m=1}^{l_k} v_{i_k,m}. \tag{3.1}$$

This technique actually explores the graph in a DFS way, simultaneously storing all the paths at every step. This approach is exponential in space, as noticed in [LT82]. It is formalised in [Sri79] as a technique similar to the matrix-power based techniques enumerated above, with a modified adjacency matrix.

Symbolic directed cycles enumeration Detecting a directed cycle in the transition system of the program can be reduced to deciding whether the next formula is satisfiable or not:

$$\exists x, \exists y, T^+(x,y) \wedge T^+(y,x) \wedge x \neq y, \tag{3.2}$$

where T^+ is the transitive closure of the transition system. If it is SAT, we extract the directed cycle from the model and iterate whilst preventing this model happening again—an iteration strategy with "recordings" [ES03]. The symbolic directed cycle detection, certainly efficient for deciding whether there is a directed cycle in the graph, is quite inefficient for enumerating, as it needs to propagate in its formula all the directed cycles it detected so far, leading to a potential space explosion. In terms of time complexity, it is also quite hard to determine the overhead compared to the approaches specifically tailored for this problem, since we use the SAT solver as a "black-box". Nothing guarantees that our SAT encoding will provide the corresponding complexity.

An alternative option is to solve the formula in equation 3.2 with a #SAT solver, e.g. SharpSAT [Thu06], which counts all the SAT solutions, that is, all the directed cycles in the graph, usually by enumerating them.

The symbolic approach is intuitively a good approach for limited search over complex structures/graphs, leaving all the hard work to the SAT solver or the BDDs. However, for the complete exploration of a whole graph, given a specific problem, an algorithm with an explicit search specifically tailored for this problem would be likely to be more efficient, exploiting its instantiated structures that the SAT solver would infer progressively through learning.

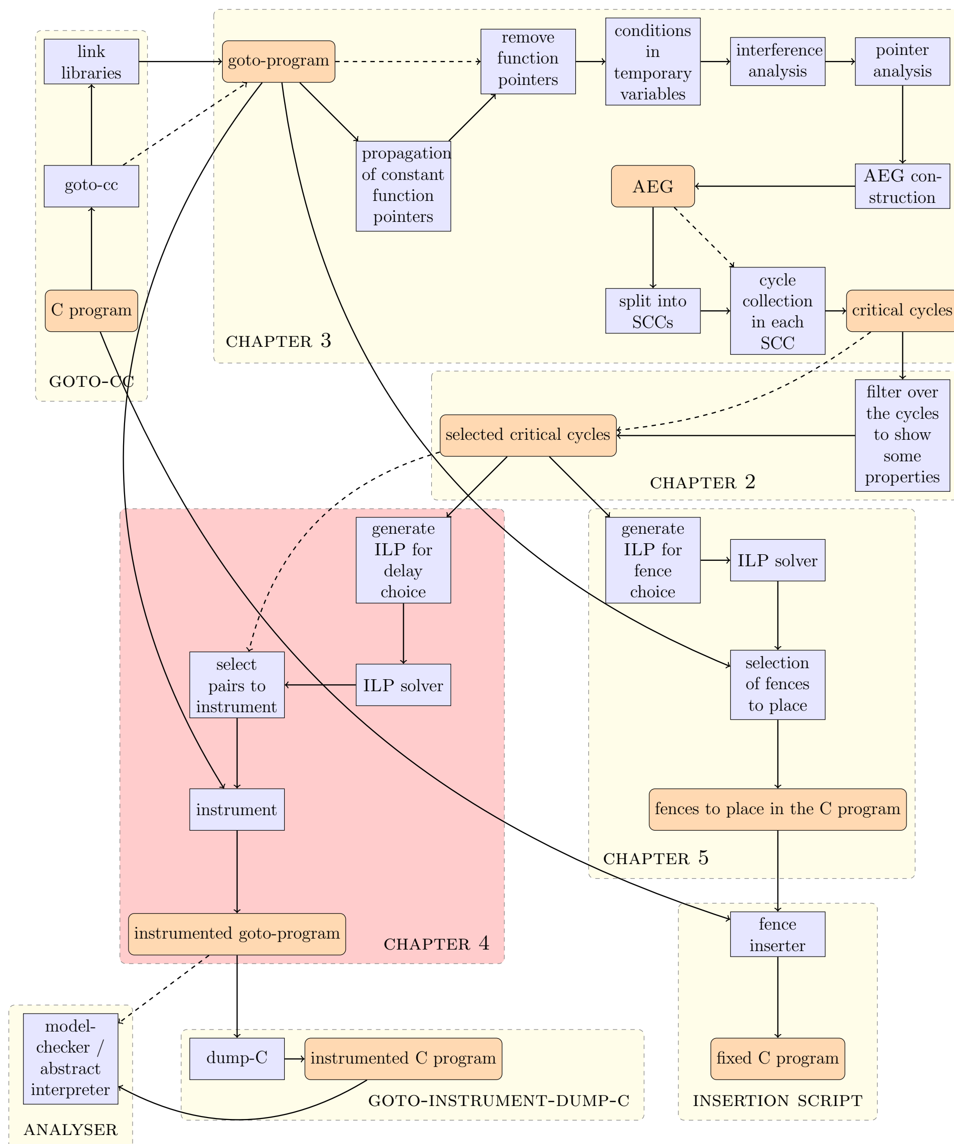
3.4 Summary

In this chapter, we introduced a sound abstraction of the CFG, namely the AEG, that captures all the executions modelled by Alglave's framework. We showed that critical cycles were abstracted by static critical, directed cycles in the AEG. We explained how we implemented this search, and discussed its complexity.

Given a program, we can extract static, critical cycles covering critical cycles that may occur during an execution of this program on a given architecture. In the next chapter, we introduce formally the properties that can hold for programs and program analyses with respect to weak memory consistency, and explain how to use these static, critical cycles to address them statically.

Chapter 4

Instrumentation over Weak Memory



In Chap. 2, we explained that a program P can be transformed into a program P' so that P' would behave under SC as if the original program P were run on a processor implementing weak memory (e.g. x86 or ARM).

In this chapter, we propose a program transformation which implements this. The core idea consists of inserting in the program buffers, queues and non-deterministic writes and flushes emulating the buffers and cache memory systems implemented in the actual processors. Atig et al. proposed in [ABP11] a technique that replaces the accesses to shared memory by non-deterministic writes to a buffer or the memory and non-deterministic flushes of the buffer in order to emulate x86. Similarly, we propose a transformation technique that also covers weaker architectures, like Power and ARM, which are significantly weaker than x86 in that they allow more reorderings. In particular, the read-read or read-write reorderings can be challenging to simulate in a static setting, as postponing a read means that we must force the evaluation of an expression to happen later—and there is no native mechanism to do such in C.

The purpose of this program transformation is to allow any analyser handling SC concurrency and non-determinism—some model-checkers like SatAbs [CKSY05] or Q Program Verifier[1] (aka Poirot), or some abstract interpreters like ConcurInterProc[2]—to be aware of the targeted architecture on which the program will run. In practice, it means that if we input the program transformed for x86 into e.g. SatAbs, SatAbs would be able to find assertions violated because of an execution involving a reordering specific to the x86 processor, whereas these assertions would not be violated by any interleavings under SC. Analysing with ConcurInterProc a transformed program for ARM would allow the abstract interpreter to guarantee the safety of the program for given assertions or properties, taking into account the specific memory consistency of the ARM processor.

We implemented the instrumentation[3] of Atig et al. and the instrumentation for weaker architectures including Power and ARM. We then connected 5 model-checkers that handle SC concurrency:

- CBMC, a bounded model checker;
- Corral, a model checker implementing context-bounded translation to sequential programs;

[1] `http://research.microsoft.com/en-us/projects/verifierq/`. This verifier, formerly called Poirot, uses the model-checker Corral [LQL12] for Boogie programs [Lei08] as back-end.

[2] `http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi`. ConcurInterProc is an on-line abstract interpreter for concurrent programs that uses the library APRON [JM09].

[3] The instrumentation of Atig et al. is implemented as the option –cav11 in goto-instrument.

- ESBMC, a bounded model checker;
- SatAbs, a model checker based on predicate abstraction;
- Threader, a thread-modular verifier;

and compared to 4 weak memory aware model checkers:

- Blender;
- CBMC MT[4];
- CheckFence;
- MMChecker.

Details about these tools, the versions used, their licenses and who to contact to retrieve them are listed in Sec. E.3 in App. E. We observed that programs of more than 50 lines would become too complex after instrumentation for the model-checkers to be solved in a time that would allow a practical use—that is, a time measured in minutes. Instrumenting to simulate the additional behaviours is not sufficient, we also want to limit the instrumentation—which adds buffers, queues and non-deterministic choices in the program—to the places where it is really necessary.

Our solution to limit the instrumentation consists of a static analysis that occurs right before the insertion of buffers, queues and their related operations. By [SS88], we know that only memory accesses that would be involved in critical cycles would affect the semantics of the program. By computing the over-approximation that we described in Chap. 3 and collecting the critical cycles in it, we can target which variables need to be instrumented and where in the code. Our experiments showed that this selection has a significant impact on the scalability of the verification after instrumentation strategy, and we could analyse an excerpt of Pgsql of about 4000 lines of C codes.

Once we have detected a cycle, we actually need to instrument only one of its delays in order to allow the behaviour forbidden by the critical cycle. The choice of this delay also had an impact on the verification performance. Based on the instrumentation strategy for each of the delay pairs, we assigned costs to these delays and chose to instrument the pairs that will minimise the global cost. We performed this selection by solving an integer linear program. With the help of this technique, we

[4]CBMC MT is a modification of CBMC implemented for [AKT13]. The tool can be retrieved as a patch for CBMC at `www.cprover.org/wmm/cav13`.

reduced the verification time[5] of 26% in average on our benchmark.

Plan of the chapter
We first explain how to address the problem of instrumentation at the level of event structures, as defined in Chap. 2. We recall the details of an abstract machine that is equivalent to event structures. The states of this machine maintain write buffers and read sets in addition to the memory environment, events can be delayed or flushed according to a set of rules. These delays and flushes will be the base of our instrumentation. We explain briefly how the instrumentation fits with the critical cycles that may appear in the structures. We extend both of these aspects in the following section for C programs, and introduce the integer linear program that selects an instrumentation which is less complex for the model-checker (or abstract interpreter) to analyse afterwards. We detail the experiments that we ran in the next section, with a specific focus on reproducing a weak memory bug that had been observed in Pgsql. We finally compare our approach to the existing techniques for handling weak memory behaviours in the context of verification.

4.1 Instrumentation of event structures

In this section, we summarise the strategy introduced in [AKNT13] by Alglave to instrument the event structures defined in Chap. 2. We will propose in the next section a static instrumentation for concurrent C programs that extends this theory to actual CFG.

We first give an operational description of memory models in terms of an *abstract state machine* (Sec. 4.1.1). This abstract machine is equivalent to the axiomatic model presented in Chap. 2. The readers can find the detailed proof of this equivalence in [AKNT13]. We explain how this equivalence guides our instrumentation strategy.

4.1.1 Abstract machine

We define a non-deterministic state machine that reads a sequence of *labels*. The machine has a designated error state $\perp$, which denotes the stalling of the machine, and all other states of the machine represent system configurations, i.e. the memory environment, write buffers, and the set of pending reads. We write addr, evt and $\mathrm{rln}(S)$ for the set of memory addresses, the set of events and the set of relations over S (i.e., $\wp(S \times S)$) respectively. W and R are the sets of respectively the write and read

[5] Including the additional time spent in solving the linear programming problem with **glpk**.

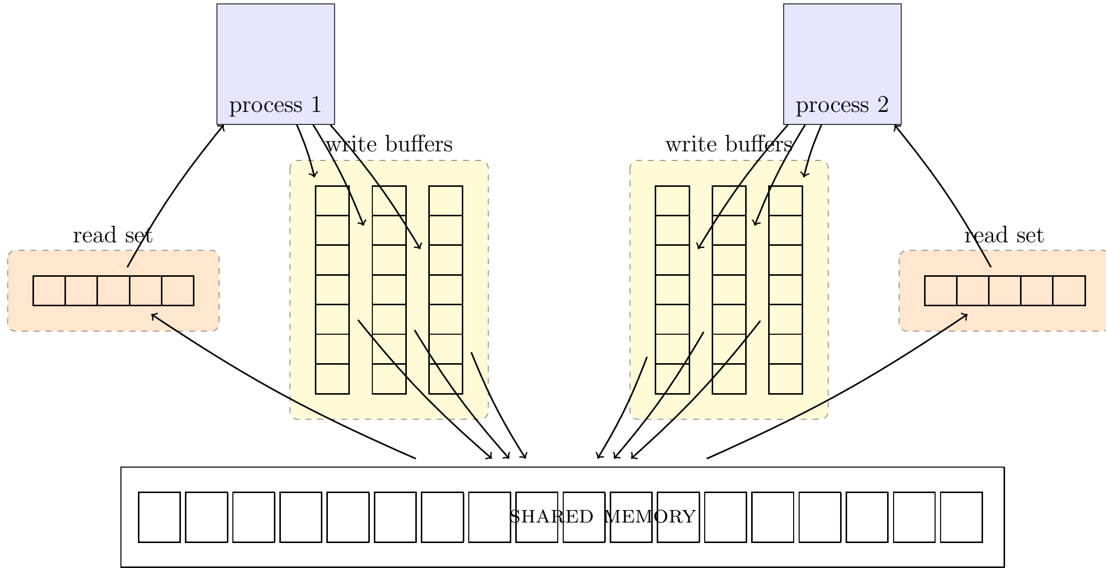


Figure 4.1: Representation of the system (for 2 threads) whose configurations are modelled by the abstract machine.

events, and $\mathrm{rr}(R, S)$ restricts the relation R to the set S, that is, $\mathrm{rr}(R, S) \triangleq R \cap (S \times S)$. The initial values of variables are modelled by writes that happen before any other event. We will often omit them in the representations when shared variables are initialised by default (to 0).

Definition 11 (Machine state)**.** *A* state *of the machine is either* $\perp$ *or a triple* $(mem, \boldsymbol{b}, \boldsymbol{rs})$*, where*

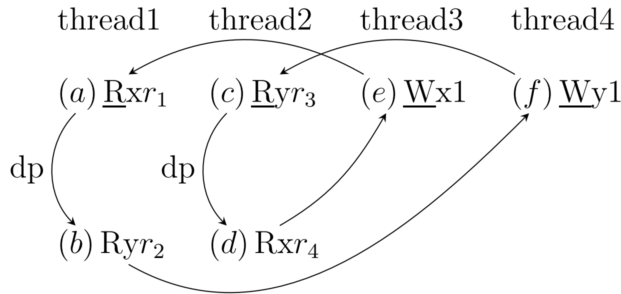
- *the* memory ($mem \in addr \to evt$) *maps a memory address* ℓ *to a write to* ℓ*;*
- *the* write buffer ($\boldsymbol{b} \in rln(evt)$) *is a total order over writes to the same address; the buffer has a special symbol* $\perp_{\boldsymbol{b}}$*, placed before all events in the buffer;*
- *the* queue ($\boldsymbol{rs} \in \wp(evt)$)*is a set of read events.*

Fig. 4.1 represents a system that would be modelled by these states. We have a single set of reads, but one totally ordered buffer per address. Existing formalisations [OSS09, ABP11] use per-thread buffers, whereas our buffers are solely per-address objects. This allows us to model not only store buffering (which per-thread objects would allow) like in Fig. 2.14, but also caching scenarios (fully non-atomic stores) as possibly exhibited[6] by **(iriw+dps)** in Fig. 4.2. The event structure in

[6] In the program of Fig. 4.2, the *read_pair* structures are passed as arguments rather than placed in memory as global variables in order to avoid some potential additional reorderings of these reads. The **(iriw+dps)** idiom would nevertheless still be observable with the reading variables as global variables.

```
#include <assert.h>
#include <pthread.h>

/* shared memory */
volatile unsigned x = 0;
volatile unsigned y = 0;

void* thread1 (void* arg) {
  pair_t* read_pair = (pair_t*)arg;

  /* r1 = x */
  *(read_pair->first) = x;

  /* tmp = r1 ^ r1 -- robust to
     GCC optimisations since r1 is
     casted as volatile */
  unsigned tmp = *(read_pair->first)
    ^ *(read_pair->first);

  /* r2 = *(&y + tmp) */
  *(read_pair->second) = *(&y+tmp);
}

void* thread3 (void* arg) {
  x = 1;
}

/* args (with volatile casts) */
typedef struct {
  volatile unsigned* first;
  volatile unsigned* second;
} pair_t;

void* thread2 (void* arg) {
  pair_t* read_pair = (pair_t*)arg;

  /* r3 = y */
  *(read_pair->first) = y;

  /* tmp = r3 ^ r3 */
  unsigned tmp = *(read_pair->first)
    ^ *(read_pair->first);

  /* r4 = *(&x+tmp) */
  *(read_pair->second) = *(&x+tmp);
}

void* thread4 (void* arg) {
  y = 1;
}

int main () {
  pthread_t p[4];
  unsigned r1 = 0, r2 = 0, r3 = 0, r4 = 0;
  pair_t arg1 = {.first = &r1, .second = &r2};
  pair_t arg2 = {.first = &r3, .second = &r4 };

  pthread_create(p,   0, (*thread1), (void*)&arg1);
  pthread_create(p+1, 0, (*thread2), (void*)&arg2);
  pthread_create(p+2, 0, (*thread3), 0);
  pthread_create(p+3, 0, (*thread4), 0);
  unsigned i;
  for(i = 0; i<4; ++i) pthread_join(p[i], 0);

  assert( ! (r1==1 && r2==0 && r3==1 && r4==0) );
  return 0;
}
```

Figure 4.2: Implementation (in full) of the idiom **(iriw+dps)** with POSIX threads. The assertion can be violated under ARM and IBM Power.



assert: $\neg(r_1 \equiv 1 \wedge r_2 \equiv 0 \wedge r_3 \equiv 1 \wedge r_4 \equiv 0)$.

Figure 4.3: Litmus test **(iriw+dps)** which violates the final assertion under Power and ARM architectures.

Fig. 4.3 depicts graphically such a scenario: on Power and ARM, we can observe the case where the reads of thread1 and thread2 read respectively 1 and 0 for the first and second statements, even though we placed the two reads of each thread in dependency so that they cannot be reordered.

The machine performs transitions depending on *delay* and *flush* labels. Intuitively, a delay label pushes an object in the write buffer or read set. A flush label makes it exit the write buffer or read set. The details of transitions are described below.

Definition 12 (Delay/flush label)**.** *For a write event w, delay*$(\mathrm{w}(w))$ *denotes its* delay label*, and* $flush(\mathrm{w}(w))$ *its* flush label*. For a read event r, its delay label (with direction* r*, read) is denoted by delay*$(\mathrm{r}(w, r))$*, and its flush is denoted by* $flush(\mathrm{r}(w, r))$*.*

Transitions We write $s \xrightarrow{l} s'$ to denote that the machine can make a transition from state s to state s' reading label l. Let the machine be in a state $(\mathrm{mem}, \mathbf{b}, \mathbf{rs})$. Given a label, the machine performs transitions from one state to another if the conditions described below are fulfilled. Otherwise, the machine transitions to $\bot$ (it gets stuck).

Write to buffer: a write *delay*$(\mathrm{w}(w))$ to address ℓ can always enter the buffer $\mathbf{b}$, taking its place $\mathbf{b}$-after all the writes to ℓ that are already in $\mathbf{b}$.

$$\frac{\top}{s \xrightarrow{delay(\mathrm{w}(w))} (\mathrm{mem}, \mathrm{updb}(\mathbf{b}, w), \mathbf{rs})},$$

where updb is defined by $updb(\mathbf{b}, w) \triangleq \mathbf{b} \cup \{(w_1, w) \mid w_1 = \bot_{\mathbf{b}} \vee ((\bot_{\mathbf{b}}, w_1) \in \mathbf{b} \wedge \mathrm{addr}(w_1) = \mathrm{addr}(w))\}$.

Delay read: a read $delay(\mathrm{r}(w,r))$ can always enter the read $\mathbf{rs}$.

$$\frac{\top}{s \xrightarrow{delay(\mathrm{r}(w,r))} (\mathrm{mem}, \mathbf{b}, \mathrm{updrs}(\mathbf{rs}, r)),}$$

where $\mathrm{updrs}(\mathbf{rs}, r) \triangleq \mathbf{rs} \cup \{r\}$.

Write from buffer to memory: a write $flush(\mathrm{w}(w))$ to address ℓ exits the buffer $\mathbf{b}$ and updates the memory at ℓ if:

(W1) there is no event e in the buffer which is $\mathsf{ppo} \cup \mathsf{ab}$-before w;

(W2) *and* there is no event e in the read set which is $\mathsf{ppo} \cup \mathsf{ab}$-before w;

(W3) *and* there is no read from ℓ in the buffer which is po-before w;

(W4) *and* there is no write to ℓ in the buffer which is $\mathbf{b}$-before w.

$$\frac{\begin{array}{lr} \mathrm{rr}(\mathbf{b}, \{e \mid (e,w) \in \mathsf{ppo} \cup \mathsf{ab}\}) = \emptyset \wedge & \text{(W1)} \\ \mathbf{rs} \cap \{e \mid (e,w) \in \mathsf{ppo} \cup \mathsf{ab}\} = \emptyset \wedge & \text{(W2)} \\ \mathbf{rs} \cap \{r \mid (r,w) \in \mathsf{po\text{-}loc}\} = \emptyset \wedge & \text{(W3)} \\ \mathrm{last}(\mathrm{rr}(\mathbf{b}, \{e \mid \mathrm{addr}(e) = \ell\}), w) & \text{(W4)} \end{array}}{s \xrightarrow{flush(\mathrm{w}(w))} (\mathrm{updm}(\mathrm{mem}, w), \mathrm{delb}(\mathbf{b}, w), \mathbf{rs}),}$$

where $\mathrm{updm}(\mathrm{mem}, w) \triangleq \mathrm{mem}[l \mapsto _ \backslash l \mapsto w]$, $\mathrm{delb}(\mathbf{b}, w) \triangleq \mathbf{b} \backslash (\{w\} \times W \cup W \times \{w\})$ and $\mathrm{last}(\mathbf{b}, w)$ is the last write[7] in the buffer that targets the same address as w.

Read from set: a read $flush(\mathrm{r}(w,r))$ from ℓ (Condition (R1)) exits $\mathbf{rs}$ if:

(R2) there is no read in the read set that is dp-before w;

(R3) *and* there is no event in the buffer that is $\mathsf{ppo} \cup \mathsf{ab}$-before r;

(R4) *and* there is no event in the read set that is $\mathsf{ppo} \cup \mathsf{ab}$-before r;

(R5) *and either* w is in memory, and there is no write to ℓ in the buffer that is po-before r;

(R6) *or* if w is not in memory, w is in the buffer and is *visible to* r (a notion defined below).

[7] More formally, $\mathrm{last}(\mathbf{b}, w) \triangleq (\neg(\exists w', (\bot_{\mathbf{b}}, w') \in \mathbf{b}) \wedge w = \bot_{\mathbf{b}}) \vee ((\exists w', (\bot_{\mathbf{b}}, w') \in \mathbf{b}) \wedge (\bot_{\mathbf{b}}, w) \in \mathbf{b} \wedge \neg(\exists w', (w', w) \in \mathbf{b}))$.

$$\frac{\begin{array}{lr} r \in \mathbf{rs} \wedge & \text{(R1)} \\ \mathbf{rs} \cap \{r' \mid (r', w) \in \mathsf{dp}\} = \emptyset \wedge & \text{(R2)} \\ \mathrm{rr}(\mathbf{b}, \{e \mid (e, r) \in \mathsf{ppo} \cup \mathsf{ab}\}) = \emptyset \wedge & \text{(R3)} \\ \mathbf{rs} \cap \{e \mid (e, r) \in \mathsf{ppo} \cup \mathsf{ab}\} = \emptyset \wedge & \text{(R4)} \\ \big(\mathrm{rfm}(\mathrm{mem}, \mathbf{b}, w) \vee & \text{(R5)} \\ (w \neq \mathrm{mem}(\mathrm{addr}(r)) \wedge w \in \mathbf{b} \wedge \mathrm{visible}(w, r))\big) & \text{(R6)} \end{array}}{s \xrightarrow{flush(\mathrm{r}(w,r))} (\mathrm{mem}, \mathbf{b}, \mathrm{delrs}(\mathbf{rs}, r))},$$

where $\mathrm{delrs}(\mathbf{rs}, r) \triangleq \mathbf{rs} \backslash \{r\}$ and $\mathrm{rfm}(\mathrm{mem}, \mathbf{b}, w) \triangleq w = \mathrm{mem}(\mathrm{addr}(r)) \wedge \mathrm{rr}(\mathbf{b}, \{w \mid (w, r) \in \mathsf{po\text{-}loc}\}) = \emptyset$.

To define a write w as *visible to a read* r, we need a few auxiliary functions. We define the part of the buffer visible to a read r as follows: $\mathbf{b}_r \triangleq \{w \mid (\bot_{\mathbf{b}}, w) \in \mathbf{b} \wedge ((\mathsf{rfi} \subseteq \mathrm{safe}_A) \implies \mathrm{proc}(w) = \mathrm{proc}(r)) \wedge ((\mathsf{rfe} \subseteq \mathrm{safe}_A) \implies \mathrm{proc}(w) \neq \mathrm{proc}(r))))\}$. Now, w is visible to r when:

(V1) w and r share the same address ℓ;

(V2) w is in the part of the buffer visible to r, namely if rfi (resp. rfe) is safe then w cannot be on the same (resp. a different) thread as r ($w \in \mathbf{b}_r$);

(V3) w is $\mathbf{b}$-before the first write w_a to ℓ that is po-after r;

(V4) w is equal to, or $\mathbf{b}$-after, the last write w_b to ℓ that is po-before r.

All states except $\bot$ are accepting states. Thus, the abstract machine accepts a sequence p of labels $l_0, l_1, \ldots$ if there is a sequence of states $s_0, s_1, \ldots$ such that $s_i \xrightarrow{l_i} s_{i+1}$ and $s_i \neq \bot$ for all i.

Definition 13 (Accepting sequence)**.** *A* sequence *p is a total order over L compatible with the program order, i.e. for two events $(x, y) \in \mathsf{po}$, their delay labels appear in the same order in p. It is* accepting *iff the sequence p is accepted by the abstract machine.*

4.1.2 Illustration using examples

We illustrate the machine by revisiting the **(sb)** test of Fig. 2.15 for TSO and the **iriw** test for Power—that is, the test of Fig. 4.2 without the dependencies between the read accesses. Fig. 4.4 and 4.5 reproduce on the left the event graphs from Fig. 2.15 and 4.3. On the right, they show the counterparts in the abstract machine. We will explain the labels on the arrows in the next paragraph. We use the following graphical

conventions. In the axiomatic world (i.e. on the left of our figures), we reflect a pair that an architecture relaxes by a dashed arrow. For example, in the **(sb)** test of Fig. 4.4 on TSO, the write-read pairs (a, b) and (d, c) can be relaxed. Likewise, in the **(iriw+dps)** test of Fig. 4.5 on Power, the read-from pairs (e, a) and (f, c) can be relaxed (as opposed to the read-read pairs (a, b) on P_0 and (c, d) on P_1, which are safe because of dependencies).

In any given execution, the abstract machine may choose to relax any pair that is not safe. Such pairs are depicted with a dashed arrow. Pairs that the machine does not relax are depicted with a thick arrow.

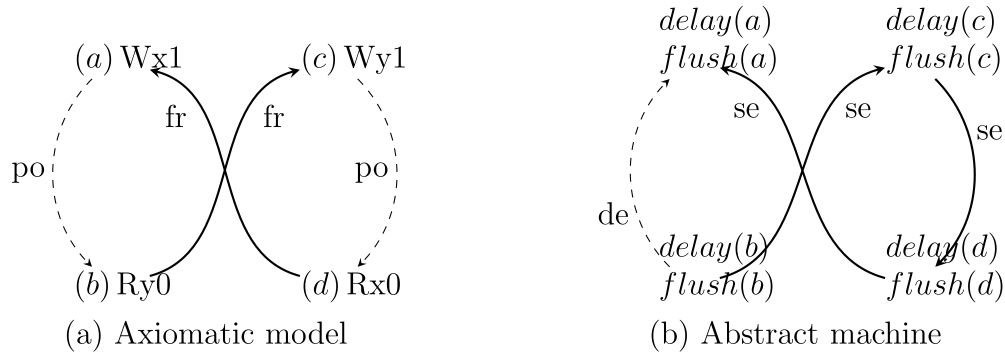
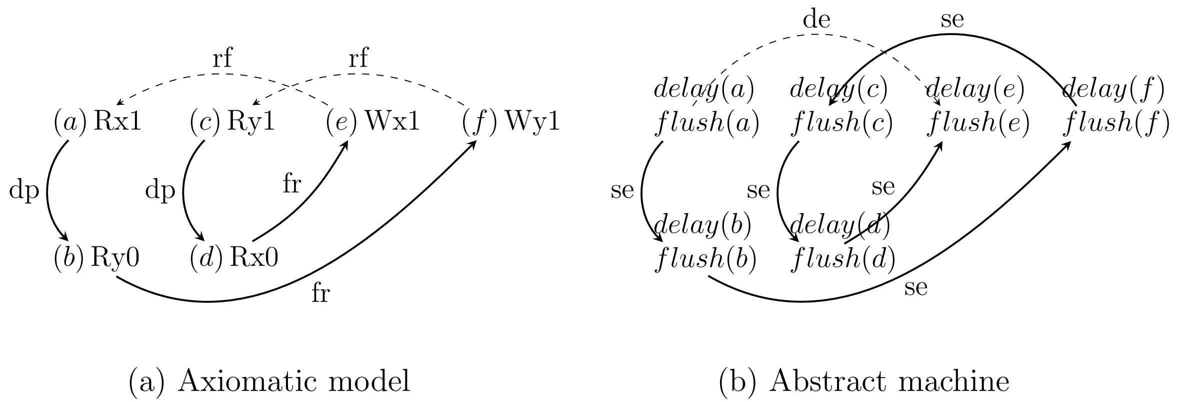


Figure 4.4: Revisiting **(sb)** (presented in Fig. 2.15) on TSO with the abstract machine of Sec. 4.1.1.

In Fig. 4.4(a), the pairs (a, b) on P_0 and (c, d) on P_1 are relaxed on TSO. Our machine may simulate the behaviour permitted on TSO by following the scenario in Fig. 4.4(b), which corresponds to the path $delay(a) \to delay(b) \to delay(c) \to delay(d) \to flush(b) \to flush(c) \to flush(d) \to flush(a)$. In the figure, the label "se" corresponds to a safe exit, and "de" to a delay exit, which are formalised below. The machine delays all events w.r.t. program order. In this scenario, the machine chooses to relax the pairs (a, b) by flushing the read b before the write a, ensuring that the registers `r1` and `r2` hold 0 in the end.

In Fig. 4.5(a), assume dependencies between the reads on P_0 and P_1, so that (a, b) on P_0 and (c, d) on P_1 are safe on Power. Yet (e, a) and (f, c) may be relaxed on Power, because Power has non-atomic writes. Our machine may simulate the weak behaviour exhibited on Power by following Fig. 4.5(b), which corresponds to the path $delay(e) \to delay(a) \to flush(a) \to delay(b) \to flush(b) \to delay(f) \to flush(f) \to delay(c) \to flush(c) \to delay(d) \to flush(d) \to flush(e)$. Since (a, b) and (c, d) are safe on Power, our machine flushes a before b (resp. c before d). Since $(b, f) \in \mathsf{fr}$ (resp. $(d, e) \in \mathsf{fr}$), which is always safe, the machine flushes b before f (resp. d before



(a) Axiomatic model

(b) Abstract machine

Figure 4.5: Revisiting **(iriw+dps)** (implemented in Fig. 4.2) on Power with the abstract machine of Sec. 4.1.1.

e), ensuring that b and d read from memory, thus `r2` and `r4` hold 0 in the end. Finally, in this scenario, the machine chooses to relax the pairs (e, a) by flushing a before e, ensuring that `r1` and `r3` hold the value 1 in the end.

4.1.3 Instrumentation

Alglave proved in our paper [AKNT13], theorems 1 and 2, that the abstract machine was equivalent to the axiomatic description, i.e., that a sequence accepted by the machine would be valid for the axiomatic model and vice-versa.

Theorem 2 (Equivalence between axiomatic model and abstract machine)**.** *Given an event structure E and an architecture A, there is a (finite) execution X valid for E under A iff there is a path of labels p such that the abstract machine for E accepts p.*

The proof provides two mappings ptoX and Xtop which go respectively from the paths of labels to the executions and vice-versa. More formally, $\text{ptoX}(p, L)$ is the set of executions that are compatible with the path p of labels in L, and $\text{Xtop}(X, E, \mathsf{ndelay})$ is the set of paths of labels that can construct the execution X for a given event structure E that would have ndelay as edges that cannot be delayed under the architecture considered. ndelay is taken such that $\mathsf{ndelay}(E, X) \supseteq (\mathsf{ws} \cup \mathsf{rf} \cup \mathsf{fr}) \cap \text{safe}_A$, ndelay transitive and irreflexive.

Thm. 2 leaves freedom in the instrumentation strategy. We can exploit the choice of delay pairs and the choice of the linearisation in order to reduce the overhead of running or verifying an instrumented program.

Choice of delay pairs The conditions on the ndelay relation restrict the choice of delay pairs. We have to put at least all the safe pairs into ndelay, by the first condition.

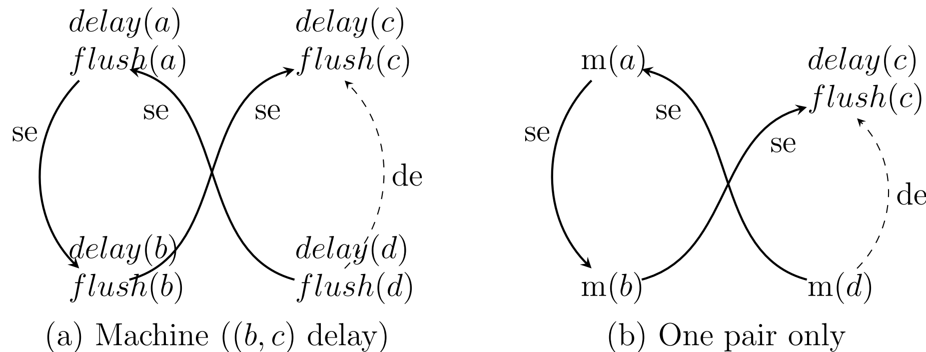
Since ndelay is transitive and irreflexive, it is acyclic. An execution (E, X) presents a cycle iff it is not SC (if it is SC, all pairs are safe and there is no cycle). [AM11, Thm.1] shows that an execution is valid on A but not on SC iff it contains *critical cycles*. Thus we can put all pairs in ndelay, except one unsafe pair per critical cycle, which corresponds to the last condition over ndelay.

In Fig. 4.4(b), we build an accepting path corresponding to the axiomatic execution of Fig. 4.4(a) by choosing the unsafe pair (a, b) on the cycle to be a delay. In Fig. 4.6(a), we choose the unsafe pair (c, d). Similarly for Fig. 4.5(a), we can build an accepting path corresponding to the axiomatic execution of Fig. 4.5(a) by choosing e.g. (e, a) as delay (cf Fig. 4.5(b)). In Fig. 4.7(a), we choose (f, c) as delay.

Our examples are symmetric, thus the choice of which pair to delay should not make a difference. In Fig. 4.4(a), (a, b) and (c, d) are write-read pairs. Similarly in Fig. 4.5(a), (e, a) and (f, c) are of the same nature, namely rfe pairs. For asymmetric examples, the chosen delayed pair can make a crucial difference, if the instrumentation of one pair causes more execution or verification time overhead than the other.

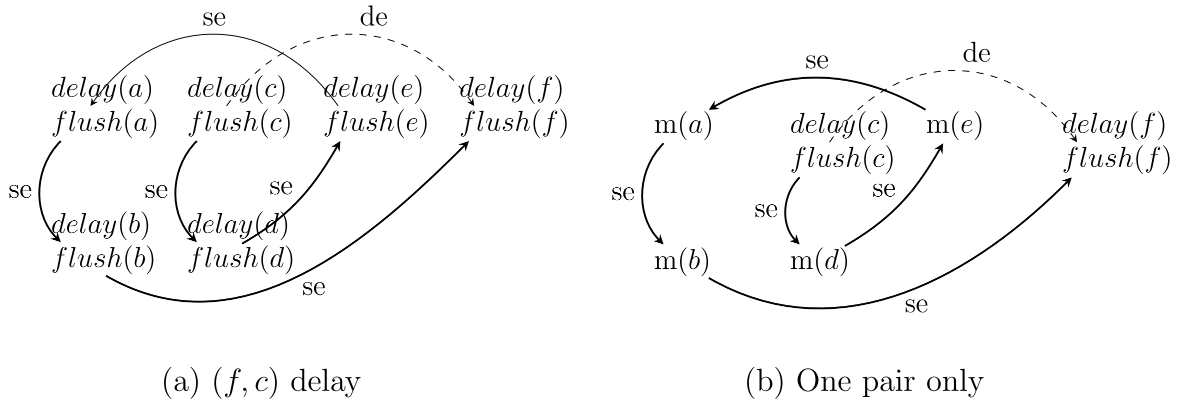
Choice of the linearisation Thm. 2 accepts any linearisation of $(\text{Xtop}(E, X, \mathsf{ndelay}))^+$. Yet, some require less instrumentation than others. Consider Fig. 4.6(a) and (b): in both we choose to delay the pair (c, d). On the left, we can pick any interleaving (compatible with Xtop) of the delayed and flushed events, e.g. $delay(a) \to delay(b) \to delay(c) \to delay(d) \to flush(b) \to flush(d) \to flush(c) \to flush(a)$.

On the right, we write $\text{m}(e)$ when the delayed and flushed part of an event happen without intervening events in between. Observe that in this case, the event e occurs



(a) Machine ((b, c) delay) (b) One pair only

Figure 4.6: Choices for instrumenting **(sb)** for TSO.



(a) (f, c) delay

(b) One pair only

Figure 4.7: Choices for instrumenting **(iriw+dps)** for Power.

w.r.t. memory: if it is a read, it reads from the memory; if it is a write, it writes to memory. In Fig. 4.6(b), we pick a particular interleaving, namely the one where all events are w.r.t. memory, except for the event c. This interleaving requires instrumenting only one instruction, as opposed to all of them on the left.

Similarly in Fig. 4.7(a) and (b), we choose in both cases to delay the pair (f, c). On the left, we instrument all instructions. On the right, we instrument only the pair (f, c).

4.2 Instrumentation of C programs

In order to instrument following the delays of critical cycles, we first construct the AEG of the input program with the method described in Chap. 3. We then collect the critical cycles and end up with a collection of delays—that are the static counterparts of the delays not in ndelay in Sec. 4.1.3—that need to be instrumented.

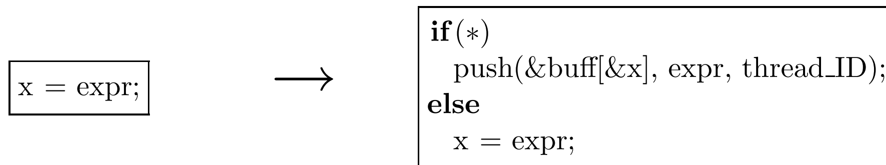
4.2.1 Delaying an access to shared memory

The above cycle detection yields candidates for unsafe pairs of abstract events to be delayed in each cycle. Following Sec. 4.1.3, we instrument one pair to delay per cycle. We may select these pairs arbitrarily, but we describe below a weighted instrumentation that reduces verification time, as we show in Sec. 4.3.

We first normalise the program such that all shared memory accesses appear in assignments only. Any reads in branching conditions or function call parameters are moved to temporary variables as follows: **if** $(\phi(\text{x}))$ ...; $\longmapsto$ tmp $= \phi(\text{x})$; **if**(tmp) ...; for an expression ϕ over a shared memory address x. In the following, we thus restrict ourselves to assignment statements.

For each memory address x of events in unsafe pairs we introduce an array $\mathbf{b}(x)$. In addition to the properties described in Sec. 4.1.1, we also keep track of the originating thread of the write to x. We introduce an additional pointer for each local variable reading from a shared memory address, i.e. an r such that $r = x$;. In a pair to delay, in one of the critical cycles or after, we equip r with a pointer $\mathbf{rs}(r)$, which implements the read set of Sec. 4.1.1. We now describe the instrumentation of writes, reads then external communications. To soundly over-approximate all possible behaviours, all instrumented operations are guarded by **if**$(*)$, expressing non-deterministic choice[8]. We also surround these instrumentations with begin_atomic() and end_atomic(), which keep them atomic. If the model-checker analysing the instrumented program does not handle atomic primitives, the instrumentation remains sound by removing them—but some additional spurious results might arise.

Instrumenting writes (WR and WW) We implement here the two operations associated with the weak-memory effects of a write w, as defined in Sec. 4.1.1: (1) delaying a write, $delay(W(w))$, by appending to the buffer, and (2) flushing a write, $flush(W(w))$, removing it from the buffer. A delayed write amounts to appending an element to the array buff that implements $\mathbf{b}$:



Note that the identifier of the (static) thread to which this push belongs, is contained in the abstract events of the AEG, and was computed by the interference analysis (in Chap. 3) prior to the delay detection.

According to Sec. 4.1.1, each delay is accompanied by a flush. Yet the point in time when the flush happens is not determined. We would thus need to add non-deterministic flush instructions at each statement in the program. This transformation would make the program highly non-deterministic, and very hard for a model checker to analyse. Therefore, we insert flushes only where they might have an effect, i.e., before each potential read from the address that was written to (or before relevant memory fences), and make them flush a non-deterministic number of writes in FIFO-manner. The function *take* implements the semantics of "write from buffer

[8]CProver provides non_det() primitives that simulates the non-determinism in input programs. See `http://www.cprover.org/cprover-manual/modeling-nondet.shtml` for more details.

```
void* thread1 (void* arg) {
  /* [...] */
  r1 = x;
  unsigned tmp = r1 ^ r1;
  r2 = *(&y+tmp);
}

void* thread2 (void* arg) {
  /* [...] */
  r3 = y;
  unsigned tmp = r3 ^ r3;
  r4 = *(&x+tmp);
}

void* thread4 (void* arg) {
  y = 1;
}

void* thread3 (void* arg) {
  x = 1;
}

int main () {
  /* [...] */
  assert( ! (r1==1 && r2==0 && r3==1 && r4==0) );
  return 0;
}
```

```
void* thread1 (void* arg) {
  /* [...] */
  begin_atomic();
  if(length(&buff[&x]) != 0 && *) {
    delay[&x] = TRUE;
    delay_tmp[&x] = x;
    x = last(&buff[&x], thread_ID_1);
  }
  r1 = x;
  if(delay[&x] && *) {
    x = delay_tmp[&x];
    delay[&x] = FALSE;
  }
  end_atomic();
  unsigned tmp = r1 ^ r1;
  r2 = *(&y+tmp);
}

void* thread2 (void* arg) {
  /* [...] */
  r3 = y;
  unsigned tmp = r3 ^ r3;
  r4 = *(&x+tmp);
}

void* thread4 (void* arg) {
  y = 1;
}

void* thread3 (void* arg) {
  begin_atomic();
  if(*)
    push(&buff[&x], 1, thread_ID_3);
  else
    x = 1;
  end_atomic();
}

int main () {
  /* [...] */
  assert( ! (r1==1 && r2==0 && r3==1 && r4==0) );
  return 0;
}
```

Figure 4.8: The implementation of **(iriw+dps)** from Fig. 4.2 on top and the instrumented program below.

to memory" of Sec. 4.1.1 for a non-deterministic number of elements, and returns the resulting in-memory value at address &x.

```
expr = x;    ⟶    if(*)
                    x = take(&buff[&x], thread_ID);
                  expr = x;
```

We illustrate the write instrumentation with the transformations of **(sb)** and **(mp)** in Sec. C.

Instrumenting reads (RW and RR) Here we are to implement the two operations for reads: delaying a read $delay(R(w, r))$ and reading from the set, $flush(R(w, r))$. We delay a read by recording the memory address to be read from. Note that, given our program normalisation, our reads manifest as assignments to local variables. For a local variable r1, we delay the read of x as follows:

```
r1 = x;    →    if(*)
                  rdelay[&r1] = &x;
                else
                  r1 = x;
```

For flushing the read, considerations analogous to the write case are made: we flush non-deterministically upon an actual read (then of r1) only, instead of every program point. The flush dereferences the address previously recorded:

```
r2 = r1;    →    if(rdelay[&r1] != NULL && *) {
                   r1 = *rdelay[&r1];
                   rdelay[&r1] = NULL;
                 }
                 r2 = r1;
```

We illustrate the write instrumentation with the transformation of **lb** in Sec. C.

Instrumenting communications (caches) Architectures like Power and ARM allow rfe to be reordered. If we consider the program **(iriw+dps)** in Fig. 4.2(a), the read (a) Rx1 can read the value written by the write (e) Wx1 even though (d) Rx0 reads 0 for the same address. This is due to the fact that the write (e) was buffered, the thread of (e) could read directly into this buffer but the thread of (d) could not.

```
r1 = x;    →    if(length(&buff[&x]) != 0 && *) {
                  delay[&x] = TRUE;
                  delay_tmp[&x] = x;
                  x = last(&buff[&x], thread_ID_1);
                }
                r1 = x;
                if(delay[&x] && *) {
                  x = delay_tmp[&x];
                  delay[&x] = FALSE;
                }
```

Given a pair of events in rfe, we do not instrument only the first event, as we did for the previous reorderings. We instrument the write and the read. The write is instrumented in the exact same way we instrumented it earlier for poWR and poWW—that is, we delay the write by placing it into a buffer. Since the read is supposed to read from the buffer directly, we save the current value in the memory location of the variable about to be read, then take the last value in the buffer and assign it to the memory. Note that last is identical to take, except that it does not modify the buffer afterwards. Then the read will read from the value that exists in buffer—that we have here temporarily affected to the variable itself in memory. After the read, we restore the (previous) value in the memory and leave the buffer as it is. This strategy simulates the direct read into the buffer, that can be non-visible to the other threads.

We illustrate the communication instrumentation with the transformation of **(iriw+dps)** in Fig. 4.2.

Note that, for **(iriw+dps)**, instrumenting a read-read pair would have have been sufficient to make the final state specific to ARM or Power observable under SC. Indeed, even though we take into account dependencies at detection, we do not insert any mechanism in the instrumentation that would maintain the dependencies at runtime. Arguably, this allows us to use the read instrumentation instead of the communication instrumentation: the rfe would be followed in the thread of the read by either a poRR or a poRW. If a fence (full or lightweight) is placed between those last two events, then the communication is forbidden by cumulativity. If there is none, then those events could be reordered, and the instrumentation of the critical does therefore not require instrumenting the communication. If there is a fence, we would detect it but could still ignore it in the instrumentation: the last case still applies. In the implementation, we however kept the communication instrumentation for two reasons. First, we are not sure that the read instrumentation is cheaper than the communication instrumentation (we gave them the same weight in

the weighted selection of Sec. 4.2.2). Read instrumentation requires the introduction of a pointer in the implementation, whereas pointers can be avoided for the communication instrumentation—the difficulty of the instrumented program thus depends on the model-checker ability to handle efficiently pointers. Second, we might in the future integrate dependencies in the instrumentation.

4.2.2 Weighted selection of unsafe pairs

Above, we selected an arbitrary unsafe pair per cycle, as this suffices to reveal all weak-memory effects (cf Sec. 4.1.3). We do observe, however, that the choice of pairs has a strong effect on verification time. We thus assign an empirically devised cost cost to candidate pairs. With our implementation, we chose cost(poW*)=1 (pairs in program order where the first event is a write), cost(poRW)=2 (read-write pairs in program order), cost(rfe)=2 (write-read pairs on different threads), cost(poRR)=3 (read-read pairs in program order). Given a set of critical cycles $C = \{C_1, \ldots, C_n\}$ and the set of delays $delays(C_i)$ for each cycle C_i, we solve the integer linear program of Fig. 4.9 using GLPK. Our experiments in Sec. 4.3.1 show that this encoding yields a speedup of 26% over all architectures with an SC bounded model-checker.

Input: the set of cycles $\{C_1, \ldots, C_n\}$, the delays $delays(C_i)$ of each cycle C_i
Problem: minimise $\sum_{e \in \bigcup_{1 \leq i \leq n} delays(C_i)} b_i \times \mathsf{cost}(e)$
Constraints: $\forall 1 \leq i \leq n, \sum_{e \in delays(C_i)} b_e \geq 1$ (ensures soundness)
where
e is a pair to potentially instrument,
b_e is a Boolean variable stating whether we instrument e,
and cost() is the cost of an instrumentation.
Output: the set of b_e set to 1 by the solver, stating which pairs e to instrument

Figure 4.9: Integer linear programming problem to choose the pairs to instrument

We explain in detail some ideas towards a formal soundness arguments for the static construction in App. B.

4.2.3 Boundedness of the buffers and read set

Implementation In goto-instrument, we did not implement our buffers and read sets as (bounded) C arrays, as few model-checkers would have been able to handle them. We used instead as many variables as needed, in addition to Boolean variables

keeping the state of the write buffer/read set. For each write that would be added to the buffer, we operate, if it is possible, a shift of the existing elements and place this write as the last write, keeping track of the thread that delayed it. In goto-instrument, the variables simulating the buffer in the instrumented program are implemented for up to two writes per shared variable. The function take—which flushes the buffer—follows the conditions detailed in the table below:

before x = take(&buff[&x],thd[i])			after x = take(&buff[&x],thd[i]))		
last in buffer	previous in buffer	memory	last in buffer	previous in buffer	memory
–	–	m	–	–	m
a (thd_i)	–	m	–	–	a
a (thd_j)	–	m	a (thd_j)	–	m
			–	–	a
a (thd_j)	b (thd_j)	m	a (thd_j)	b (thd_j)	m
			a (thd_j)	–	b
			–	–	a
a (thd_j)	b (thd_i)	m	a (thd_j)	–	b
			–	–	a
a (thd_i)	b (thd_j)	m	–	–	a
a (thd_i)	b (thd_i)	m	–	–	a

To implement the delay set, we add one pointer (to the addresses of shared memory) per local variable that would be involved in a critical cycle where a read could be potentially delayed. As we will see in the next paragraph, there is, however, no need to keep track of more than one read in the delaying set per local variable. Our read delay instrumentation is unbounded.

(Un-)boundedness for detection and instrumentation Our instrumentation strategy involves two steps: finding the potential critical cycles in the program, and instrumenting the program so that the buffering and caching effects would be revealed to an SC analyser. The detection of cycles is an analysis of the relations between accesses to shared memory, based on an axiomatic model that do not involve bounds. The detection is unbounded by nature. The instrumentation, however, relies on the insertion of actual buffers and read set in the program that simulate a weak memory operational semantics.

The strategy employed for the read set does not involve a bound. Let us consider a variant of the load buffer lb, as implemented in Fig. 4.10, with multiple reads that

need to be reordered with the following write in the left thread in order to violate the assertion. We do not need a "buffer" of reads: if two reads read from the same shared variable but are flushed at different time, then these delayed reads are supported by the additional pointers associated to the two (distinct) local variables to which the value read should be assigned (for expressions, temporary variables are introduced). If the values read are assigned to the same local variable, the local memory rules apply and the second read will overwrite the first one, making the first assignment invisible.

```
void thd_1() {        void thd_2() {
  r_1 = x;              r_0 = y;
  r_2 = x;              asm("sync");
  r_3 = x;              x = 1;
  y = 1;                x = 2;
}                       x = 3;
                      }

assert ( ! (r_0==1 && r_1==1
  && r_2==2 && r_3==3) );
```

Outcomes for r_0, r_1, r_2 and r_3:

$SC, ..., RMO$	$Power, ARM$
(0, a, b, c), where a, b and c in $\{0, 1, 2, 3\}$ and $a \leq b \leq c$; (1, 0, 0, 0)	(0, a, b, c) and **(1, a, b, c)**, where a, b and c in $\{0, 1, 2, 3\}$ and $a \leq b \leq c$

Figure 4.10: A variant of load-buffering with several reads to the same shared variable.

The write buffer involved in the instrumentation is, however, bounded. Since a critical cycle involves a maximum of two events per thread and there is one buffer per variable—and not a global buffer for all the writes—most of the instrumentation only require a buffer of size 1 to simulate a write-write or write-read reordering. In our experiments, this was sufficient for all the benchmarks.

There can nevertheless be situations where several cycles would relate a succession of writes and reads, and one particular state only reachable under weak memory consistency would require a buffer of size n to be detected by an SC model-checker. Let us consider the short program of Fig. 4.11. It is a variant of store-buffering, where writes and reads can be reordered. In this example, one needs to have all the writes on the left thread to be reordered with the read so that we can observe a violation of the assertion. The cycle detection finds all the critical cycles, but the instrumentation must simulate a write buffer of size n to make the violating state discoverable by an analyser afterwards.

Property 12. ***(sb^n+fence)*** *(in Fig. 4.11) needs to be instrumented with a write buffer of size n or greater if we want to observe the final state specific to TSO (or weaker), namely r_0$= 0 \wedge$r_1$= 0 \wedge \ldots \wedge$r_n$= 0$, with an interleaving.*

```
void thd_1() {         void thd_2() {
  x = 1;                 y = 1;
  ...                    asm("mfence");
  x = n;                 r_1 = x;
  r_0 = y;               ...
}                        r_n = x;
                       }

  assert( ! (r_0==0 && r_1==0
      && ... && r_n==0) );
```

Outcomes for r_0, r_1 ... r_n:

SC	$TSO, ..., Power$
(1, 0, ..., 0)	(1, 0, ..., 0)
(1, 0, ..., 0, v_j, ..., v_n)	(1, 0, ..., 0, v_j, ..., v_n)
(1, v_1, ..., v_n)	(1, v_1, ..., v_n)
(0, v_1, ..., v_n)	**(0, 0, ..., 0)**
where $\forall j,\ v_j \geq 1$	(0, 0, ..., 0, v_j, ..., v_n)
and $\forall i \leq k,\ v_i \leq v_k$	(0, v_1, ..., v_n)
	where $\forall j,\ v_j \geq 1$
	and $\forall i \leq k,\ v_i \leq v_k$

Figure 4.11: A variant of store-buffering where a n-buffer is needed to violate the assertion. (**sb**^{n}**+fence**)

Proof. We want to show that an interleaving would require a buffer of size n or greater to observe r_0= $0 \wedge$*r_1=* $0 \wedge \ldots \wedge$*r_n=* 0*. r_0=y must read 0. This has to happen before y=1;asm("mfence");, since the fence will flush the buffer of y, writing 1 in the memory of* y*. This implies that the* $Ry0$ *must happen before all the reads following the fence. We also need the r_1, ..., r_n in thread 2 to all read 0. Each of these reads must happen before any write in thread 1. Any linearisation* l *of this set of constraints will start with* $Ry0; Wy1; fence; Rx0; ...; Rx0;$ *followed by the writes in thread 1. Any interleaving reaching the (0, ..., 0) state thus needs to have this prefix, and needs to reorder all the writes on thread 1 with the final read. This requires a buffer of size n or more, as otherwise some writes would touch the memory and invalidate the possibility of reading 0 again from x in thread 2.* □

One might argue that, in the example of Fig. 4.11, we do not need to keep track of all the values written in x for reaching the (0, ..., 0) case specific to TSO. However, we can construct an example that would require reading each of these values after a store-buffering. We construct an example in Fig. 4.12, where we need a buffer to show operationally that r_0= $0 \wedge$r_1= $0 \wedge \ldots \wedge$r_n= 0—that is, to pass the write-read reordering— and the $\geq n$-buffer to store the writes of thread 1 and read their effects in thread 2: s_1= $1 \wedge \ldots \wedge$s_n= n.

These examples are artificial. An interesting question is which class of programs require a write buffer of size n, and it there are instances of such programs, for example in the Debian packages. This is left as future work.

If we were to insert unbounded write buffers, the verification problem solved by the analyser after instrumentation would become undecidable, as proved by Bouajjani

void thd_1() {	void thd_2() {
x = 1;	y = 1;
...	asm("mfence");
x = n;	r_1 = x;
r_0 = y;	...
}	r_n = x;
	s_1 = x;
	...
	s_n = x;
	}

```
assert ( ! (r_0==0 && r_1==0
&& ... && r_n==0 && s_1==1
&& ... && s_n==n) );
```

Outcomes for r_0, r_1 ... r_n, s_1 ... s_n:

SC	$TSO, ..., Power$
(1, 0, ..., 0)	(1, 0, ..., 0)
(1, 0, ..., 0, v_j, ..., v_{2n})	(1, 0, ..., 0, v_j, ..., v_{2n})
(1, v_1, ..., v_{2n})	(1, v_1, ..., v_{2n})
(0, v_1, ..., v_{2n})	(0, 0, ..., 0)
where $\forall j,\ v_j \geq 1$	**(0, 0, ..., 0, v_j, ..., v_{2n})**
and $\forall i \leq k,\ v_i \leq v_k$	(0, v_1, ..., v_{2n})
	where $\forall j,\ v_j \geq 1$
	and $\forall i \leq k,\ v_i \leq v_k$

Figure 4.12: A variant of **(sb^n+fence)** (Fig. 4.11) where we read from the n-buffer.

et al. in [ABBM12]. Abstractions over the (bounded) write buffers were suggested as an alternative in [KVY11, DMVY15].

Assertion to preserve soundness despite the buffer bound As the buffer is bounded (the current implementation of goto-instrument can put into buffer up to two writes), we insert an assertion checking that we do not exceed the maximum size of the buffer. We give a low priority to this assertion, since our main objective is to find bugs due to weak memory. If the instrumented program is proved correct despite this assertion, it means it is safe for weak memory. If this assertion is failed, but no other assertion can be violated, one needs to increase the bound of the buffer—in the same way as we perform loop unwinding for bounded model checking.

As expected, in case of unbounded loops, this reachability problem becomes undecidable, as under SC. The unbounded store-buffering of Fig. 4.13 gives an example in which we would need a buffer as large as the unwinding bound. It implements the store-buffering of Fig. 4.11, except that the number of writes is not known in advance and we cannot predict a bound. Note that, in addition to x and y, max is a variable shared between threads 1 and 2. Its purpose is only to set an end[9] to the iteration of thread 2—it is not an issue if thread 2 uses a former value of max.

[9] We could have actually replaced the bound j<max by a non-deterministic choice $*$, but this would just increase the state space of the local memory without bringing new behaviours that would involve new environments in the shared memory state space.

```
void thd_1() {                      void thd_2() {
  for (i = 0; *; max = i++)           y = 1;
    x = i+1;                          for (j = 0; j<max; ++j)
  s = y;                                r[j] = x;
}                                   }

          for(k = 0; k<=max; sum += r[k++]);
          assert( ! (s==0 && sum==0) );
```

Figure 4.13: A variant of store-buffering with an unbounded loop.

4.2.4 Trade-off between detection and instrumentation precisions

Our instrumentation technique is based on two steps:

- the detection of the critical cycles, a static technique relying on an axiomatic semantics;
- the addition of buffers and read sets to the program, following an operational model, so that additional behaviours are revealed dynamically (i.e., during the analysis of an abstract interpreter, symbolic execution engine or model-checker).

These two steps are complementary in terms of precision. Fig. 4.14 displays the trade-off between the precision of these two steps. A very precise instrumentation with a completely imprecise cycle detection would lead to instrumenting all the shared variables—as suggested in [ABP11] and implemented in goto-instrument with the option –cav11. At the other extremity of the spectrum, a very precise cycle detection—value-sensitive, for instance—coupled with a simple instrumentation that would deterministically bufferise specific writes without constraints, would require splitting all the cases, based on the values, and introduce a combinatorial explosion of paths. At both of these extremities, we can expect an important increase of the size of the instrumented program—in terms of number of variables or state space—which makes the verification task at the end of the chain significantly harder.

Placing one's tool in the middle of the spectrum is not only reducing potentially the quantity and complexity of instrumentation. It also allows us to consider some specific aspects of the weak memory behaviours under either the axiomatic model (at the detection) or under the operational model (at the instrumentation). Properties relating variables would for example be more conveniently checked statically under the

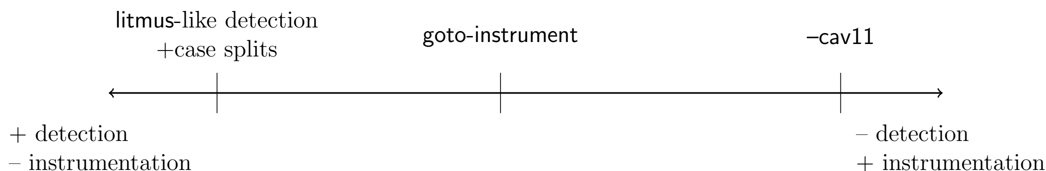


Figure 4.14: Trade-off between static detection precision and instrumentation precision.

axiomatic model, whereas non-relational properties could be checked more accurately directly at "runtime", that is, in the instrumentation.

For example, we take into account dependencies in the AEG and at the cycle detection, since it is an additional relation that is reasonable to compute statically and has only a limited impact on the cost of detecting critical cycles. It can nevertheless decrease significantly the number of cycles under Power and ARM. Implementing dependencies between e.g. reads, at the instrumentation level would have required more work. In particular, the pointers associated to local variables involved in critical cycles would not have been sufficient to model precisely the read set with dependencies—more complex structures relating previous read events would have been needed.

Another example on the side of the instrumentation concerns the writes. In the AEG, we completely ignored the values the writes could assign to places in memory. We abstracted the values in order to avoid the calculation of the memory environment step-by-step—which would have required an operational semantics handling weak memory so that the tool would remain sound. The instrumentation however does place some assignments to memory, leading at verification time to the propagation of actual values for the variables.

In our approach, we kept the detection of weak memory patterns (the critical cycles) and the execution/verification of the program to simulate weak memory behaviours completely separate. Introducing symbolic execution at the static cycle detection time would increase the precision, reducing the critical cycles and thus the instrumentation—possibly also introducing some branchings based on specific values inducing some weak memory behaviours. This would be at the cost of the scalability of the static analysis part of the technique, prior to the instrumentation itself. The operational semantics that would have to be put in place at this detection time would also be complex for weak architecture like Power or ARM. The benefit of this compared to using the same operational semantics directly onto the program would be uncertain.

4.3 Experiments

We carried out our method and measured its cost using 8 tools, as summarised in Fig. 4.15. We considered 5 ANSI-C model checkers: a bounded model checker based on CBMC [AKT13] ; SatAbs [CKSY05], a verifier based on predicate abstraction, using Boom as the model checker for the Boolean program; ESBMC [CF11], a bounded model checker; Threader [GPR11], a thread-modular verifier; and Q Program Verifier[10] (also known as Poirot), which implements a context-bounded translation to sequential programs and checks it with Corral [LQL12]. These tools cover a broad spectrum of symbolic algorithms for verifying SC programs. We also experimented with Blender [KVY11], CheckFence [BAM], and MMChecker [HR06]. We ran our experiments on Linux 2.6.32 64-bit machines at 3.07 GHz (only Poirot was run on a Windows system). In this section, we give an analysis summarising the results. All the results are available at `http://www.cprover.org/wmm/esop13`.

Disclaimer: The results reported in the dissertation were those obtained with the tools listed in Sec. E.3, in early 2013 for goto-instrument –mm (published in [AKNT13]) and in 2014 for musketeer (published in [AKNP14]). goto-instrument –mm and musketeer were based on CProver 4.3, released in February 2013. Results are also available in detail on the webpage `www.cprover.org/esop13` and `www.cprover.org/cav14`.

At the time of writing, we worked at porting goto-instrument –mm and musketeer to the trunk version of CProver, posterior to the release 4.9, in order to get it integrated to CProver for the release 5.0. This work is finished (both tools are already accessible from CProver's SVN[11]) and we are aiming to reproduce all the results we reported in the dissertation. Most of the results can be obtained again; some of the experiments still need some adjustments to reproduce some results at the time of writing. The installation process of the new tools is simpler, and allows us to make direct use of the new format of goto-binaries used by goto-cc/cbmc.

4.3.1 Validation

First, we systematically validate our setup using 555 litmus tests exposing weak memory artefacts (e.g., instruction reordering, store buffering and write atomicity relaxation) in isolation. The diy tool automatically generates x86, Power and ARM assembly programs implementing an idiom that cannot be reached on SC, but can be

[10] `http://research.microsoft.com/en-us/projects/verifierq/`
[11] http://www.cprover.org/svn/cbmc/

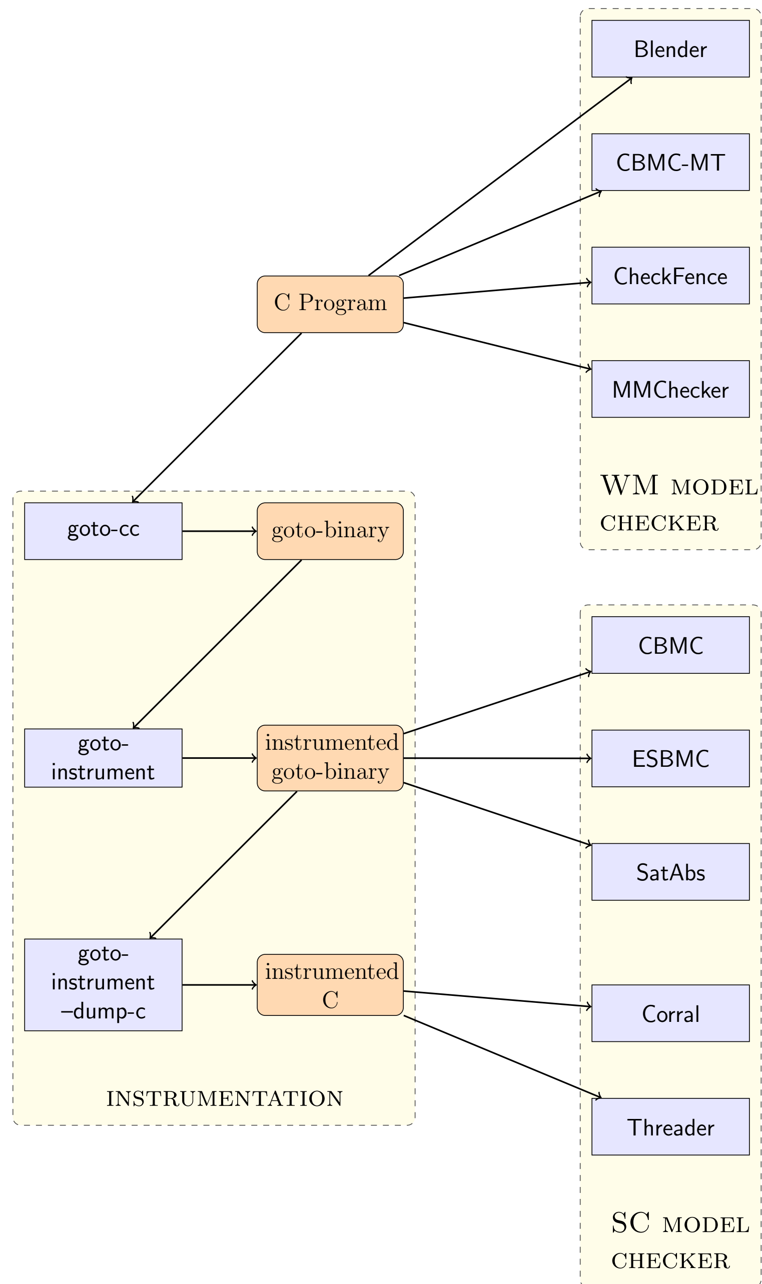


Figure 4.15: Tools used for the experiments of Sec. 4.3.

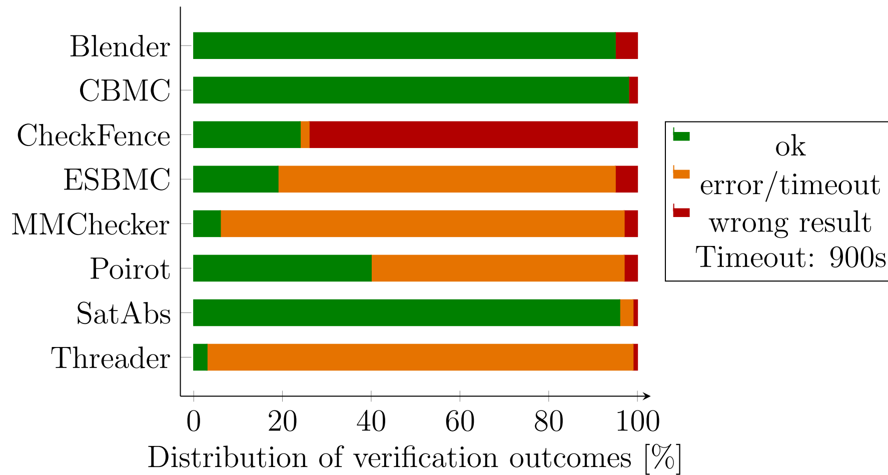


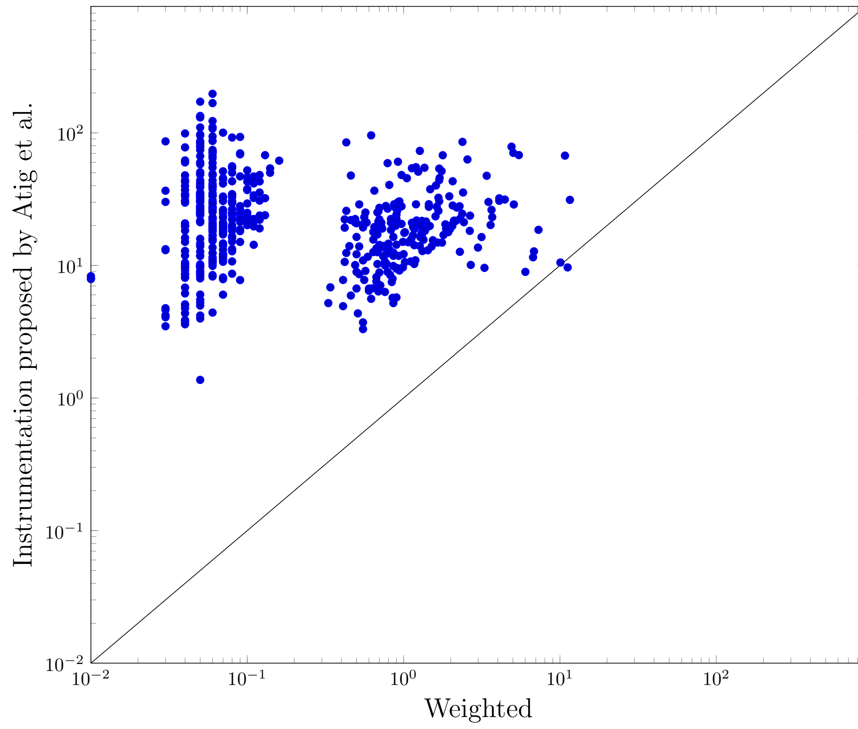
Figure 4.16: All tools on all litmus tests and models.

reached on a given model. For example, **(sb)** (Fig. 4.4(a)) exhibits store buffering, thus the final state can be reached on any weak model, from TSO to Power.

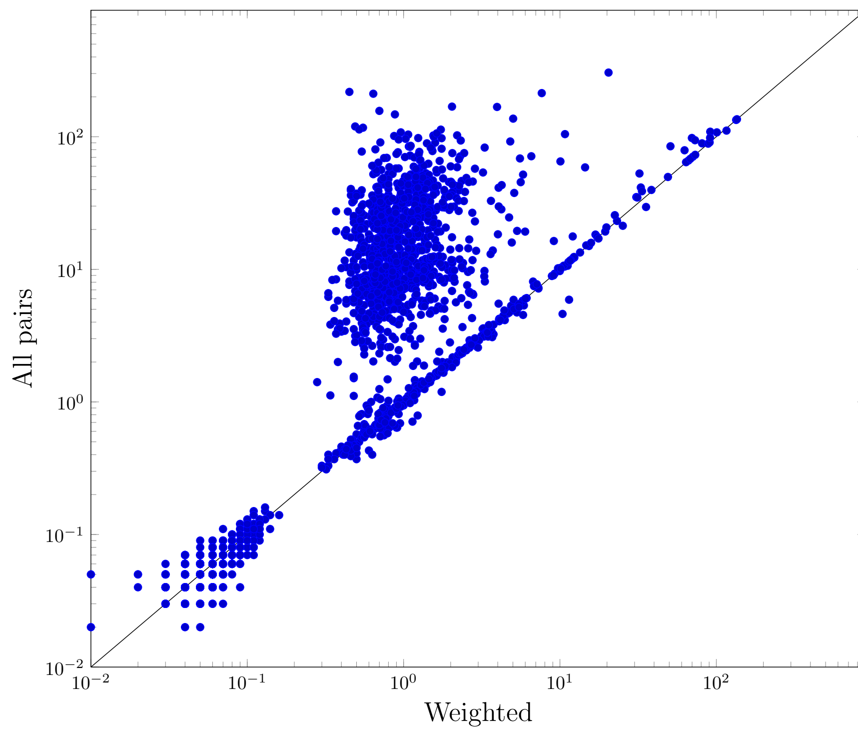
Each litmus test comes with an assertion that models the SC violation exercised by the test, e.g., the outcomes of Fig. 4.4(a) and 4.5(a). Thus, verifying a litmus test amounts to checking whether the model under scrutiny can reach the specified outcome. We then convert these tests automatically into C code, leading to programs of 48 lines on average, involving 2 to 4 threads.

These examples provide assurance that we soundly implement the theory of Sec. 4.1.1: we verify each test w.r.t. SC, i.e., without transformation, then w.r.t. TSO, PSO, RMO, and Power. Despite the tests being small, they provide challenging concurrent idioms to verify. Fig. 4.16 compares the tools on all tests and models. Most tools, with the exception of Blender, CBMC and SatAbs, time out or give wrong results on a vast majority of tests. Blender only expectedly fails on tests involving `lwsync` fences; CBMC and SatAbs return spurious results in 1.5% of the tests, caused by the over-approximation in the implementation of our instrumentation.

Fig. 4.17 compares the verification time using CBMC over all litmus families (e.g. rfe tests exercise store atomicity, podwr tests exercise the write-read reordering) for different instrumentation options. First, with the restriction to TSO, Fig. 4.17(a) compares the instrumentation of all shared memory accesses proposed in [ABP11] to the weighted transformation (Sec. 4.2.2). On average, we observe a more than 300-fold speedup in verification time. In addition, the reduced instrumentation also yields 246 fewer spurious results. We also quantify the specific benefit of the weighted selection of pairs in Fig. 4.17(b). We compare the cost of the instrumentation of all



(a) All accesses [ABP11] vs. weighted selection



(b) All pairs vs. weighted selection

Figure 4.17: Comparison of verification times of CBMC (in seconds, with logarithmic scale) for different instrumentations.

```
1  #define WORKERS 2
2  volatile _Bool latch[WORKERS];
3  volatile _Bool flag[WORKERS];
4  void worker(int i)
5  { while(!latch[i]);
6    for (;;)
7    { assert(!latch[i] || flag[i]);
8      latch[i] = 0;
9      if(flag[i])
10     { flag[i] = 0;
11       flag[(i+1)%WORKERS] = 1;
12       latch[(i+1)%WORKERS] = 1; }
13     while(!latch[i]);   } }
```

Figure 4.18: Token passing in `pgsql.c`.

pairs on critical cycles with that of the weighted transformation (Sec. 4.2.2) for all models, tools and tests. The average speedup over all models and tests is still more than one order of magnitude. We give the detailed results for all experiments online.

We also verified several TSO examples that have been used in the literature. Note that these examples in fact only exhibit idioms already covered by our litmus tests (e.g. Dekker corresponds to the **(sb)** test of Fig. 2.15). Furthermore, we applied the instrumentation to code taken from the Read-Copy-Update algorithm in the Linux kernel and scheduling code in the Apache HTTP server, as well as industrial code from IBM. We observe that the instrumentation tool completes even on such code of up to 28,000 lines in less than 1 second, and in 32 seconds on IBM's code. We now study one real-life example in detail, an excerpt of the relational database software PostgreSQL.

4.3.2 A case study: worker synchronisation in PostgreSQL

Mid 2011, PostgreSQL developers observed that a regression test occasionally failed on a multi-core PowerPC system.[12] The test implements a protocol passing a token in a ring of processes. Further analysis drew the attention to an interprocess signalling mechanism. It turned out that the code had already been subject to an inconclusive discussion in late 2010.[13]

The code in Fig. 4.18 is an inlined version of the problematic code, with an additional assertion in line 7. Each element of the array "latch" is a Boolean variable

[12] http://archives.postgresql.org/pgsql-hackers/2011-08/msg00330.php
[13] http://archives.postgresql.org/pgsql-hackers/2010-11/msg01575.php

stored in shared memory to facilitate interprocess communication. Each working process waits to have its latch set and then expects to have work to do (from line 9 onwards). Here, the work consists of passing around a token via the array "flag". Once the process is done with its work, it passes the token on (line 11), and sets the latch of the process the token was passed to (line 12).

Starvation seemingly cannot occur: when a process is woken up, it has work to do (has the token). Yet, the PostgreSQL developers observed that the wait in line 13 (which in the original code is bounded in time) would time out, thus signalling starvation of the ring of processes. The developers identified the memory model of the platform as possible culprit: it was assumed that the processor would at times delay the write in line 11 until after the latch had been set.

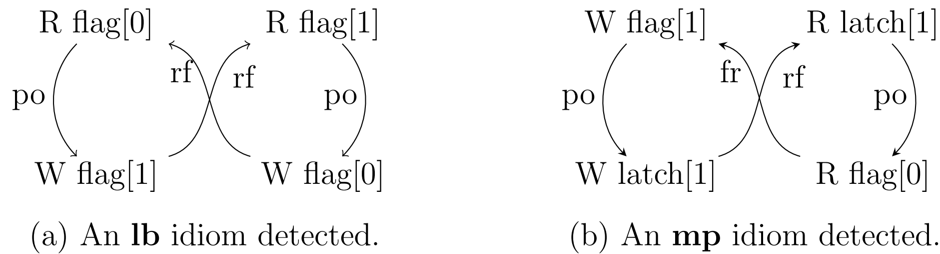
We transform the code of Fig. 4.18 for two workers under Power. The event graphs show two idioms: **(lb)** (load buffering) and **(mp)** (message passing), in Fig. 4.19. The code fragments on the left-hand side give the corresponding line numbers in Fig. 4.18.

The **(lb)** idiom contains the two *if* statements controlling the access to both critical sections. Since the **(lb)** idiom is yet unimplemented by Power machines (despite being allowed by the architecture [SSA$^+$11]), we believe that this is not the bug observed by the PostgreSQL developers. Yet, it might lead to actual bugs on future machines.

In contrast, the **(mp)** case is commonly observed on Power machines (e.g. 1.7G/167G on Power 7 [SSA$^+$11]). The **(mp)** case arises in the PostgreSQL code by the combination of some writes in the critical section of the first worker, and the access to the critical section of the second worker; the relevant code lines are in Fig. 4.19(b).

We first check the fully transformed code with SatAbs. After 21.34 seconds, SatAbs provides a counterexample (given online), where we first execute the first worker up to line 13. All accesses are w.r.t. memory, except at lines 11 and 12, where the values 0 and 1 are stored into the buffers of flag[0] and flag[1]. Then the second worker starts, reading the updated value 1 of latch[1]. It exits the blocking while (line 5) and reaches the assertion. Here, latch[1] still holds 1, and flag[1] still holds 0, as Worker 0 has not yet flushed the write waiting in its buffer. Thus, the condition of the *if* is not true, the critical section is skipped, and the program arrives at line 13, without having authorised the next worker to enter the critical section, and loops forever.

As **(mp)** can arise on Power e.g., because of non-atomic writes, we know by Sec. 4.1.3 that we only need to transform one rfe pair of the cycle, and relaunch the



(a) An **lb** idiom detected. (b) An **mp** idiom detected.

Figure 4.19: **(lb)** and **(mp)** idioms detected in `pgsql.c`.

verification. SatAbs spends 1.29 seconds to check it (and finds a counterexample, as previously).

PostgreSQL developers discussed fixes, but only committed comments to the code base, as it remained unclear whether the intended fixes were appropriate. We proposed a provably correct patch solving both **(lb)** and **(mp)**. After discussion with the developers[14], we improved it to meet the developers' desire to maintain the current API. The final patch introduces two `lwsync` barriers: after line 8 and before line 12.

4.4 Novelty of our contribution w.r.t. the related work

In this chapter, we focussed on safety verification, that is, the error state/violated assertion reachability—and not on the SC-robustness restoration, in which all the non-SC behaviours need to be prevented. We address the latter in Chap. 5. The reachability problem is non-primitive recursive for TSO [ABBM10]. Even if loops are bounded, it is still undecidable if read/write or read/read pairs can be reordered, as in RMO-like models [ABBM10]. Forbidding *causal loops* restores decidability; relaxing write atomicity makes the problem undecidable again [ABBM12].

Existing solutions use various bounds over the objects of the model [ABP11, KVY10], over-approximate the possible program behaviours [KVY11, JYKS12], or relinquish termination [LW11]. For TSO, [AAC+12] presents a sound and complete solution. We present a provably sound method that allows to lift any SC method or tool to a large spectrum of weak memory models, ranging from x86 to Power. We build an operational model; [OSS09] presented such a model, but theirs is restricted to TSO. Given the undecidability of the problem, we cannot provide completeness, as we focus on soundness. We do not use any bound in our theoretical model (Sec. 4.1.1),

[14] http://archives.postgresql.org/pgsql-hackers/2012-03/msg01506.php

but our implementation uses finite buffers with assertions checking that we do not exceed the bound (Sec. 4.2).

Our approach also reduces the amount of instrumentation in a provably sound manner. Unlike [ABP11], we only instrument selected shared memory accesses. For TSO this would follow immediately from [BMM11], but we generalise to models such as Power.

4.5 Summary

In this chapter, we first explained how to instrument an event structure as defined in Chap. 2 so that it would permit all the executions valid on a targeted architecture. This consists of placing non-deterministic writes and reads to buffers and queues in the delays of the critical cycles that could exist in the event structure. We exploited the results of Chap. 3 to extend this instrumentation to actual C programs. We first detect static critical cycles that might occur in the code, using the static analysis described in Chap. 3. Then we insert some static non-deterministic delays of writes and reads to memory, with the help of additional queues and buffers. We finally optimise the global placement of these instrumentations with an integer linear program in order to minimuse the runtime overhead induced by the instrumentation.

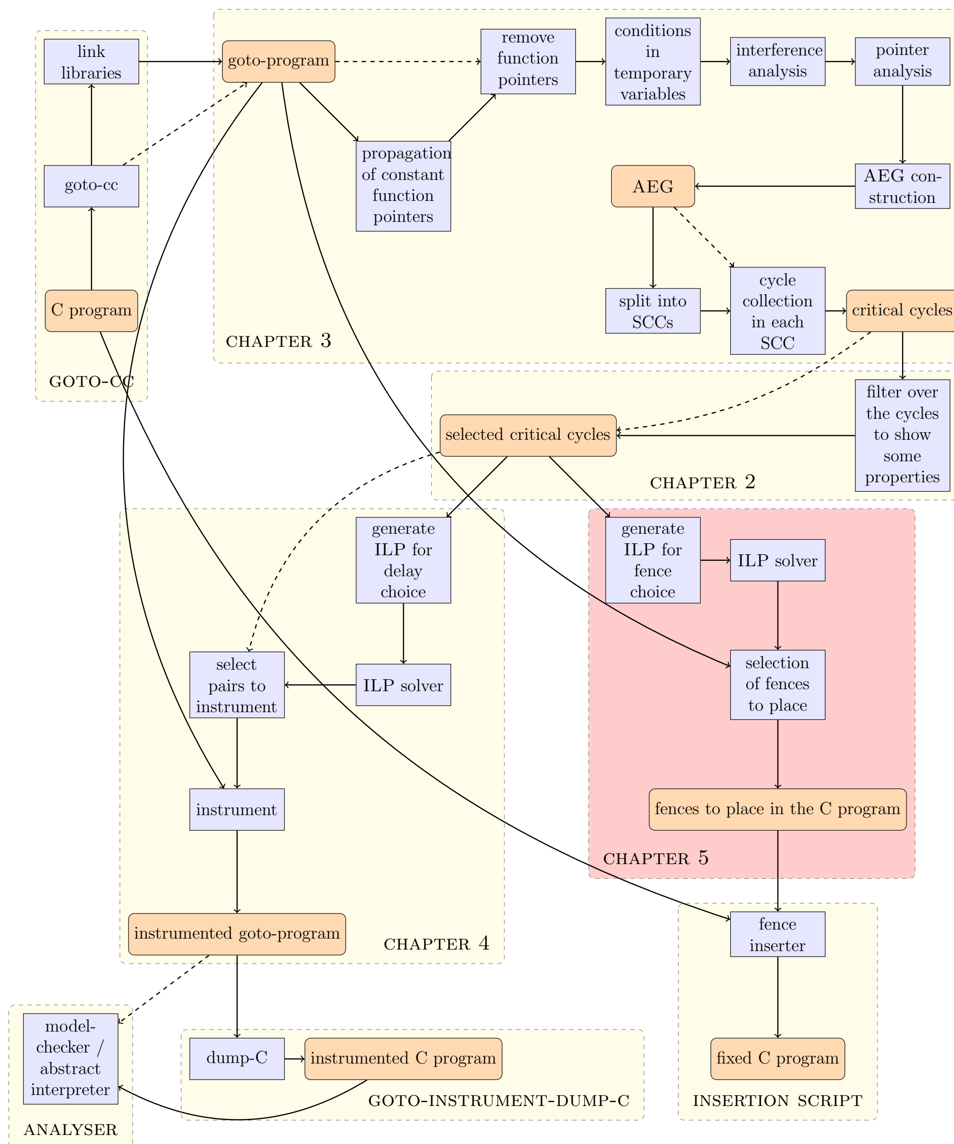
We implemented this approach in goto-instrument and validated it over 500+ Litmus tests[15], that are short examples that test corner-cases of multi-core processor concurrency. We then experimented it on some classic algorithms from the literature—like Peterson's mutual exclusion algorithm—and we reproduced a bug of weak memory detected[16] in Pgsql.

[15] `http://diy.inria.fr/doc/litmus.html`

[16] http://www.postgresql.org/message-id/24241.1312739269@sss.pgh.pa.us

Chapter 5

Synchronisation Synthesis over Weak Memory



In this chapter, we introduce a synchronisation synthesis technique to restore primarily sequential consistency. As we explained in Chap. 2, reasoning with interleavings in mind is simpler than with all the processor specifics. Restoring sequential consistency might slightly worsen the performance of the final program, but it can ensure a better readability, understanding and thus simplify the maintenance of the program and its correctness. It does not mean that all the accesses to shared memory will happen in order. Some access reorderings that do not impact the semantics of the program are still allowed (as opposed to under strict consistency).

The technique that we describe in this chapter makes use of the critical cycle detection introduced in Chap. 3. Once we have collected the cycles, we construct an integer linear program (ILP) that implicitly encodes the semantics of the synchronisations preventing reorderings. The solution of this ILP optimises the placement and type of synchronisations inserted so that the program becomes sequentially consistent—hence, any execution of this program can be modelled as an interleaving.

This technique is not limited to the restoration of sequential consistency. One can restore to a memory model weaker than SC, such as x86 or PSO. Porting a program written for x86 to ARM would not necessarily require restoring SC. For instance, if the program is known to be correct for x86—and thus more efficient than under SC—then one may try to prevent only reorderings that would not be allowed under x86.

Plan of the chapter

In this chapter, we first explain the problem of synchronisation synthesis in the context of weak memory models. We then suggest an optimisation problem that would address it, and explain one ILP encoding that solves it. We detail how to introduce the synchronisation mechanisms in the program in practice, before measuring and analysing the impact of the inferred synchronisations over program runtime. We test the technique over a large set of benchmarks. We finally propose some alternative, more sophisticated encodings and show that, despite involving less variables, they actually have very little impact over the performance of the tool. We conclude with a comparison against the other existing techniques for synthesising memory synchronisations.

	SC	x86	Power
poWR	yes	mfence	sync
poWW	yes	yes	sync, lwsync
poRW[1]	yes	yes	sync, lwsync, dp
poRR	yes	yes	sync, lwsync, dp, branch;isync

Figure 5.1: ppo and fences per architecture

5.1 Introduction to optimised fence synthesis

Concurrent programs are hard to design and implement, especially when running on multiprocessor architectures. Multiprocessors allow more behaviours than SC. This has a dramatic effect on programmers, most of whom learned to program with SC. Fortunately, architectures provide special *fence* (or *barrier*) assembly instructions to prevent certain behaviours. Processors implementing Power and ARM architectures also ensure that accesses to shared memory that are in *dependency* cannot be reordered. We will explain the concept of dependency in the next paragraph. Both the questions of *where* and *how* to insert fences are contentious, as fences are architecture-specific and expensive.

Attempts at automatically placing fences include the Visual Studio 2013 compiler, which offers an option to guarantee acquire/release semantics. The C++11 standard provides an elaborate API for inter-thread communication, giving the programmer some control over which fences are used, and where. The use of such APIs might be a hard task, even for expert programmers. For example, Norris and Demsky reported a bug found in a published C11 implementation of a work-stealing queue [ND13].

We address here the question of how to *synthesise* fences, i.e. automatically place them in a program to enforce robustness/stability [BMM11, AM11], defined in Chap. 2.3. This ensures that when the program is run on a processor implementing a weaker memory consistency, no additional behaviours impacting the semantics of the program would be observed. This should lighten the programmer's burden. The fence synthesis tool is based on the AEGs of Chap. 3, which relies on the precise model of weak memory introduced by [Alg10].

Finally, considering models à la Power makes the problem significantly more difficult than Intel x86, that offers only one fence (mfence). Power offers a variety of synchronisation: fences (e.g. sync and lwsync), or dependencies (address, data or control). This diversity makes the optimisation more subtle: one cannot simply minimise the number of fences, but rather has to consider the costs of the different synchronisation mechanisms. It might be cheaper to use one full fence than four dependencies.

We summarise in Fig. 5.1 the effect of the fences for x86 and Power. An mfence under x86 will prevent the write-read reorderings. The other orders are anyway enforced by the architecture. The memory consistency of Power is significantly weaker than x86's one, and all the orders without dependencies can be relaxed. A full fence sync is available, but isync and lwsync should be used whenever possible as they are cheaper than sync. The dependencies between two events mean that these two events either manipulate the same data or address register (data or address dependency) or they are conditioned by a conditional control. Processors implementing Power will enforce this order, i.e., they will not allow the reorderings of events in dependencies. Dependency is a notion from the hardware, and the compiler simply ignores it, as we discuss in Sec. 5.3. We provide in the same section a way to enforce[2] the dependency from the C program down to the machine code.

ARM fences are similar to Power with the following mapping: the isb is equivalent to isync, the dmb and dsb are full fences like sync, and dependencies are similarly preserved. There is however no lightweight fence lwsync available.

Axiomatic approach In verification, models commonly adopt an operational style, where an execution is an interleaving of transitions accessing the memory (as in strict consistency). To address weaker architectures, the models are augmented with buffers and queues that implement the features of the hardware. Similarly, a good fraction of the fence synthesis methods, e.g. [LW13, KVY10, KVY11, LNP+12, AAC+13, BDM13], rely on operational models to describe executions of programs.

Thus, methods using operational models inherit the limitations of methods based on interleavings, *e.g.* the *"severely limited scalability"*, as [LNP+12] puts it. Indeed, none of them scale to programs with more than a few hundred lines of code, due to the very large number of executions a program can have. Another impediment to scalability is that these methods establish if there is a need for fences by exploring the executions of a program one by one.

[1] A conditional branching would also prevent a poRW reordering. However, we do not synthesise artificial conditional branchings, as we observed that GCC was simplifying it in most of the cases, even with -O0—thus allowing the unwanted poRW reordering.

[2] Daniel Poetzl noted that even though we force the compiler to respect the "hardware" orders that we impose, it might itself reorder accesses to shared memory for non-volatile declared variables. Our study is restricted to the hardware concurrency—we assume that the compiler do not reorder memory accesses in the assembly code it generates. As noted by Daniel Kroening, this might however constitute a memory model by itself, comparable to the one we use here, with the potential of reusing the same techniques and tools. We will briefly discuss this opportunity in the further work section of Chap. 6.

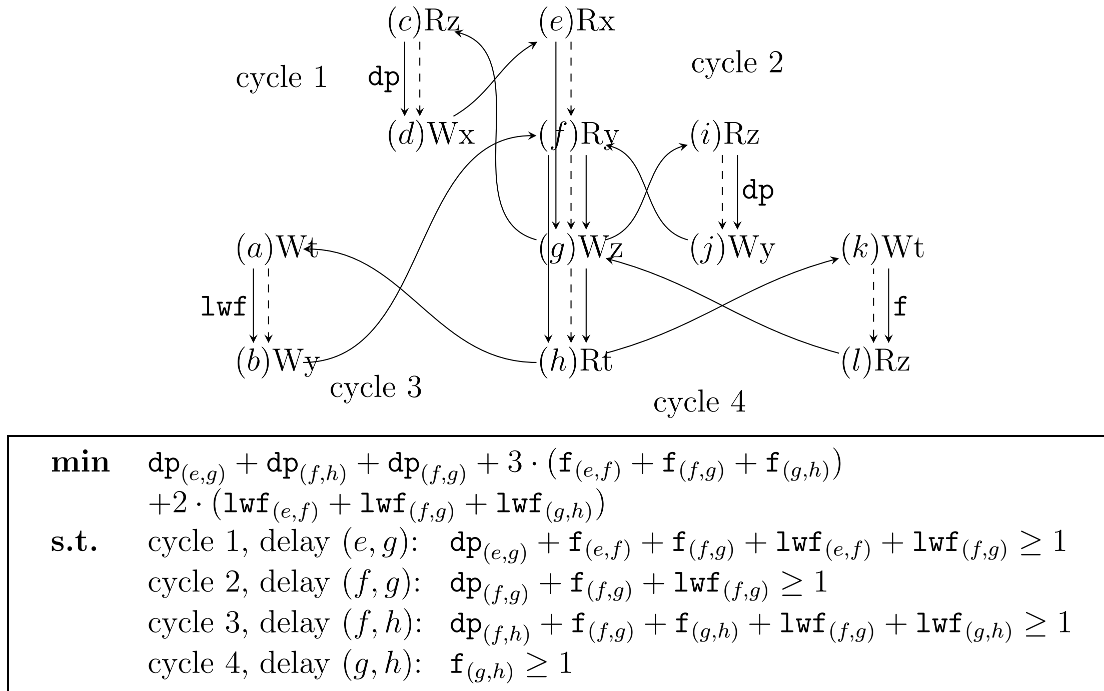


Figure 5.2: Example of resolution with between

In this approach, we rely on the AEG that we computed in Chap. 3 to find all the critical cycles and exploit this (static) knowledge to determine where to place fences and which type of fences to place without enumerating the combinations of fences that would satisfy the problem—that is, restoring SC.

5.2 Synthesis

In Fig. 5.2, we have an AEG with five threads: $\{a, b\}, \{c, d\}$, $\{e, f, g, h\}$, $\{i, j\}$ and $\{k, l\}$. Each node is an abstract event computed as in the previous section. The dashed edges represent the po_s between abstract events in the same thread. The full lines represent the edges involved in a cycle. Thus the AEG of Fig. 5.2 has four potential critical cycles. We derive the set of constraints in a process we define later in this section. We now have a set of cycles to forbid by placing fences. Moreover, we want to optimise the placement of the fences.

Challenges If there is only one type of fence (as in TSO, which only features mfence), optimising only consists of placing a minimal amount of fences to forbid as many cycles as possible. For example, placing a full fence sync between f and g in Fig. 5.2 might forbid cycles 1, 2 and 3 under Power, whereas placing it somewhere else might forbid at best two amongst them.

Input: AEG ($\mathbb{E}_s$,po_s,cmp) and potential critical cycles $C = \{C_1, ..., C_n\}$
Problem: minimise $\sum_{(l,\mathsf{t}) \in \mathsf{potential\text{-}places}(C)} \mathsf{t}_l \times \mathsf{cost}(\mathsf{t})$
Constraints: for all $d \in \text{delays}(C)$
(* for TSO, PSO, RMO, Power *)
if $d \in \mathsf{poWR}$ then $\sum_{e \in \mathsf{between}(d)} \mathsf{f}_e \geq 1$
if $d \in \mathsf{poWW}$ then $\sum_{e \in \mathsf{between}(d)} \mathsf{f}_e + \mathsf{lwf}_e \geq 1$
if $d \in \mathsf{poRW}$ then $\mathsf{dp}_d + \sum_{e \in \mathsf{between}(d)} \mathsf{f}_e + \mathsf{lwf}_e \geq 1$
if $d \in \mathsf{poRR}$ then $\mathsf{dp}_d + \sum_{e \in \mathsf{between}(d)} \mathsf{f}_e + \mathsf{lwf}_e \quad + \sum_{e \in \mathsf{ctrl}(d)} \mathsf{cf}_e \geq 1$
(* for Power *)
if $d \in cmp$ then $\sum_{e \in \mathsf{cumul}(d)} \mathsf{f}_e + \sum_{e \in \mathsf{cumul}(d) \cap \neg \mathsf{poWR} \cap \neg \mathsf{poRW}} \mathsf{lwf}_e \geq 1$
Output: the set $\mathsf{actual\text{-}places}(C)$ of pairs (l, t) s.t. t_l is set to 1 in the ILP solution

Figure 5.3: ILP for inferring fence placements.

Since we handle several types of fences for a given architecture (e.g. dependencies, lwsync and sync on Power), we can also assign some cost to each of them. For example, following the folklore, a dependency is less costly than an lwsync, which is itself less costly than a sync. Given these costs, one might want to minimise their sum along different executions: to forbid cycles 1, 2 and 3 in Fig. 5.2, a single lwsync between f and g can be cheaper at runtime than three dependencies respectively between e and g, f and g, and f and h. However, if we had only cycles 1 and 2, the dependencies would be cheaper. We see that we have to optimise both the placement and the type of fences at the same time.

We model our problem as an *integer linear program* (ILP) (see Fig. 5.3), which we explain in this section. Solving our ILP gives us a set of fences to insert to forbid the cycles. This set of fences is optimal in that it minimises the cost function. More precisely, the constraints are the cycles to forbid, each variable represents a fence to insert, and the cost function sums the cost of all fences.

5.2.1 Cost function of the ILP

We handle several types of fences: full (f), lightweight (lwf), control fences (cf), and dependencies (dp). On Power, the full fence is sync, the lightweight one lwsync. We write $\mathbb{T}$ for the set $\{\mathsf{dp}, \mathsf{f}, \mathsf{cf}, \mathsf{lwf}\}$. We assume that each type of fence has an *a priori* cost (e.g. a dependency is cheaper than a full fence), regardless of its location in the code. We write $\mathsf{cost}(\mathsf{t})$ for $\mathsf{t} \in \mathbb{T}$ for this cost.

We take as input the AEG of our program and the potential critical cycles to

fence. We define two sets of pairs (l, t) where l is a po_s edge of the AEG and t a type of fence. We introduce an ILP variable t_l (in $\{0, 1\}$) for each pair (l, t).

The set potential-places is the set of such pairs that can be inserted into the program to forbid the cycles. The set actual-places is the set of such pairs that have been set to 1 by our ILP. We output this set, as it represents the locations in the code in need of a fence and the type of fence to insert for each of them. We also output the total cost of all these insertions, i.e. $\sum_{(l,\mathsf{t})\in\mathsf{potential\text{-}places}(C)} \mathsf{t}_l \times \mathsf{cost}(\mathsf{t})$. The solver should minimise this sum whilst satisfying the constraints.

5.2.2 Constraints in the ILP

We want to forbid all the cycles in the set that we are given after filtering, as explained in the preamble of this section. This requires placing an appropriate fence on each delay for each cycle in this set. Different delay pairs might need different fences, depending e.g. on the directions (write or read) of their extremities. Essentially, we follow the table in Fig. 5.1. For example, a write-read pair needs a full fence (e.g. mfence on x86, or sync on Power). A read-read pair can use anything amongst dependencies and fences. Our constraints ensure that we use the right type of fence for each delay pair.

Inequalities as constraints We first assume that all the program order delays are in po_s and we ignore Power and ARM special features (dependencies, control fences and communication delays). This case deals with relatively strong models, ranging from TSO to RMO. We relax these assumptions below.

In this setting, potential-places(C) is the set of all the po_s delays of the cycles in C. We ensure that every delay pair for every execution is fenced, by placing a fence on the static po_s edge for this pair, and this for each cycle given as input. Thus, we need at least one constraint per static delay pair d in each cycle.

If d is of the form poWR, as (g, h) in Fig. 5.2 (cycle 4), only a full fence can fix it (cf Fig. 5.1), thus we impose $\mathsf{f}_d \geq 1$. If d is of the form poRR, as (f, h) in Fig. 5.2 (cycle 3), we can choose any type of fence, i.e. $\mathsf{dp}_d + \mathsf{cf}_d + \mathsf{lwf}_d + \mathsf{f}_d \geq 1$.

Our constraints cannot be equalities because it is not certain that the resulting system would be satisfiable. To see this, suppose our constraints were equalities, and consider Fig. 5.2 limited to cycles 2, 3 and 4. Using only full fences, lightweight fences, and dependencies (i.e., ignoring control fences for now), we would generate the constraints (i) $\mathsf{lwf}_{(f,g)} + \mathsf{f}_{(f,g)} = 1$ for the delay (f, g) in cycle 2, (ii) $\mathsf{dp}_{(f,h)} + \mathsf{lwf}_{(f,h)} +$

$\mathsf{f}_{(f,h)} + \mathsf{lwf}_{(g,h)} + \mathsf{f}_{(g,h)} = 1$ for the delay (f,h) in cycle 3, and **(iii)** $\mathsf{f}_{(g,h)} = 1$ for the delay (g,h) in cycle 4.

Preventing the delay (g,h) in cycle 4 requires a full fence, thus $\mathsf{f}_{(g,h)} = 1$. By the constraint **(ii)**, and since $\mathsf{f}_{(g,h)} = 1$, we derive $\mathsf{f}_{(f,g)} = 0$ and $\mathsf{lwf}_{(f,g)} = 0$. But these two equalities are not possible given the constraint **(i)**. By using inequalities, we allow several fences to live on the same edge. In fact, the constraints only ensure the soundness; the optimality is fully determined by the cost function to minimise.

Delays are in fact in po_s^+, not always in po_s: in Fig. 5.2, the delay (e,g) in cycle 1 does not belong to po_s but to po_s^+. Thus given a po_s^+ delay (x,y), we consider all the po_s pairs which appear between x and y, i.e.: $\mathsf{between}(x,y) \triangleq \{(e_1,e_2) \in \text{po}_s \mid (x,e_1) \in \text{po}_s^* \wedge (e_2,y) \in \text{po}_s^*\}$. For example in Fig. 5.2, we have $\mathsf{between}(e,g) = \{(e,f), (f,g)\}$. Thus, ignoring the use of dependencies and control fences for now, for the delay (e,g) in Fig. 5.2, we will not impose $\mathsf{f}_{(e,g)} + \mathsf{lwf}_{(e,g)} \geq 1$ but rather $\mathsf{f}_{(e,f)} + \mathsf{lwf}_{(e,f)} + \mathsf{f}_{(f,g)} + \mathsf{lwf}_{(f,g)} \geq 1$. Indeed, a full fence or a lightweight fence in (e,f) or (f,g) will prevent the delay in (e,g).

Dependencies need more care, as they cannot necessarily be placed anywhere between e and g (in the formal sense of $\mathsf{between}(e,g)$): $\mathsf{dp}_{(e,f)}$ or $\mathsf{dp}_{(f,g)}$ would not fix the delay (e,g), but simply maintain the pairs (e,f) or (f,g), leaving the pair (e,g) free to be reordered. Thus if we choose to synchronise (e,g) using dependencies, we actually need a dependency from e to g: $\mathsf{dp}_{(e,g)}$. Dependencies only apply to pairs that start with a read; thus for each such pair (see the poRW and poRR cases in Fig. 5.3), we add a variable for the dependency: (e,g) will be fixed with the constraint $\mathsf{dp}_{(e,g)} + \mathsf{f}_{(e,f)} + \mathsf{lwf}_{(e,f)} + \mathsf{f}_{(f,g)} + \mathsf{lwf}_{(f,g)} \geq 1$.

Control fences placed after a conditional branch (e.g. bne on Power) prevent speculative reads after this branch (see Fig. 5.1). Thus, when building the AEG, we built a set poC for each branch, which gathers all the pairs of abstract events such that the first one is the last event before a branch, and the second is the first event after that branch. We can place a control fence before the second component of each such pair, if the second component is a read. Thus, we add cf_e as a possible variable to the constraint for read-read pairs (see poRR case in Fig. 5.3, where $\mathsf{ctrl}(d) = \mathsf{between}(d) \cap \mathsf{poC}$).

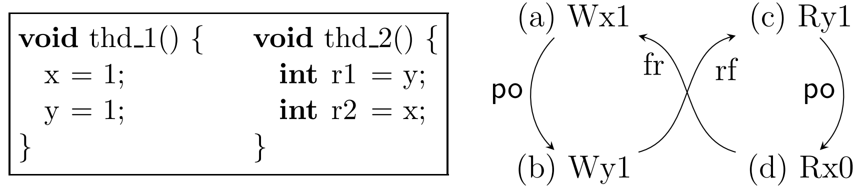


Figure 5.4: Message Passing from Fig. 2.14 with its corresponding event structure.

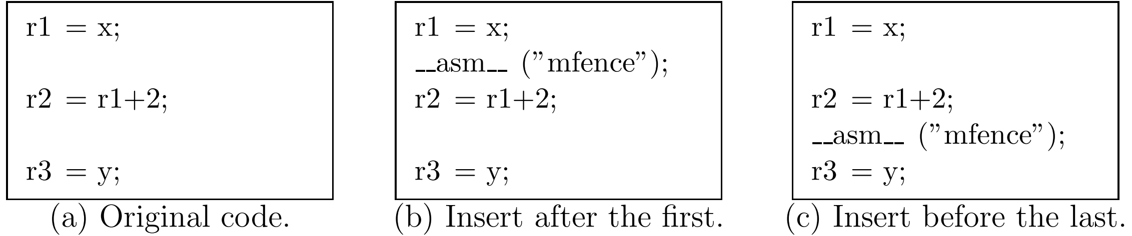


Figure 5.5: Choices for placing a fence.

Cumulativity For architectures like Power, where stores are non-atomic, we need to look for program order pairs that are connected to an external read-from (e.g. (c, d) in Fig. 5.4 has an rf connected to it via event c). In such cases, we need to use a *cumulative fence*, e.g. lwsync or sync, and not, for example, a dependency.

The locations to consider in such cases are: before (in po_s) the write w of the rfe, or after (in po_s) the read r of the rfe, i.e. $\mathsf{cumul}(w, r) = \{(e_1, e_2) \mid (e_1, e_2) \in \text{po}_s \wedge ((e_2, w) \in \text{po}_s^* \vee (r, e_1) \in \text{po}_s^*)\}$. In Fig. 5.2 (cycle 2), (g, i) over-approximates an rfe edge, and the edges where we can insert fences are in $\mathsf{cumul}(g, i) = \{(f, g), (i, j)\}$.

We need a cumulative fence as soon as there is a potential rfe, even if the adjacent po_s pairs do not form a delay. For example in Fig. 5.4, suppose there is a dependency between the reads on T_1, and a fence maintaining write-write pairs on T_0. In that case we need to place a cumulative fence to fix the rfe, even if the two po_s pairs are themselves fixed. Thus, we quantify over all po_s pairs when we need to place cumulative fences. As only f and lwf are cumulative, we have $\mathsf{potential\text{-}places}(C) \triangleq \{(l, t) \mid (t \in \{\mathsf{dp}\} \wedge l \in \text{delays}(C)) \vee (t \in \mathbb{T} \backslash \{\mathsf{dp}\} \wedge l \in \bigcup_{d \in \text{delays}(C)} \mathsf{between}(d)) \vee (t \in \{\mathsf{f}, \mathsf{lwf}\} \wedge l \in \text{po}_s(C))\}$.

5.3 Insertion of fences and dependencies

Given an AEG, we return the static program order edges where we should place a fence to forbid the critical cycles. Then we have some freedom for the fence placement in the actual code. Consider e.g. the program on the left of Fig. 5.5. The corresponding AEG is $(Rx, Ry) \in \text{po}_s$. To fence this edge, we can place a fence either as in

Fig. 5.5(b) or in Fig. 5.5(c), namely just after the first component of a delay pair, or just before the last. Our tool offers these two options. We next illustrate how we concretely insert fences and dependencies in a piece of C code.

Fences are all handled the same way; we simply inline an assembly fence. For example, for a read-read pair separated by a branch (lines 3 and 5 in Fig. 5.6 on the left), we can insert a control fence, e.g. isb on ARM. The compiler keeps the fence in place, as one can see in the compiled code in Fig. 5.6 on the right. The while loop (including the read of x) is implemented by lines 3 to 6, then comes the isb (line 7), and the read of y corresponds to lines 8 and 9.

Dependencies force us to rewrite the code. Consider a read-read pair, corresponding to lines 3 and 9 on the left of Fig. 5.7. We enforce an *address dependency* from the read of x to the read of y, by using a register (r3) to perform some computation which always returns 0 (in this case xor-ing a register with itself), then add this result to the address of y. Again, the compiler does not optimise this dependency (see lines 4 to 8 on the right of Fig. 5.7).

5.4 Experiments and Impact

We implemented our new method, in addition to all the methods described in Sec. 5.7, in our tool musketeer, using glpk (`http://www.gnu.org/software/glpk`) as the ILP solver. musketeer is a completely automated source-to-source transformation for concurrent C program. Once the locations and types of fences have been inferred, the

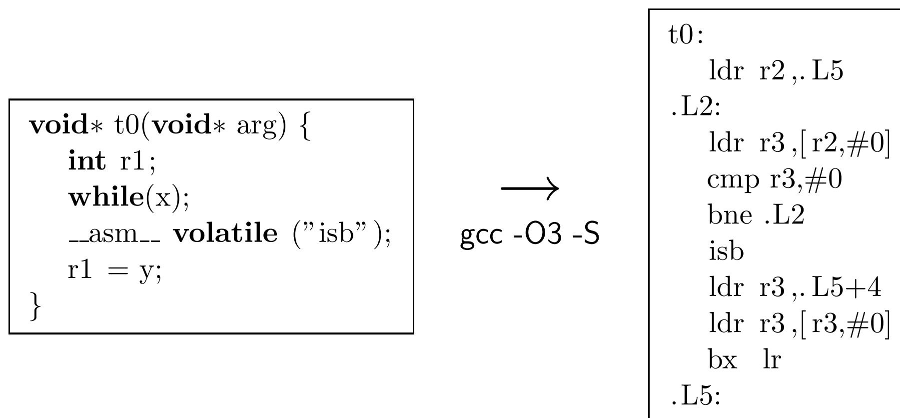


Figure 5.6: `isb.c` and `isb-O3.s`.

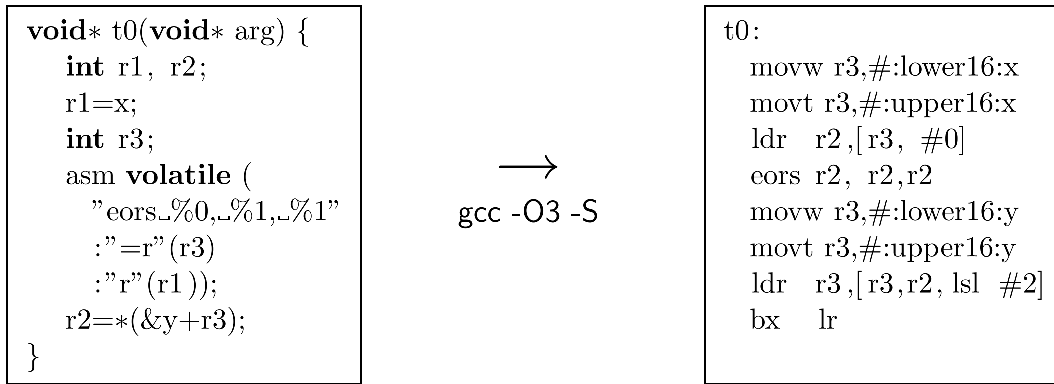


Figure 5.7: `addr.c` and `addr-O3.s`.

insertion in the source itself is performed by a script. This step is reasonably straightforward for memory fences. The difficulty mainly lies in extracting accesses from expressions into temporary assignments when necessary and then taking into account the syntactical structure of the program before inserting the inline assembly—which is sometimes not trivial in the presence of loops or advanced controls. Inserting dependencies in C is more challenging, due to the multiple optimisations that the compiler will perform. We explained how we address this issue in Sec. 5.3.

We compare our method and the methods we reimplemented to the existing tools listed in Sec. 5.7 on classic examples from literature and some Debian executables in Sec. 5.4.1. We finally check the impact on runtime of the fences inferred and inserted in memcached, a Debian executable of about 10000 lines of code.

Currently, in order to infer fences for a program, musketeer needs an entry point (usually the *main* function) in order to over-approximate the interfering threads. In [AMT13], Michael Tautschnig used a specific harness calling the functions of a

	CLASSIC									
	Dek		Pet		Lam		Szy		Par	
LoC	50		37		72		54		96	
dfence	–	–	–	–	–	–	–	–	–	–
memorax	0.4	2	1.4	2	79.1	4	–	–	–	–
musketeer	**0.0**	**5**	**0.0**	**3**	**0.0**	**8**	**0.0**	**8**	**0.0**	**3**
offence	0.0	2	0.0	2	0.0	8	0.0	8	–	–
pensieve	0.0	16	0.0	6	0.0	24	0.0	22	0.0	7
remmex	0.5	2	0.5	2	2.0	4	1.8	5	–	–
trencher	1.6	2	1.3	2	1.7	4	–	–	0.5	1

Figure 5.8: All tools on the CLASSIC series for TSO

	FAST											
	Cil		CL		Fif		Lif		Anc		Har	
LoC	97		111		150		152		188		179	
dfence	7.8	3	6.2	3	∼	0	∼	0	∼	0	∼	0
memorax	–	–	–	–	–	–	–	–	–	–	–	–
musketeer	**0.0**	**3**	**0.0**	**1**	**0.1**	**1**	**0.0**	**1**	**0.1**	**1**	**0.6**	**4**
offence	–	–	–	–	–	–	–	–	–	–	–	–
pensieve	0.0	14	0.0	8	0.1	33	0.0	29	0.0	44	0.1	72
remmex	–	–	–	–	–	–	–	–	–	–	–	–
trencher	8.6	3	–	–	–	–	–	–	–	–	–	–

Figure 5.9: All tools on the FAST series for TSO

library non-deterministically inside a loop. A similar approach could be implemented in musketeer to analyse APIs.

Note that musketeer currently makes no assumption regarding potential APIs used in programs that would allow reorderings on purpose (e.g., the Read Copy Update API in the Linux kernel). Because musketeer does not evaluate conditions, it might infer unnecessary fences in case of, e.g., compare and swaps. One way to avoid these fences could consist of annotating the programs or APIs so that cycles involving these intended reorderings would be ignored by the tool. Other strategies based on filtering of the cycles (for instance, only minimising the existing fences inserted by the users) could also be applied.

5.4.1 Experiments and benchmarks

Our tool analyses C programs. dfence also handles C code, but requires some high-level specification for each program, which was not available to us. memorax works on a process-based language that is specific to the tool. offence works on a subset of assembler for x86, ARM and Power. pensieve originally handled Java, but we did not have access to it and have therefore re-implemented the method. remmex handles Promela-like programs. trencher analyses transition systems. Most of the tools come with some of the benchmarks in their own languages; not all benchmarks were however available for each tool. We have re-implemented some of the benchmarks for offence.

We now detail our experiments. CLASSIC and FAST gather examples from the literature and related work. The DEBIAN benchmarks are packages of Debian Linux 7.1. CLASSIC and FAST were run on a x86-64 Intel Core2 Quad Q9550 machine with 4 cores (2.83 GHz) and 4 GB of RAM. DEBIAN was run on a x86-64 Intel Core i5-3570 machine with 4 cores (3.40 GHz) and 4 GB of RAM.

			TSO		Power	
	LoC	nodes	fences	time	fences	time
memcached	9944	694	3	13.9s	70	89.9s
lingot	2894	183	0	5.3s	5	5.3s
weborf	2097	73	0	0.7s	0	0.7s
timemachine	1336	129	2	0.8s	16	0.8s
see	2626	171	0	1.4s	0	1.5s
blktrace	1567	615	0	6.5s	–	timeout
ptunnel	1249	1867	2	95.0s	–	timeout
proxsmtpd	2024	10	0	0.1s	0	0.1s
ghostess	2684	1106	0	25.9s	0	25.9s
dnshistory	1516	1466	1	29.4s	9	64.9s

Figure 5.10: musketeer on selected benchmarks in DEBIAN series for TSO and Power

classic consists of Dekker's mutex (Dek) [Dij65]; Peterson's mutex (Pet) [Pet81]; Lamport's fast mutex (Lam) [Lam87]; Szymanski's mutex (Szy) [Szy88]; and Parker's bug (Par) [Dic09]. We ran all tools in this series for TSO (the model common to all). For each example, Fig. 5.9 and Fig. 5.8 give the number of fences inserted, and the time (in sec) needed. When an example is not available in the input language of a tool, we write "–". The first four tools place fences to enforce stability/robustness [AM11, BMM11]; the last three to satisfy a given safety property. We used memorax with the option `-o1`, to compute one *maximal permissive* set and not all. For remmex on Szymanski, we give the number of fences found by default (which may be non-optimal). Its "maximal permissive" option lowers the number to 2, at the cost of a slow enumeration. As expected, musketeer is less precise than most tools, but outperforms all of them.

fast gathers Cil, Cilk 5 Work Stealing Queue (WSQ) [FLR98]; CL, Chase-Lev WSQ [CL05]; Fif, Michael et al.'s FIFO WSQ [MVS09]; Lif, Michael et al.'s LIFO WSQ [MVS09]; Anc, Michael et al.'s Anchor WSQ [MVS09]; Har, Harris' set [DFG+00]. For each example and tool, Fig. 5.8 gives the number of fences inserted (under TSO) and the time needed to do so. For dfence, we used the setting of [LNP+12]: the tool has up to 20 attempts to find fences. We were unable to apply dfence on some of the FAST examples: we thus reproduce the number of fences given in [LNP+12], and write $\sim$ for the time. We applied musketeer to this series, for all architectures. The fencing times for TSO and Power are almost identical, except for the largest example, namely Har ($0.1\,s$ vs $0.6\,s$).

debian gathers 374 executables. These are a subset of the goto-programs that have been built from packages of Debian Linux 7.1 by Michael Tautschnig. A small excerpt of our results is given in Fig. 5.10. The full data set is provided at `http://www.cprover.org/wmm/musketeer`. For each program, we give the lines of code and number of nodes in the AEG. We used musketeer on these programs to demonstrate its scalability and its ability to handle deployed code. Most programs already contain fences or operations that imply them, such as compare-and-swaps or locks. Our tool musketeer takes these fences into account and infers a set of additional fences sufficient to guarantee SC. The largest program we handle is memcached ($\sim$ 10000 LoC). Our tool needs 13.9 s to place fences for TSO, and 89.9 s for Power. A more meaningful measure for the hardness of an instance is the number of nodes in the AEG. For example, ptunnel has 1867 nodes and 1249 LoC. The fencing takes 95.0 s for TSO, but times out for Power due to the number of cycles. Not all fences inferred by musketeer are necessary to enforce SC, due to the imprecision introduced by the AEG abstraction. However, as Section 5.4.2 will show, the execution time overhead of the program versions with fences inserted by musketeer is still very low.

5.4.2 Impact of inferred fences on runtime

Before optimising the placement of fences, we investigated whether naive approaches to fence insertion indeed have a negative performance impact. To that end, we measured the overhead of different fencing methods on a stack and a queue from the liblfds lock-free data structure package (`http://liblfds.org`). For each data structure, we built a harness (consisting of 4 threads) that concurrently invokes its operations. We built several versions of the above two programs:

- (M) with fences inserted by our tool musketeer;
- (P) with fences following the *delay set analysis* of the pensieve compiler [SFW+05], i.e. a static over-approximation of Shasha and Snir's eponymous (dynamic) analysis [SS88] (see also the discussion of Lee and Padua's work [LP01] in Sec. 5.7);
- (V) with fences following the *Visual Studio* policy, i.e. guaranteeing acquire/release semantics (in the C11 sense [c1111]), but not SC, for reads and writes of `volatile` variables (see `http://msdn.microsoft.com/en-us/library/vstudio/jj635841.aspx`, accessed 04-11-2013). On x86, no fences are necessary as the model is sufficiently strong already; hence, we only provide data for ARM;

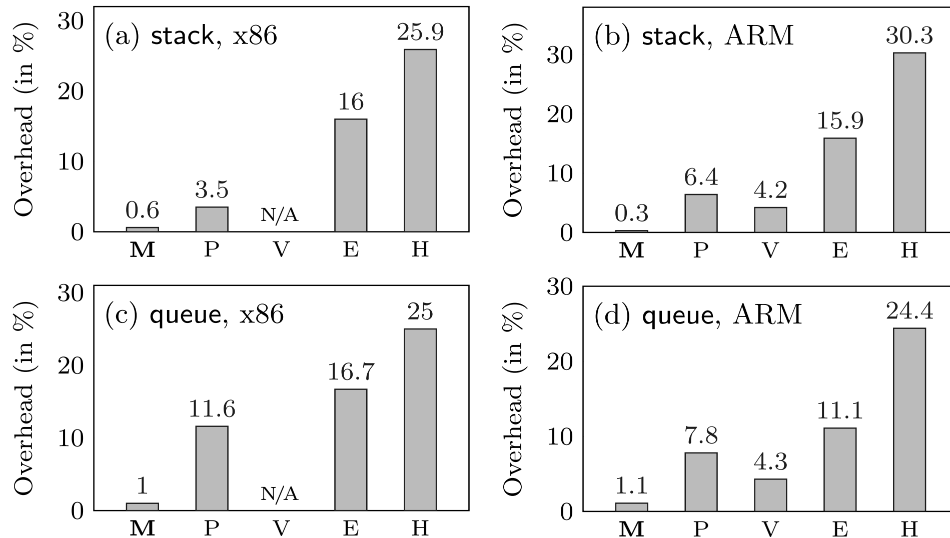


Figure 5.11: Overheads for the different fencing strategies

- (E) with fences after each access to a shared variable;
- (H) with an mfence (x86) or a dmb (ARM) after every assembly instruction that writes (x86) or reads or writes (ARM) *static global* or *heap data*.

We emphasise that these experiments required us to implement (P), (E) and (V) ourselves, in order for them to handle the architectures that we considered. This means in particular that our tool provides the pensieve policy (P) for TSO, Power and ARM, whereas the original pensieve targeted Java only.
We ran all versions 100 times, on an x86-64 Intel Core i5-3570 with 4 cores (3.40 GHz) and 4 GB of RAM, and on an ARMv7 (32-bit) Samsung Exynos 4412 with 4 cores (1.6 GHz) and 2 GB of RAM.

For each program version, Fig. 5.11 shows the mean overhead w.r.t. the unfenced program. We give the overhead (in %) in *user time* (as given by Linux `time`), i.e. the time spent by the program in user mode on the CPU. Amongst the approaches that guarantee SC (i.e. all but V), the best results were achieved with our tool musketeer.

We checked the statistical significance of the execution time improvement of our method over the existing methods by computing and comparing the confidence intervals for a sample size of $N = 100$ and a confidence level $1 - \alpha = 95\%$ in Fig. 5.14. If the confidence intervals for two methods are non-overlapping, we can conclude that the difference between the means is statistically significant.

	stack on x86	stack on ARM	queue on x86	queue on ARM
(O)	[9.757; 9.798]	[11.291; 11.369]	[11.947; 11.978]	[20.441; 20.634]
(M)	[9.818; 9.850]	[11.316; 11.408]	[12.067; 12.099]	[20.687; 20.857]
(P)	[10.077; 10.155]	[11.995; 12.109]	[13.339; 13.373]	[22.035; 22.240]
(V)	N/A	[11.779; 11.834]	N/A	[21.334; 21.526]
(E)	[11.316; 11.360]	[13.071; 13.200]	[13.949, 13.981]	[22.722; 22.903]
(H)	[12.286; 12.325]	[14.676; 14.844]	[14.941, 14.963]	[25.468; 25.633]

Figure 5.12: Confidence intervals for data structure experiments.

5.4.3 A case study: Memcached

We finally measure the impact of fences for the program memcached, running experiments similar to those in Sec. 5.4.2. As we mentioned in Sec. 5.4.1, musketeer inferred 3 fences under TSO and 70 under Power. We built new versions of memcached according to the fencing strategies described in Sec. 5.4.2. We used in particular the memtier benchmarking tool to generate a workload for the memcached daemon, and killed the daemon after 60 s. Fig. 5.11 shows the mean overhead w.r.t. the original, unmodified program.

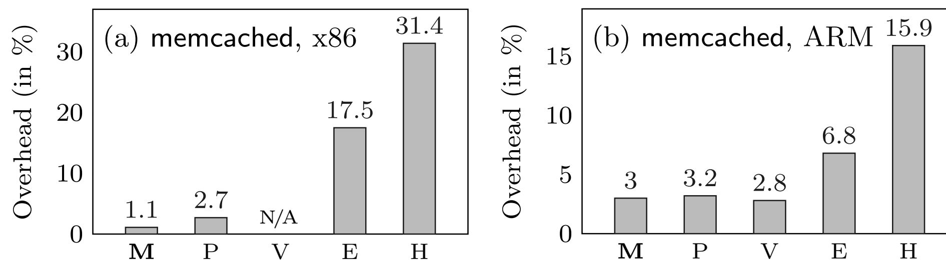


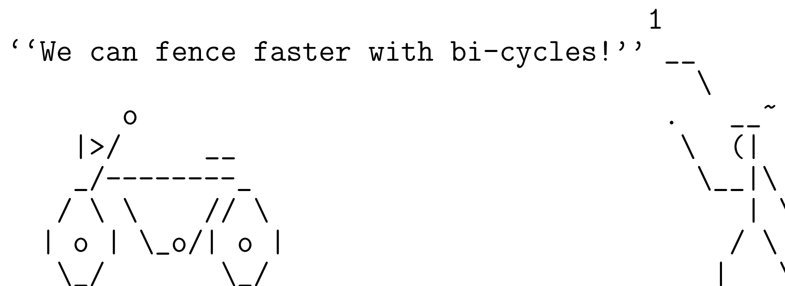
Figure 5.13: Runtime overheads due to inserted fences in memcached for each strategy

Adding a fence after *every* access to static or heap data has a significant performance effect. Similarly, adding fences via an escape analysis is expensive, yielding overheads of up to 17.5 %. Amongst the approaches guaranteeing SC (i.e. all but V), the best results were achieved with our tool musketeer. We again computed the confidence intervals to check the statistical significance in Fig. 5.14.

	stack on x86	stack on ARM	queue on x86	queue on ARM
(O)	[9.757; 9.798]	[11.291; 11.369]	[11.947; 11.978]	[20.441; 20.634]
(M)	[9.818; 9.850]	[11.316; 11.408]	[12.067; 12.099]	[20.687; 20.857]
(P)	[10.077; 10.155]	[11.995; 12.109]	[13.339; 13.373]	[22.035; 22.240]
(V)	N/A	[11.779; 11.834]	N/A	[21.334; 21.526]
(E)	[11.316; 11.360]	[13.071; 13.200]	[13.949, 13.981]	[22.722; 22.903]
(H)	[12.286; 12.325]	[14.676; 14.844]	[14.941, 14.963]	[25.468; 25.633]

Figure 5.14: Confidence intervals for data structure experiments

5.5 Alternative encodings



The key point of the encoding that we introduced in Sec. 5.2 is that our ILP variables are po_s edges and they are associated with simple rules—the constraints that we construct following the five (meta-) “*Constraints*” lines in Fig. 5.3. Other techniques would consider all the po_s^+ edges that could be involved in some critical cycles and how they relate to each other—a fence on a po_s^+ edge can affect another sub-po_s^+ edge and so on. po_s edges are also convenient because they are easy to collect from the AEG and linear in the size of the graph, whereas po_s^+ edges can be in exponential number.

Working on the po_s edges is, however, the lowest level of abstraction in the AEG, and one might not need this level of granularity to infer fences. In Fig. 5.15, for instance, we do not need to take into account all the po_s edges (i.e. $\{(a,b),(c,d),(e,f_1),(f_n,g)\} \cup \bigcup_{i=1,\ldots,n-1}\{(f_i,f_{i+1})\}$) to infer the fence required between (e,g) to prevent a store-buffering. We can achieve this by reasoning over $\{(a,b),(c,d),(e,g)\}$ only.

We have therefore studied other encodings with different variables. The objectives remain however the same: the fence placement must remain sound and optimal for the given AEG. Fig. 5.16 summarises in a table these different choices of variables, their respective costs of collection in AEGs, their numbers (that impacts the size of

[1]This play on words was suggested by Ganesh Narayanaswamy.

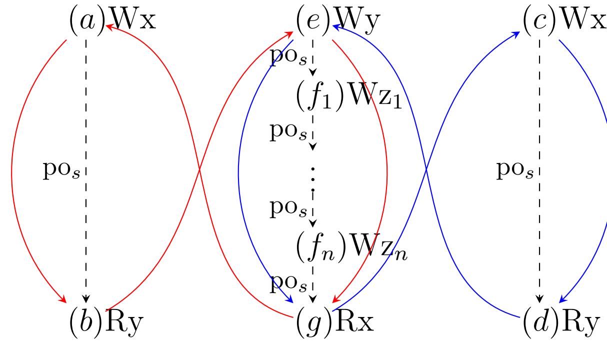


Figure 5.15: A bi-cycle example where most of the po_s edges are useless for the fence inference.

the ILP and thus the time spent by the ILP solver) and an intuitive description of them.

We explain here the notations used to quantify the number of variables of the ILP. Given a set of critical cycles collected from the AEG of the input program, we write delays for the set of delays of these cycles that lie in po_s^+ (delays$\in \wp(po_s^+)$). A delay $d \in$delays (and in po_s^+) can be decomposed into all the po_s edges composing it with $\gamma_v(\{d\})$, where $\gamma_v \in \wp(po_s^+) \rightarrow \wp(po_s)$. The inverse function that recompose into (possibly several) po_s^+ edge(s) is $\alpha_v \in \wp(po_s) \rightarrow \wp(po_s^+)$. For some of the encodings, we distinguish the delays that are isolated to those which intersect. We partition delays into delays$_{\nearrow}$ and delays$_{\curlyvee}$ such that delays=delays$_{\nearrow}\cup$delays$_{\curlyvee}$, $\forall d,\, d' \in$delays$_{\nearrow}$, $\gamma_v(d) \cap \gamma_v(d') = \emptyset$ and $\forall d \in$delays$_{\curlyvee}$, $\exists d' \in$delays$_{\curlyvee}$, $\gamma_v(d) \cap \gamma_v(d') \neq \emptyset$.

We now explain in detail each of these alternative encodings.

Basic encoding (po_s encoding) In the basic encoding, in addition to the delays themselves that could host a dependency, we collect all the po_s involved in critical cycles as we mentioned in the previous section (in linear time with respect to the number of cycles) and use them as variables in the ILP. There are as many variables as there are po_s edges in the delays of the critical cycles.

Restricted encoding (restricted po_s encoding) In the restricted encoding, we only consider po_s edges for the delays of entangled critical cycles. For the other delays, we use one variable per delay (in po_s^+). The number of variables is lower, but we need to detect the entangled cycles, which is quadratic in the number of cycles.

[2]These numbers actually exclude the specific case of dependencies, that can be placed on each delay of delays. If we take dependencies into account, like for Power or ARM, one needs to add #delays to the total. Some of the variables might already be existing, meaning that we get an over-approximation of the actual number of variables.

encoding	ILP variables	number of variables[2]	complexity
basic	po_s in the critical cycles	$\# \bigcup_{d \in \mathrm{delays}} \gamma_v(\{d\})$	$O(\#cycles)$
restricted	po_s in the intersections of pairs of critical cycles	$\#\mathrm{delays}_{\nearrow} + \# \bigcup_{a,b \in \mathrm{delays}_{\nwarrow\!\!\!\nearrow}} (\gamma_v(\{a\}) \cap \gamma_v(\{b\}))$	$O(\#\mathrm{cycles}^2)$
GCEs	po_s^+ at the intersections of any set of critical cycles	$\#\mathrm{delays}_{\nearrow} + \# \bigcup_{p \in \wp(\mathrm{delays}_{\nwarrow\!\!\!\nearrow}) \wedge \#p > 1} \alpha_v \left(\bigcap_{d \in p} \gamma_v(\{d\}) \right)$	$O(2^{\#\mathrm{cycles}})$
pair-wise GCEs	po_s^+ at the intersections of any pair of critical cycles	$\#\mathrm{delays}_{\nearrow} + \# \bigcup_{a,b \in \mathrm{delays}_{\nwarrow\!\!\!\nearrow}} \alpha_v (\gamma_v(\{a\}) \cap \gamma_v(\{b\}))$	$O(\#\mathrm{cycles}^2)$

Figure 5.16: Comparison of the alternative encodings in terms of variables.

Greatest common edges ($\mathbf{po}_s^+$ encoding) In the greatest common edges encoding (GCEs), we abstract away from the po_s edges and try to reason over the po_s^+ edges. We consider all the delays of critical cycles that are not intersected by any other cycles, and all the po_s^+ edges that are shared by several cycles. The number of variables is thus lower than when considering all the po_s edges. Finding the intersections of the cycles, that is, the po_s^+ edges shared between several cycles, requires exploring all the combinations of critical cycles, which is exponential in the number of cycles.

Pair-wise greatest common edges (pair-wise $\mathbf{po}_s^+$ encoding) The pair-wise greatest common edges encoding is similar to the previous encoding, except that we do not explore all the combinations of critical cycles, but only every pair of cycles. Indeed, calculating the greatest po_s^+ edges shared by a collection of cycles is identical to finding the intersection of a set of segments. Given a set of segments, the intersection of all these segments is either empty or the intersection of two of these[3]. We

[3] Indeed, the intersection of two segments $[a1, a2]$ and $[b1, b2]$, $[a1, a2] \sqcap [b1, b2]$ is defined as empty or $[\max(a1, b1), \min(a2, b2)]$. If we compose $\sqcap$ twice, i.e. $([a1, a2] \sqcap [b1, b2]) \sqcap [c1, c2]$, this intersection will be equal to either empty, $[a1, a2] \sqcap [b1, b2]$, $[c1, c2] \sqcap [b1, b2]$ or $[a1, a2] \sqcap [c1, c2]$. To show this, consider each of the possible outputs (e.g. $([a1, a2] \sqcap [b1, b2]) \sqcap [c1, c2] = [a1, c2]$) and use the min and max in the intersection expression to determine orders between segments' extremities and

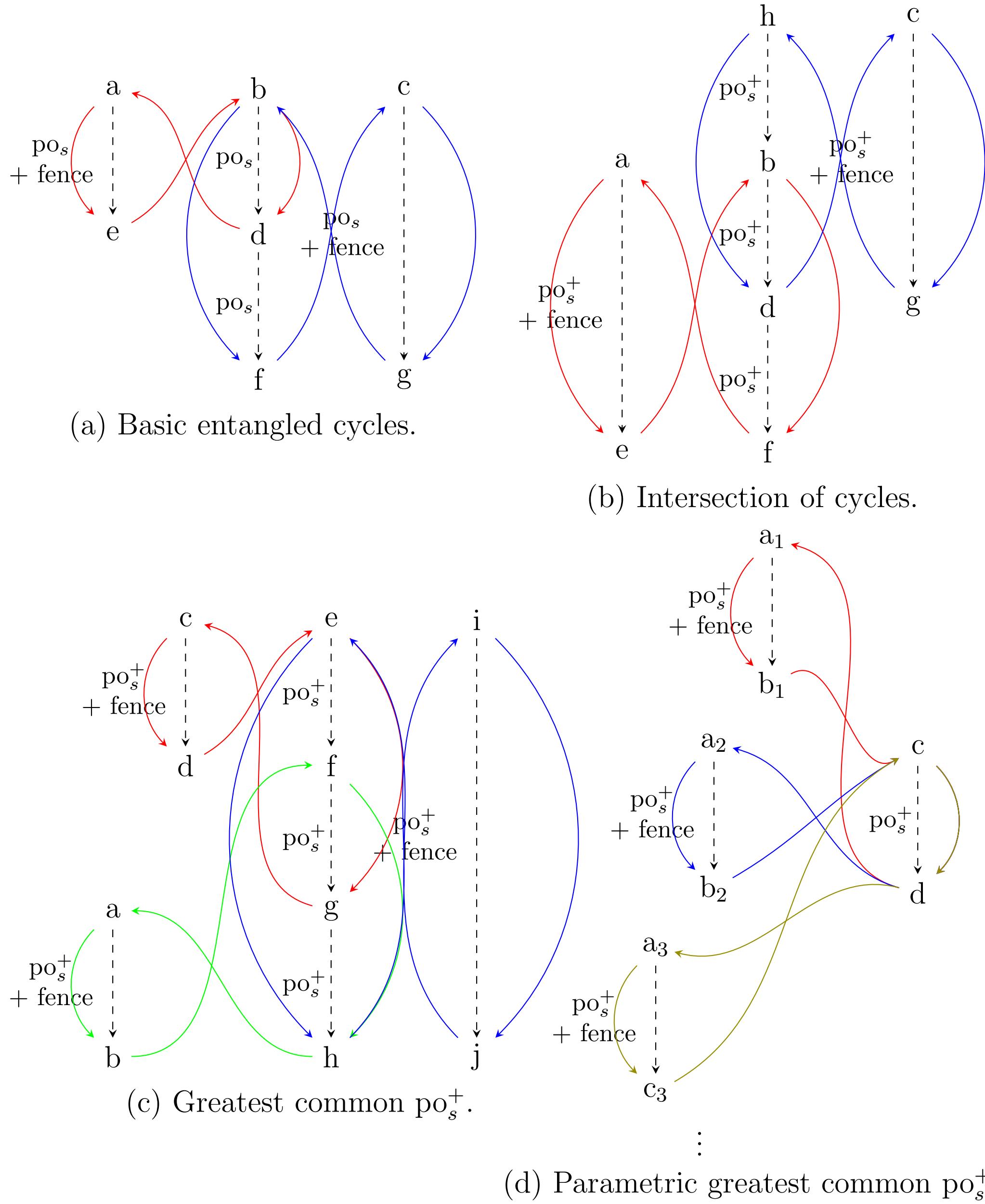
thus compute only the po_s^+ of every pair of cycles, which is quadratic in the number of critical cycles.

Comparison We compare these four encodings with four cases: a simple AEG of two entangled critical cycles in Fig. 5.17(a), an AEG where two cycles are entangled but do not share a delay in Fig. 5.17(b), an AEG with three cycles that are two-by-two entangled but do not share a delay in Fig. 5.17(c), and an AEG where m critical cycles are entangled with sharing a delay in Fig. 5.17(d). We write in Fig. 5.18 the number of variables used with each of the encoding for each of the cases, and in Fig. 5.19 the computation complexity to retrieve these variables. We compute separately the numbers with and without taking into account dependencies (for Power and ARM), as the dependencies affect delays in po_s^+ regardless of the sequences of po_s they might be composed of—and this could blur the results on our short examples.

In Fig. 5.17(a), there are two delays to address: (b, d) and (b, f). The former is composed of only one po_s; the latter is composed of (b, d) and (d, f). The basic encoding will consider each cycle and add a variable for each po_s it encounters in the delays of each cycle. Since this exploration is performed cycle by cycle, we can directly infer the dependency variables for each delay of the cycles. Therefore there is no difference between the costs with or without dependencies. The other encodings will consider uniquely the pair of cycles, and take as variable the only po_s edge at the intersection of the two cycles. Therefore they need only one variable to encode this problem. Note that in case of dependencies, for each of these encodings, one needs to enumerate the cycles one by one to add the variables encoding the placement of a dependency—which explains the additional computational cost in case of dependencies.

The AEG in Fig. 5.17(b) is similar to the previous one, except that the intersection of the two cycles is neither a delay, nor a po_s edge. Actually, the edges (h, b), (b, d) and (d, f) all lie in po_s^+. In order to get a clearer idea of these measures, we will assume that each of them is composed of n po_s edges. Because the basic encoding relies on a direct collection of the po_s edges, the number of variables will depend on n, even though inferring a fence for this AEG would not need to consider all these po_s edges. The restricted encoding only considers the intersection of the cycles and does better. It is however still function of n. The GCEs and pair-wise GCEs encodings

thus the corresponding intersection of two segments (e.g., $a1 = \max(\max(a1, b1), c1) \Rightarrow a1 > c1$; $c2 = \min(\min(a2, b2), c2) \Rightarrow a2 > c2$; hence $[a1, a2] \sqcap [c1, c2]$). By induction, it applies to the whole collection.



(a) Basic entangled cycles.

(b) Intersection of cycles.

(c) Greatest common po_s^+.

(d) Parametric greatest common po_s^+.

Figure 5.17: Illustrations of the different encodings for the ILP construction.

	without dependencies				with dependencies			
	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)
basic	2	$3n$	$3n$	n	3	$3n+2$	$3n+3$	$n+1$
restricted	1	n	$3n$	n	2	$n+2$	$3n+3$	$n+1$
GCEs	1	1	3	1	2	3	4	1
pair-wise GCEs	1	1	3	1	2	3	4	1

Figure 5.18: Number of variables per encoding and case.

avoid this expensive collection of po_s and just encode the intersection of the cycles in po_s^+ as a unique variable.

Fig. 5.17(c) is the first example where the pair-wise GCEs encoding performs better than the GCEs encoding. The reason for this is that GCEs encoding would require exploring the intersection of all the combinations of two cycles or more, which makes $2^{\#\text{cycles}} - \#\text{cycles}$ intersections to explore, whereas the pair-wise GCEs encoding requires $\frac{\#\text{cycles}(\#\text{cycles}-1)}{2}$ intersections. This exponential explosion can be measured in Fig. 5.17(d), where m cycles are all entangled and sharing one delay, namely (c, d). One variable is required for both GCEs and pair-wise GCEs—the other approaches require n variables—but their computation time is respectively exponential and quadratic.

We also evaluated these encodings empirically. In addition to the basic encoding used to run the experiments of Sec. 5.4, we implemented the pair-wise greatest common edges encoding. We ran the parametric benchmarks for this encoding. We observed that the runtime difference between these two encodings was negligible. For the parametric example $(\mathbf{mp}^n)$, whose results for the two encodings and the other tools are presented in Fig. 5.20, we observe that the GCEs encoding might be slightly more efficient for large parameters. The implementation of this encoding is, however, less robust, therefore we decided to develop and maintain the simplest po_s encoding, "basic".

	without dependencies				with dependencies			
	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)
basic	3	$4n$	$7n$	$m \times n$	3	$4n$	$7n$	$m \times n$
restricted	1	n	$5n$	$\frac{m(m-1)}{2} \times n$	3	$n+2$	$5n+3$	$\frac{m(m-1)}{2} \times n + m$
GCEs	1	1	4	$2^m - 1 - m$	3	3	7	$2^m - 1$
pair-wise GCEs	1	1	3	$\frac{m(m-1)}{2}$	3	3	6	$\frac{m(m+1)}{2}$

Figure 5.19: Computational cost for retrieving the variables per encoding and case.

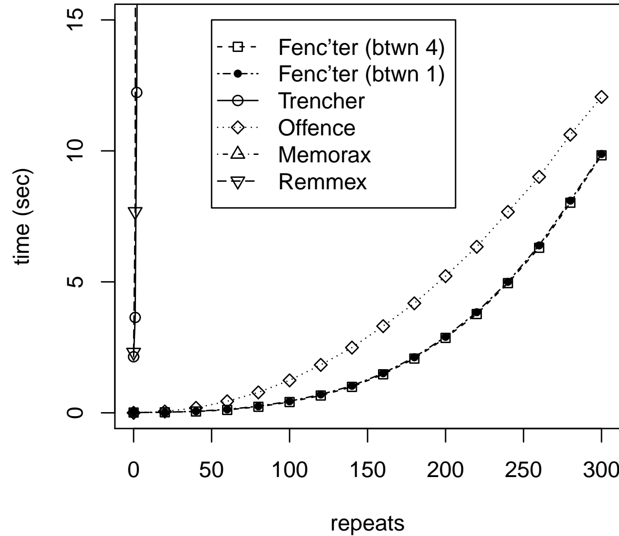


Figure 5.20: Parametric example $(\mathbf{mp}^n)$ run for the basic encoding, the GCEs encoding and the other tools.

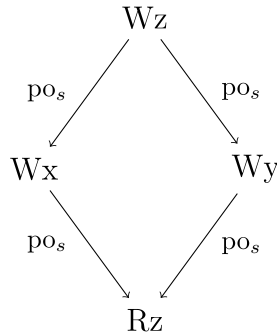
5.6 Optimality and control flows

The synthesis method we proposed is sound and optimal for a given cost of the respective fences/dependencies and for the AEG of the input program that we produced in Chap. 3. We encoded within the ILP constraints the expected semantics of the potential fences, and the ILP solver finds a solution which satisfies all these constraints and provides a sound, minimal placement of fences in the program without enumerating or encoding all the potential combinations of fences.

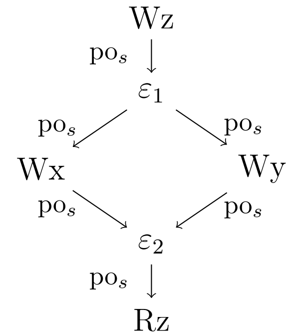
There is however one case where the AEG we described in Chap. 3 might induce a collection of fences that might not appear minimal: branching conditions not involving shared memory accesses that would lie in the middle of two or more cycles. The AEG is indeed computed so that it is minimal itself, in that none of the abstract event nodes can be empty, and there is no empty transition. All these potential cases are absorbed at the construction of the graph into the multiple, non-deterministic in- and out-going edges. This has the advantage of limiting the size of the graph, for a negligible additional cost at construction. Yet, if we observe the program in Fig. 5.21(a) and its associated AEG as it would be built in Chap. 3 in Fig. 5.21(b), we can see that $Wz \xrightarrow{\text{po}_s} Wx$ and $Wz \xrightarrow{\text{po}_s} Wy$ are not related. If the cycle detection finds two cycles and would declare that $Wz \xrightarrow{\text{po}_s} Wx$ and $Wz \xrightarrow{\text{po}_s} Wy$ both require

```
z = 2;
if(local==1) {
  x = 1;
}
else {
  y = 1;
}
local = z;
```

(a) if-diamond



(b) AEG as described in Chap. 3



(c) AEG with empty events

Figure 5.21: Example that requires empty events placed at the branchings.

a fence to prevent a reordering, then the AEG would be too abstract to insert one single fence before the branching condition as it should, and would place two fences in possibly each branch.

To solve this, a post-treatment could be sufficient: if there are fences in all the branches of a conditional branching, we could automatically replace these with one single fence before the condition check. This might arguably impact the optimality of the solution, since the ILP solver took into consideration those two emplacements, and not one single placement. Yet, if the solution it picked contains all these fences, then any other solutions would be more expensive. Therefore, replacing all these fences by a necessarily cheaper, single fence will preserve the optimality of the solution, as no other solution could become cheaper by the same single fence substitution.

We, however, prefer to refine the construction described in Chap. 3: every time the transformer encounters a branching in the program—or a branching junction—it inserts an empty event in the AEG and connects it as expected with po_s edges. The resulting AEG for the example is in Fig. 5.21(c). With this graph, the ILP we construct afterwards takes into account the po_s before the branching, and the ILP solver is hence able to find solution that would insert fences affecting multiple branches. In the example, it would insert a fence between $Wz \xrightarrow{\text{po}_s} \varepsilon_1$.

5.7 Novelty of our contribution w.r.t. the related work

5.7.1 Semantics for the weak memory related analyses

As we did in the related work of Chap. 3, we present the fence synthesis tools by modelling paradigm: axiomatic or operational. We note that the axiomatic approach

authors	tool	model style	objective
Abdulla et al. [AAC+13]	memorax	operational	reachability
Alglave et al. [AMSS10]	offence	axiomatic	SC
Bouajjani et al. [BDM13]	trencher	operational	SC
Fang et al. [FLM03]	pensieve	axiomatic	SC
Kuperstein et al. [KVY10]	fender	operational	reachability
Kuperstein et al. [KVY11]	blender	operational	reachability
Linden et al. [LW13]	remmex	operational	reachability
Liu et al. [LNP+12]	dfence	operational	specification
Sura et al. [SFW+05]	pensieve	axiomatic	SC

Figure 5.22: Fence synthesis tools

was mostly adopted by the compiler and distributed systems communities, whereas operational semantics were preferred by the verification community. One possible reason for this is that members of the verification community tend to expect *completeness*—no false positives in our case—in addition to soundness, whereas static analysis researchers just expect soundness. The intuition behind this is that the operational models convey easily all the details of the semantics, whereas the axiomatic models would tend to focus more on the essentials, at the cost of an abstraction that does not always allow a return to the concrete semantics. The work of [DMVY13] combining abstract interpretation and fence synthesis might be a first step towards unifying the two. All the approaches that we cite here still rely on the foundational work of Shasha and Snir [SS88].

Operational models As we noted in Chap. 3, Linden and Wolper [LW13] explore all executions to simulate the reorderings occurring under TSO and PSO and use a mapping to place fences (store-read and store-store). Abdulla et al. [AAC+13] couple predicate abstraction for TSO with a counterexample-guided strategy. If an error state is reachable, they calculate what they call the *maximal permissive* sets of fences that forbid this error state. Their method guarantees that the fences they find are *necessary*, i.e., removing a fence from the set would make the error state reachable again.

Kuperstein et al. [KVY10] explore all executions for TSO, PSO and a subset of RMO, or use abstraction interpretation in [KVY11] by abstracting the order in the write buffers. They build, during the process, constraints encoding reorderings leading to error states. The fences can be derived from the set of constraints at the error states. Liu et al. [LNP+12] offer a *dynamic synthesis* approach for TSO and PSO,

enumerating the possible sets of fences to prevent an execution picked dynamically from reaching an error state.

Bouajjani et al. [BDM13] build on an operational model of TSO. and look for critical cycles by enumerating delays. Like us, they use linear programming. However, they first enumerate all the solutions, then encode them as an ILP, and finally ask the solver to pick the least expensive one. Our method directly encodes the whole decision problem as an ILP. The solver thus both constructs the solution (avoiding the exponential-size ILP problem) and ensures its optimality.

All the approaches above focus on TSO and its siblings PSO and RMO, whereas we also handle the significantly weaker Power, including quite subtle barriers (e.g. lwsync) compared to the simpler mfence of x86.

Axiomatic models Krishnamurthy et al. [KY96] apply Shasha and Snir's method to *single program multiple data* systems. They however place fences on all the delays.

Lee and Padua [LP01] propose an algorithm based on Shasha and Snir's work. They use dominators in graphs to determine which fences are redundant. This approach was later implemented by Fang et al. [FLM03] in pensieve, a compiler for Java. Sura et al. later implemented a more precise approach in pensieve [SFW+05] (see (P) in Sec. 5.4.2). They pair the cycle detection with an analysis to detect synchronisation that could prevent cycles.

Alglave and Maranget [AM11] revisit Shasha and Snir for contemporary memory models and insert fences following a refinement of [LP01]. Their offence tool handles snippets of assembly code only, where the memory locations need to be explicitly given.

Others We cite the work of Vafeiadis and Zappa Nardelli [VZN11], who present an optimisation of the certified CompCert-TSO compiler to remove redundant fences on TSO. Marino et al. [MSM+11] experiment with an SC-preserving compiler, showing overheads of no more than 34 %. Nevertheless, they emphasise that *"the overheads, however small, might be unacceptable for certain applications"*.

5.7.2 Comparison with trencher

We illustrate the difference between trencher [BDM13] and our approach using Fig. 5.23. There are three cycles that share the edge (a, b). They differ in the path taken between nodes c and g. Suppose that the user has inserted a full fence between a and b. To forbid the three cycles, we need to fence the thread on the right.

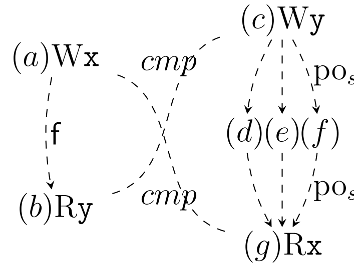


Figure 5.23: Cycles sharing the edge (a, b)

The trencher algorithm first calculates which pairs can be reordered: in our example, these are (c, g) via d, (c, g) via e and (c, g) via f. It then determines at which locations a fence could be placed. In our example, there are 6 options: (c, d), (d, g), (c, e), (e, g), (c, f), and (f, g). The encoding thus uses 6 variables for the fence locations. The algorithm then gathers all the *irreducible* sets of locations to be fenced to forbid the delay between c and g, where "irreducible" means that removing any of the fences would prevent this set from fully fixing the delay. As all the paths that connect c and g have to be covered, trencher needs to collect all the combinations of one fence per path. There are 2 locations per path, leading to 2^3 sets. Consequently, as stated in [BDM13], trencher needs to construct an exponential number of sets.

Each set is encoded in the ILP with one variable. For this example, trencher thus uses $6 + 8$ variables. It also generates one constraint per delay (here, 1) to force the solver to pick a set, and 8 constraints to enforce that all the location variables are set to 1 if the set containing these locations is picked.

By contrast, musketeer only needs 6 variables: the possible locations for fences. We detect three cycles, and generate only three constraints to fix the delay. Thus, on a parametric version of the example, trencher's ILP grows exponentially whereas musketeer's is linear-sized.

5.8 Summary

In this chapter, we introduced an ILP encoding that allows us to determine both the places and the types of fences to insert to restore SC or weaker. We combine the construction of the AEG from the input program and the detection of critical cycles described in Chap. 3 with the construction of the ILP, its resolution and the actual insertion of memory fences and dependencies in the input program to get a completely automated source-to-source program transformer: musketeer.

We compared our tool musketeer to other existing tools synthesising fences in the verification community on classic examples from literature. The results suggest that musketeer scales to larger programs (memcached is composed of 9944 lines of C code), at the cost of an over-approximation. We also reimplemented other existing static analyses techniques and compared them with musketeer. We measured both the time of analysis required and ran some additional experiments to study the impact of the fences inferred by musketeer and the other techniques over runtime. We concluded that the analysis time in general matches the building time, and the runtime overhead for memcached with fences inferred by musketeer was not exceeding 3%. This would suggest that the strategy could be integrated in a compilation process.

We finally proposed alternative encodings, which appeared to be more complex but less efficient in practice. We shortly discussed how the control flow of the program reflected into the AEG and the fences inferred by musketeer. We finally summarised the axiomatic and operational existing techniques for synthesising fences.

Chapter 6

Future Work

6.1 Future work in program analyses

Equivalence In Chap. 2, we showed that a program analysis based on a non-relational domain was sound w.r.t. weak memory. It is not the case for some analyses working on relational domains. We believe that it is the case for all the relational analyses. If it is indeed the case, then by contraposition an analysis is non-relational if and only if it is sound for weak memory.

(trace) vs. (inter) Our proof relied on traces, meaning that we could only address analyses under their **(trace)** solution and, if it is sound for weak memory, extend the result to the **(inter)** solution because of Eq. 2.2.1. It might, however, be the case that, given an analysis, the imprecision due to the calculation of **(inter)** would cover the points missed by the **(trace)** solution. In other words, some analyses might be sound for weak memory under their **(inter)** form but not under their **(trace)** form.

Repairing strategy Finally, we provided in [AKL+11] a method for repairing analyses that are unsound for weak memory. The strategy consists of analysing each thread in isolation, feeding back the results of each thread to the other threads analyses, and iterating before reaching a fixed point. This strategy breaks relations between variables, but might break too much. There is probably room for precision improvement, by targeting only these relations that need to be broken.

6.2 Future work in critical cycle detection

In Chap. 3, we introduced a static detection of critical cycles that potentially prevent some reorderings of events under certain architectures. This method is sound in

its principles but incomplete. The imprecision could nevertheless be reduced with additional techniques.

Local analysis The analysis we constructed does not evaluate the conditionals used for jump. The result is that two events that could not exist in a same execution of the program on a real machine would appear in a same path in an exploration of our AEG. Some of the unfeasible cycle involving such events could be discarded by running symbolic execution engines between these events—a "*local analysis*". Note that some care must be taken, as conditionals can themselves depend on weak memory reorderings, and a reordering appearing in a cycle could allow another reordering in another cycle. The symbolic execution engine might need to be weak memory aware. None of these were available in the CProver framework at the time we wrote goto-instrument –mm. The work of [AKT13] has now been integrated in the framework for TSO and PSO, and we could exploit directly this engine.

Reachability analysis The reachability issue can appear in the heart of a potential critical cycle, but also between different cycles. A weak-memory analysis similar to the one we described earlier—but this time performed on the whole program—could also discard some cycles that are simply never reachable. This operation will be undecidable, unless some restrictions are made.

Synchronisation analysis Even though our analysis deals with low-level synchronisation, we did not combine it with a higher-level synchronisation analysis in our implementation. This means that we might assume that some (parts of) threads would interfere, whereas synchronisation mechanisms like locks, semaphores and others would prevent them from interfering in any execution. We believe that such an execution would strongly improve the scalability potential of the approach, as this could reduce exponentially the difficulty of some of the examples—since the complexity of our analysis is mostly due to the communications between threads.

Herding cats format In [AMT13], Alglave *et al.* defined a language to characterise succinctly a model with axioms as those that were used for example in Sec. 2.1.3 in Chap. 2. A definition of the Power model, more precise than the one we used in Sec. 2.1.3, can be expressed like in Fig. 6.1. They implemented an interpreter that parses the model and checks if an execution would be feasible under an architecture or not—i.e., if a sequence of events can be accepted by the model. We believe that

there exists a strong connection with context-free grammars and the string acceptors generated by parser generators like yacc/bison[1]. The language itself is based on partial orders rather than strings, but if we implement a theory for the WW/RW operators—that respectively mean that a string satisfying this predicate must start by a write (respectively read) and finish by a read—and for the transitive closure operator—whose rule would be very close to the string one—, then we can generate an acceptor given a .cat model for a sequence of events (i.e., an execution), after a slight disambiguation of the recursive part involving ii, ic and so on. A parser generated by yacc/bison would not be adapted to this context, since if it explores a sequence that is inconclusive, it will not backtrack but only return a parsing error. The Herd models contain non-deterministic choice, justifying the use of a generator that can explore several traces—btyacc[2] would be an option with this respect. The complexity of the string acceptance check is no longer linear, but the acceptor would be sound.

In the context of our static detection, this means that we could plug a sequence acceptor inside our static cycle detection that was directly generated by btyacc for a given model, and then have a generic approach to weak memory models. We can benefit from new model definitions. There are however some limitations to this approach. Firstly, all the optimisations to reduce the exploration of the AEG must be disabled, otherwise we might lose the soundness of the approach. This could strongly impact the scalability. Secondly, this only concerns the detection itself[3]. It is sufficient for concurrency cartography, for instance, but inserting instrumentations (like in Chap. 4) or synchronisations (like in Chap. 5) would not benefit from the new model. In fact, generating an operational mechanism from such axioms (for the instrumentation) or deriving the optimal fences required to ensure robustness, based on the axiom given, is a serious challenge. One option to avoid this last issue would consist of adopting different models for the static detection (generic) and the instrumentation/fence generation (hard-coded or semi-generic).

6.3 Future work in program instrumentation

We explained in Chap. 4 how to instrument the code to make explicit the event reorderings to SC analysers, and how to use the cycle detection before this instru-

[1]GNU Bison: `http://www.gnu.org/software/bison`.
[2]btyacc: `http://www.siber.com/btyacc`.
[3]This observation was initially made by Daniel Poetzl.

```
(* sc per location *)
acyclic po-loc|rf|fr|co

(* ppo *)
let dp = addr|data
let rdw = po-loc & (fre;rfe)
let detour = po-loc & (coe;rfe)

let ii0 = dp|rdw|rfi
let ic0 = 0
let ci0 = (ctrl+isync)|detour
let cc0 = dp|po-loc|ctrl|(addr;po)

let rec ii = ii0|ci|(ic;ci)|(ii;ii)
and ic = ic0|ii|cc|(ic;cc)|(ii;ic)
and ci = ci0|(ci;ii)|(cc;ci)
and cc = cc0|ci|(ci;ic)|(cc;cc)
let ppo = RR(ii)|RW(ic)

(* fences *)
let fence = RM(lwsync)|WW(lwsync)|sync

(* no thin air *)
let hb = ppo|fence|rfe
acyclic hb

(* prop *)
let prop-base = (fence|(rfe;fence)); hb*
let prop = WW(prop-base)|(com*;prop-base*;sync;hb*)

(* observation *)
irreflexive fre;prop;hb*

(* propagation *)
acyclic co|prop
```

Figure 6.1: An expression of the Power model with the Herding cat language, as defined in [AMT13].

mentation in order to simplify the instrumented program, and improve the scalability of the whole analysis.

Instrumentation strategy Given a critical cycle detected, we suggested three instrumentation strategies:

- instrumenting all the pairs of events that could reorder in each cycle;
- instrumenting one arbitrary pair of events that could reorder in each cycle;
- treating the whole set of pairs to instrument in all the cycles as an ILP and solving it to minimise a global cost of these instrumentations.

Other strategies can also be implemented—instrumenting for instance the writes first, if the write instrumentation is more efficient than the read one, or even selecting these pairs manually. We also assumed that all the cycles could equally be involved in executions in the cost function of the ILP, that may or may not reflect the reality. One option to confirm these coefficients or to give a larger priority to cycles that often occur during an execution is to set them dynamically by running the original program (augmented with some counters of visits in cycles) on the target machine and finding their visit frequencies. This approach could be later combined with machine learning techniques.

Bounds extension Even though the static detection of potential critical cycles does not assume any bound regarding buffers or queues thanks to the axiomatic model, the instrumentation itself is operational, and actual buffers are used to delay writes (we explained in Sec. 4.2.3 that the read queue was not bounded). A first question, as we mentioned in Sec. 4.2.3, is the class of programs that precisely require a write buffer of size n, and whether such programs can be found in e.g. Debian packages.

The general problem requires unbounded buffers, which leads to undecidability. Some abstractions over the write buffers, as suggested in [KVY11, DMVY15], would nevertheless increase the number of programs that could be proven.

In goto-instrument –mm, we implemented with one buffer per variable with a bound 2, which was sufficient for our experiments. These buffers (and their access functions) were actually hard-coded using predicates to facilitate the verification task for SC model checkers. There would be room for a more generic implementation of these buffers—but we need to care about the difficulty of the instrumented program and the possibilities offered by the third-party model checkers/analysers.

Dependency mechanism in instrumentation For Power and ARM, we detected data and address dependencies between events during the static detection—as a separated yet combined in practice analysis. The instrumentation itself does not enforce an operational mechanism ensuring dependencies—meaning that some reorderings not allowed due to dependencies by the processor(s) would be allowed in our instrumented program.

Operational TSO improvement The static cycle detection performs a thin detection of TSO and PSO potential reorderings. The instrumentation itself has been implemented as the most generic one, modelling in particular write buffers per variable for PSO and weaker. Because of this implementation choice, some instrumentations for TSO might produce a few spurious executions that should not be permitted on TSO. One way to circumvent that issue consists of modifying the access rules to the buffers and constraint them so that they behave as if they were one single buffer. Another more direct approach would be to reimplement directly with one single buffer.

Warnings w.r.t. dynamic verification with randomisation/scheduling The instrumentation allows several possible executions thanks to non-deterministic choices inserted in the code. Actually, the non-deterministic choice for the write to buffer or memory can be replaced by a pseudo-random choice[4] or a decision by a scheduler[5] if we want to exploit this transformation for testing or any technique relying on executing the program on an actual machine.

6.4 Future work in fence synthesis

Dynamic evaluation of the fence cost The method we described in Chap. 5, like all the other fence methods [AAC+13, AMSS10, KVY11, KVY10, LW13], is subject to the following criticism: why minimise the number and cost of fences regardless of how many times they are executed? To answer this concern, Bouajjani *et al.* mentioned in [BDM13] the possibility of parametrising the cost of a fence location by its frequency of use in the code, defined empirically or by analysis of the program. We believe that this approach would be practically hard to use, as it virtually impose to measure the frequencies of all the locations where we may have to insert fences.

[4]We could however not provide guarantees regarding the coverage in this case—the program becomes indeed only one possible instance of the original program.

[5]We would however need the scheduler to cover all the possible schedules so that properties of the scheduled program could transfer to the original one.

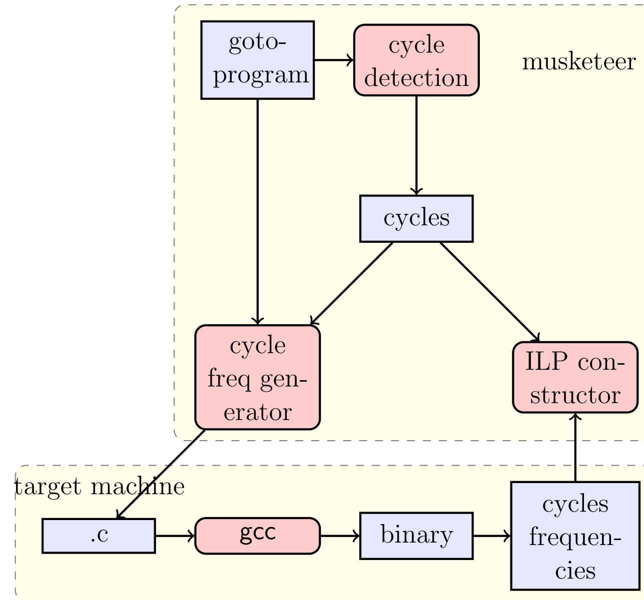


Figure 6.2: Critical cycle frequency analysis to tune the ILP formula.

A similar idea would be a dynamic method measuring the frequencies of critical cycles reached within a certain number of iterations of the original program on the target machine, and adjusting the cost w.r.t. these frequencies. The equation in Subsec. 5.2.1 would become:

$$\sum_{e \in \mathsf{po}} \Big[(\mathtt{dp}_e * \mathsf{cost}(\mathtt{dp}) + \mathtt{lwf}_e * \mathsf{cost}(\mathtt{lwf}) + \mathtt{cf}_e * \mathsf{cost}(\mathtt{cf}) + \mathtt{f}_e * \mathsf{cost}(\mathtt{f}) + \mathtt{br}_e * \mathsf{cost}(\mathtt{br})) \times \Big(\sum_{e \in C_j} \mathrm{freq}(C_j) + \varepsilon \Big) \Big].$$

We add an $\varepsilon > 0$ to the frequency of each critical cycle in order to ensure that all the cycles are always taken into account. This ensures that each edge likely to be fenced appears in the cost function to minimise. The opposite would result in *invisible variables* in the linear program, leading to no specific optimisation if all the frequencies measured are null. Note that the soundness is still always ensured by the constraints.

In practice, as depicted in Fig. 6.2, musketeer would add to the input C program a counter for each critical cycle that it detected earlier. The user takes this C program, compiles it on its target machine, then runs it a large number of times. We then generate a file with the frequencies measured, and the user can then transfer this file to the machine running musketeer and call it this time with this configuration-specific settings.

Chapter 7

Conclusion

Reasoning with weak memory models is not trivial, especially when it comes to writing programs. We presented in Chap. 1 the example of an implementation of Peterson's mutual exclusion algorithm, where the absence of memory fences induced some uncontrolled weak memory reorderings that may lead to the violation of mutual exclusion. Weak memory behaviours do not only interfere with specific synchronisation strategies in a theoretical context. In Chap. 4, we tested our instrumentation tool on a fragment of PostgreSQL that contained a genuine weak memory bug reported by the developers community. The non-expected weak memory behaviours could indeed induce a deadlock of the system, due to the loss of a token.

In order to facilitate concurrent programming with shared memory—and to re-exploit a good part of the existing code that is sometimes written with sequential consistency or a specific target architecture in mind—we developed in this dissertation automated static analyses to either make these weak memory behaviours directly apparent in the source, or simply forbid them. *We demonstrated the feasibility of automatic static analysis of axiomatically-defined weak memory models.*

In Chap. 4, we conceived an automated strategy that instruments the code with write buffers and read queues in the location where some reorderings of shared accesses might happen at runtime. This allows analysers handling interleavings to detect bugs that would manifest themselves only in the presence of specific reorderings. We evaluated the tool on a large set of Litmus tests. The resulting instrumented benchmarks have since then been used in [WKO13] and recently submitted to the Software Verification Competition, thanks to Björn Wachter. The tool itself is also available in Debian, in the package of `cbmc`.

In Chap. 5, we developed a strategy that inserts memory fences in the code so that it restores SC. The resulting code can thus be run without weak memory behaviours

affecting its semantics. The strategy optimises the placement of the fences in order to limit the impact of these on the runtime performance of the program. We measured for example a runtime overhead of only 3% on Power for Memcached.

These two strategies rely on a static detection of critical cycles, performed on an abstraction that we explained in Chap. 3. This detection can also be used directly for establishing a (over-approximating) cartography of the potential weak memory behaviours existing in a program. We provided a script with the tool which maps the cycles detected to an existing, simpler Litmus-like program in our databases of test that can be used for educational purposes. In [AMT13], Tautschnig re-used this detection to survey the potential critical cycles in a large set of program source from Debian packages.

On critical inspection of our work, we can identify the following reasons for the good scalability and applicability of our approaches:

1. where [ABP11] instrumented every access to shared memory in the program, we only targeted very specific places where weak memory reorderings might happen before instrumenting;
2. where [AAC+12] takes the whole semantics into account, including local variables operations, our static analysis completely abstracts the evaluations of expressions and conditions—at the cost of a lower precision;
3. our AEG and the detection performed on it rely on a reasonably simple axiomatic model [Alg10], that did not involve data structures and relations (like caches per thread and their related semantics) too complex to maintain during the program analysis;
4. when we combined the static detection—axiomatic—with the program instrumentation followed by a third-party tool analysis—operational—we did not try to have a perfectly precise approach for either the static detection or the instrumentation: we accepted a comprise between the two of them;
5. for the practicality and experiments, we wrote our tools in a framework for C program verification, CProver[1], that has been maintained for more than 10 years and allows us to write reasonably precise analyses for C with about 20 instructions—at the cost of diving into a very large, lightly documented[2] code base.

[1] `http://www.cprover.org`
[2] But Michael Tautschnig was an excellent code diving professor!

The axiomatic model allowed us to reason on well-understood static structures like graphs—which would also suggest the possibility of re-using some strategies coming from graph theory, as we briefly mentioned in Chap. 6—and we believe that it was the key to build an efficient static analysis for weak memory.

As we discussed in Chap. 6, there are still many points that need to be addressed in our approach to weak memory via static analysis. We have not measured the imprecision of the approach. Our empirical experiments showed in Chap. 5 that, despite the insertion of some spurious fences amongst those we inserted in memcached, we were getting at most a runtime overhead of 3% on ARM machines for our tests. These 3% would be fine for most applications; some applications might however require these 3%. In this case, some refinement of the abstraction might be a good direction to follow—at the probable cost of a slower static analysis. The work of Meshman et al. [MDVY14] would show a step in this direction.

We did not formalise the compiler optimisations[3] in our studies. The compiler optimisations could have an important impact over the semantics of the code actually executed on the machine, as the compiler can itself reorder some accesses to shared memory or ignore some dependencies or memory fences we would place in order to prevent weak memory reorderings. Based on empirical observations of `gcc -O0`, we assumed a simple generation of assembly code from C programs. We also checked empirically that our fences and dependencies inserted were maintained—even with `-O3` on simple programs. Describing the actual correspondence between the assembly and the C via compilers with optimisations, and the soundness of our approaches with respect to these, are all vast, open questions that we did not address.

The semantics generating event structures from goto-programs was also not defined–which forced us to make three assumptions regarding it. Albeit reasonably intuitive in words, the formalisation of this semantics—in a sound *and complete* sense—is also a challenge, since such a formalisation would actually fill the gap between the static world of C programs and the dynamic world of event structures and their related executions, valid or invalid on architectures. The AEG is a first answer to this question—but remains incomplete, i.e., it remains an over-approximation.

We always targeted *soundness* in our work. If *completeness* is a requirement for a problem, then static analysis such as the one we developed is probably not a good answer to this. If one targets TSO, PSO or RMO, one might in that case prefer to consider the tools developed by other research teams that we listed in Fig. 5.22 in

[3]By "optimisations", we mean any operation that would lead to simplify the code generated—even at `-O0`.

Chap. 5. We also provide the versions we used in our experiments in Sec. E.3 in App. E, along with their respective licenses and their authors to contact.

Finally, most of this work was based on the research of Shasha and Snir reported in [SS88]. This work, which was inspired from databases research, had some consequences in the distributed system community, the verification community and perhaps also the database community. There will certainly be in the future some great opportunities in transferring results from one of these branches to the others. A lot of interesting research still lies ahead.

Bibliography

[AAC+12] P. Abdulla, M. F. Atig, Y. Chen, C. Leonardsson, and A. Rezine, *Counter-example guided fence insertion under TSO*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2012.

[AAC+13] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine, *Memorax, a precise and sound tool for automatic fence insertion under TSO*, TACAS, 2013.

[ABBM10] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi, *On the verification problem for weak memory models*, POPL, 2010.

[ABBM12] ———, *What's decidable about weak memory models?*, European Symposium on Programming (ESOP), 2012.

[ABP11] M. F. Atig, A. Bouajjani, and G. Parlato, *Getting rid of store-buffers in the analysis of weak memory models*, Computer Aided Verification (CAV), 2011.

[AKL+11] Jade Alglave, Daniel Kroening, John Lugton, Vincent Nimal, and Michael Tautschnig, *Soundness of data flow analyses for weak memory models*, APLAS, 2011, pp. 272–288.

[AKNP14] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl, *Don't sit on the fence - A static analysis approach to automatic fence insertion*, Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, 2014, pp. 508–524.

[AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig, *Software verification for weak memory via program transformation*, ESOP, 2013.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig, *Partial orders for efficient bounded model-checking of concurrent software*, CAV, 2013.

[Alg10] J. Alglave, *A shared memory poetics*, Ph.D. thesis, Université Paris 7 and INRIA, 2010.

[AM11] J. Alglave and L. Maranget, *Stability in weak memory models*, Computer Aided Verification (CAV), LNCS, vol. 6806, Springer, 2011, pp. 50–66.

[AMSS10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell, *Fences in weak memory models*, CAV, 2010.

[AMSS11] ______, *Litmus: Running tests against hardware*, Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, 2011, pp. 41–44.

[AMT13] Jade Alglave, Luc Maranget, and Michael Tautschnig, *Herding cats*, CoRR **abs/1308.6810** (2013).

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

[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer, *Checking and enforcing robustness against TSO*, ESOP, 2013.

[BJG08] J. Bang-Jensen and G.Z. Gutin, *Digraphs: theory, algorithms and applications*, Springer, 2008.

[BMM11] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann, *Deciding robustness against total store ordering*, ICALP (2), 2011, pp. 428–440.

[BvL87] G. J. Bezem and J. van Leeuwen, *Enumeration in graph*, Tech. report, Rijskuniversiteit Utrecht, May 1987.

[c1111] *Information technology – Programming languages – C*, BS ISO/IEC 9899:2011, 2011.

[CC76] P. Cousot and R. Cousot, *Static determination of dynamic properties of programs*, Proceedings of the Second International Symposium on Programming, Dunod, Paris, France, 1976, pp. 106–130.

[CC92] ———, *Abstract interpretation frameworks*, Journal of Logic and Computation **2** (1992), no. 4, 511–547.

[CF11] Lucas Cordeiro and Bernd Fischer, *Verifying multi-threaded software using smt-based context-bounded model checking*, Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, 2011, pp. 331–340.

[CH78] P. Cousot and N. Halbwachs, *Automatic discovery of linear restraints among variables of a program*, Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Tucson, Arizona), ACM Press, New York, NY, 1978, pp. 84–97.

[Cha68] J.P. Char, *Master circuit matrix*, Electrical Engineers, Proceedings of the Institution of **115** (June 1968), no. 6, 762–770.

[CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav, *SATABS: SAT-based predicate abstraction for ANSI-C*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), Lecture Notes in Computer Science, vol. 3440, Springer Verlag, 2005, pp. 570–574.

[CL05] David Chase and Yossi Lev, *Dynamic circular work-stealing deque*, SPAA, 2005.

[Cou96] Patrick Cousot, *Abstract interpretation*, ACM Comput. Surv. **28** (1996), no. 2, 324–328.

[cpp11] *Information technology – Programming languages – C++*, BS ISO/IEC 14882:2011, 2011.

[CVJL08] Ravi Chugh, Jan Wen Voung, Ranjit Jhala, and Sorin Lerner, *Dataflow analysis for concurrent programs using datarace detection*, PLDI, 2008, pp. 316–326.

[Dan68] G. Danielson, *On finding the simple paths and circuits in a graph*, Circuit Theory, IEEE Transactions on **15** (September 1968), no. 3, 294–295.

[DDN11] Arnab De, Deepak D'Souza, and Rupesh Nasre, *Dataflow analysis for datarace-free programs*, Proc. of the 20th European Symposium on Programming (ESOP'11) (Saarbrücken, Germany), LNCS, vol. 6602, Springer, March 2011.

[Deu92] A. Deutsch, *A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations*, International Conference on Computer Languages, 1992, pp. 2–13.

[DFG+00] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul A. Martin, Nir Shavit, and Guy L. Steele Jr., *Even better DCAS-based concurrent deques*, DISC, 2000.

[Dic09] David Dice, November 2009.

[Dij65] Edsger W. Dijkstra, *Solution of a problem in concurrent programming control*, Commun. ACM **8** (1965), no. 9, 569.

[DM14] Egor Derevenetc and Roland Meyer, *Robustness against Power is PSpace-complete*, Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II, 2014, pp. 158–170.

[DMD13] Mathieu Desnoyers, Paul E. McKenney, and Michel R. Dagenais, *Multicore systems modeling for formal verification of parallel algorithms*, Operating Systems Review **47** (2013), no. 2, 51–65.

[DMVY13] Andrei Marian Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav, *Predicate abstraction for relaxed memory models.*, Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings, 2013, pp. 84–104.

[DMVY15] Andrei Dan, Yuri Meshman, Martin Vechev, and Eran Yahav, *Effective abstractions for verification under relaxed memory models*, VMCAI 2015, 2015, p. to appear.

[D'S10] Vijay D'Silva, *Software verification lectures.*

[ES03] Niklas Eén and Niklas Sörensson, *Temporal induction by incremental SAT solving*, Electr. Notes Theor. Comput. Sci. **89** (2003), no. 4, 543–560.

[Fer08] Pietro Ferrara, *Static analysis via abstract interpretation of the happens-before memory model*, TAP, 2008, pp. 116–133.

[FK10] Azadeh Farzan and Zachary Kincaid, *Compositional bitvector analysis for concurrent programs with nested locks*, SAS, 2010, pp. 253–270.

[FLM03] Xing Fang, Jaejin Lee, and Samuel P. Midkiff, *Automatic fence insertion for shared memory multiprocessing*, ICS, 2003.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall, *The implementation of the Cilk-5 multithreaded language*, PLDI, 1998.

[GPR11] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko, *Threader: A constraint-based verifier for multi-threaded programs*, Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, 2011, pp. 412–417.

[HK09] Jacob M. Howe and Andy King, *Logahedra: A new weakly relational domain*, ATVA, 2009, pp. 306–320.

[HKSV97] R.H. Hardin, R.P. Kurshan, S.K. Shukla, and M.Y. Vardi, *A new heuristic for bad cycle detection using bdds*, Computer Aided Verification (Orna Grumberg, ed.), Lecture Notes in Computer Science, vol. 1254, Springer Berlin Heidelberg, 1997, pp. 268–278.

[HR06] Thuan Quang Huynh and Abhik Roychoudhury, *A memory model sensitive checker for C#*, FM, 2006.

[Jea09] Bertrand Jeannet, *Relational interprocedural verification of concurrent programs*, SEFM, 2009, pp. 83–92.

[JGR05] Bertrand Jeannet, Denis Gopan, and Thomas W. Reps, *A relational abstraction for functions*, SAS, 2005, pp. 186–202.

[JM09] Bertrand Jeannet and Antoine Miné, *Apron: A library of numerical abstract domains for static analysis*, Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings, 2009, pp. 661–667.

[Joh75] Donald B. Johnson, *Finding all the elementary circuits of a directed graph*, SIAM J. Comput. **4** (1975), no. 1, 77–84.

[JYKS12] H. Jin, T. Yavuz-Kahveci, and B. A. Sanders, *Java memory model-aware model checking*, TACAS, 2012.

[Kam67] T. Kamae, *A systematic method of finding all directed circuits and enumerating all directed paths*, Circuit Theory, IEEE Transactions on **14** (June 1967), no. 2, 166–171.

[KD94] Uday P. Khedker and Dhananjay M. Dhamdhere, *A generalized theory of bit vector data flow analysis*, ACM Transactions on Programming Languages and Systems **16** (1994), 1472–1511.

[Knu68] D. Knuth, *The art of computer programming*, 1968.

[KSKZ09] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang, *Static data race detection for concurrent programs with asynchronous calls*, FSE, 2009.

[KSV96] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer, *Parallelism for free: Efficient and optimal bitvector analyses for parallel programs*, ACM Trans. Program. Lang. Syst. **18** (1996), no. 3, 268–299.

[KVY10] Michael Kuperstein, Martin T. Vechev, and Eran Yahav, *Automatic inference of memory fences*, FMCAD, 2010.

[KVY11] ———, *Partial-coherence abstractions for relaxed memory models*, PLDI, 2011.

[KY96] Arvind Krishnamurthy and Katherine A. Yelick, *Analyses and optimizations for shared address space programs*, J. Par. Dist. Comp. **38** (1996), no. 2.

[Lam79] Leslie Lamport, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Trans. Computers **28** (1979), no. 9, 690–691.

[Lam87] ———, *A fast mutual exclusion algorithm*, ACM Trans. Comput. Syst. **5** (1987), no. 1.

[Lei08] K. Rustan M. Leino, *This is boogie 2*, Tech. report, June 2008.

[LNP+12] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav, *Dynamic synthesis for relaxed memory models*, PLDI, 2012.

[LP01] J. Lee and D.A. Padua, *Hiding relaxed memory consistency with a compiler*, IEEE Transactions on Computers **50** (2001), 824–833.

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri, *Corral: A solver for reachability modulo theories*, Computer-Aided Verification (CAV), July 2012.

[LR92] William Landi and Barbara G. Ryder, *A safe approximate algorithm for interprocedural pointer aliasing*, 1992.

[LT82] G. Loizou and P. Thanisch, *Enumerating the cycles of a digraph: A new preprocessing strategy*, Information Sciences **27** (1982), no. 3, 163 – 182.

[LW11] A. Linden and P. Wolper, *A verification-based approach to memory fence insertion in relaxed memory systems*, SPIN, 2011.

[LW13] Alexander Linden and Pierre Wolper, *A verification-based approach to memory fence insertion in PSO memory systems*, TACAS, 2013.

[MDVY14] Yuri Meshman, Andrei Marian Dan, Martin T. Vechev, and Eran Yahav, *Synthesis of memory fences via refinement propagation*, Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings, 2014, pp. 237–252.

[Min01] Antoine Miné, *The octagon abstract domain*, WCRE, 2001, pp. 310–.

[Min11] ______, *Static analysis of run-time errors in embedded critical parallel C programs*, Proc. of the 20th European Symposium on Programming (ESOP'11) (Saarbrücken, Germany), LNCS, vol. 6602, Springer, March 2011.

[MS96] Maged M. Michael and Michael L. Scott, *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*, PODC, 1996.

[MSM+11] Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy, *A case for an SC-preserving compiler*, PLDI, 2011.

[MVS09] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat, *Idempotent work stealing*, PPOPP, 2009.

[ND13] Brian Norris and Brian Demsky, *CDSchecker: checking concurrent data structures written with C/C++ atomics*, OOPSLA, 2013.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin, *Principles of program analysis*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[NPW79] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel, *Petri nets, event structures and domains*, Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2-4, 1979, 1979, pp. 266–284.

[OSS09] S. Owens, S. Sarkar, and P. Sewell, *A better x86 memory model: x86-TSO*, Theorem Proving in Higher Order Logics (TPHOLs), LNCS, vol. 5674, Springer, 2009, pp. 391–407.

[Pat69] Keith Paton, *An algorithm for finding a fundamental set of cycles of a graph*, Commun. ACM **12** (1969), no. 9, 514–518.

[Pet81] Gary L. Peterson, *Myths about the mutual exclusion problem*, Inf. Process. Lett. **12** (1981), no. 3.

[PM64] J. W. La Patra and B. R. Myers, *Algorithm for circuit enumeration*, IEEE internat. Convention Record **1** (1964), 368–373.

[Pon66] J. Ponstein, *Self-avoiding paths and the adjacency matrix of a graph*, SIAM Journal on Applied Mathematics **14** (1966), no. 3, 600–609.

[ppc09] *Power ISA version 2.06*, 2009.

[RLL07] K. Rustan, M. Leino, and F. Logozzo, *Using widenings to infer loop invariants inside an SMT solver, or: A theorem prover as abstract domain*, International Workshop on Invariant Generation, 2007.

[RR99] Radu Rugina and Martin C. Rinard, *Pointer analysis for multithreaded programs*, PLDI, 1999, pp. 77–90.

[SFW+05] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David A. Padua, *Compiler techniques for high performance sequentially consistent Java programs*, PPOPP, 2005.

[SKH02] Axel Simon, Andy King, and Jacob M. Howe, *Two variables per linear inequality as an abstract domain*, LOPSTR, 2002, pp. 71–89.

[SL76] Jayme L. Szwarcfiter and Peter E. Lauer, *A search strategy for the elementary cycles of a directed graph*, BIT Numerical Mathematics **16** (1976), 192–204 (English).

[SNM+12] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd D. Millstein, and Madanlal Musuvathi, *End-to-end sequential consistency*, 39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA, 2012, pp. 524–535.

[spa94] *Sparc Architecture Manual Version 9*, 1994.

[Sri79] P.K. Srimani, *Generation of directed circuits in a directed graph*, Proceedings of the IEEE **67** (Sept. 1979), no. 9, 1361–1362.

[SS79] Pradip K. Srimani and Abhijit Sengupta, *Algebraic determination of circuits in a directed graph*, International Journal of Systems Science **10** (1979), no. 12, 1409–1413.

[SS88] Dennis Shasha and Marc Snir, *Efficient and correct execution of parallel programs that share memory*, ACM Trans. Program. Lang. Syst. **10** (1988), no. 2, 282–312.

[SSA+11] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, *Understanding Power multiprocessors*, PLDI, 2011.

[SSN+09] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave, *The semantics of x86-cc multiprocessor machine code*, Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009, 2009, pp. 379–391.

[Ste96] Bjarne Steensgaard, *Points-to analysis in almost linear time*, POPL, 1996, pp. 32–41.

[Sys73] Maciej M. Syslo, *Algorithm 459: the elementary circuits of a graph*, Commun. ACM **16** (1973), no. 10, 632–633.

[Szy88] Boleslaw K. Szymanski, *A simple solution to Lamport's concurrent programming problem with linear wait*, ICS, 1988.

[Tan95] Andrew S. Tanenbaum, *Distributed operating systems*, Prentice Hall, 1995.

[Tar72] Robert Endre Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput. **1** (1972), no. 2, 146–160.

[Tar73] ______, *Enumeration of the elementary circuits of a directed graph*, SIAM J. Comput. **2** (1973), no. 3, 211–216.

[Thu06] Marc Thurley, *sharpsat - counting models with advanced component caching and implicit BCP*, SAT, 2006, pp. 424–429.

[Tie70] James C. Tiernan, *An efficient search algorithm to find the elementary circuits of a graph*, Commun. ACM **13** (1970), no. 12, 722–726.

[Val79] Leslie G. Valiant, *The complexity of enumeration and reliability problems*, SIAM J. Comput. **8** (1979), no. 3, 410–421.

[VZN11] Viktor Vafeiadis and Francesco Zappa Nardelli, *Verifying fence elimination optimisations*, SAS, 2011.

[War62] Stephen Warshall, *A theorem on boolean matrices*, J. ACM **9** (1962), no. 1, 11–12.

[Wei72] Herbert Weinblatt, *A new search algorithm for finding the simple cycles of a finite directed graph*, J. ACM **19** (1972), no. 1, 43–56.

[Win82] Glynn Winskel, *Event structure semantics for CCS and related languages*, Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings, 1982, pp. 561–576.

[WKO13] Björn Wachter, Daniel Kroening, and Joël Ouaknine, *Verifying multi-threaded software with impact*, Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, 2013, pp. 210–217.

[Yan04] Y. Yang, *Formalizing shared memory consistency models for program analysis*, Ph.D. thesis, University of Utah, 2004, `http://www.cs.utah.edu/~yyang/papers/thesis.pdf`.

[Yau68] S. Yau, *Generation of all hamiltonian circuits, paths, and centers of a graph, and related problems*, Circuit Theory, IEEE Transactions on **14** (March 1968), no. 1, 79–81.

Glossary

Weak memory model framework

Memory event	A memory event is a write or read event (in $\mathbb{E}$)	p. 22
$\text{es} = (\mathbb{E}, \mathsf{po})$	Event structure	p. 23
$\text{X} = (\mathsf{rf}, \mathsf{ws})$	Execution	p. 24
Existence	Existence of an event structure, given a program	p. 23
Compatibility	Compatibility of an execution with an event structure, regardless of the architecture	p. 24
Validity	Validity of an execution w.r.t. an event structure, given an architecture	p. 29
(uniproc)	Uniproc property	p. 29
(GHB)	Global happen before	p. 29
(thin-air)	Thin-air property	p. 29
$<_X$	Total order on the events, given a valid execution X	p. 28
$\text{W}\mathsf{a}v$	Write event of the value v at address a	p. 22
$\text{R}\mathsf{a}v$	Read event of the value v from address a	p. 22
po	Program order	p. 23
po-loc	Program order per location	p. 29
ppo	Preserved program order, given an architecture	p. 29
rf	Read-from	p. 25
rfe	External read-from	p. 66
rfi	Internal read-from	p. 100
fr	From-read	p. 25
fre	External read-from	p. 66
ws	Write coherence	p. 25
grf	Global read from (i.e., maintained communications)	p. 29
dp	Control-, address- and data-dependency	p. 27
Architecture $= (\mathsf{ppo}, \mathsf{grf})$	Maintained pairs of events	p. 29
$<$	Strength order between the architectures	p. 45

Static extension to the weak memory model framework

AEG $= (\mathbb{E}_s, \mathrm{po}_s, \mathrm{cmp})$	Abstract event graph	p. 61
po_s	Static program order	p. 66
cmp	Competing pairs	p. 66
$\mathrm{sym}(R)$	Symmetric relation of R	p. 71
$\otimes$	Competing pair operator	p. 71
$\bar{\emptyset}$	Triple of empty sets	p. 71

Scope of the analyses

(full)	Addressing all the behaviours	p. 31
(prop)	Addressing only the behaviours that affect a property or an assertion	p. 32

Properties for weak memory models

$\mathsf{traces}_A(P)$	Set of executions valid for an event structure extracted from the program P valid under the architecture A	p. 45
$\mathsf{robust}_A(P)$	Robustness of program P under architecture A	p. 45
$\mathsf{robust}_{A/B}(P)$	Parametric robustness	p. 45
$\mathsf{trans}_{B\to A}(P, P')$	Transformation of P into P' so that P' run under architecture A contains all the behaviours of P run under architecture B	p. 47
$\mathsf{sound}_A(f)$	Soundness of program analysis f under architecture A	p. 48
$\mathsf{sound}_{A/B}(f)$	Parametric soundness of program analysis f under architecture A	p. 49
$\mathsf{sound}_A(f, P)$	Soundness of program analysis f under architecture A for a program P	p. 48
$\mathsf{sound}_{A/B}(f, P)$	Parametric soundness of program analysis f under architecture A for a program P	p. 49
$\mathsf{repair}_{A\to B}(f, f')$	f' assuming architecture B returns all the results of f assuming architecture A	p. 50
$\mathsf{ccycles}_A(P)$	Set of potential critical cycles in P under architecture A	p. 52

Weak memory instrumentation

Machine state = $(mem, \mathbf{b}, \mathbf{rs})$	State of the machine	p. 96
mem	Memory mapping	p. 96
$\mathbf{b}$	Write buffer	p. 96
$\mathbf{rs}$	Read set	p. 96
$\mathrm{rr}(R, S)$	Restricts relation R to $S \times S$	p. 96
addr	Address of an event	p. 98
last	Last write event of a buffer	p. 99
$delay$	Label for an event	p. 98
$flush$	Label for an event	p. 98
se	Safe exit	p. 101
de	Delayed exit	p. 101
ptoX	Maps an accepted path of labels to an execution	p. 102
Xtop	Maps an execution to an accepted path of labels	p. 102
$\mathrm{ndelays}(E, X)$	Pairs that cannot be relaxed under the architecture	p. 102
cost	(Assumed) Cost of a given fence	p. 109
$delays(C)$	Pairs of events potentially relaxed in the static critical cycle C	p. 109
poWR	Pairs of (write event, read event) in po_s	p. 108
poWW	Pairs of (write event, write event) in po_s	p. 108
poRW	Pairs of (read event, write event) in po_s	p. 108
poRR	Pairs of (read event, read event) in po_s	p. 108
$\mathrm{delays}_{\nearrow}$	Set of po_s^+ delays that do not intersect with other delays	p. 142
$\mathrm{delays}_{\nwarrow\!\!\!\nearrow}$	Set of po_s^+ delays that intersect with other delays	p. 142

Memory fence synthesis

f	Full fence variable	p. 130
lwf	Lightweight fence variable	p. 130
cf	Control fence variable	p. 130
dp	Dependency variable	p. 130
actual-places	Set of places where fences should be inserted, according to the ILP solution	p. 129
potential-places	Set of possible places where fences can be inserted	p. 131
between	po_s edges shared between two (or more) critical cycles	p. 132
ctrl	Places in which a control fence can be inserted	p. 132
cumul(p)	Places in which a fence with cumulativity would maintain the pair of events p	p. 133
γ_v	Decomposes a set of po_s^+ edges into a set of po_s edges	p. 142
α_v	Recomposes a set of po_s^+ edges from a set of po_s edges	p. 142

Program analyses for weak memory

(mop)	Meet-over-path solution	p. 33
(mfp)	Maximum-fixed-point solution	p. 33
(trace)	Interleaving semantics	p. 34
(inter)	Interference semantics	p. 34
aes	Abstract event structure	p. 36
conc	Concretisation mapping	p. 37
S^{aes}	Maps a program to its aes	p. 41
$[\![.]\!]$	Analysis	p. 38
$\langle\!\langle . \rangle\!\rangle$	Non-relational analysis	p. 40
$\text{values}_A(E, R)$	values possible for E with R as initial environment, under an architecture A	p. 38
$\widehat{\text{values}_A(E}, R)$	(non-relational) values possible for E with R as initial environment, under an architecture A	p. 40
$\widetilde{\text{values}}(P)$	values yielded by the program P under architecture A	p. 41

Appendix A

A short survey of sound and unsound domains

A.1 Numerical abstractions

The objective of a numerical abstraction is to represent a set of values with a simpler, more abstract object. Ideally, the abstraction α and concretisation γ form a Galois connection from the concrete lattice to the abstract lattice, i.e., they satisfy $\gamma \circ \alpha \supseteq id_{\mathcal{D}}$ and $\alpha \circ \gamma \sqsubseteq id_{\mathcal{A}}$ [NNH99, p. 234], where id is the identity function. A program analysis written as a Galois connection will be sound by definition. If its domain is relational, and if it is sound for concurrent program under SC, then Thm. 1 guarantees that the analysis is sound w.r.t. weak memory models.

Box abstraction Let us consider the box abstraction [CC76]. Every variable is mapped to an interval, which contains all the possible values. The abstract lattice[1] is $< Var \rightarrow Val \times Val, \sqsubseteq, \sqcup, \sqcap >$ is defined with $[a,b] \sqsubseteq [c,d] \triangleq c \leq a \wedge b \leq d$, $[a,b] \sqcup [c,d] \triangleq [min(a,c), max(b,d)]$ (providing none of them is empty) and $[a,b] \sqcap [c,d] \triangleq [max(a,c), min(b,d)]$. The variables are not related, so we can express the values back to the concrete domain with $\wp(Var \times Val)$: this abstract domain is non-relational.

Weakly relational abstractions In the *octagon abstraction* [Min01], pairs of variables can be related in an octagon. This means that a pair (x,y) of variables can be bounded by constraints of the form $ax + by \leq c$, for two integers a and b in $\{-1, 0, 1\}$ and c in $\mathbb{Q}$. As we explained in the previous subsection, the concretisation of a set of octagons lies in $\wp\wp(Var \times Val)$: the domain is relational.

[1] We assume here that the segment $[a,b]$ with $b < a$ is the empty set.

The *logahedron abstraction* is a generalisation of the octagon [HK09]. The only difference is that, for a given n, the constraints are of the form $ax + by \leq c$, where x and y are the values of the variables, a and b two integers in $\{-2^n, 0, 2^n\}$ and c in $\mathbb{Q}$.

Two Variables Per Inequality (TVPI) [SKH02] generalises logahedron, with relating pairs of variables in generic inequalities, that is, using constraints of the form $ax + by \leq c$, where $a, b, c \in \mathbb{Q}$. As both of these domains are more precise than octagon abstract domain, they are also relational.

Polyhedron abstraction Cousot and Halbwachs's polyhedron [CH78] relates all the variables in inequalities: $\sum_i a_i x_i \leq d$, with $\forall i, a_i \in \mathbb{Q}$ and $d \in \mathbb{Q}$. The domain is relational.

Octagon, logahedron and TVPI abstract interpretations are called *weakly relational* domains, as they offer a compromise between the precision (obviously lower than the polyhedron one) and efficiency (polynomial complexity, whereas polyhedra are exponential) .

Abstract interpretation for concurrent programs The abstract interpretations we introduced previously were originally designed for sequential programs [CH78, Min01, HK09, SKH02]. They can be adapted to concurrent programs using **(trace)** or **(inter)** solutions. [Jea09] proposes to adapt [CH78, Min01] (and could extend in the same way [HK09, SKH02]) for concurrent programs, combining the previous numerical abstraction with a stack abstraction. The stack abstraction makes the analysis context-sensitive, and by instrumenting the program, the analysis can handle concurrent program while preserving safety properties: if the abstract numerical domain is non-relational, then the analysis for concurrent programs is also sound w.r.t. weak memory models.

A.2 Pointers analyses

Alias and points-to analyses The goal of a (*may*) pointer analysis is to reveal at least all the locations that the pointers (or references) may point to during a program execution. There are mainly two approaches: the *points-to* analysis and the *alias* analysis [Ste96]. The former computes all the addresses (or abstract memory locations) which can be hold in memory at a pointer address. The latter collects and maintains classes of aliases to a same object. The memory locations are however not modelled in this case: the whole path to the pointed object has to be stored. For example, if

$p \rightarrow next \rightarrow data$, stored at address $\&pnd$, and $q \rightarrow data$, stored at address $\&qd$, point to the same object r, the points-to analysis will report $\{(\&pnd, \&r), (\&qd, \&r)\}$, whereas the alias analysis will store $\{\{p \rightarrow next \rightarrow data, q \rightarrow data\}\}$, where $\{p \rightarrow next \rightarrow data, q \rightarrow data\}$ encodes the class of pointers pointing to r. Alias analyses cannot be directly expressed in terms of Var and Val, because the addresses and values have been abstracted. They actually work on relations between variables only, i.e., $Var \rightarrow \wp(Var)$, which can be expressed as $\wp\wp(Var)$—and generalised to $\wp\wp(Var \times Val)$—but not as $\wp(Var)$. Therefore, the domain is relational, as in [LR92, Deu92].

Collecting points-to analyses A flow-sensitive points-to analysis returns a set of pairs in $Addr \times Val$ per line, representing the pointers and the locations they point to after each statement, usually displayed with a graph [NNH99]. To adapt this strategy to a concurrent program, we compute all the interleavings of the program and run the analysis on them. The analysis is SC-sound.

The number of interleavings to compute is, however, factorial, so other strategies should be preferred for actual code analysis. Rinard and Rugina proposed in [RR99] to compute the interferences of the threads instead of considering all the interleavings. The analysis does not propagate one graph of pointers to locations, it propagates three points-to graphs. The idea behind this is to analyse each thread with the initial points-to graph, and collect all the relations possibly created during the analysis of the thread in isolation. This interference graph of potentially new edges is fed back into all the threads except this one, and analyses are run again, taking into account these new edges, and iterated until a fixed point is reached.

Both analyses are SC-sound and non-relational, since their results are expressible using $\wp(Var \times Val)$.

Partition refinement points-to analyses Steensgaard proposed in [Ste96] a points-to analysis which is flow-insensitive. The analysis is less precise than the other ones, but it is almost linear in time and can be directly applied to concurrent programs without modification, ensuring SC-soundness. The algorithm builds for every pointer a type, and successively refines the environment of types using a set of rules for the language, until the whole program is "well-typed" *w.r.t.* this environment. The types actually represent some parts of the memory, and we end up with a partition of the memory. The final domain is non-relational, representing the memory, and the flow-insensitivity entails the SC-soundness.

A.3 (Other) Classes of Analyses

Separable and non-separable analyses Khedker and Dhamdhere introduced in [KD94, p. 13] the *separability* property for analyses. The abstract domain of a separable analysis is expressed as a Cartesian product of sets, i.e., $A_1 \times \ldots \times A_n$, where each $\langle A_i, \sqsubseteq_i, \sqcap_i \rangle$ forms a semi-lattice. All the abstract transformers work independently on every component of a vector in this domain. For example, a meet $\sqcap$ in the (semi-) lattice of the abstract domain $\langle A_1 \times \ldots \times A_n, \sqsubseteq, \sqcap \rangle$, is defined as $a \sqcap b = (a_1 \sqcap_1 b_1, \ldots, a_n \sqcap_n b_n)$. If we consider analyses satisfying this property, and working with Var and Val, it is clear that they can be expressed on the abstract domain $\wp(Val)^{\#Var}$, where $X^N \triangleq \underbrace{X \times \ldots \times X}_{N \text{ times}}$. Indeed, $\wp(Val)^{\#Var}$ is isomorphic to $Var \rightarrow \wp(Val)$ (as long as we work on a finite number of variables), so the domain is non-relational.Yet, we cannot conclude about the non-separable analyses.

Bit-vector analyses The *bit-vector* framework is a subset of the monotonic data flow equations framework, a generic framework which defines analyses through equations with monotonic functions [NNH99, p. 65]. Let us consider we are working on a lattice with tuples of bits. Khedker and Dhamdhere defines the bit-vector framework as a data flow framework working with only monotonic bit-vector transformers [KD94, p. 14]. That is, transformers in $\mathcal{A} \rightarrow \mathcal{A}$ such that a transformer, for a given statement, can only work independently on every bit. Symbolically, if h_s is such a transformer for a statement s which works on the bit tuple X to propagate Y, i.e., $h_s(X) = Y$, for all i, $Y_i = h_s^i(X_i)$. The transformer can be split into n separated transformers, working on the boolean lattice. The transformers have to be monotonic, so it is clear that none of the bits can be switched, which means that none of the h_s^i can compute the negation of the corresponding bit. For a given statement, the bit can only be propagated, always put to 1 or always to 0. This (sequential) framework is separable, and thus works on a non-relational domain.

Concurrent bit-vector analyses The previous framework for bit-vector analyses is extended to concurrent program analyses in [KSV96]. It means that if an analysis working on Var and Val is defined with this framework, the abstract domain is non-relational.

[FK10] also extends bit-vector analyses to concurrent programs, albeit using an alternative definition of the framework. For a given statement s, the abstract transformer corresponding to this statement is of the form $(X \cup Gen(s)) \setminus Kill(s)$, for X

the incoming flow (represented here as a set). In other words, the created and killed values only depend on the statement, and not on the flow. For finite lattices, this is exactly equivalent[2] to Khedker and Dhamdhere's definition [KD94, p. 14]. Indeed, we can represent a (finite) set $S \subseteq \mathcal{A}$ by a tuple of bits b, where every bit b_i represents the property $a_i \in S$, for all the a_i of the domain $\mathcal{A}$. $Gen(s)$ represents the bits always raised to 1 at s, $Kill(s)$ the 0 ones, and the $\cup$ and $\setminus$ operators maintain all the other bits, i.e., they propagate.

[2][NNH99, p. 137] proposes a third equivalent definition, with $(X \cap Kill(s)) \cup Gen(s)$.

Appendix B

Ideas towards soundness arguments

We describe in this appendix some ideas towards formal soundness arguments for the AEG construction in Chap. 3 and for the static instrumentation explained in Chap. 4. The soundness of the fence synthesis can be derived from the soundness of the AEG construction for the critical cycle detection and the mappings detailed in the work of Alglave et Maranget in [AM11].

Contrary to the rest of the dissertation, which was published via the articles [AKL+11, AKNT13, AKNP14], the ideas described in this appendix have not been peer-reviewed by the verification community.

B.1 Assumptions regarding $\mathrm{S}^?$, the semantics from C to event structures

We will assume a set of basic rules that the semantics $\mathrm{S}^?$ should meet, and use these assumptions in our soundness proof to show that the AEG indeed captures all the executions.

The first rule connects the existence of memory events in an event structure E to shared memory accesses in the input goto-program P. We call a trace t a path in the control-flow graph of P (written $\mathrm{CFG}(P)$) and $t[i]$ the ith instruction. We recall that $E.\mathbb{E}$ is the set of events of the event structure E, and $\mathrm{evts}(i)$ the events yielded by instruction i.

$$\frac{e_1 \in E.\mathbb{E} \qquad E \in \mathrm{S}^?(P)}{\exists t \in \mathrm{CFG}(P), \exists i, e_1 \in \mathrm{evts}(t[i])} \tag{B.1}$$

The second rule ensures the existence of a path between two events of the event structure that would be connected by a po edge. We assume in this case the existence of a path in the CFG of the program that would connect the two accesses to the shared

memory that yielded the events. We write in this case $t[i] \xrightarrow{\text{CFG}} t[j]$. We only require the path to follow the CFG of P, we do not require that the guards along the path are met. The reason for this is that the AEG that we will construct is abstracting all the guards, since some of the paths might actually need a weak memory reordering to exist. For example, if we would write a conditional jump after SB in Fig. 2.15 like **if**(r1==0 && r2==0) {/**body**/}, the path would need to have access to the weak memory semantics, which is precisely what we try to avoid by using the model of Alglave afterwards. We recall that $E.\mathsf{po}$ stands for the program order defined in the event structure E.

$$\frac{e_1 \xrightarrow{\mathsf{po}} e_2 \in E.\mathsf{po} \qquad E \in \mathrm{S}^?(P)}{\exists t \in \mathrm{CFG}(P), \exists i \neq j, e_1 \in \mathrm{evts}(t[i]) \wedge e_2 \in \mathrm{evts}(t[j]) \wedge t[i] \xrightarrow{\mathrm{CFG}} t[j]} \tag{B.2}$$

The third and last rule ensures that, given an event structure, if there is a (valid) execution with a communication between two events of two different threads, then there exists a path in the CFG which goes through a thread call instruction and one of the events, and another path in the called thread that reaches the second event. We recall that $X.\mathsf{com}$ is the communication relation of the execution X valid under architecture A for the event structure E—i.e., $A.\mathrm{valid}(E, X)$.

$$\frac{e_1 \xrightarrow{\mathsf{com}} e_2 \in X.\mathsf{com} \qquad E \in \mathrm{S}^?(P) \qquad A.\mathrm{valid}(E, X)}{\begin{array}{c}\exists t,\, t' \in \mathrm{CFG}(P), \exists j, k, l, m\ t[j] \text{ is } \mathsf{START_THREAD}\ \mathrm{T},\ t'[m] \text{ is } \mathsf{START_FUNCTION}\ \mathrm{T}, \\ e_1 \in \mathrm{evts}(t[k]),\ e_2 \in \mathrm{evts}(t'[l]),\ t[j] \xrightarrow{\mathrm{CFG}} t[k] \wedge t'[m] \xrightarrow{\mathrm{CFG}} t'[l]\end{array}} \tag{B.3}$$

B.2 Soundness ideas for AEG construction

To prove the soundness of the AEG construction, we first introduce three lemmas. The lemmas 3 and 4, and the rules B.1 to B.3 rely on traces in the CFG. Yet, we do not analyse the CFG trace by trace. For instance, when we encounter the diamond **if**(c) a; **else** b; d;, we do not compute τ[**assume**(c); a; d;]$(A) \cup \tau$[**assume**(!c); b; d;](A) but τ[**if**(c) a; **else** b; d;](A). One needs to ensure that

$$\forall A,\ \tau[\textbf{if}(\mathrm{c})\ \mathrm{a};\ \textbf{else}\ \mathrm{b};\ \mathrm{d};](A) \mathrel{\dot{\supseteq}} \tau[\textbf{assume}(\mathrm{c});\ \mathrm{a};\ \mathrm{d};](A) \cup \tau[\textbf{assume}(!\mathrm{c});\ \mathrm{b};\ \mathrm{d};](A)$$

where $\dot{\subseteq}$ is the component-wise inclusion relation[1]. That is, that the AEG we compute would capture the same executions as those we would capture by computing it with single traces in the CFG. This is the purpose of the trace inclusion lemma.

Lemma 2 (Trace inclusion lemma). *For any path t in a* CFG *C, $\tau[t](\bar{\emptyset})\dot{\subseteq}\tau[C](\bar{\emptyset})$*

Proof. We need to show for each instruction i (potentially guarded) of the list in Fig. 3.1 that, if there is a guard, the AEG *resulting of each of the branch would be captured by our* AEG*. In other words, we show that $\tau[skip](A) \cup \tau[i](A)\dot{\subseteq}\tau[\,[c]\,i](A)$, and perform a structural induction over the* CFG*. Details can be found in App. B.* □

The connection lemma guarantees that two events will be connected by po_s^+ in the AEG if a restricted set of instructions is encountered in the CFG when producing the AEG.

Lemma 3 (Connection lemma). *If a trace t in the* CFG *of a program P encounters only instructions different from START_THREAD, then $\forall i < j$, $\forall e, e'$ such that $e \in evts(t[i])$ and $e' \in evts(t[j])$, then $e \xrightarrow{po_s^+} e'$ in the* AEG *of P.*

Proof. We want to show that if t is a sequence of instructions restricted to those above, then for any two events e and e' yielded respectively by instructions $t[i]$ and $t[j]$ such that $i < j$, then $e \xrightarrow{po_s^+} e'$. In other words, we want to show that $(e, e') \in \left(\tau[t_{i,j}](\bar{\emptyset}).po_s\right)^+$. By induction over the trace, we show that $\forall k \leq j$, $\exists(e, x)$ such that $(e, x) \in \left(\tau[t_{i,k}](\bar{\emptyset}).po_s\right)^+$, using the fact that $\left(\tau[t_{i,k}](\bar{\emptyset}).po_s\right)^+ = \left(\left(\tau[t_{i,k-1}](\bar{\emptyset}).po_s\right)^+ \cup \tau[t_{k-1,k}]\left(\tau[t_{i,k-1}](\bar{\emptyset})\right).po_s\right)^+$. Details of the intermediate step for each of the instructions can be found in App. B. We then use lemma 2 to show that it also belongs to the transitive closure of the AEG *we computed, that is, $(e, e') \in \left(\tau[\mathrm{CFG}](\bar{\emptyset}).po_s\right)^+$.* □

The thread spawning lemma ensures that if two events are such that one of them came from a new thread, then these events are in cmp.

Lemma 4 (Thread spawning lemma). *Given two events e and e' are such that, for two traces t and t' in the* CFG *of the program P, and for j, k, l, m, $e \in evts(t[k]) \wedge e' \in evts(t'[l]) \wedge t[j] \xrightarrow{\mathrm{CFG}} t[k] \wedge t'[m] \xrightarrow{\mathrm{CFG}} t'[l]$, $t[j]$ is the start of a new thread T and $t[j]$ is a start_thread instruction calling T, then the abstractions of e and e', namely e_s and e'_s, are in cmp in the* AEG *of P.*

[1] That is, $S_1\dot{\subseteq}S_2 \triangleq S_1.\mathbb{E}_s \subseteq S_2.\mathbb{E}_s \wedge S_1.\mathrm{po}_s \subseteq S_2.\mathrm{po}_s \wedge S_1.\mathrm{cmp} \subseteq S_2.\mathrm{cmp}$.

Proof. *We want to show that* $(e, e') \in \gamma\,(\tau[t_{j,k}](A).cmp)$*, for any* A*.* $t[j]$ *is a thread creation, so we have* $\tau[t_{j,k}](A).cmp \supseteq \tau[t_{j+1,k}](A).\mathbb{E}_s \otimes \tau[t'_{m,l}](\bar{\emptyset}).\mathbb{E}_s$*. By construction of* $\otimes$*, we get the result.* □

We now prove that the AEG of a program P captures statically all the orders involved in the event structures and executions of P. To do so, for any execution and event structure, we first use the rules B.1 to B.3 in order to find a trace in the CFG that would support them. Then, we use the lemmas 3 and 4 to prove that it maps to a path in the AEG. Finally, we show that this path indeed abstracts the orders of the event structure and execution.

Property 13. *Any execution valid on an architecture supported by the framework of [Alg10] is captured by this abstraction.*

Proof. *We need to show that* $\forall (E, X)$ *valid under an architecture* A *for a program* P*, i.e.,* $E \in S^{?}(P)$ *and* $A.valid((E, X)$*, there is a path* t *in the* AEG *of the program* P *such that:*

$$\gamma(t.po_s^{+}) \supseteq E.\mathit{po} \cup X.\mathit{wsi} \cup X.\mathit{rfi} \cup \mathit{fri}(X.\mathit{ws}, X.\mathit{rf}), \quad \text{(B.4)}$$

$$\gamma(t.cmp) \supseteq X.\mathit{wse} \cup X.\mathit{rfe} \cup \mathit{fre}(X.\mathit{ws}, X.\mathit{rf}), \quad \text{(B.5)}$$

$$\gamma_e(t.\mathbb{E}_s) \supseteq E.\mathbb{E}, \quad \text{(B.6)}$$

where γ *is the concretisation function mapping from* $\wp(\mathbb{E}_s \times \mathbb{E}_s)$ *to* $\wp(\mathbb{E} \times \mathbb{E})$ *and* γ_e *from* $\mathbb{E}_s$ *to* $\mathbb{E}$*. To show this, we use the rules B.1 to B.3 in order to find a trace* t' *in the* CFG *of* P *that is responsible for these* E *and* X*, then use the connection lemma and thread spawning lemma (lemmas 3 and 4) in order to show that this trace* t' *corresponds to a path* t *in the* AEG*, and this* t *captures statically all the orders of* E *and* X*. The details of the proof are in App. B.* □

Note that Prop. 13 just states that any valid execution which can be constructed out of the program is existing in the graph. In other words, all the possible partial orders involved in the validity of an execution are contained statically in the graph.

Property 14. *Any critical cycle forbidding an execution on an architecture exists in the graph as a cycle.*

Proof. *The orders involved in the critical cycles all appear in the orders lying in the executions (and the event structures corresponding the program). By Prop. 13, all these orders are abstracted by the static orders* po_s *and* cmp *in the* AEG*. Hence any cycle in the dynamic orders will also appear as a cycle in* $cmp \cup po_s$*.* □

This is a direct consequence of Prop. 13. In particular, we are interested in cycles

that contain delays, i.e., cycles that do not prevent an execution from happening on a weak memory consistent architecture due to some reorderings.

B.3 Assumption for loop-yielded memory events

The soundness proof sketch we suggested in Sec. B.2 did not have to cover the cases where two events are yielded by the same static assignment. This is due to the rule B.2, which assumes that i and j are different. We can relax this constraint and replace the rule B.2 by the rule below.

$$\frac{e_1 \xrightarrow{\mathsf{po}} e_2 \in E.\mathsf{po} \qquad E \in \mathrm{S}^?(P)}{\exists t \in \mathrm{CFG}(P), \exists i,\, j, e_1 \in \mathrm{evts}(t[i]) \wedge e_2 \in \mathrm{evts}(t[j]) \wedge t[i] \xrightarrow{\mathrm{CFG}} t[j]} \tag{B.7}$$

To complete the proof of Sec. B.2 with rule B.7, we insert the duplications of the events of loops described above in the connection lemma. The rest follows.

B.4 Soundness ideas for static instrumentation

We want to prove that the instrumentation produced with the AEG is sound, that is, that the instrumented program simulates any execution of the original program run under a given architecture A. In order to do so, one needs to show that any path of labels accepted by the abstract machine under A is covered by an instrumentation of the program decided by our AEG-based approach.

We recalled in Sec. 4.1.1 the equivalence between the abstract machine accepting label sequences and the event structures that decide the validity of a given execution. In Chap. 3, we also established that the AEG was a sound abstraction of the executions as defined under Alglave's framework. We can therefore base our soundness proof on the AEG—as depicted in Fig. B.1.

Theorem 3 (Soundness of the static instrumentation)**.** *Given the* AEG *of a (goto-)program P, the instrumented (goto-)program P' based on the* AEG *is capturing any path of labels accepted by the abstract machine.*

Proof. Given a (finite) AEG *G, any simple path p in the* AEG *is a sequence of accesses to shared memory. Following γ_{exec} then Xtop, it maps to a set of sequences of delay labels that are valid for an abstract machine (or equivalently event structure) captured by the* AEG*. For each of these label sequences, we show that it is captured by the instrumented goto-program. We establish this with a case analysis, as described in the table below.*

If A and B are on the same thread:

label sequence \ abstract events pair (A, B) *in*	$W \times R$	$W \times W$	$R \times W$	$R \times R$
mem(A); mem(B)	*(1)*	*(1)*	*(1)*	*(1)*
delay(A); flush(A); mem(B)	*(2)*	*(2)*	*(3)*	*(3)*
mem(A); delay(B); flush(B)	*(2)*	*(2)*	*(3)*	*(3)*
delay(A); mem(B); flush(A)	*(4)*	*(4)*	*(5)*	*(5)*
delay(A); flush(A); delay(B); flush(B)	*(2)*	*(2)*	*(3)*	*(3)*
delay(A); delay(B); flush(A); flush(B)	*(4)*	*(4)*	*(5)*	*(5)*
delay(A); delay(B); flush(B); flush(A)	*(4)*	*(4)*	*(5)*	*(5)*

If A and B are on different threads and target the same memory address:

label sequence \ abstract events pair (A, B) *in*	$W \times R$	$W \times W$	$R \times W$	$R \times R$
mem(A); mem(B)	*(1)*	*(1)*	*(1)*	*(1)*
delay(A); flush(A); mem(B)	*(1)*	*(1)*	*(1)*	*(1)*
mem(A); delay(B); flush(B)	*(1)*	*(1)*	*(1)*	*(1)*
delay(A); mem(B); flush(A)	*(6)*	*(1)*	*(1)*	*(1)*
delay(A); flush(A); delay(B); flush(B)	*(1)*	*(1)*	*(1)*	*(1)*
delay(A); delay(B); flush(A); flush(B)	*(6)*	*(1)*	*(1)*	*(1)*
delay(A); delay(B); flush(B); flush(A)	*(1)*	*(1)*	*(1)*	*(1)*

(1) *no instrumentation needed, detected at the cycle detection stage—same behaviour.*

(2) *delays the write, then flushes it immediately:* **if**(∗) push(&buff[&x], v, id); **else** x=v;. *Reachable by taking the second branch of the non-deterministic if.*

(3) *delays the read, then flushes it immediately:* **if**(∗) rdelay[&r1]=&x; **else** r1=x;. *Reachable by taking the second branch of the non-deterministic if.*

(4) *delays the write, then flushes it later:* **if**(∗) push(&buff[&x], v, id); **else** x=v; y=w; *and, for any read of x in the critical cycle,* **if**(∗) x=take(&buff[&x], id); expr=x;. *Reachable by taking the first branch on the non-deterministic addition to buffer, then flushing the buffer afterwards using take when encountering a read of x.*

(5) *delays the reads, then flushes it later:* **if**(∗) rdelay[&r1]=&x; **else** r1=x; *and, for any operation reading from the local variables r1 in or after the critical cycles,* **if**(rdelay[&r1]!=NULL&&∗) { r1=∗rdelay[&r1]; rdelay[&r1]=NULL; } r2=r1;. *Reachable by taking the first branch of the postponement of the read, then flushed afterwards by assigning to the read local variable the value of x in the current memory environment.*

(6) *delays the write, then the read reads from the write buffer before it is flushed. The write is first placed in the buffer by following the first branch of* **if**(∗) push(&buff[&x], v, id); **else** x=v;. *Then, on the thread of the read, we follow the first branch of* **if**(length(&buff[&x])!=0 && ∗) { delay[&x]=TRUE; delay_tmp[&x]=x; x=last(&buff[&x], thread_ID); } *which stores the current value of* x *and assign to it the value currently in the buffer. The read is performed with* r1 = x; *and the previous value of* x *is restored by* **if**(delay[&x]&&∗) { x=delay_tmp[&x]; delay[&x]=FALSE; }. *Since we executed these*

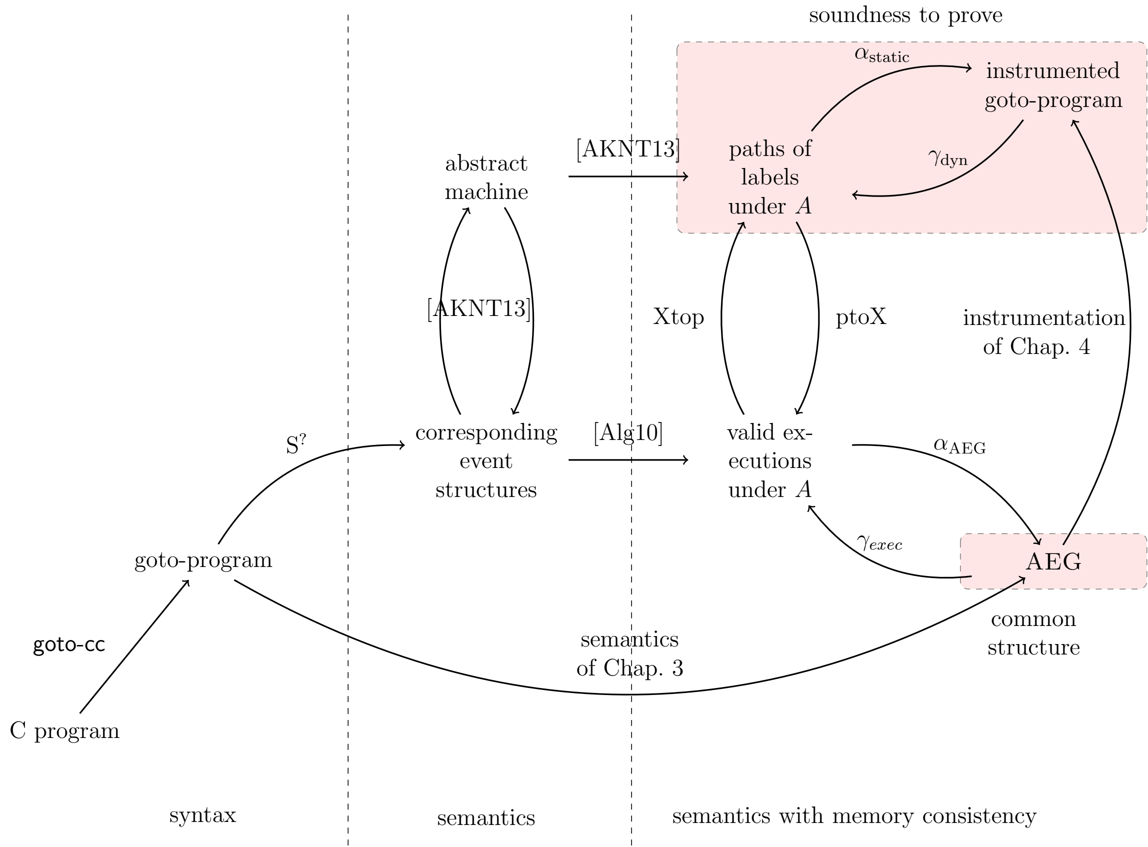


Figure B.1: Soundness of the instrumentation, using the soundness depicted in Fig. 3.4.

steps atomically, the other threads did not see the temporary change of value. The write buffer can then be flushed later.

□

Appendix C

Transformations of a few classic programs

C.1 Store buffering (sb)

C.2 Load delaying (lb)

C.3 Message passing (mp)

C.4 Independent reads independent writes (iriw+dps)

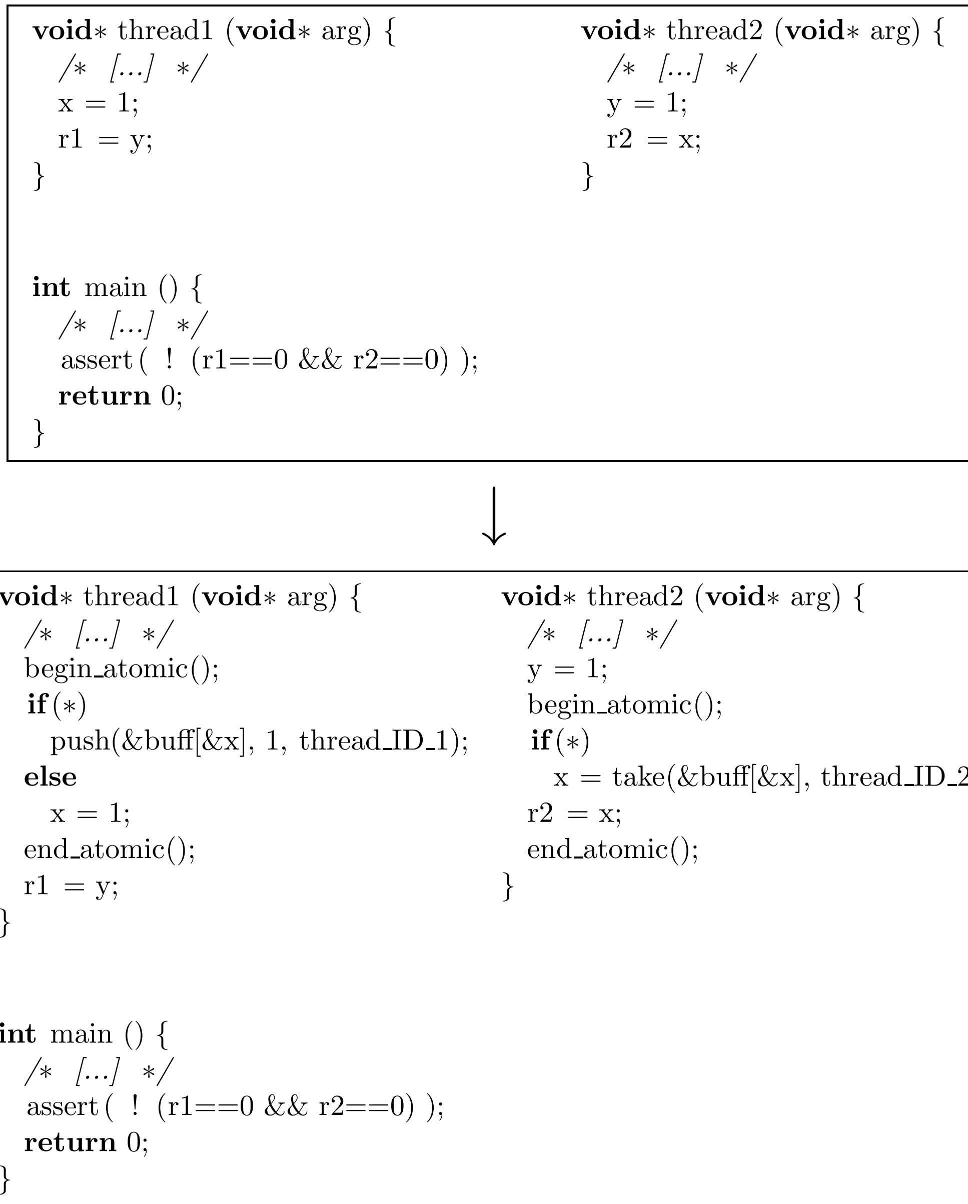


Figure C.1: The implementation of **(sb)** on top and the instrumented program below.

```
void* thread1 (void* arg) {
  /* [...] */
  r1 = y;
  x = 1;
}

void* thread2 (void* arg) {
  /* [...] */
  r2 = x;
  y = 1;
}


int main () {
  /* [...] */
  assert( ! (r1==1 && r2==1) );
  return 0;
}
```

```
void* thread1 (void* arg) {
  /* [...] */
  begin_atomic();
  if(*)
    rdelay[&r1] = &y;
  else
    r1 = y;
  end_atomic();
  x = 1;
}

void* thread2 (void* arg) {
  /* [...] */
  r2 = x;
  y = 1;
}


int main () {
  /* [...] */
  begin_atomic();
  if(rdelay[&r1]!=NULL && *) {
    r1 = *rdelay[&r1];
    rdelay[&r1] = NULL;
  }
  end_atomic();
  assert( ! (r1==0 && r2==0) );
  return 0;
}
```

Figure C.2: The implementation of **(lb)** on top and the instrumented program below.

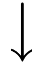
```
void* thread1 (void* arg) {
  /* [...] */
  x = 1;
  y = 1;
}

void* thread2 (void* arg) {
  /* [...] */
  r1 = y;
  r2 = x;
}

int main () {
  /* [...] */
  assert ( ! (r1==1 && r2==0) );
  return 0;
}
```

↓

```
void* thread1 (void* arg) {
  /* [...] */
  begin_atomic();
  if(*)
    push(&buff[&x], 1, thread_ID_1);
  else
    x = 1;
  end_atomic();
  y = 1;
}

void* thread2 (void* arg) {
  /* [...] */
  r1 = y;
  begin_atomic();
  if(*)
    x = take(&buff[&x], thread_ID_2);
  r2 = x;
  end_atomic();
}

int main () {
  /* [...] */
  assert ( ! (r1==1 && r2==0) );
  return 0;
}
```

Figure C.3: The implementation of **(mp)** on top and the instrumented program below.

```
void* thread1 (void* arg) {
  /* [...] */
  r1 = x;
  unsigned tmp = r1 ^ r1;
  r2 = *(&y+tmp);
}

void* thread2 (void* arg) {
  /* [...] */
  r3 = y;
  unsigned tmp = r3 ^ r3;
  r4 = *(&x+tmp);
}

void* thread4 (void* arg) {
  y = 1;
}

void* thread3 (void* arg) {
  x = 1;
}

int main () {
  /* [...] */
  assert( ! (r1==1 && r2==0 && r3==1 && r4==0) );
  return 0;
}
```



```
void* thread1 (void* arg) {
  /* [...] */
  begin_atomic();
  if(length(&buff[&x]) != 0 && *) {
    delay[&x] = TRUE;
    delay_tmp[&x] = x;
    x = last(&buff[&x], thread_ID_1);
  }
  r1 = x;
  if(delay[&x] && *) {
    x = delay_tmp[&x];
    delay[&x] = FALSE;
  }
  end_atomic();
  unsigned tmp = r1 ^ r1;
  r2 = *(&y+tmp);
}

void* thread2 (void* arg) {
  /* [...] */
  r3 = y;
  unsigned tmp = r3 ^ r3;
  r4 = *(&x+tmp);
}

void* thread3 (void* arg) {
  begin_atomic();
  if(*)
    push(&buff[&x], 1, thread_ID_3);
  else
    x = 1;
  end_atomic();
}

void* thread4 (void* arg) {
  y = 1;
}

int main () {
  /* [...] */
  assert( ! (r1==1 && r2==0 && r3==1 && r4==0) );
  return 0;
}
```

Figure C.4: The implementation of **(iriw+dps)** from Fig. 4.2 on top and the instrumented program below.

Appendix D

Basic combinatoric details

D.1 Duplication in presence of nested loops

We explained in Subsec. 3.1.3 that, in the case of nested loops, we duplicate each of the sub-bodies only once in order to avoid an exponential explosion.

Let us suppose that we have a loop l_1 of $\#l_1$ events, that contains a loop l_2 of $\#l_2$ events, containing itself a loop l_3 and so on until a n$^{\text{th}}$ loop l_n. If we duplicate all the loops, for each loop l_i with $i < n$, we copy twice the events in l_i that are not in l_{i+1}, i.e., $2(\#l_i - \#l_{i+1})$, and the events in l_{i+1} will be copied twice because they are also in l_i, and twice more because l_{i+1} needs to be duplicated itself. We actually have $2(\#l_1 - \#l_2 + 2(\#l_2 - \#l_3 + 2(... + 2l_n)...)$ events after duplication. If we write $\max_{\Delta\#}\{l_1, ..., l_n\}$ (respectively $\min_{\Delta\#}\{l_1, ..., l_n\}$) for the maximum (respectively minimum) difference of successive cardinalities, that is,

$$\max_{\Delta\#}\{l_1, ..., l_n\} = \max(\{\#l_i - \#l_{i+1} \mid 1 \leq i < n\} \cup \{\#l_n\}),$$

we can bound this sum with $\min_{\Delta\#}\{l_1, ..., l_n\} \sum_{i=1}^{n} 2^i$ and $\max_{\Delta\#}\{l_1, ..., l_n\} \sum_{i=1}^{n} 2^i$, which are

$$2(2^n - 1)\min_{\Delta\#}\{l_1, ..., l_n\} \text{ and } 2(2^n - 1)\max_{\Delta\#}\{l_1, ..., l_n\}.$$

The number is exponential in the number of nested loops.

If we duplicate only once for each loop, we have twice the number of events inside l_1 but outside l_2, then we have three times the number of events in l_2 that are not in l_3 (twice because the events of l_2 are also in l_1, plus one duplication), and so on. We actually have $2(\#l_1 - \#l_2) + 3(\#l_2 - \#l_3) + ... + (n + 1)\#l_n$. Using the same maximum and minimum as in the previous case, we have $\min_{\Delta\#}\{l_1, ..., l_n\} \sum_{i=2}^{n+1} i$ and $\max_{\Delta\#}\{l_1, ..., l_n\} \sum_{i=2}^{n+1} i$, which are

$$\frac{n(n+3)}{2}\min_{\Delta\#}\{l_1, ..., l_n\} \text{ and } \frac{n(n+3)}{2}\max_{\Delta\#}\{l_1, ..., l_n\}.$$

The number of events is quadratic in the number of nested loops.

D.2 Static analysis of pointers to function

The default strategy initially available in the CProver codebase consists of replacing the call to function pointers by a non-deterministic choice between all the functions whose addresses are used at some point in the code and whose prototypes are compatible with the prototype of the pointer.

In the context of our analysis, this over-approximation can strongly impact the size of the graph, particularly the number of cmp. Indeed, pthread_create welcomes a function of the generic type **void**$*$ $(*)$ (**void**$*$). With the default strategy, any function whose address is taken (and thus any function called for a new thread) can be non-deterministically reached from this call. Let us consider SB^n in Fig. 3.19(a). If we inline the functions precisely, we have exactly n cmp connections in the AEG. If we apply the default strategy: we count the cmp for every pair of threads ($\binom{n}{2}$ possibilities). Each thread has a non-deterministic choice between n functions, and each of these n functions can have 3 cmp. Indeed, suppose that we consider, in the first thread, the function with Wx1 Rx2. Wx1 can be connected to the Wx1 of the same function on the other thread; or it can be connected to the Rx1 of the previous function on the other thread; or Rx2 can be connected to the Wx2 of the next function on the other thread. There are thus $3n\binom{n}{2}$, that is,

$$\frac{3n^2(n-1)}{2} \text{ cmp.}$$

The same reasoning can be applied for W_m^n in Fig. 3.19(b). We have the number of pairs of threads multiplied by the number of functions that are potentially hosted by the thread multiplied by the number of cmp that a function in the first thread can make with the other functions of the other thread:

$$\binom{n}{2} \times \left(\text{number of functions} \times \begin{array}{c}\text{number of cmp between}\\ \text{one function and the}\\ \text{other thread's functions}\end{array} \right).$$

If we apply the default strategy, given a function on a thread, there are $m^2 \times n$ possible cmp. We have a total of

$$\frac{m^2 n^3(n-1)}{2} \text{ cmp.}$$

If we apply a precise inlining of the functions, we only have 1 function per thread and the possible cmp per function w.r.t. the other (unique) function in the other thread

is m^2. Thus there are

$$\frac{m^2 n(n-1)}{2} \text{ cmp.}$$

Appendix E

Data, tools and experiments reproducibility

In this appendix, we explain how to reproduce the experiments we ran with our tools

- goto-instrument –mm, which instruments C programs so that SC analyser can prove programs or detect bugs involving non-SC behaviours (Chap. 4);
- musketeer, which inserts memory fences in C programs to restore SC (Chap. 5).

. We first explain how to install our tools in a Linux environment. They should in principle also work for Windows/Mac/Solaris, as cbmc does, but this was untested. We then provide a short tutorial on how to use each of these applications. We finally explain how to reproduce our experimental results.

Note that goto-instrument –mm has also been tested with some analysers that might not be publicly available. We invite the readers wishing to reproduce the experiments to contact the authors of these tools.

We ran some comparative experiments for musketeer, using tools implemented by other research teams. We will explain how we ran them. Similarly to the previous case, we invite the readers to contact the authors to obtain the software if it is not publicly available.

We provide in the last section of this appendix a table listing the tools we used and their developers or contacts.

Disclaimer: As we mentioned in Chap. 4, the results reported in the dissertation were those obtained with the tools listed in Sec. E.3, in early 2013 for goto-instrument –mm (published in [AKNT13]) and in 2014 for musketeer (published in [AKNP14]). goto-instrument –mm and musketeer were based on CProver 4.3, released in February

2013. Results are also available in detail on the webpage `http://www.cprover.org/wmm/esop13` and `http://www.cprover.org/wmm/musketeer/`.

At the time of the dissertation writing, we worked at porting goto-instrument –mm and musketeer to the trunk version of CProver, posterior to the release 4.9, in order to get it integrated to CProver for the release 5.0. This work is finished (both tools are already accessible from CProver's SVN[1]) and we are aiming to reproduce all the results we reported in the dissertation. Most of the results can be obtained again; some of the experiments still need some adjustments to reproduce some results at the time of writing.

The installation process of the new tools is simpler, and allows to make direct use of the new format of goto-binaries used by goto-cc/cbmc.

E.1 goto-instrument –mm

E.1.1 Setting the experimental environment

The list of packages required to compile goto-instrument and the whole CProver framework on a Debian can be found at `http://www.cprover.org/svn/cbmc/trunk/COMPILING`.

The original source and binaries used in [AKNT13] can be downloaded from `http://www.cprover.org/wmm/esop13/manual.shtml`, along with the version released with cbmc 4.5. To compile them, please follow the instructions listed in `http://www.cprover.org/svn/cbmc/trunk/COMPILING`.

The model-checkers used in combination with the instrumenter in our experiments are listed in Sec. E.3 with their respective websites. Please follow the installation instructions provided on these websites.

Please note that the minimisation strategy developed in Sec. 4 also requires glpk.

The goto-instruments provided in the releases 4.3, 4.4 and 4.5 of cbmc also have the weak memory instrumentation functionality. The source can be retrieved from the SVN at `http://www.cprover.org/svn/cbmc/releases/`. From 4.6, however, we observed some significant regressions. We would not recommend using the versions 4.6, 4.7, 4.8 and 4.9 for instrumentation.

[1] http://www.cprover.org/svn/cbmc/

E.1.2 A short tutorial

A manual for goto-instrument –mm is available online at `http://www.cprover.org/wmm/esop13/manual.shtml`. It also provides scripts written by Michael Tautschnig to connect some model-checkers that do not directly analyse C programs.

We reproduce below a short example of how to use goto-instrument –mm. Let us suppose that we use SatAbs as model-checker (with Boom as model-checker for Boolean programs).

Checking a program for a given architecture

We want to check that a program (for example, `safe006.c`, available at `http://www.cprover.org/wmm/esop13/x86_litmus/safe006.c`) is safe for PSO (and then safe for TSO), but not for RMO (and for Power, by inclusion). We first convert the program into a goto-program:

```
goto-cc -o safe006.goto safe006.c
```

Then we instrument it for PSO:

```
goto-instrument --mm pso safe006.goto safe006_pso.goto
```

We check it with SatAbs:

```
satabs --concurrency --full-inlining --max-threads 3 --model-checker \
boom safe006_pso.goto
```

And Satabs returns that the program is validated after 1.64 sec:

```
Statistics of modelchecker:
Broadcast assignment operations executed: 0
Non-broadcast assignment operations executed: 220
Time spent in broadcast assignment operations: 0
Time spent in non-broadcast assignment operations: 0.002453

VERIFICATION SUCCESSFUL
```

We have two RW pairs which cannot be reordered on PSO. We now investigate for RMO. We instrument with this weaker architecture, and check again with SatAbs.

```
goto-instrument --mm rmo safe006.goto safe006_rmo.goto
satabs --concurrency --full-inlining --max-threads 3 --model-checker \
boom safe006_rmo.goto
```

The model-checker returns after 7.14 sec this counter-example and a failure message below.

```
Violated property:
file safe006.c line 27 function main
assertion
$tmp_guard

VERIFICATION FAILED
```

The two RW pairs can indeed be reordered under RMO, and the assertion can be violated.

E.1.3 How to reproduce the experimental data

All the benchmarks can be downloaded from the experimental results webpage `http://www.cprover.org/wmm/esop13/experiments.shtml`.

E.1.4 How to install the new version of the tool

The source of the new version of the tool can be retrieved on the SVN from the trunk or from release 5.0. Binaries can be directly used from the cbmc tarball[2] available on the CProver website or from the Debian package of cbmc. The installation process is the same as for the older version.

E.1.5 A short note regarding the license

The license is a 4-clause BSD and is the same as the general CProver code base license. It can be found in the main folder (if compiled from source) or in the package.

[2] www.cprover.org/cbmc

E.2 musketeer

E.2.1 Setting the experimental environment

The list of packages required to compile goto-instrument and the whole CProver framework on a Debian can be found at `http://www.cprover.org/svn/cbmc/trunk/COMPILING`.

The original source and binaries used in [AKNP14] can be downloaded from `http://www.cprover.org/wmm/musketeer`.

E.2.2 A short tutorial

Let us suppose that we want to insert some fences in a program `my_program.c` for an architecture A.

We first generate a goto-program:

```
goto-cc -o my_program.gb my_program.c
```

We then analyse the goto-program for MM in tso, pso, rmo, power, arm:

```
musketeer --mm MM my_program.gb
```

We finally apply the fence insertion script for A in x86, ARM and F in fence, dp (inserts fences only or fences and dependencies):

```
fence-inserter.py A F results.txt
```

Fences have been inserted into the source to prevent weak memory reorderings.

E.2.3 How to reproduce the experimental data

The classic examples extracted from the literature and the parametric ones can be found at this address: `http://www.cprover.org/wmm/musketeer/benchmarks.tar.gz`.

The Debian benchmarks, prepared by Michael Tautschnig, can be retrieved from `http://dkr-debian.cs.ox.ac.uk:8080/`. Note that since the server recompiles regularly these programs, the list of available programs may change.

E.2.4 How to install the new version of the tool

The source of the new version of the tool can be retrieved on the SVN from the trunk or from release 5.0. The installation process is the same as for the older version.

E.2.5 A note regarding the license

The license is a 4-clause BSD and is the same as the general CProver code base license. It can be found in the main folder (if compiled from source) or in the package.

E.3 Third-party tools

We list below the third-party tools used in our experiments and their developers or contacts.

tool	version	license	authors/contacts	institution
Blender	unkn.	unkn.	Michael Kuperstein, Martin Vechev, Eran Yahav	ETH Zürich, Technion
CBMC[1]	4.3✗	BSD 4-clause	Daniel Kroening, Michael Tautschnig	University of Oxford, Carnegie Mellon University
CheckFence[2]	unkn.	BSD 3-clause	Sebastian Burckhardt	University of Pennsylvania
Corral[3]	1.0.0.0✗	Apache v2	Akash Lal, Shuvendu Lahiri, Shaz Qadeer	Microsoft Research
DFence[4]	unkn.	NCSA	Martin Vechev, Eran Yahav	ETH Zürich, Technion
ESBMC[5]	1.18✗	BSD 3-clause	Lucas Cordeiro	University of Southampton
MMChecker[6]	1.0✓	unkn.	Abhik Roychoudhury	National University of Singapore
Memorax[7]	0.1.1✓	GPL v3	Carl Leonardsson	Uppsala Universitet
Offence[8]	0.01 (1.00)✓	LGPL v3	Luc Maranget, Jade Alglave	INRIA Rocquencourt
Remmex	unkn.	unkn.	Alex Linden, Pierre Wolper	Université de Liège
SatAbs[9]	3.2✓	BSD 4-clause	Daniel Kroening	University of Oxford
Threader[10]	0.9 ✓	BSD 3-clause	Ashutosh Gupta, Corneliu Popeea, Andrey Rybalchenko	Technische Universität München
Trencher[11]	unkn.	unkn.	Egor Derevenetc, Roland Meyer	Technische Universität Kaiserslautern

✓: up-to-date

×: newer version available

New versions available at the time of writing:

> CBMC 4.9,
> Corral 1.0.0.0 (new version),
> ESBMC 1.24.1.

[1] http://www.cprover.org/cbmc/
[2] http://checkfence.sourceforge.net/
[3] http://corral.codeplex.com/
[4] https://github.com/eth-srl/DFENCE/
[5] http://www.esbmc.org/
[6] http://www.comp.nus.edu.sg/~release/mmchecker/
[7] https://github.com/memorax/
[8] http://offence.inria.fr/
[9] http://www.cprover.org/satabs/
[10] https://www7.in.tum.de/~popeea/research/threader.html
[11] http://concurrency.cs.uni-kl.de/trencher.html