Model checking LLVM IR using LTSmin

Using relaxed memory model semantics

Author: F. I. van der Berg
Date: 20th December 2013

UNIVERSITY OF TWENTE.

UNIVERSITY OF TWENTE.

UNIVERSITY OF TWENTE
Faculty of Electrical Engineering, Mathematics and Computer Science
Formal Methods and Tools

MASTER'S THESIS

Model checking LLVM IR using LTSmin

Using relaxed memory model semantics

Author:
Freark VAN DER BERG

Committee:
Prof. Dr. Jaco VAN DE POL
Dr. Stefan BLOM
Alfons LAARMAN, MSc

20th December 2013

Abstract

Advancements in computer architectures have resulted in an exponential increase in the speed of processors and memory capacity. However, memory latencies have not improved to the same extent, thus access times are increasingly limiting peak performance. To address this, several layers of cache are added to computer architectures to speed up apparent memory access and instruction are reordered to maximize memory throughput.

This worked well for single-processor systems, but because of physical limits, modern computer architectures gain performance by adding more processors instead of increasing the clock speed. In multi-processor systems, the cache and instruction reordering make communication complex, because reads and writes of one processor may be observed in a different orders by different processors. To mitigate this, some computer architectures add complex hardware at the cost of performance, power requirements and die size. Other architectures employ a *relaxed memory model* and add synchronization instructions, *memory barriers*, to the instruction set. This means the software has to deal with the complexity. By placing memory barriers, an ordering on reads and writes can be enforced, causing processors to synchronize.

However, memory barriers are expensive instructions and need only to be placed where absolutely needed if performance is of importance. To this end, we present our tool, LLMC. The target of LLMC is concurrent programs written in LLVM IR, an intermediate representation language with numerous front-ends, e.g. for C, C++, Java, .NET, and Erlang. Using the model checker LTS_{min}, we explore the state space of these programs in search of assertion violations, deadlocks and livelocks. We do this for the memory models TSO, PSO and a limited version of RMO. To the best of our knowledge, this is the first tool that model checks LLVM IR programs running on PSO and a limited version of RMO. We applied LLMC to a well-known concurrent queue, the Michael-Scott queue, and were able to confirm the necessity of the required memory barriers for correctness under RMO.

Preface

The reason I started this project is because of months of struggling to wrap my head around getting the implementation of a concurrent queue correct on an ARMv7 architecture. This was before my thesis. Back then, I did not know all the intricate details of relaxed memory models and the memory model of the ARMv7 instruction set is one of the most relaxed. But after being at it for months, I had learned a great deal about concurrent data structures and implementing them on relaxed memory models. But still, this concurrent queue was a beast.

So, months later, still shaking off some frustration, the thought crept into my mind to develop a tool which could help me do this! A tool that will tell me if my concurrent queue implementation is correct on ARMv7. And thus, a long while later, LLMC was a reality. While it is still a long way from the tool I had envisioned, I think it will still be useful for the next time I need to implement a concurrent queue.

In making LLMC a reality, the members of my committee were essential. I would like to take this opportunity to thank Jaco, Stefan and Alfons, for supporting me in this project. I received useful feedback, applicable suggestions and guidance to reach the end. I would like to thank Jaco for giving me the opportunity and confidence to define my own master's thesis. He helped me define my goals and narrow down the scope of this project. I would like to thank Stefan for the discussions on a wide range of topics, the various technical inspirations and for the added features to LTS$_{min}$. I would like to thank Alfons for the discussions on memory models, for concise, no-nonsense feedback and for helping me structure this thesis.

Looking back at this project, it has been quite the ride: a lot of hours went into supporting as many features as I wanted, to make LLMC more useful and in writing all this down. I learned a lot from this. In particular, I learned that one should not try to solve everything at once: science is an iterative process, gathering knowledge one step at a time. I should not forget this.

Contents

1 Introduction 1

1.1 Bugs . . . 1

1.2 Hardware . . . 1

1.3 Program Verification . . . 2

1.3.1 LTSmin . . . 2

1.4 The LLVM Project . . . 2

1.5 Problem Statement . . . 2

1.5.1 Research Questions . . . 2

1.6 Contribution . . . 3

1.7 Organization . . . 3

2 Preliminaries 5

2.1 Computer Architectures . . . 5

2.1.1 Memory instruction reordering . . . 6

2.2 The LLVM Project . . . 9

2.2.1 Intermediate Representation . . . 9

2.2.2 Memory Model . . . 12

2.2.3 Motivation for The LLVM Project . . . 13

2.3 LTSmin . . . 15

2.3.1 The PINS Interface . . . 15

2.3.2 Motivation for LTSmin . . . 17

2.4 Related Work . . . 18

2.4.1 Related Approaches . . . 18

2.4.2 Related tools . . . 18

2.4.3 Comparison . . . 20

3 LLMC Design **21**

3.1 Design choices . . . 21

3.2 The Execution Model . . . 22

3.2.1 Preliminaries . . . 22

3.2.2 The Program . . . 22

3.2.3 The Execution of a Program . . . 23

3.2.4 Differences . . . 24

3.2.5 Example . . . 25

3.3 Mapping LLVM IR and LTSmin . . . 27

3.3.1 Mapping the state . . . 27

3.3.2 Initial state . . . 28

3.3.3 Next-state . . . 28

3.3.4 Thread Management . . . 31

3.3.5 Dependency Matrix . . . 31

3.4 Exploration strategy . . . 33

3.4.1 Soundness and completeness . . . 33

3.4.2 Deadlock and livelock detection . . . 34

4 LLMC Implementation **35**

4.1 Implementational Details . . . 35

4.1.1 Pointers . . . 35

4.1.2 Bounded buffer . . . 35

4.1.3 Exploration . . . 35

4.1.4 Features . . . 35

5 Results **39**

5.1 Validation . . . 39

5.2 Experiments . . . 41

5.2.1 Concurrent counting . . . 41

5.2.2 Michael-Scott queue . . . 42

5.3 Benchmarks . . . 47

5.3.1 Performance . . . 47

5.3.2 Implementation bottlenecks . . . 47

6 Conclusions **49**

6.1 Summary . . . 49
6.2 Evaluation . . . 50
6.2.1 Considerations . . . 50
6.2.2 So where does that leave LLMC? . . . 50
6.3 Future Work . . . 51
6.3.1 Future Features . . . 51
6.3.2 Future Research . . . 51
6.3.3 Future test cases . . . 52

A Glossary **59**
Glossary . . . 59

B Litmus Tests **61**
B.1 Store Buffer Litmus Test (SB) . . . 61
B.1.1 Summary of inserted barriers . . . 61
B.1.2 C and LLVM IR implementations . . . 61
B.1.3 Traces to error . . . 65
B.2 Load Buffer Litmus Test (LB) . . . 66
B.2.1 Summary of inserted barriers . . . 66
B.2.2 C and LLVM IR implementations . . . 66
B.2.3 Traces to error . . . 68
B.3 Dependent Load Litmus Test (DL) . . . 69
B.3.1 C and LLVM IR implementations . . . 69
B.4 Store propagation litmus test (IRIW) . . . 70
B.4.1 Summary of inserted barriers . . . 70
B.4.2 C and LLVM IR implementations . . . 70
B.4.3 Traces to error . . . 72
B.5 Store prop.+dep. litmus test (IRIW+addr) . . . 73
B.5.1 Summary of inserted barriers . . . 73
B.5.2 C and LLVM IR implementations . . . 73
B.6 Message Passing Litmus Test (MP) . . . 75
B.6.1 Summary of inserted barriers . . . 75
B.6.2 C and LLVM IR implementations . . . 75
B.6.3 Traces to error . . . 77
B.7 Message Passing Litmus Test with dep. (MP-dep) . . . 78
B.7.1 Summary of inserted barriers . . . 78

B.7.2 C and LLVM IR implementations 78

C Implementations of Experiments **81**

C.1 Concurrent counting 81

C.2 Michael-Scott queue 83

C.3 Recursive Fibonacci algorithm 86

1 Introduction

Behind many great projects lies a large collection of software components. Not only scientific endeavours such as the Large Hadron Collider [HKK+13], the Mars Rover [WC05] and nuclear power plants [VPP], but also projects people use every day, such as planes, trains and automobiles, contain large code bases. The software in these projects have to function according to their specification. If they do not, they contain Bugs. The effects from a bug may differ from project to project. A bug in your favourite messenger will not cause the loss of billions of dollars, but a bug in the Mars Climate Orbiter will [BV05]. A nuclear power plant requires software reacting to the environment in real-time. It would be quite unhealthy for the surroundings if there would be a catastrophic bug in the program operating the control rods of the reactor. When not controlled correctly, some medical equipment can have a devastating effect on the lives of people. The Therac-25 machine used in radiation therapy was one of those, causing at least five deaths. [Trc] Trains, transporting numerous of persons, rely on the correctness of the software as well. Many lives could be lost if the control software were to direct two trains on a collision course.

It is vital for the success of these projects that the software functions adequately. Not only could it cause the loss of billions of dollars, but also human lives are at stake. This is why finding bugs before they happen is critical.

1.1 Bugs

Bugs are caused by that the logic of the program does not reflect the intended behaviour. Either the implementation is not correct according to the algorithm it tries to implement, or the algorithm is not correct itself. An example of this kind of bug could be that some implementation of a protocol does not handle a certain message correctly. Another example is that the protocol itself allows for unwanted behaviour such as deadlocks. A second kind of bug is where the program relies on certain facts about its environment. These bugs can prove highly elusive, as it could involve multiple aspects of various programs. An example is a program relying on a specific version of a library being available, assuming a certain contract. The environment of a program is not limited to software: the hardware the program is running can influence the correctness of a program as well.

1.2 Hardware

In modern hardware, the executed instructions of a program only vaguely resembles the original code of that program: a lot of optimizations are done on-the-fly to make the code more performant. This includes removing, replacing and reordering instructions. This is not an issue for single-threaded programs, but can cause problems for multi-threaded programs.

In multi-processor hardware using shared memory, these optimizations pose a problem for communication between processors. While the optimized reordering of memory operations does not alter the local behaviour of a process, another process could observe an unintended state of that process. What kind of memory instruction reorderings are allowed is governed by the *memory*

model of the hardware. Hardware that allows memory instructions to be reordered are said to have a *relaxed memory model*.

Programmers tend to think in a sequential consistent way of their code, but this does not hold for hardware with a relaxed memory model. This makes writing concurrent software that is both correct and performant a daunting task.

1.3 Program Verification

The programmer is faced with the question: *is my multi-threaded code correct?* To be able to answer this, the programmer needs to go through all the possible scenarios where it might go wrong, possibly due to memory instruction reordering. The number of these scenarios is exponential in the number of threads; too high for a mere human to reason about.

To this end, we can call on the help of *formal verification*: proving the correctness or incorrectness of the code using formal methods. Various verification techniques have been researched; one of which is *model checking*. Model checking means to systematically perform an exhaustive exploration to find all the states the program can be in, using all possible interleavings of threads and memory instruction reorderings, thus finding the possible scenarios. This set of states the program can be in is called the *state space*. The combinatorial blow-up of the number of states is known as the *state space explosion*: all the possible interleavings of multiple threads cause an exponential growth of the state space.

The idea is to find if the state space contains states that have a certain property. For example, we could define erroneous states by states that have an outcome we do not desire, like in Figure 2.2.

1.3.1 LTSmin

LTS$_{min}$ is a toolset for model checking and manipulating labelled transition systems. It uses a partitioned next-state interface (PINS) to separate language modules from exploration tools. This modular approach yields a high reusability of modules: a new language module can automatically benefit from all the algorithms and tools implementing PINS. New back-end tools provide enhancements for all the language modules, though sometimes the language modules need to be slightly updated. We will discuss LTS$_{min}$ in more detail in Section 2.3.

1.4 The LLVM Project

The LLVM Project [LLV] contains modular and reusable compiler and toolchain technologies. It uses language-independent instruction set and type system. Instructions are in static single assignment form, allowing simple variable dependency analysis. This instruction set is named LLVM Intermediate Representation (LLVM IR).

There exist multiple front-ends that combined compile many languages to LLVM IR, for example C, C++, Java, Ruby, and Rust. Having such a wide range of input languages makes The LLVM Project interesting: if our program code is in generic LLVM IR, we automatically support all the languages that have a compiler to LLVM IR. We will discuss The LLVM Project in more detail in Section 2.2.

1.5 Problem Statement

This research aims to marry the projects LLVM and LTS$_{min}$ to produce a model checker that can model check LLVM IR. We want to verify LLVM IR programs using various memory models and provide guarantees per memory model. This way, we can determine on what hardware the LLVM IR will behave correctly and on what hardware the LLVM IR may exhibit undesired behaviour. Our primary target is to verify the correctness of concurrent, lock-free data structures.

1.5.1 Research Questions

Using this problem statement as a basis for our research, we must answer the following questions:

P2 *How can we model the execution of multi-threaded LLVM IR programs on a relaxed memory model?* Exploration of the state space of multi-threaded LLVM IR requires defining an execution model and a threading model. Modeling the relaxed memory model semantics requires taking into account the presence of caches and write buffers.

P1 *How can we construct a next-state function from this model?* By using the model checker LTS_{min}, we need to implement PINS and thus define a next-state function. This next state function needs to take into account the registers, stack and global memory of the program. It also needs to consider the memory instruction reordering.

P3 *When is the multi-threaded program deemed correct and when is it deemed incorrect?* The program could be incorrect even in the absence of memory instruction reordering. If we inspect the states of the state spaces, we must decide what constitutes an erroneous state. We must also differentiate the causes of erroneous states: whether or not it is only reachable using a more relaxed memory model or also reachable using a sequential memory model.

P4 *How can we limit memory usage?* Saving entire LLVM process stacks, global memory and heap memory can be a daunting task.

P5 *How can we make our LLVM IR model checker as forward compatible with future LLVM IR versions as possible?* The LLVM Project is ever evolving with new features being added and thus the LLVM IR changes with it. Limiting the efforts to incorporate the new features is beneficial to the maintainability of an LLVM IR model checker.

We aim to make guarantees given a program with a limited number of threads, for example a test program for a concurrent data structure. We do not address the issue of providing guarantees under any number of threads.

1.6 Contribution

We design and implement our approach in LLMC, the low-level model checker. To the best of our knowledge, this is the first model checker that accepts generic LLVM IR and explores its state space assuming a relaxed memory model.

We specified an execution model of an LLVM IR program running on a relaxed memory model and used this model to implement state space exploration. The advantage of targeting LLVM IR is that there are a lot of languages that can be compiled to LLVM IR, including C and C++. Using LLMC, we were able to confirm the necessity of the required memory barriers for correctness in the well-known Michael-Scott queue, running on our relaxed memory model.

LLMC uses an original LLVM interpreter, but modified to accommodate our needs. By reusing the LLVM interpreter, future LLVM interpreter versions can easily be merged. This allows for new features to be integrating in the existing tool without significant problems. This comes at a performance penalty: serializing and re-initializing the LLVM Interpreter takes more than half the work.

LLMC is also an attempt to bring software model checking to the toolset LTS_{min}. While a lot of language modules already exist, until now there have been none for software model checking, without performing an abstraction step. We hope this tool can form a basis for future software model checking research using LTS_{min}.

1.7 Organization

We first provide required background information. We start by briefly covering the relevant history of multi-processor hardware: why do we have multiple processors in the first place and why do we have to deal with these relaxed memory models (Section 2.1). We then describe the LLVM Project and its low-level intermediate representation LLVM IR (Section 2.2), following by a description of the toolset LTS_{min} (Section 2.3). Finally, we comment on related techniques and tools (Section 2.4).

We then describe our tool, LLMC. We describe our design choices (Section 3.1), provide a design, including execution model (Section 3.2) and how we mapped LLVM IR to PINS (Section 3.3). We

then describe a strategy of exploration that gradually relaxes the memory model and we comment on the soundness and completeness (Section 3.4), followed by a brief description of some implementational details (Section 4.1).

We apply LLMC to various litmus tests (Section 5.1) and execute multiple experiments (Section 5.2) to indicate the validity and applicability of LLMC. For reference, we provide a number of benchmarks based on these experiments (Section 5.3).

We conclude with a brief summary (Section 6.1) and evaluate design choices and achieved goals (Section 6.2). Finally, we suggest future improvements and future possible topics of research (Section 6.3).

2 Preliminaries

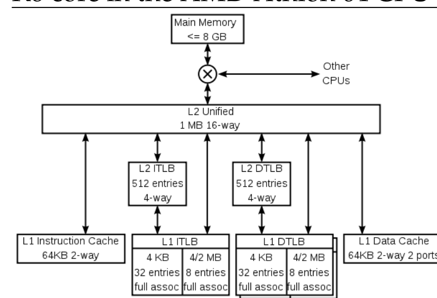
2.1 Computer Architectures

Computer architectures implement a certain instruction set. This instruction set dictates what instructions a program can perform and what the effects of the instructions are. There are many such instruction sets in existence. Popular ones include X86, SPARC and various ARM versions. They are similar in many respects: they all have instructions for memory instructions to load from and store to memory. To actually perform calculations, they usually have basic arithmetic instructions.

At the center of a computer architecture is the central processing unit (CPU). This is the part that executes these instructions. The data that is used during execution is classically stored in memory. The speed of the complete system is influenced by two important factors: 1) the speed of the processor, i.e. how fast it can execute basic arithmetic; and 2) the bandwidth and latency of the memory, i.e. how fast can the processor load and store values.

Ever since the year 1958 the performance of CPUs have roughly doubled every two years [M+65]. However, memory latency decrease has been lagging behind by a significant margin since 1980 [Car02]. This means that inevitably performance will hit a *memory wall* [WM95]: performance will be limited by the speed of accessing memory.

Figure 2.1 The cache hierarchy of the K8 core in the AMD Athlon 64 CPU[1]



2.1.1 Cache

To speed up apparent memory access, CPUs are given a *cache*. This cache is a faster type of memory and acts as a barrier between the CPU and the slower shared memory. See Figure 2.1 for an illustration of this. The idea of the cache is to speed up operations on the same memory addresses over and over again. When the CPU would request the value of an address in the memory, it would be cached. The next request of this address would not go through to the slower shared memory, but the value could be obtained from the cache.

Most computer architectures even employ multiple layers of cache. Because of the limited size of the cache, only a limited number of addresses can be cached. Thus, over time some cached values are flushed to memory in order to make room for other values. The operation of this depends on the heuristics (replacement policy) used; one heuristic could be to flush the 'oldest' cached memory addresses when space is needed.

Depending on the architectural implementation the cache may be coherent or not. A cache is *coherent* iff writes to a single location are serialized so every process observes the same order of writes [MSS12]. A cache is *causal* iff a read of a location does not return the value of a write until all

[1] Source: https://en.wikipedia.org/wiki/CPU_cache

observers observe that write. A cache that is both coherent and causal is *multi-copy atomic* [ARM10].

2.1.2 Write buffer

Writing to memory is also sped up by writing the value to the cache instead. The cache then writes to the shared memory, using a write buffer. A write buffer is part of the CPU cache and buffers writes from the cache to the shared memory. This speeds up apparent processing as instead of waiting for the write to complete, the cache and by extension the CPU can continue other work. One further optimization is to merge writes to consecutive locations in memory, allowing writes to complete out-of-order.

2.1.3 Multi-processor

This allowed computer architectures to gain performance by setting the clock-rate of the processor faster and faster. However, since an electric signal takes time to reach its destination, the clock-rate was bounded by the time to complete an instruction. Thus, to further increase the clock-rate, the execution of an instruction had to be divided into a sequence of steps. The hardware was divided accordingly into stages, the so-called instruction pipeline, the output of one stage being the input of another. The execution time of one stage is significantly smaller than the execution time of the entire instruction. Thus, the clock-rate could be increased, now only bounded by the slowest stage, but still bounded.

Because of this bound on the clock-rate, at a certain point [Sut] a different approach had to be taken. Instead of making one processor faster, the focus shifted towards having multiple processors. However, this approach has a fundamental issue: a single sequential program does not utilize multiple processors, because it is only made to be executed on one. For a program to fully utilize multiple processors, the work has to be divided in such a way that the processors can do part of the work load in parallel. At a certain point the workers running in parallel may have to communicate with one another, for example to signal they are done. The synchronization of parallel workers is not a trivial task as this is usually done by writing to and reading from shared memory. This creates a difficulty, because these operations go through the cache and write buffer. These subsystems can cause two processors to observe a different state of the memory.

2.1.1 Memory instruction reordering

One side-effect of the introduction of cache and write buffers is that the execution time of loading and storing is not a fixed number of cycles. Two loads or stores to different addresses may take a different number of clock cycles to get the value, because one could be cached while the other is not. This would allow load and stores to complete out-of-order. The merging of stores in the write buffer allows stores to complete out-of-order as well.

Thus, the evident execution order is different from the program order. This is because of the different execution times of the instructions and that the processor in question did not wait for an instruction to be completed before going to the next. The operation of single sequential programs is not altered by this reordering, because the processor takes them into account.

Not all processors employ this policy of allowing loads and stores being reordered this way. Table 2.1 shows some instruction sets and which memory operations they allow to be reordered.

2.1.1.1 Memory models

For single processor architectures it is not a problem to allow memory operations to be reordered this way. The hardware implementation guarantees that the effects of reordering are not observable by the program running on the single processor. However, a problem arises when multiple processors are introduced, all with their own cache and write buffer. If one processor executes its memory operations out of order, it could mean that another processor observes this fact. Even though this reordering does not alter the semantics of the first processor, it could inadvertently cause the second processor to observe a state of the memory that was not intended to be observed. When and how a processor observes writes from another processor is governed by the memory model of the instruction set.

Figure 2.2 Reordering on X86, is $\texttt{R1} = 1 \vee \texttt{R2} = 1$ guaranteed?

P_1	P_2
`X ← 1` `R1 ← Y`	`Y ← 1` `R2 ← X`

Table 2.1 Some instruction sets and their policy on reordering memory operations

		RMO		PSO	TSO	
Relaxation	**Alpha**	**ARMv7**	**IA-64**	**SPARC PSO**	**x86**	**AMD64**
Loads reordered after loads	✓	✓	✓			
Stores reordered after loads	✓	✓	✓	✓	✓	✓
Loads reordered after stores	✓	✓	✓			
Stores reordered after stores	✓	✓	✓	✓		
Atomic reordered with loads	✓	✓	✓			
Atomic reordered with stores	✓	✓	✓	✓		
Dependent loads reordered	✓					

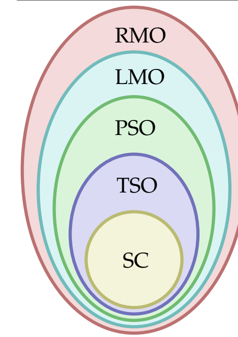
A memory model dictates the conditions under which writes of one processor become observable to another processor and places constraints on read operations.

An example of this is shown in Figure 2.2. The memory model of X86 allows a processor to reorder loads before stores to different addresses. Thus, the operation `R1` ← `Y` may read the value of `Y` before 1 is written to the memory where `X` is stored. In that event, R1 will be equal to 0. Then P_2 runs in its entirety. Because 1 was not yet written to `X`, R2 will also be equal to 0. Then finally the store operation of P_1 finishes and thus P_1 is also done. The end state is `X` == `Y` == 1 and `R1` == `R2` == 0, which is not something one might expect.

This is just one type of reordering of memory operations: store after load. There are in total four types of these relaxations on memory instruction order as can be seen in Table 2.1. The three other cases, atomic and dependent operations, are special cases of these four. Not every processor allows for the same reordering to happen, but even on the X86, which only allows stores to be reordered after loads, it is a source of bugs in multi-threaded programs.

We consider four commonly used memory models: sequentially consistent (SC), total store order (TSO), partial store order (PSO) and relaxed memory order (RMO). In addition to these, we describe a limited relaxed memory order (LMO) memory model that we will use for LLMC.

Figure 2.3 Memory models hierarchy



- SC We use sequentially consistent to denote the memory models that specifies that all memory operations are observed by all observers at the same time: no reordering is allowed.
- TSO `[w→r]` Total store order means that all observers agree on a single total order of all store operations. Reads are allowed to be reordered after writes.
- PSO `[w→r/w]` Partial store order means that in addition to the relaxation of TSO, the store operations issued by a process may overtake other store operations and atomic operations.
- LMO `[w→r/w + r→*r/w]` Limited relaxed memory order is an extension to PSO: in addition to allowing `w→r/w`, LMO allows `r→r/w` in some cases, but not all. We will specify these cases and give a formal description of LMO in Section 3.2. Without this limitation, it would be equal to RMO.
- RMO `[r/w→r/w]` Relaxed memory order means a memory model that allows any number of relaxations, with the exception of the reordering of dependent loads.

Note that we left the memory model of the Alpha, which allows dependent loads to be reordered with each other. This is a relic of the past and we will not consider it.

2.1.1.2 Multi-processor communication

To restrain the processor to reorder memory operations at a certain point in the program order, the programmer can use memory barriers, also called *memory fences*. These are instructions used to prohibit certain memory operations from being reordered. Simply put, memory barriers work by not allowing memory operations to "jump" over the barrier, i.e. being reordered past a barrier.

There are four types of barriers, each to restrain a certain reordering: `LoadLoad`, `LoadStore`, `StoreLoad` and `StoreStore`. For example, the `StoreLoad` barrier guarantees that all store in-

structions before the barrier are executed and observed before load instructions after the barrier. Table 2.2 shows with which instructions x86, ARMv7 and SPARC [Spa98] implement these four barriers. These instructions are needed for a processor to cooperate *correctly* with another processor. If the code running on one processor makes assumptions on the state of a second processor based on the state of the memory, the memory should be in a state consistent with the state of the second processor. Thus, barriers need to be explicitly added where needed.

The example of Figure 2.2 is fixable by adding memory barriers. A fixed version is shown in Figure 2.4. The reordering of the store after the load needs to be prohibited, thus we insert a `StoreLoad` barrier between these instructions. Now the processor is not allowed to reorder the store past the load and thus we can guarantee that in the end $R1 = 1 \vee R2 = 1$. This is just a simple toy example. As a multi-threaded program grows and its concurrency complexity increases it becomes more difficult to know where there barriers are needed.

Figure 2.4 $R1 = 1 \vee R2 = 1$ is guaranteed!

P_1	P_2
X ← 1 SL-barrier R1 ← Y	Y ← 1 SL-barrier R2 ← X

Even more difficult is to know when a memory barrier can be left out. This is important, because these instructions cause processors to synchronize with each other. Synchronization yields a performance hit as it removes some parallelism from the program. Thus, memory barriers have to be placed with care. Alternating every instruction of a program with memory barriers will make it sequentially consistent, but its performance will be significantly degraded.

2.1.1.3 Reasons of relaxation

So this raises the question: why do instruction sets not just implement a sequential consistent memory model? Having a relaxed memory model has various advantages.

A relaxed memory model allows more optimizations. Because the hardware does not have to deal with sequential consistency, it may optimally reorder instructions. Instruction reordering is a feature that provides a healthy speed increase, by fully exploiting instruction level parallelism (ILP) in the instruction pipeline [HP06]. The unconstrained buffering of writes hides the write latency of slow memory. The faster cache hides the latency of reads and writes.

If these advantages were desired *and* a stronger memory model were a requirement, then a lot of complex hardware would need to be added to keep both the performance and correctness, depending on the required guarantees. Adding complex hardware not only further increases the time and costs of development, but also the production costs. The added hardware also means an increase in die size, thermal design power (TDP) and power requirements as well. A higher TDP means that the hardware gets hotter, which means better cooling is required. In this era of measuring performance in instructions per watt, these are important factors.

Thus, a relaxed memory model benefits the hardware. However, the software running on this hardware has the disadvantage of having to cope with less guarantees. This makes the software more complex, because the synchronization now has to deal with multiple processors possibly observing a different state of the memory. A lot of responsibility now rests upon the shoulders of the programmer to write correct multi-threaded code using a relaxed memory model.

[1] Sparc-V9's `membar` supports more control than specified here, e.g. flushing the lookaside buffer

Table 2.2 Some instruction sets and their memory barrier instruction

Type	x86	ARMv7	Sparc-V9[1]
load-load	`lfence`	`dmb`	`membar #LoadLoad`
store-load	`mfence`	`dmb`	`membar #StoreLoad`
load-store	`mfence`	`dmb`	`membar #LoadStore`
store-store	`sfence`	`dmb`	`membar #StoreStore`

2 The LLVM Project

The LLVM Project [LLV] is a collection of modular and reusable compiler and toolchain technologies. The origin of the LLVM Project lies with the Master's Thesis of Chris Lattner [Lat02]. LLVM used to be an acronym for Low-Level Virtual Machine, but they presumably changed this to simply LLVM to iterate the fact that LLVM is more than a virtual machine. It is a complete infrastructure for compilers, using a language-independent register-based instruction set and type system. This instruction set is LLVM Intermediate Representation (LLVM IR) and is at the heart of the LLVM project. LLVM IR instructions are in static single assignment (SSA) form, meaning every variable (a typed register), is assigned once. One of the advantages of SSA is that it allows simple variable dependency analysis.

as can be seen in Figure 2.5. This figure also depicts where LLMC would fit in the existing LLVM toolchain.

There are many front-ends that generate LLVM IR, enabling LLVM to support a wide variety of languages [LLV]: ActionScript, Ada, D, Fortran, GLSL, Haskell [TC10], Java bytecode, Julia, Objective-C, Python, Ruby, Rust, Scala, C#, and Erlang [SST12]. After these front-ends compile a program from a language to the LLVM IR, the LLVM tool chain takes it from there.

An important next step is optimization. The generated LLVM IR may contain redundant code that the front-end generated naively; some registers may be optimized out; or inlining instructions may improve performance. The LLVM collection has numerous optimization passes, supporting compile-time, link-time, run-time, and "idle-time" optimization of programs. Some of these can be performed on any LLVM IR, regardless of the machine the code will be executed on. Others are only available when the target architecture is known or only for a specific target.

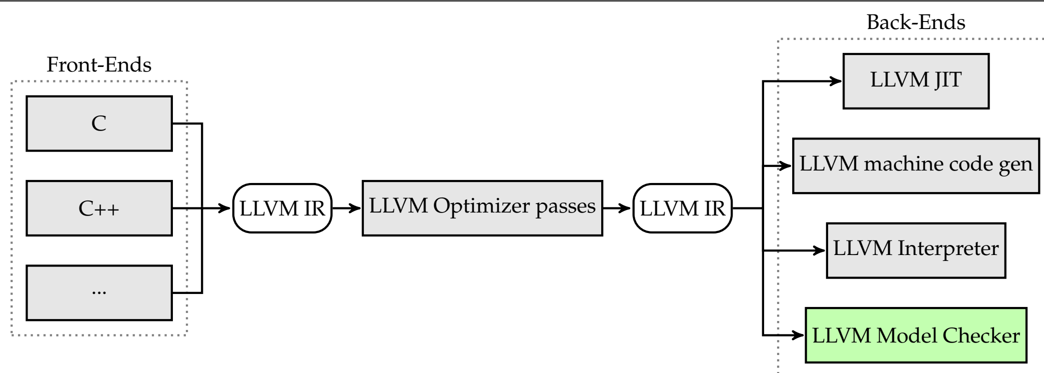
After the optimization step the optimized LLVM IR is used to generate machine code. This can either be done statically, resulting in a binary that can only be executed on the target architecture, or it can be done in a just-in-time (JIT) fashion.

2.2.1 Intermediate Representation

The LLVM IR has three distinct goals. It is well-suited to be 1) used by a compiler in-memory; 2) to be used as an on-disk file and later compiler by a JIT compiler; 3) to be used as a human readable assembly language [LLI]. In all of these scenarios the LLVM IR is equivalent. LLVM IR aims to be a representation that is low enough to not sacrifice performance, while providing the means such that high-level concepts can be mapped to it in a clean fashion.

LLVM programs are built from `Module`s containing LLVM IR. A module is usually the result of one of the front-ends translating a single unit into a single `Module`. Each module may contain functions, global variables, and symbol table entries. The LLVM linker can link these modules together, thus forming a new `Module` that is the results of merging the linked modules. During the merge, optimizations may have taken place.

Figure 2.5 The flow of data: 1) front-ends; 2) optimizer passes; 3) back-ends



2.2.1.1 Type System

One of the key assets of the LLVM IR is its type system. It provides enough information to allow various optimizations directly on the LLVM IR without surplus analysis. Combined with the fact that is in the SSA form, it allows for easy analysis and transformations. Table 2.3 lists the available types.

Primitive Types

Primitive types form the basis of the LLVM Type System. The primitive types are: `label, void, integer, floating point, x86mmx, metadata`. Table 2.4 lists the available primitive types and shows a description of each.

Table 2.4 LLVM IR Primitive Types

Type name	Description
`label`	Labels are references to specific positions in the LLVM IR.
`void`	A void type is a type without size or value.
`integer`	Integers are used to describe whole numbers. Any integer type with a bit width ranging from 1 to $2^{23} - 1$ can be created.
`floating point`	Floating points are used to describe real numbers. There are various floating point types, offering different domains and resolution.
`x86mmx`	This type is only available on an x86 machine. It is used to describe an MMX register.
`metadata`	The metadata type represents embedded metadata.

Aggregate Types

Aggregate types are types that are composed from other types. Table 2.5 lists the available methods of creating new aggregate types and shows a description of each.

Table 2.5 LLVM IR Aggregate Types

Type name	Description
Arrays	Array types describe a number of elements of any one certain type sequentially in memory.
Structs	Struct types describe a collection of data members of various types, grouped together in memory.

Function Types

Function types are types that describe the signature of a function. It contains a return type and a list of parameter types. It is made up of basic blocks, containing instructions. These basic blocks can be jumped to, allowing the familiar `goto`, `if`, and `while` to be implemented. The last instruction of every basic block should be a terminator instruction, listed in Table 2.7.

Pointer Types

Pointer types are used to point to a specific location of a specific type in memory.

Vector Types

Table 2.3 LLVM IR Types

Class	Types
integer	`i`n, where $1 \leq n \leq 127$, e.g. `i8`, `i10`, `i64`
floating point	`half`, `float`, `double`, `x86_fp80`, `fp128`, `ppc_fp128`
first class	`integer, floating point, pointer, vector, structure, array, label`, `metadata`
primitive	`label`, `void`, `integer`, `floating point`, `x86mmx`, `metadata`
derived	`array`, `function`, `pointer`, `structure`, `vector`, `opaque`

Vector types describe a vector of elements. They are not equal to an array and are not considered an aggregate type. Vector types are used for SIMD (single instruction multiple data) instructions, where a single instructions operates on the elements in parallel.

Examples

Table 2.6 shows a list of examples of LLVM types.

Table 2.6 LLVM IR Examples of Types

LLVM IR	Description
`i100`	An integer of 100 bits
`[float x 23]`	Array of 23 single precision floating point values
`[10 x [14 x i8]]`	10x14 array of 8-bit integer values
`<{ i8, i16 }>`	A packed structure of exactly 3 bytes
`{ i8, i16 }`	A non-packed structure where padding between elements may be inserted
`float (i16, i8*)*`	Pointer to a function that takes an `i16` and a pointer to `i8`, returning `float`
`{i16, i16} (float)`	A function taking a `float`, returning a structure containing two `i16` values

2.2.1.2 Instruction Set

Terminator instructions

Terminator instructions are used to terminate a basic block; see Table 2.7 for a list. They decide what the next basic block to be executed is. A `br` for example jumps to the specified basic block. A `ret` finishes the current function and optionally returns a value, then allowing the caller to continue execution of the basic block where the callee was called.

Table 2.7 LLVM IR Terminator Instructions

LLVM IR	Description
`ret`	The return instruction hands back control to the function caller and optionally returns a value
`br`	The branch instruction branches to a target label, optionally depending on a conditional value
`switch`	The switch instruction branches to any one label of a list of labels and values, depending on a value
`indirectbr`	The indirect branch instruction branches to any one label of a list of labels and values, depending on an address
`invoke`	The invoke instruction calls a specific function and provides a mechanism for the function to throw an exception
`resume`	The resume instruction is used to resume an exception that is currently in-flight
`unreachable`	The unreachable instruction is used to tell the optimizer that this instruction should not be reachable

Binary instructions

LLVM IR supports a number of binary instructions: `add`, `fadd`, `sub`, `fsub`, `mul`, `fmul`, `udiv`, `sdiv`, `fdiv`, `urem`, `srem`, `frem`, `shl`, `lshr`, `ashr`, `and`, `or`, `xor`.

Memory instructions

The memory instructions of LLVM IR are of special interest to this project; see Table 2.8 for a list. LLVM IR provides atomic memory instructions and defines behaviour in their presence. This can be used for our multi-threaded LLVM IR programs. Together with the memory model LLVM IR defines, we can abstract from hardware memory models and only concern ourselves with the software memory model LLVM defines. We will discuss this further in Section 2.2.2.

Table 2.8 LLVM IR Memory Instructions

LLVM IR	Description
alloca	Allocates memory on the current stack frame
load	Loads a value from a location in memory into a register
store	Stores a value to a location in memory
fence	A memory fence can be used to introduce dependencies between instructions
cmpxchg	Atomically compares and modifies memory
atomicrmw	Atomically modify memory
getelementptr	Get the address of an element of an aggregate type

2.2.1.3 LLVM IR Example

An example implementation of the program shown in Figure 2.2:

```
1  @x = global i32 0
2  @y = global i32 0
3
4  define i32 @proc_1_t0() {
5  entry:
6    store i32 1, i32* @x ; X  ← 1
7    %R1 = load i32* @y   ; R1 ← Y
8    ret i32 %R1          ; for valid IR
9  }
10
11 define i32 @proc_1_t1() {
12 entry:
13   store i32 1, i32* @y ; Y  ← 1
14   %R2 = load i32* @x   ; R2 ← X
15   ret i32 %R2          ; for valid IR
16 }
```

2.2.2 Memory Model

LLVM version 3.0 introduced a memory model to define the behaviour of LLVM IR in the presence of multiple threads executing LLVM IR code [LLI]. This model is derived from the C++11 memory model [ABH+04]. We will first discuss the C++11 memory model and then the LLVM memory model.

Figure 2.6 data should be available, when P_2 observes shared.status = 1

P_1	P_2

shared.X ← data
ssfence
shared.status ← 1
	if(shared.status)
	  llfence
	  data ← shared.X

2.2.2.1 C++11 Memory Model

The two most important keywords in the C++11 memory model are `release` and `acquire`. Together with `seq_cst` for sequentially consistent and `relaxed` for no guarantees, they govern the possible memory order semantics of memory barriers, i.e. `std::atomic_thread_fence(std::memory_order)`. They can also be used when calling `std::atomic` methods, e.g. `std::atomic<int>::store(int, std::memory_order)`. Note that this generally only makes sense in the presence of other shared variables.

Figure 2.7 shows a C++11 interpretation of the program shown in Figure 2.6. Notice that in the second C++11 version the placement of the memory barrier is different from the other examples. It is placed before the conditional jump generated by the `if` statement instead of after it like the other examples. This can cause a performance hit when running on architectures that need an explicit memory barrier at that point. If the condition of the `if` statement evaluates to `false`, the fence is not needed in this example.

The supported memory ordering specifications are listed below together with their intended semantics.

Figure 2.7 C++11 interpretations of Figure 2.6

P1 / **P2** **(explicit fence)**

P1:
```
shared.X ← data
atomic_thread_fence(memory_order_release);
shared.status ← 1
```

P2:
```
if(shared.status==1) {
  atomic_thread_fence(memory_order_acquire);
  data ← shared.X
}
```

P1 / **P2** **(atomic store/load)**

P1:
```
shared.X ← data;
shared.status.store(1, memory_order_release);
```

P2:
```
if(shared.status.load(memory_order_acquire)==1)
  data ← shared.X;
```

relaxed The `relaxed` memory order specifies that no ordering whatsoever is guaranteed in relation to other variables. It does specify that for all write operations to any single memory location there is a single total order. When compiling for an instruction set that has a coherent cache, this does not add any more guarantees than the instruction set guarantees.

release The `release` memory order specifies that memory operations before the fence will be observed by all observers before any `store` after the fence. [LS+SS FENCE]

acquire The `acquire` memory order specifies that `load` operations before the fence will be observed before any memory operation after the fence. [LL+LS FENCE]

seq_cst The `seq_cst` memory order specifies that memory operations before the fence will be observed by all observers before any memory operation after the fence. When put before and after every memory instruction, this guarantees that all observers agree on a total order of those memory operations. [LL+LS+SL+SS FENCE]

2.2.2.2 LLVM Memory Model

The LLVM memory model employs the same semantics as C++11 for the specified memory ordering keywords. In addition, it specifies *unordered* and *monotonic*. These ordering specifications are applicable to the atomic LLVM IR instructions such as `fence` or `cmpxchg`.

monotonic The `monotonic` memory order is the equivalent of C++11's **relaxed** memory order.

unordered The `unordered` memory order guarantees only that a value can only be read if it was previously written. This is a very weak guarantee, but strong enough to model Java's non-volatile shared variables. This is a more relaxed guarantee than **monotonic**, because it allows observers to observe a different order of write operations to a single location. However, since we assumed a multi-copy atomic cache, we cannot model this relaxation.

Figure 2.8 shows the LLVM IR interpretations of the program shown in Figure 2.6. Notice that the placement of fences matches that of the C++11 example perfectly.

LLVM IR can be compiled to various instruction sets. By providing a software memory model, LLVM IR abstracts from the memory model used in various instruction sets. LLVM IR that correctly implements this memory model and is verified to be correct under this model, is guaranteed to run on supported instruction sets, regardless of its memory model. This is assuming the compiler from LLVM IR to machine code correctly implemented the mapping from the LLVM memory model to the memory model of the instruction set in question.

2.2.3 Motivation for The LLVM Project

There are a number of compelling arguments for using The LLVM Project. Firstly, having a low-level intermediate representation, The LLVM Project provides a precise mapping to machine instructions. The LLVM IR was designed to be a platform-independent, low-level representation of a program. Thus, it resembles an assembly language. This is an advantage over JVM and .NET, because it more closely resembles the generated machine code.

A second advantage is that the LLVM Project has numerous front-ends, supporting many languages, including C++11 and Java. The LLVM community is of considerable size and enjoys support from various companies. It is still growing as well, more features being added to the tool chain at a fast rate [LLV]. The performance of the generated machine code is on par with GCC generated machine code [Lara, Larb]. A major advantage to GCC is that the compile times of LLVM generated machine code are considerably lower than those of GCC.

A third advantage is the use of SSA to unique registers. In LLVM IR, there are no limits to the number of registers used, so each assignment to a register 'creates' a new register. This is useful because it means once a register is created, its value will not change.

A disadvantage of using LLVM is that we obtain answers for LLVM IR and not in the language that was used to compile to LLVM IR, for example C++11. Using LLVM IR gives us a generic way of reasoning, but it does not automatically map back to the language of every front-end. Another disadvantage is that because LLVM IR is primarily used by generating it from other, higher order languages, potentially information is lost. For example in C, it is valid to optimize `exit(0)` in `main()` to `return 0`, but this knowledge does not have to be passed down.

Figure 2.8 LLVM IR interpretations of Figure 2.6

Using explicit fence

```
@shared_data = global i32 0
@shared_status = global i32 0

define i32 @proc_1_t0() {
entry:
  store i32 1234, i32* @shared_data
  fence release
  store i32 1, i32* @shared_status
  return
}

define i32 @proc_1_t1() {
entry:
  %status = load i32* @shared_status
  %_eq_ = icmp eq i32 %status, 1
  br i1 %_eq_, label %then, label %merge

then:
  fence acquire
  %data = load i32* @shared_data
  br label %merge

merge:
  return
}
```

Using atomic store/load

```
@shared_data = global i32 0
@shared_status = global i32 0

define i32 @proc_1_t0() {
entry:
  store i32 1234, i32* @shared_data
  store atomic i32 1, i32* @shared_status release
  return
}

define i32 @proc_1_t1() {
entry:
  %status = load atomic i32* @shared_status acquire
  %_eq_ = icmp eq i32 %status, 1
  br i1 %_eq_, label %then, label %merge

then:
  %data = load i32* @shared_data
  br label %merge

merge:
  return
}
```

2.3 LTSmin

For the verification we have chosen LTS$_{min}$ [LPW11a, BPW10]. This is a toolset providing a modular high-performance model checker. It contains multiple language modules supporting various input specification languages, such as μCRL, mCRL2, DVE, PROMELA [BL12], UPPAAL and ETF. The modularity stems from the use of a single, specified interface: the PINS interface.

In this section, we will first describe LTS$_{min}$ and then motivate the choice.

2.3.1 The PINS Interface

PINS, **P**artitioned **N**ext-**S**tate **I**nterface, is an interface between the various parts of LTS$_{min}$, see Figure 2.9 for an illustration. All LTS$_{min}$ modules work with this interface and thus modules implementing optimization algorithms can be reused by any language module. The result of this clean interface is the separation of concerns into three areas: language modules, PINS optimization modules and model checking algorithm modules.

There are four primary model checking tools that implement PINS: sequential, multi-core, distributed and symbolic:

- The sequential back-end offers LTL model checking using partial-order reduction [LPPW13]. The storage can optionally be done using BBD-based state storage.
- The multi-core back-end [LPW10] optimizes exploration on a single machine using multiple processors and shared memory. It supports LTL model checking and uses a tree-based compression method to store states [LPW11b]. Both multi-threaded and multi-process exploration is supported.
- The distributed back-end [BLPW09] allows a cluster of compute nodes to explore the state space. It supports multi-core exploration as well, but is not as optimized for single machine operations as the multi-core back-end. Exploration is limited to safety checking.
- The symbolic back-end [BPW10] supports CTL/μ-calculus model checking [BPW09] using various BDD/MDD packages, including the parallel BDD package Sylvan [DLP13].

2.3.1.1 Next-State

A transition system (TS) is a structure $\langle S, \rightarrow, s^0 \rangle$, where S is a set of states, $\rightarrow \subseteq S \times S$ is a transition relation and $s^0 \in S$ is the initial state.

For example, take the transition system $\Gamma = \langle S_\Gamma, \rightarrow_\Gamma, s^0_\Gamma \rangle$, where

- $S_\Gamma = \{\langle i, j \rangle \mid i, j \in \{0, 1, 2\}\}$,
- $\rightarrow_\Gamma = \{\langle \langle i, j \rangle, \langle i+m, j+n \rangle \rangle \mid i, j, m, n \in \{0, 1\}, m \neq n\}$,
- $s^0_\Gamma = \langle 0, 0 \rangle$.

Figure 2.10 illustrates this transition system.

Figure 2.10 State space of Γ example

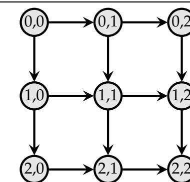
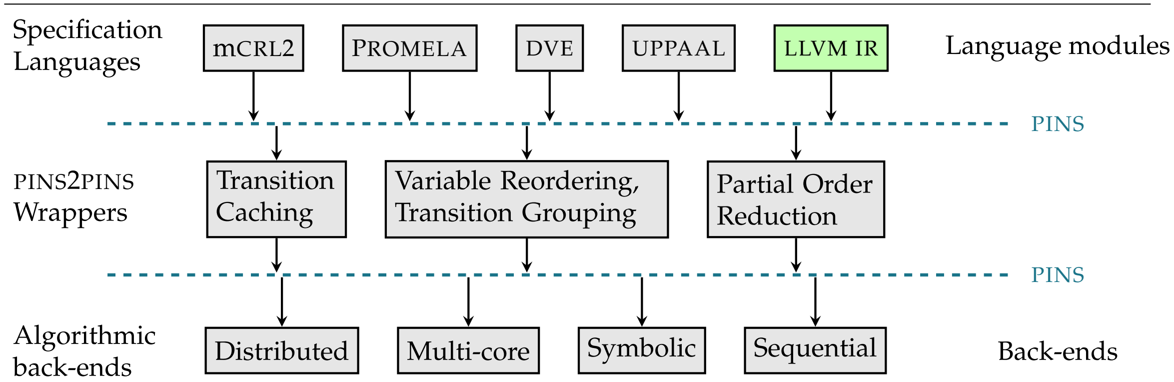


Figure 2.9 PINS, **P**artitioned **N**ext-**S**tate **I**nterface



2.3.1.2 Partitioned Next-State

LTS*min* uses a *partitioned* next state interface, which uses a partitioned transition systems (PTS). In a PTS, the set of states is a Cartesian product and the transition relation is the union of *transition groups*. A PTS is a structure $\mathcal{P} = \langle\langle S_1, ..., S_N\rangle, \langle\rightarrow_1, ..., \rightarrow_K\rangle, \langle s_1^0, ..., s_N^0\rangle\rangle$, where

- the sets of elements $S_1, ..., S_N$ define the set of states $S_\mathcal{P} = S_1 \times \cdots \times S_N$;
- the transition groups $\rightarrow_i \subseteq S_\mathcal{P} \times S_\mathcal{P}$, $1 \leq i \leq K$ define the transition relation $\rightarrow = \bigcup_{i=1}^{K} \rightarrow_i$;
- the initial state $s^0 = \langle s_1^0, ..., s_N^0\rangle$.

The defined TS of $\mathcal{P}$ is $\langle S_\mathcal{P}, \rightarrow, s^0\rangle$. A state $s \in \langle S_1, ..., S_N\rangle$ in a PTS is in fact a vector of N *slots* or variables.

This provides the ability to define transition groups that read or modify certain slots. These *dependencies* can be specified in a dependency matrix $D_{K\times N}$, a matrix with K rows (transition groups) and N columns (state vector slots). $D_{i,j}$ specifies whether transition group i depends on state vector slot j. The dependency matrix is relayed from the language module to the back-end via PINS.

An example partitioned transition system based on the Γ example is $\Delta = \langle\langle S_\mathtt{i}, S_\mathtt{j}\rangle, \langle\rightarrow_\mathtt{i}, \rightarrow_\mathtt{j}\rangle, \langle s_\mathtt{i}^0, s_\mathtt{j}^0\rangle\rangle$, where

- $S_\mathtt{i} = S_\mathtt{j} = \{0, 1, 2\}$,
- $\rightarrow_\mathtt{i} = \{\langle\langle i, j\rangle, \langle i+1, j\rangle\rangle \mid i \in \{0,1\}, j \in \{0,1,2\}\}$
  $\rightarrow_\mathtt{j} = \{\langle\langle i, j\rangle, \langle i, j+1\rangle\rangle \mid j \in \{0,1\}, i \in \{0,1,2\}\}$,
- $s_\mathtt{i}^0 = s_\mathtt{j}^0 = 0$.

Figure 2.11 Dependency matrix

Transition Groups	State vector i	j
$\rightarrow_\mathtt{i}$	+	
$\rightarrow_\mathtt{j}$		+

r: read, w: write, +: read/write

Notice that the transition groups do not read or write all state vector variables when calculating the next states of a state. Transition group $\rightarrow_\mathtt{i}$ reads and writes only i and analogically $\rightarrow_\mathtt{j}$ reads and writes only j. The advantage of a PTS is that if a transition group does not depend on all the state vector slots, reachability tools can exploit this.

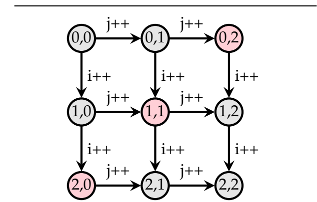
The dependency matrix of this example is shown in Figure 2.11.

2.3.1.3 Labels

More information can be added to the transition system by specifying *state labels* and *edge labels*. State labels are similar to state vector variables and their value is based on a subset of state vector variables. State labels can be used to describe a certain property of a state. For example, a state label `totalIsTwo` could be added to the earlier Γ, calculated by

$$SL_{\mathtt{totalIsTwo}}(\langle i, j\rangle) = \begin{cases} 1, & (i+j=2) \\ 0, & \text{otherwise} \end{cases}$$

Figure 2.12 State space of labeled Γ example



The fact that their value is solely based on the state vector allows them to be calculated on demand.

Edge labels are labels associated with a transition and are not solely based on a single state. Instead, edge labels are calculated by the language module when reporting a new transition to LTS*min* via PINS. Thus, they are not generated on demand, but calculated once for every new transition.

In Figure 2.12 a labeled version of the Γ example is shown. In this version, we coloured the state s red iff $SL_{\mathtt{totalIsTwo}}(s) = 1$. We added an edge label in the transition system to provide a description of each transition.

2.3.1.4 Trace generation

We can also define traces on a transition system. A *trace* is a path from one state to another state, including all states in between. A trace ρ can be written as $\rho = s_0 s_1 ... s_n$, $\forall 0 \leq i \leq in : s_i \rightarrow s_{i+1}$. For example, $\langle 0,0\rangle\langle 1,0\rangle\langle 1,1\rangle\langle 1,2\rangle\langle 2,2\rangle$ is one of the six traces from $\langle 0,0\rangle$ to $\langle 2,2\rangle$ in the Γ example.

We can ask LTS_{min} to search the state space for a state with a certain property, e.g. an erroneous state. Finding this erroneous state tells us that the program is not correct, but does not tell us why: for this we need LTS_{min} to generate a trace to the state. This will tell us how the state can be reached: what transitions have been taken and what the intermediate states are.

2.3.1.5 Linear Temporal Logic

LTS_{min} supports Linear Temporal Logic (LTL). LTL is a model temporal logic which can be used to create formulae that reason about traces. For example, a formula stating that a certain condition will eventually be true or that it remains true in all possible paths until some other condition is satisfied. In the case of LTS_{min}, conditions can be about state vector variables or state labels.

2.3.1.6 Chunk Mapping

The toolset LTS_{min} supports a feature that is called *chunk mapping*. This boils down to a table of chunks of data, where a chunk is identified by a single integer. These chunks are available to all back-ends and all workers. The language module can *upload* a chunk map to LTS_{min}, getting back a chunk identifier. This chunk identifier can then be put into the state vector. Later, when reading back the state, this identifier can be used to *download* this chunk from LTS_{min}. If there are significantly less versions of a chunk than there are states, this reduces the memory needed to describe the state space.

2.3.2 Motivation for LTSmin

Much research has been done over the last decade to make LTS_{min} what it is now [LTS]. It is a model checker and complete suite of tools, supporting multiple exploration algorithms and various input specifications. Because LTS_{min} separates language modules from analysis tools using PINS, future improvements to the analysis tools will apply to older language modules as well. This may require a patch to the original language module, but even then it is a insignificant compared to the advantages of gaining algorithmic and implementational advancements. Moreover, by implementing PINS, we automatically enable the use of a wide range of reachability tools: e.g. distributed, multi-core and symbolic. This is very useful for this research: we get these reachability tools for free and benefit from future improvements to them as well.

Furthermore, it is interesting to investigate how LTS_{min} can cope with large state vectors containing entire registers, stacks and memory. Together with the state space explosion caused by memory operation reordering this forces to investigate into memory footprint reducing techniques.

Multiple approaches have been investigated [GWZ+11, BL13] in an attempt to make software model checking more practical. This research is an attempt to pave the way for this research to continue using the LTS_{min} toolset as a basis. This is one thing that has been missing from the wide range of input specifications LTS_{min} can handle: source code. Because the chosen target is LLVM, this would widen the input range to a whole new audience. Instead of having to create a model first and then feed it to LTS_{min}, it would be possible to directly model check a program using its source code. This makes this research interesting for the development of LTS_{min}.

2.4 Related Work

There are a lot of similar approaches and tools already existing. However, they all have one or more aspects that are not similar. In this section we list a few approaches and tools that relate to the goal of this project in some way.

2.4.1 Related Approaches

2.4.1.1 SAT/SMT solving

SAT solving involves determining whether or not a boolean formula can be satisfied using a certain interpretation. A small example is determining for what interpretation of a, b and c the formula $a \vee b \vee c \Leftrightarrow a \wedge b \wedge c$ holds (there are two of these interpretations for this example).

SMT solving is similar to SAT solving: an SMT formula is a generalization of a boolean SAT formula. SMT adds predicates over a set of non-boolean variables to the mix.

In recent years, research [Duc13, Mos09] has been done to bring SMT solving to low-level software supporting pointer arithmetic and to allow to reason about the heap. This research makes SMT solving feasible to use in the context of verifying LLVM IR.

However, both approaches would require mapping every LLVM IR instruction to a formula. Capturing the precise meaning of every instruction is prone to human error and may prove to be cumbersome.

2.4.1.2 CEGAR

A Counter-Example Guided Abstraction Refinement [Kur94, CGJ+00, BL13] loop is a technique to symbolically build an abstract model of a model by iteratively applying refinement techniques using automatically generated counterexamples. When used on software model checking, a counterexample is a path that leads to an error state. If the path is not possible in the original model, the abstract model is subsequently refined accordingly.

This technique has also been successfully applied [CI10] to assembly code, thus avoiding assumptions about higher level languages and supporting pointer arithmetic. This makes it an interesting approach as well.

2.4.1.3 Static Analysis

Static analysis of source code is done by verifying that invariants hold. These invariants can be manually specified, but research [DH07] has done to automatically deduce invariants, even on low-level code allowing recursive data structures and pointer arithmetic.

2.4.2 Related tools

2.4.2.1 MCP

MCP [TBS] is a **M**odel checker for **C** **P**lus plus. It is an explicit-state software model checker for LLVM bitcode, but aimed at model checking C++ code. An important difference is that MCP assumes no memory model and does not mandate a specific threading model. Moreover, it specifically targets C++ instead of generic LLVM IR.

It does contain a few interesting features. Firstly, it uses a versioned heap, allowing the run-time to remember the history of the memory. By itself this is not more than what LTS$_{min}$ offers, but the implementation is at least interesting: instead of saving states, state deltas are saved.

2.4.2.2 Klee

KLEE [CDE08] is a symbolic virtual machine built on top of the LLVM compiler infrastructure. It can be utilized as an automatic generator of coverage tests. It produced some significant results, e.g. uncovering bugs in the COREUTILS library that had been missed for over 15 years.

While KLEE also targets LLVM IR, coverage test generation is a different approach to debugging software than model checking. It is a testing technique to generate tests that combined *cover* as much of the software as possible. The measurement of coverage can be based on for example what statements are executed at least once or the combination of conditions in `if-else` statements. Research has been done to combine software testing and model checking and [BG04].

2.4.2.3 LLBMC

LLBMC [SFM10], the Low-Level Bounded Model Checker, is a static software analysis tool for finding bugs in LLVM IR programs. It is based on the technique of Bounded Model Checking and primarily targeted at verifying low-level programs. It does not support concurrency.

2.4.2.4 DiVinE

DiVinE 3.0 [BBH+13] is a tool that closely matches the goal of this project. While targeting multi-threaded C/C++ programs, they use generic LLVM IR as input that in theory can be generated by other LLVM front-ends. The difference is that DiVinE does not reason about programs running on instruction sets employing a relaxed memory model such as the ARMv7. They do support reasoning about programs running on TSO and automatic memory barrier insertion under TSO.

2.4.2.5 JPF and MoonWalker

The Java PathFinder is an explicit-state model checker for Java programs. It is made by NASA with the primary purpose of finding concurrency bugs like data races and deadlocks. JPF uses on-the-fly partial order reduction [Pel96] to minimize the state space.

JPF is similar to the goal of this project, but it assumes the Java memory model and model checks only Java programs.

A similar tool for .NET programs is called MoonWalker [ANR09].

2.4.2.6 Verisoft

VeriSoft is a model checker for concurrent C and C++. It does not save visited state, but uses persistent sets and sleep sets to explore the state space. It can detect dead locks, live locks, divergence and assertion violations. However, VeriSoft assumes a sequentially consistent memory model.

2.4.2.7 CHESS

CHESS [MQB+07] is a stateless model checker for finding and reproducing Heisenbugs in concurrent programs. CHESS repeatedly runs a program and on every run tries a different interleaving. If an interleaving results in an error, CHESS can report it.

Sober [BM08] extends CHESS by combining a store buffer safety monitor and thus supporting relaxed memory models.

2.4.2.8 SLAM

SLAM [BLR11] is a Microsoft Research project which found many bugs in Windows Device Drivers. The SLAM Project uses CEGAR as a means to search for bugs and violations in C. It only targets sequential programs, but there are some extensions that allow concurrent programs.

2.4.2.9 Saturn

The Saturn project [XA05] is is another tool for bug-finding and verification in C using static analysis. Saturn uses three main concepts: 1) it's summary-based, the analysis of a function f is a summary of the behaviour of f; 2) it's constraint-based, analysis is expressed in a set of boolean constraint formulae; and 3) program analysis is expressed in a logic programming language.

The tool scales well [DDA08] and found numerous locking errors in the Linux kernel. It supports heap manipulation, but only the operations `load` and `store` on pointers. Moreover, it only supports sequential programs. However, it is not entirely bit-accurate, because functions are abstracted into small finite-state property automata.

Table 2.9 Comparison of approaches

Aspect	Target	Approach	Concur.	TSO	PSO	RMO	Dynamic[1]
MCP	C/C++[2]	Model checking	✓				✓
Klee	LLVM IR	Coverage test gen.					
LLBMC	LLVM IR	Static analysis					
DiVinE	LLVM IR	Model checking	✓	✓			✓
JPF	Java	Model checking	✓[3]				✓
MoonWalker	.NET	Model checking	✓[4]				✓
Verisoft	C/C++	Model checking	✓				✓
CHESS	Binaries	Model checking	✓	✓[5]			✓
SLAM	C	CEGAR	✓[5]				✓
Saturn	C	Static analysis					✓[6]
Calysto	LLVM IR	Static analysis					✓[7]
CheckFence	C	Static analysis	✓	✓	✓	✓	✓
CBMC	C/C++	Bounded model ch.	✓	✓	✓	✓	✓
LLMC	LLVM IR	Model checking	✓	✓	✓	(LMO)	✓

2.4.2.10 Calysto

Calysto [BH08], an extended static checker, is inspired by Saturn and scales even better. Calysto has arguably better statistics to show. A notable difference between the two is that Saturn is only intraprocedurally path-sensitive, abstracting functions into small finite-state property automata. Calysto uses a fully formal analysis, but employs unsound approximations when dealing with loops, recursion and heap-allocated data. It uses LLVM as a front-end, allowing many languages to serve as input format. It is only targeted at sequential programs, however.

2.4.2.11 CheckFence

CheckFence [BAM07] is a SAT-based formal verification tool that analyzes C code implementing concurrent data types on multiprocessors with respect to a selected memory model. The implementation of a program is soundly verified or falsified by CheckFence for individual tests supplied by the user and covers all possible instruction interleavings and memory instruction reorderings. No annotations or formal specification is needed: CheckFence learns a specification directly from the C code A subset of C is supported, including conditionals, loops, pointers, arrays, structures, function calls, locks, and dynamic memory allocation. CheckFence lets the user specify the desired memory model. If a test fails, CheckFence provides an HTML-formatted counterexample trace that displays various views of the execution.

2.4.2.12 CBMC

CBMC [CKL04] is a Bounded Model Checker for ANSI-C and C++ programs. It checks for buffer overflows, pointer safety, exceptions and user-specified assertions. The verification is done by unwinding loops in a program and passing the resulting equation to a decision procedure. There is an extension to CBMC that adds support for SC, TSO, PSO, RMO and Alpha [AKT13].

2.4.3 Comparison

These tools all have their own merits and approaches, as listed in Table 2.9. Some approach finding bugs in programs by generating coverage tests, others by model checking. What makes our approach unique is that we model check generic concurrent LLVM IR that assumes only a relaxed memory model. By using this approach, we enable model checking of a wide range of languages in the presence of a relaxed memory model. This is by no means a complete list. Some other related tools are Bandera, Valgrind, Jinx Debugger, CMC, MaceMC and SMV.

[1] Dynamic features such as heap manipulations, recursion and loops.
[2] Using LLVM IR
[3] Using the Java Memory Model.
[4] Using the .NET Memory model.
[5] Using extensions.
[6] Only `load` and `store` operations.
[7] Unsound approximations.

3 LLMC Design

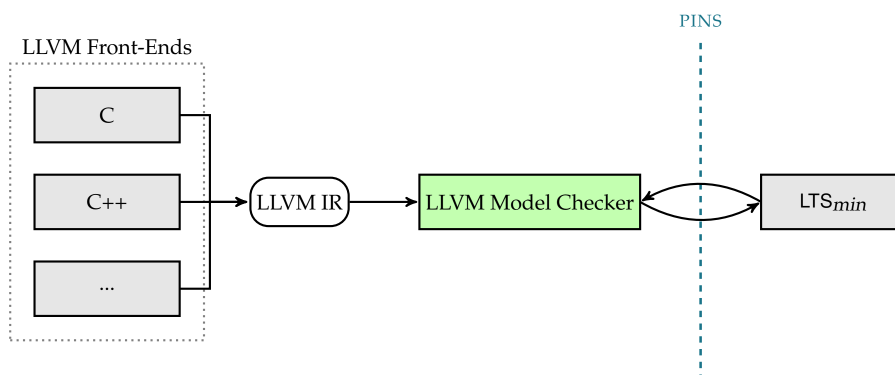
The rationale behind LLMC is that using LLVM IR as input specification, we cater for a lot of languages: all the languages that have a compiler to LLVM IR. Because we implement PINS, we get the benefit of the various back-end reachability tools of LTS_{min}. Combined this gives us a model checker for a large collection of languages. In addition to this, LLMC explores the state space of the LLVM IR program running as if it ran on a limited relaxed memory model. Figure 3.1 depicts where this would put the new LLVM IR Model Checker in relation to the other LLVM tools as well as its relation with LTS_{min}.

This chapter describes the design of our tool, LLMC. We will first describe our design choices in order to answer our research questions (Section 3.1) and define a formal model for our program (Section 3.2). We then describe how we used LLVM and LTS_{min} to reach our goal (Section 3.3). We conclude this chapter by describing an exploration strategy and comment on the soundness and completeness (Section 3.4).

3.1 Design choices

- *How can we model the execution of multi-threaded LLVM IR programs on a relaxed memory model?* To model relaxed memory and LLVM IR instructions in the presence of relaxed memory, we define a model of execution. The memory model of this execution model, LMO, governs what instructions can execute when and what instructions can be buffered and when. We discuss this in more detail in Section 3.2.
- *How can we explore the state space of generic LLVM IR programs?* The LLVM Project contains an interpreter that accepts LLVM IR. Modifying this interpreter with functionality to load and store the entire state and performing a single instruction (step) is a valid approach to explore the state space of an LLVM IR program. This avoids the need to recreate the semantics of

Figure 3.1 How LLMC fits in relation to other tools.



every instruction.
Because the LLVM Interpreter itself does not specify a threading model, nor does it natively support threading, we need to define a threading model in its presence. We opt to implement two threading models: 1) by catching `pthread` calls, we borrow the threading model of `pthreads` (`pthread`-mode); and 2) we can let the user define LLVM IR functions that will be started as if they were threads (`proc`-mode). In this second model, we can also define `init` and `fini` functions that run exclusive at respectively the start and the end of the execution.
We map the execution of LLVM IR to the PINS interface and thus gain a method of exploring the state space of LLVM IR programs. This is discussed in more detail in Section 3.3.

- *When is the multi-threaded program deemed correct and when is it deemed incorrect?* LLMC allows users to specify what are erroneous states using `assert(<condition>)` statements. LLMC intercepts these calls and if `condition` does not hold, we have found an erroneous state.
To determine under what memory model the erroneous state is reachable, we can make use of a search strategy that starts with sequential consistent and gradually relaxes memory operations. To determine what memory operation reorderings caused the erroneous state to be reached, we can look at the trace to the erroneous state: this precisely tells us the order of memory operations. We discuss this exploration strategy in more depth in Section 3.4.
- *How can we limit memory usage?* Limiting memory usage is done by saving copies of data only once and inserting them into a chunk table. This is discussed in more detail in Section 3.3.3.3.
- *How can we make our LLVM IR model checker as forward compatible with future LLVM IR versions as possible?* By reusing the LLVM Interpreter and making the required changes, newer revisions of the LLVM Interpreter and LLMC have a common ancestor. This allows for future merging of new instructions, providing our modifications are not too intrusive. By taking care to leave the LLVM Interpreter as intact as possible, sometimes at the cost of readability and common best practices, we make it easier to merge future LLVM Interpreter revisions and with it, newer LLVM IR versions.

3.2 The Execution Model

We base our definition of a program and program execution on the definitions of the memory model in Section 8.1 of [ABBM10]. That model allows the writes to be reordered after reads and reads to be reordered after any memory operation (`[w→r + r→r/w]`). Since that model is a memory model and does not define instruction that use loaded values, we need to extend it with registers and instructions using these registers. This extension allows us to map the register-based LLVM IR to the model. We will first give our definition (Sections 3.2.1 to 3.2.3) and then comment on the differences (Section 3.2.4).

3.2.1 Preliminaries

Let $[k], k \in \mathbb{N}, k \geq 1$ be the set of integers $\{1, ..., k\}$. Let $k \geq 1$ be an integer and E a set. Let $\mathbf{e} = (e_1, ...e_k) \in E^k$ be a k-dim vector over E. For every $i \in [k]$, we use $\mathbf{e}[i]$ to denote the i-th component of $\mathbf{e}$ (i.e., $\mathbf{e}[i] = e_i$). For every $j \in [k]$, and $e' \in E$, we denote by $\mathbf{e}[j \hookleftarrow e']$ the k-dim vector $\mathbf{e}'$ over E defined as follows: $\mathbf{e}'[j] = e'$ and $\forall l \in [k], l \neq j : \mathbf{e}'[l] = \mathbf{e}[l]$.

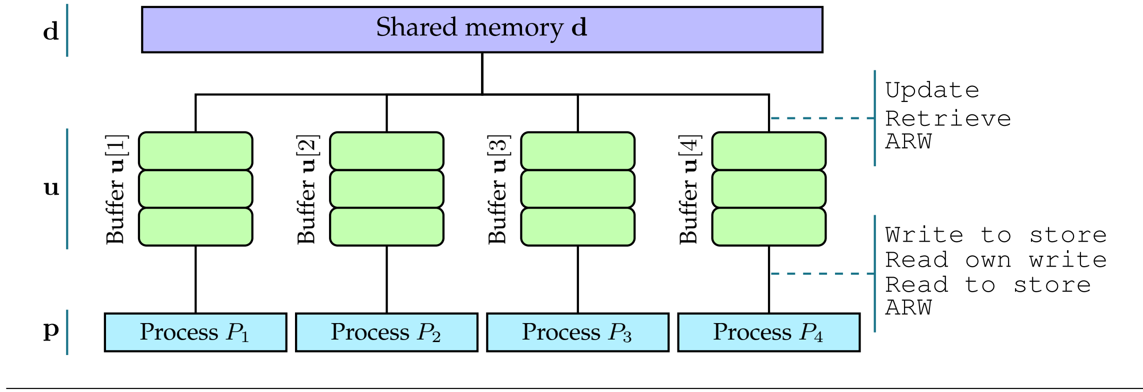
3.2.2 The Program

Let D be a finite data domain, $X = \{x_1, ..., x_m\}$ be a finite set of variables valued in D and $M = D^m$ be the set of all possible valuations of the variables in X. We will use this to model the shared memory. We define $R = D^*$ to describe the possible states of a register of a single process. We can use an unbounded number of registers, because LLVM IR allows for an unbounded number of registers.

For a given finite set of process identities I, let $\Omega(I, D, X)$ be the smallest set of operations which contains 1) the *no operation* `nop`; 2) the operations local to a process $\texttt{lop}(i, r_d, r_s)$; 3) the *read operations* $\texttt{r}(i, x, d, r_d)$; 4) the write operations $\texttt{w}(i, x, d)$; and 5) the *atomic read-write operations* $\texttt{arw}(i, x, d, d')$, where $i \in I$, $x \in X$, $d, d' \in D$, $r_d \in \mathbb{N}$, $r_s \in \mathbb{N}^*$.

We can then define a multi-threaded program over D and X by a tuple $\mathcal{N} = (\mathcal{P}_1, ..., \mathcal{P}_n)$, where for every $i \in [n]$, $\mathcal{P}_i = (P_i, \Delta_i)$ is a finite-state process where P_i is a finite set of control states and $\Delta_i \subseteq P_i \times \Omega(\{i\}, R, D, X) \times P_i$ is a finite set of label transition rules. For the sake of clarity,

Figure 3.2 A graphical depiction of what the variables in the definitions roughly mean.



we use $p \xrightarrow{\texttt{op}} p'$ instead of $(p, \texttt{op}, p') \in \Delta_i$, for any $p, p' \in P_i$ and $\texttt{op} \in \Omega(\{i\}, R, D, X)$. The exact implementation of the various operations in $\Omega(I, R, D, X)$ depends on the memory model we want to model.

3.2.3 The Execution of a Program

We assume the cache to be multi-copy atomic. This avoids the need to differentiate between what every process observes of the shared memory, since writes to a single location are serialized and all observers agree on the order of serialization. We also separate instructions that only affect the local stack from other instructions that affect the shared memory. This will give rise to possible optimizations later.

Let us now define the model of executing a multi-threaded program for a relaxed memory model, by defining the various operations in $\Omega(I, D, X)$ for LMO. We first associate a buffer to each process. This buffer is used to store both write and read operations performed by the process (`Write to Store`, `Read to store`). These buffered operations are then performed by nondeterministically choosing a process and nondeterministically choosing an *enabled* operation in the buffer of that process (`Update`).

A memory operation is *enabled* iff there are no preceding memory operations to the same location. This allows operations to overtake each other, but not allowing memory operations to the same location to overtake each other. This is the desired behaviour for our LMO definition, because we assume a coherent cache.

When a read operation to the variable x_j would be added to the buffer and there are writes to x_j in that buffer, the read will read the last of such write operation instead of being added to the buffer (`Read own write`). This allows read operations to overtake memory operations in the buffer. Atomic read-write operations can be executed when the buffer is empty (`ARW`) or when the buffer does not contain memory operations to dependent memory locations (`RelaxedARW`). Lastly, the memory barriers `SCFence`, `RelFence` and `AcqFence` can be executed when the buffer does not contain writes (`RelFence`) or reads (`AcqFence`) or both (`SCFence`).

A formal definition is as follows. Let $\mathbf{P} = P_1 \times \cdots \times P_n$ and for every $i \in [n]$, let $B_i = \{\texttt{w}, \texttt{r}\} \times \{i\} \times [m] \times D \times \mathbb{N}$ be the alphabet of the store buffer associated with $\mathcal{P}_i$. A *configuration* of $\mathcal{N}$ is a tuple $\langle \mathbf{p}, \mathbf{r}, \mathbf{d}, \mathbf{u} \rangle$, where $\mathbf{p} \in \mathbf{P}$ describes the state of the finite-state processes, $\mathbf{r} \in R^n$ describes the state of the registers, $\mathbf{d} \in M$ describes the shared memory and $\mathbf{u} \in B_1^* \times \cdots \times B_n^*$ is a valuation of the store buffers. Figure 3.2 graphically depicts the relations of these variables.

Let us now define the transition relation $\rightarrow_{\mathcal{N}}$ on configurations of $\mathcal{N}$, to be the smallest relation such that, for every $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$, for every $\mathbf{r}, \mathbf{r}' \in R^n$, for every $\mathbf{d}, \mathbf{d}' \in M$, and for every $\mathbf{u}, \mathbf{u}' \in B_1^* \times \cdots \times B_n^*$, we have $\langle \mathbf{p}, \mathbf{d}, \mathbf{u} \rangle \rightarrow_{\mathcal{N}} \langle \mathbf{p}', \mathbf{d}', \mathbf{u}' \rangle$ if there is an $i \in [n]$ and there are $p, p' \in P_i$, such that $\mathbf{p}[i] = p$, $\mathbf{p}' = \mathbf{p}[i \hookleftarrow p']$, and one of the following cases hold:

1. `nop`: $p \xrightarrow{\texttt{nop}}_i p'$, $\mathbf{r}' = \mathbf{r}$, and $\mathbf{d}' = \mathbf{d}$, and $\mathbf{u}' = \mathbf{u}$.
2. `lop`: $p \xrightarrow{\texttt{lop}(i, r_d, r_s)}_i p'$,
   (a) $\forall(\texttt{op}, i, k, d, r_o) \in \mathbf{u}[i] \bullet r_o \notin r_s$, and

(b) $\mathbf{r}' = \mathbf{r}[i \hookleftarrow \mathbf{r}[i][r_d \hookleftarrow v]], v \in X$, and $\mathbf{d}' = \mathbf{d}$, and $\mathbf{u}' = \mathbf{u}$.

3. `Write to store`: $p \xrightarrow{\mathtt{w}(i,x_j,d)}_i p'$, $\mathbf{r}' = \mathbf{r}$, and $\mathbf{d}' = \mathbf{d}$, and $\mathbf{u}' = \mathbf{u}[i \hookleftarrow (\mathtt{w}, i, j, d, 0)\mathbf{u}[i]]$.
4. `Read`: $p \xrightarrow{\mathtt{r}(i,x_j,d,r_d)}_i p'$, $\mathbf{d}' = \mathbf{d}$, and
   - `Read own write`: $\mathbf{u}' = \mathbf{u}$, $\mathbf{r}' = \mathbf{r}[i \hookleftarrow \mathbf{r}[i][r_d \hookleftarrow d]]$ if $\exists u_1, u_2 \in B_i^*$ such that 1) $\mathbf{u}[i] = u_1(w, i, j, d)u_2$, and 2) $\forall(\mathtt{op}, i, k, d', r_o) \in u_1 \bullet k \neq j$, or
   - `Read to store`: $\mathbf{u}' = \mathbf{u}[i \hookleftarrow (\mathbf{r}, i, j, d, r_d)\mathbf{u}[i]]$, $\mathbf{r}' = \mathbf{r}$ otherwise.
5. `Update`: $p' = p$ and $\exists j \in [m] \bullet \exists d \in D \bullet \exists u_1, u_2 \in B_i^*$ such that
   (a) $\mathbf{u}[i] = u_1(\mathtt{w}, i, j, d, 0)u_2$, and $\forall(\mathtt{op}, i, k, d', r_o) \in u_2 \bullet k \neq j$, and
   (b) $\mathbf{r}' = \mathbf{r}$, and
   (c) $\mathbf{d}' = \mathbf{d}[j \hookleftarrow d]$, and
   (d) $\mathbf{u}' = \mathbf{u}[i \hookleftarrow u_1u_2]$
6. `Retrieve`: $p' = p$, $\mathbf{d}' = \mathbf{d}$, and $\exists j \in [m] \bullet \exists d \in D \bullet \exists r_d \in \mathbb{N} \bullet \exists u_1, u_2 \in B_i^*$ such that
   (a) $\mathbf{u}[i] = u_1(\mathtt{r}, i, j, d, r_d)u_2$, and $\forall(\mathtt{op}, i, k, d', r_o) \in u_2 \bullet k \neq j$, and
   (b) $\mathbf{r}' = \mathbf{r}[i \hookleftarrow \mathbf{r}[i][r_d \hookleftarrow \mathbf{d}[j]]$, and
   (c) $\mathbf{u}' = \mathbf{u}[i \hookleftarrow u_1u_2]$
7. `ARW`: $p \xrightarrow{\mathtt{arw}(i,x_j,d,d')}_i p'$, and
   (a) $\mathbf{u}[i] = \varepsilon$, and
   (b) $\mathbf{d}[j] = d$, and $\mathbf{r}' = \mathbf{r}$, and $\mathbf{d}' = \mathbf{d}[j \hookleftarrow d']$, and $\mathbf{u}' = \mathbf{u}$.
8. `RelaxedARW`: $p \xrightarrow{\mathtt{arw}(i,x_j,d,d')}_i p'$, and
   (a) $\forall(\mathtt{op}, i, k, d', r_o) \in \mathbf{u}[i] \bullet k \neq j$, and
   (b) $\mathbf{d}[j] = d$, and $\mathbf{r}' = \mathbf{r}$, and $\mathbf{d}' = \mathbf{d}[j \hookleftarrow d']$, and $\mathbf{u}' = \mathbf{u}$.
9. `SCFence`: $p \xrightarrow{\mathtt{scfence}}_i p'$, and $\mathbf{u}[i] = \varepsilon$, and $\mathbf{r}' = \mathbf{r}$, and $\mathbf{d}' = \mathbf{d}$, and $\mathbf{u}' = \mathbf{u}$.
10. `RelFence`: $p \xrightarrow{\mathtt{relfence}}_i p'$, $\forall(\mathtt{op}, i, k, d, r_o) \in \mathbf{u}[i] \bullet \mathtt{op} \neq \mathtt{w}$, $\mathbf{r}' = \mathbf{r}$, and $\mathbf{d}' = \mathbf{d}$, and $\mathbf{u}' = \mathbf{u}$.
11. `AcqFence`: $p \xrightarrow{\mathtt{acqfence}}_i p'$, $\forall(\mathtt{op}, i, k, d, r_o) \in \mathbf{u}[i] \bullet \mathtt{op} \neq \mathtt{r}$, $\mathbf{r}' = \mathbf{r}$, and $\mathbf{d}' = \mathbf{d}$, and $\mathbf{u}' = \mathbf{u}$.

3.2.4 Differences

The LMO memory model of our execution model differs from the memory model in Section 8.1 of [ABBM10]. There are two key differences.

Firstly, we model registers in our execution model and define the instruction in their presence, including the read and write operations. Then, we model instructions that use these registers, for example for arithmetic or branching. An instruction can only be executed if all the load operations on which it depends are executed. This means that in our memory model, we do not model all `[r→r/w]` behaviour: memory operations after the instruction are never reordered before the instruction nor before the load operations on which that instruction depends. This is a limitation we leave for future work to solve; in Section 6.3 we suggest a solution.

Secondly, they add the `[w→w]` relaxation later in Section 8.2 by having a buffer per memory location per process. However, we achieve the `[w→w]` relaxation by using a single buffer per process, but allowing memory operations to different locations to overtake each other. These approaches have the same effect: if in their model of Section 8.2 a memory operation would be enabled, it is at the front of the FIFO queue associated with the targeted memory location, meaning there are no preceding memory operations to that memory location. In that situation, the memory operation would also be enabled in our model. Vice versa, if in our model a memory operation is enabled, it means there are no preceding memory operations to that memory location, thus it is enabled in their model as well. In both models, memory operations can overtake memory operations to different memory locations.

Furthermore, we added some instructions. One of the instruction we added is `lop`, a local operation that only reads registers and writes to a single register. Local operations can only be performed if the buffer contains no stored read operations to any of the registers the local operation reads from. We can use this later to fit local LLVM IR instruction into this model. We added

`RelaxedARW` to allow memory operations to overtake atomic memory operations. We also added `SCFence`, `RelFence` and `AcqFence`, to model memory barriers according to the release-acquire memory model of LLVM.

3.2.5 Example

Let us consider the simple example in Figure 2.2, repeated in Figure 3.3. The program has two shared variables, X and Y; two processes, P_1 and P_2, with each their own register, R_1 and R_2. The entire state space of this 4-line example is shown in Figures 3.4 and 3.5.

Figure 3.3 Reordering on X86, is $\texttt{R1} = 1 \vee \texttt{R2} = 1$ guaranteed?

P_1	P_2
`X ← 1` `R1 ← Y`	`Y ← 1` `R2 ← X`

Figure 3.4 shows the state space when running assuming a sequentially consistent memory model and Figure 3.5 when assuming a relaxed memory model.

The states are described by $XYR_1R_2|B_1|B_2$, where B_1 and B_2 are the buffer of the two processes. The buffers can be empty or contain X or Y, meaning a write of 1 to X or Y respectively. Notice that states that are reachable without memory instruction reordering are marked as □; states that are only reachable with memory instruction reordering as ▢; and paths that end up in an unexpected state as ▬. This example illustrates the significance of the state space explosion. Even this example containing only 4 instructions produces a state space of 34 states. Out of these 34 states, 21 are only reachable when allowing memory instruction reordering.

3.2.5.1 Right or Wrong?

As illustrated in Figure 3.5, there are multiple execution traces of multi-threaded programs. Some of these traces are only possible when memory instruction reordering is allowed. It depends on the desired semantics of the program whether memory instruction reordering causes incorrect behaviour. In the example, we determined that ending with $R_1 = 0$ and $R_2 = 0$ is incorrect, but only because we naturally assume sequential consistency. If the program does not rely on $R_1 \vee R_2$ being true at the end, there is no incorrect behaviour. This is why we leave this up to the user to specify erroneous behaviour using `assert()` statements in the code.

Figure 3.4 The state space of the program shown in Figure 3.3 running on a sequentially consistent memory model

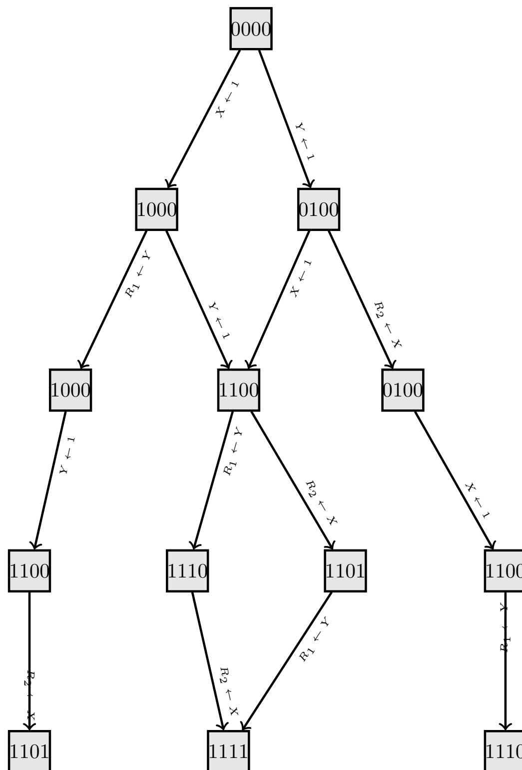
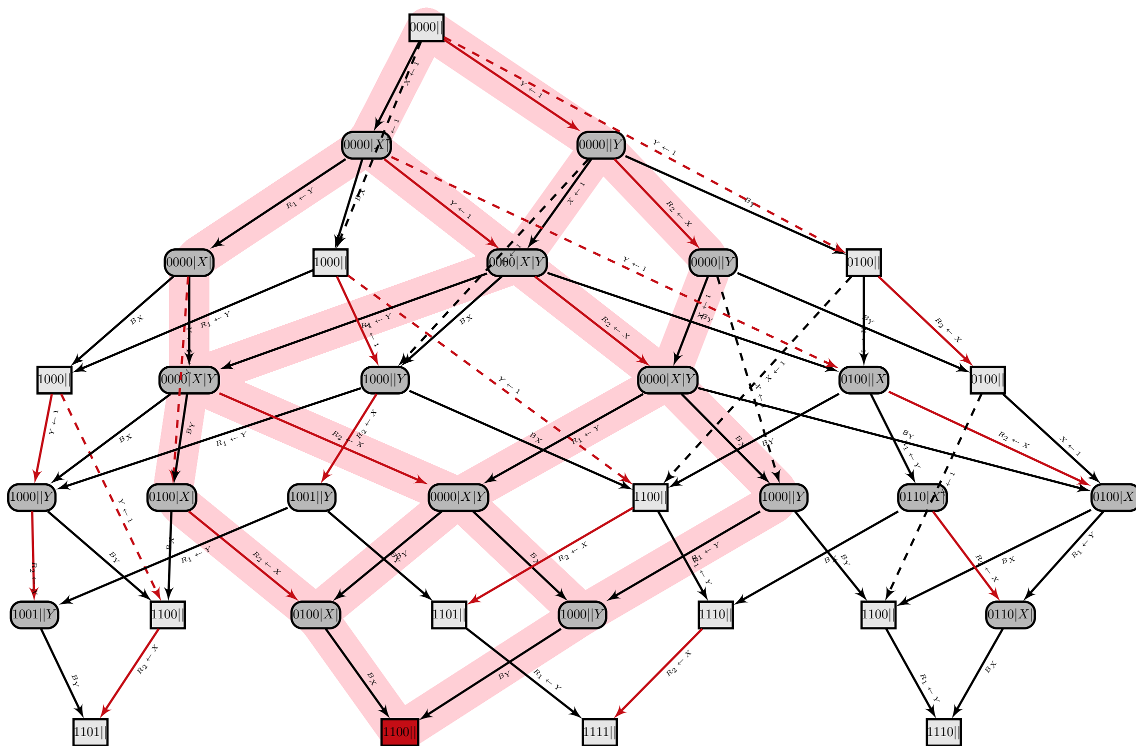


Figure 3.5 The state space of the program shown in Figure 3.3 running on a relaxed memory model



3 Mapping LLVM IR and LTSmin

In this section we will describe how we mapped LLVM IR to the model described in Section 3.1 and to the fixed-size state vector LTS*min* can understand.

The model in Section 3.1 describes a single state by $\langle \mathbf{p}, \mathbf{r}, \mathbf{d}, \mathbf{u} \rangle$ and a transition relation by $\langle \mathbf{p}, \mathbf{r}, \mathbf{d}, \mathbf{u} \rangle \rightarrow_{\mathcal{N}} \langle \mathbf{p}', \mathbf{r}', \mathbf{d}', \mathbf{u}' \rangle$, where $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$, $\mathbf{r}, \mathbf{r}' \in R^n$, $\mathbf{d}, \mathbf{d}' \in M$ and $\mathbf{u}, \mathbf{u}' \in B_1^* \times \cdots \times B_n^*$. We can map these to a transition system $\langle S_{\text{LLVM}}, \rightarrow_{\text{LLVM}}, s^0_{\text{LLVM}} \rangle$ as follows:

- $S_{\text{LLVM}} = \mathbf{P} \times \mathbf{M} \times (B_1^* \times \cdots \times B_n^*)$ (Section 3.3.1);
- $s^0_{\text{LLVM}} = \langle \mathbf{p}, \mathbf{r}, \mathbf{d}, \mathbf{u} \rangle$, where $\mathbf{p}$ are the finite-state process described by LLVM IR, $\mathbf{r}$ contains only empty LLVM IR registers, $\mathbf{d}$ contains LLVM IR global and local variables, initialized to a certain value, $\forall i \in [n] \bullet \mathbf{u}[i] = \varepsilon$ (Section 3.3.2);
- $s \rightarrow_{\text{LLVM}} s' \Leftrightarrow \langle \mathbf{p}, \mathbf{d}, \mathbf{u} \rangle \rightarrow_{\mathcal{N}} \langle \mathbf{p}', \mathbf{d}', \mathbf{u}' \rangle$, where $s = \langle \mathbf{p}, \mathbf{d}, \mathbf{u} \rangle$ and $s' = \langle \mathbf{p}', \mathbf{d}', \mathbf{u}' \rangle$. (Section 3.3.3)

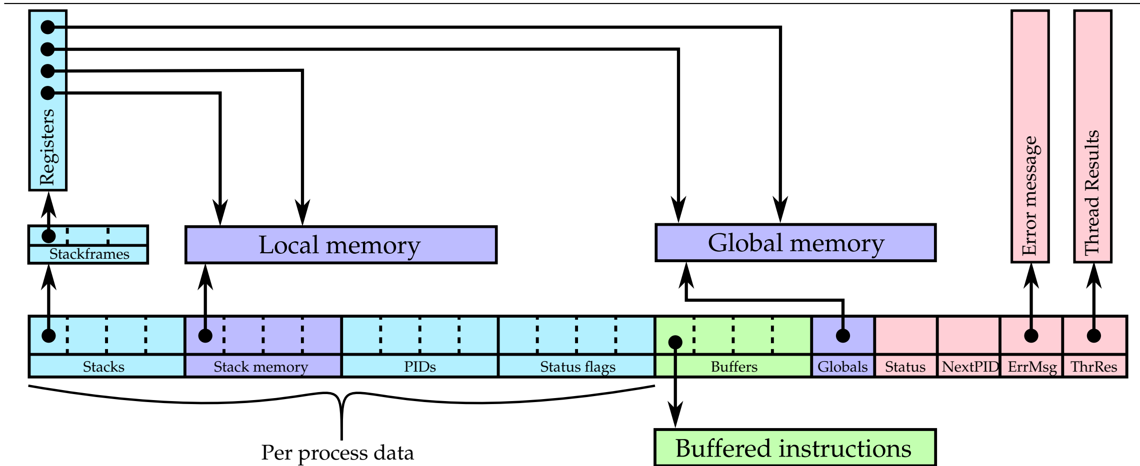
3.3.1 Mapping the state

Figure 3.6 illustrates how the state of an LLVM Interpreter is serialized to a state vector. LTS*min* describes a state using a *fixed-size* state vector. This is a problem for us if we want to support stacks, stack frames, registers and buffers of arbitrary size. To mitigate this, we used the chunk mapping functionality of LTS*min*. The downside to this is that LTS*min* has no knowledge of what is inside these chunks: they are just binary blobs to LTS*min*. This means we cannot use register values, stack memory values or buffers in LTL formulae. The only state vector variables we can meaningfully use for LTL are `PIDs`, `Status flags`, `Status` and `NextPID`. The others are all chunk map identifiers.

Each process has a stack, stack memory, a process identifier (PID) and status flags. A stack is a chunk map comprised of an arbitrary number of stack frames. A single stack frame contains the registers of the LLVM Interpreter, including the *instruction pointer* as well as some auxiliary informations such as the size of stack memory allocated. The stack memory of a process itself is stored separately. This allows for a more concise dependency matrix: local LLVM instructions that only use the registers, e.g. `bitcast`, `br`, only need a dependency on the stack itself. It also uses less duplicate memory, because local changes are in separate chunk maps from the global chunk map. Thus, a change to the stack memory does not yield a change in the global memory, thus avoiding the need to upload the entire global memory chunk map with only a minor change. A process identifier is a unique identifier during the lifetime of the LLVM program: process identifiers are not reused. The status flags of a process contain various flags, which will be explained in more detail in Section 4.1.4.5.

Notice that the depicted fixed-size state vector can only support up to four concurrently running processes. However, process slots are reused, so there is no bound on the total number of executed processes during the lifetime of the LLVM program.

Figure 3.6 The state vector containing a serialized LLVM Interpreter instance.



Given the model described in Section 3.1, the LLVM Interpreter state can be mapped to $\langle \mathbf{p}, \mathbf{r}, \mathbf{d}, \mathbf{u} \rangle$ as follows:

1. LLVM IR registers are in $\mathbf{r}$
2. LLVM IR global variables and local variables are in $\mathbf{d}$
3. The buffered LLVM IR instructions are in $\mathbf{u}$.
4. LLVM IR code is not in the state and is assumed to be immutable, so $\mathbf{p}$ is outside of the state.

3.3.2 Initial state

To start exploration of the state space, we need an initial state. We obtain the initial state by first allocating memory for all the global variables and initializing them. If present, constructors for global variables are called. This is needed to support for example global and static class variables in C++. The global constructors are assumed to be sequential: no concurrency is allowed in them. If an illegal action or an assert is triggered, the initial state will be erroneous. The global variables are serialized and put into the initial state vector. How we serialize the LLVM Interpreter we will describe in more detail in Section 3.3.3.3.

Then, we setup the initial process or processes, depending on the mode.

3.3.2.1 `pthread`-mode

In `pthread`-mode, LLVM IR is interpreted as one would expect: the function `main()` is called at the start and new threads can be created and joined using `pthread` calls. A call to the function `pthread_create(`t`,`f`)` starts a new thread that starts its execution at the function f and puts a thread identifier in t. Using this identifier, we can wait for a thread to finish execution using `pthread_join(`t`,`r`)`. When that call returns, r will have the return value of f. In Section 3.3.4 we provide detail on how this is achieved.

An example of this mode is shown in Appendix B.1.

This `main`-process is the first process, thus we give it the PID 1. We serialize the state of the LLVM Interpreter and use the first process slot of the state vector to describe this state. This gives us the state of an LLVM Interpreter, about to execute `main()`.

3.3.2.2 `proc`-mode

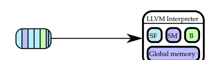
In `proc`-mode, LLVM IR is interpreted slightly differently. This mode is used when no `main()` is found in the LLVM IR. Instead, some functions are started as if they were threads. Firstly, if it exists, the function `llmc_init()` is called and executed, which can initialize global variables to specific values. Then, LLMC looks for functions starting with `llmc_proc` and initializes a stack for each of them and starts them as if it were threads. Their stacks are saved in the process slots of the initial state vector. Thus the initial state is the combined start states of all these functions running concurrently with global values initialized according to `llmc_init()`.

3.3.3 Next-state

After having uploaded the first state, LTS$_{min}$ will explore state state space by requesting new transitions and new states from LLMC. The general flow of obtaining a next state is illustrated in Figure 3.7.

LTS$_{min}$ passes a state vector to LLMC and requests to generate the next states (PINS:`getNextStates(s)`) and report them. First, LLMC re-instantiates the LLVM Interpreter and buffers from the passed state vector. Then, multiple possible next states are calculated, one for each choice of possible transitions. Lastly, the new state of the LLVM Interpreter and buffers is serialized and uploaded to LTS$_{min}$.

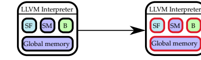
3.3.3.1 Instantiation



When we receive a state vector from LTS$_{min}$, we need to instantiate a valid LLVM Interpreter from this. We read the stack, stack memory and stack frames chunk identifiers from the state vector, download the associated chunks and recreate the stack. The global memory is one big chunk, so

is easily recreated. The buffer containing a number of buffered instruction is recreated as well. All re-instantiations are on-demand: not all transition groups need everything to be recreated. The flushing of the buffer does not need the entire LLVM Interpreter to be recreated for example, just the global memory part.

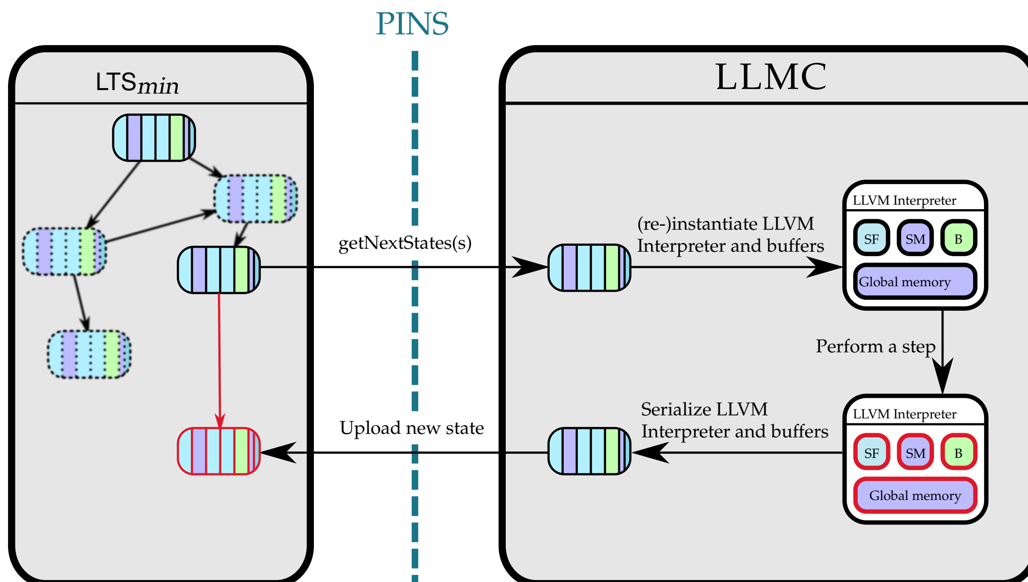
3.3.3.2 Performing a step



Having re-instantiated the LLVM Interpreter and buffers, we can perform a step. There are multiple possible steps from a single state, for example a process can perform a step or the buffer can flush an enabled item. LTS$_{min}$ uses a *partitioned* next state interface, which means we need to partition $\rightarrow_{\text{LLVM}}$ into transition groups. To optimize the state space exploration and locality, we partition the next-state into the following six transition groups:

- **Local**
  - Transitions: `lop`, `nop`
  - Alterations: Local transitions read and write only to registers and stack memory of the executing process.
  - A running process can execute any number of consecutive local transition, even after a single non-local transition. These transitions are usually squashed into previous transitions and thus this transition group is never taken in practice. A local transition reads the instruction to be executed, determines that it only affects the local state and executes it. Local instructions are only enabled when there are no dependent operations in the buffer, as per the `lop` in the model.
- **ToStore**
  - Transitions: `Write to store`, `Read to store`
  - Alterations: The global read and write operations are added to the buffer.
  - A running process can put memory operations into the buffer, waiting to be executed later by the transition group **FromStore**. Instructions can be added to the buffer whether they have dependent operations in the buffer or not. Both local and global memory operations may be added. This way, we can support reordering independent memory operations across memory operations that are dependent. So for example, `R2 ← P; R2 ← *R2; R1 ← X` can be reorded to `R1 ← X; R2 ← P; R2 ← *R2`.
- **FromStore**
  - Transitions: `Update`, `Retrieve`
  - Alterations: Operations in the buffer are executed.
  - A non-empty buffer can perform a single step per enabled item by flushing that enabled item from the buffer. Both local and global instructions can be the buffer.

Figure 3.7 The state vector containing a serialized LLVM Interpreter instance.



- **Global**
  - Transitions: `Read own write`, `ARW`, `SCFence`, `RelFence`, `AcqFence`
  - Alterations: All remaining global operations.
  - A running process can perform a single globally visible step. Local instructions of that process (`loc)`) may be executed before and after this global step and the effects squashed into a single transition. A running process can perform an instruction involving a form of synchronization in this transition group.
- **ThreadManagement**
  - LLVM `Call` instructions to `pthread` functions and to `llmc_atomic_*()` functions.
  - This is a special transition group for the management of threads (processes). This transition group handles calls to the `pthread` library. It atomically performs a `pthread_create()` or `pthread_join()`. This is only used in `pthread`-mode. As per the `pthread` specification[1], these calls synchronize memory with respect to other threads. This is achieved by only performing this call when all the buffers are empty.
- **Fini**
  - When all processes are done, a final `fini()` transition is performed to signify correct execution of the program. In `proc`-mode and if the LLVM program has a function called `llmc_fini()`, this sets up a call to `llmc_fini()`, which can be used for final assertions on the values of global variables for example.

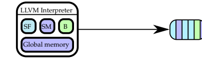
When mapping LLVM IR instruction to this transition relation, we differentiated between memory operations to the stack of a process and the global memory. Memory operations to a process' own stack are local and thus do not change the reachability of an erroneous state. This allows us to optimize these out. We can determine local memory operations by the address they are writing to or reading from.

We mapped the LLVM IR instructions to $\rightarrow_{\text{LLVM}}$ as follows.

- `lop`
  - Every terminator instruction, every binary instruction, `getelementptr` and all memory operations to stack memory of that process.
- `Write to store`
  - All `store` instructions to global memory.
- `Read own write`
  - All `load` instructions to global memory.
- `Read to store`
  - All `load` instructions to global memory.
- `ARW/RelaxedARW`
  - All `atomicrmw` and `cmpxchg` instructions to global memory, depending on the memory barrier specified as argument: `relaxed` will map to `RelaxedARW` and the others will map to `ARW`.
- `SCFence`
  - `fence seq_cst` and `fence acq_rel` instructions.
- `RelFence`
  - `fence release` instruction.
- `AcqFence`
  - `fence acquire` instruction.

[1] `pthread` specification 4.11, Memory Synchronization at
http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_11

3.3.3.3 Serialization



To serialize the state of an LLVM Interpreter instance, we must store the value of every relevant data object in the LLVM Interpreter. This includes the stack, the memory allocated on the stack and the global variables. The stack is stored per stack frame. Every individual stack frame is uploaded to LTS$_{min}$ as a chunk map. Then, the combined chunk identifiers from the stack frame chunk maps are put into an array and uploaded to LTS$_{min}$ as a chunk map as well. This chunk map represents the stack of that state of the LLVM Interpreter. This avoids duplicating every stack frame for every process and every state. The reason for this is that in the most cases the stack frames of previous stacks are not altered. Thus instead of uploading an entire stack, we upload the changed stack frames and the new array of stack frame chunk identifiers, representing the stack.

This scheme is not used for the global variables: these are just monolithically uploaded as one big chunk to LTS$_{min}$.

After uploading these various chunk maps, the chunk identifiers are stored in the state vector. Then, we pass the state vector to LTS$_{min}$ as a new state or as the initial state.

3.3.4 Thread Management

In `pthread`-mode, the creation and exiting of threads need to be managed. When a process calls `pthread_create(`t`,` f`)`, it looks for an empty process slot in the state vector. If there is one, it will initialize an LLVM Interpreter and setup a call to the function f. This state is then serialized, uploaded to LTS$_{min}$ and the associated identifier put into the process slot, along with proper status flags, empty stack memory, empty registers and an empty buffer. The PID of the process is the value of `NextPID` and then `NextPID` is incremented. If there is no empty process slot, `ErrMsg` will be set to an appropriate error message, the error status flag is set and exploration continues.

When a process finishes execution, it will store the return value of its 'main'-function (the f of the `pthread_create(`t`,` f`)` call that started it) in the thread result chunk (`ThrRes`). This is done such that it can later be retrieved using the process identifier (t).

When a process calls `pthread_join(`t`,` r`)`, it is determined if the process identified by t has completed execution. This is the case when the stack and buffer of that process are both empty. It is also the case when that process has no process slot anymore, because it has finished execution and all memory operations earlier. If the process has finished, its return value is obtained from the thread result chunk and assigned to r.

3.3.5 Dependency Matrix

The transition groups described in Section 3.3.3 do not all use every state vector variable. In Table 3.1 the associated dependency matrix is shown. All the transition groups are per process, except the **Fini** transition. The global status is read from and written to by all transition groups to ascertain the current status and possible alter it. The status flags per process can be read from and written to by every transition group as well for the same reason. The PID is always read from, with the primary goal creating an edge label that describes the performed step. The PID is used in this description to denote which process performed this step.

Because the **Local** transition only performs instructions local to that process, it only has a dependency on the data of that process. The **ToStore** transition group writes an instruction to the instruction buffer and obtains this information from the stack and stack memory. The **FromStore** flushes any one enabled item in the buffer of that process to stack memory or the global memory. The **Global** transition group is used for any other instruction that modifies global memory. Sometimes whether a global instruction can be executed depends on the buffer, so we add a read dependency. The **ThreadManagement** transition group may read and modify any one process with the exception of its buffer. The buffer is only used to check if the step can be performed. The **Fini** transition group is only enabled when no threads are running and all buffers are empty. Because it resets the stack, stack memory and PID of all processes, it also needs write permission to write to those variables.

Table 3.1 Dependency matrix

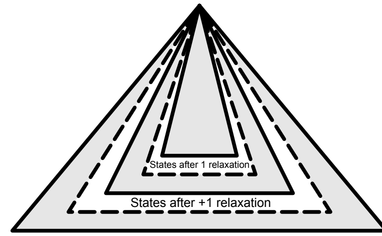
	State vector																								
Transition Groups	SFrame				SMem				PIDs				Flags				Buffers				G	S	N	E	T
Local (per Proc)	+				+				r				+	r	r	r	r					+		w	+
ToStore (per Proc)	+				r				r				+	r	r	r	+				r	+		w	+
FromStore (per Proc)	+				+				r				+				+				+	+		w	
Global (per Proc)	+				+	+	+	+	r				+	r	r	r	r	r	r	r	+	+		w	+
ThreadMananagment (per Proc)	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	r	r	r	r	+	+	+	w	+
Fini	w	w	w	w					+	+	+	+	+	+	+	+	r	r	r	r	r	+		w	

`r`: read, `w`: write, `+`: read/write

3.4 Exploration strategy

To ascertain under what memory model an assert can be triggered, we describe a specific exploration strategy. This strategy first explores the state space of a program without any memory instructions reordered (SC). Then, it will allow the reordering of `[w→w]` (TSO). Following, the `[w→r]` relaxation is allowed (PSO) and finally, we allow all the relaxations LMO allows. This exploration strategy allows us to model check for multiple memory models in one run.

Figure 3.8 An illustration of the minimal-cost search strategy.



We define this exploration strategy by first giving every transition a *cost*. The cost of a state s is determined by adding the cost of all the transitions on the trace from the initial state to s. During exploration, we will explore the next-states of the state that has the lowest such cost. Thus, the cost of the transitions will determine the order in which states are explored.

This cost is a tuple $\langle \mathtt{r}, \mathtt{ww}, \mathtt{wr} \rangle$, specifying the cost of reordering write after read (`wr`), write after write (`ww`) and read after read/write (`r`). The transitions will be given a cost as specified in Table 3.2. To determine the lowest cost, we use a lexicographical ordering. If we are only interested in a certain memory model, we can avoid exploring states that have a certain cost.

3.4.1 Soundness and completeness

We are complete with respect to LMO. But in this model, we do not allow certain `[r→r/w]` RMO memory instruction reordering scenario to occur: memory operations after an instruction that depends on a previous `load` are not allowed to be reordered with that `load`. The cause is that a local instruction is only executed when all instructions on which it depends are executed. An illustration of this is shown in Figure 3.9: under RMO, the load of `Y` would be allowed to be reordered before the other two instructions. This makes our model not complete with respect to RMO. In Section 6.3.1 we outline a possible solution.

Figure 3.9 Not all RMO allowed reorderings are modeled.

Program order	RMO allowed
R1 ← X R1 ← R1 + 1 R2 ← Y	R2 ← Y R1 ← X R1 ← R1 + 1

We also assume the modeled cache to be coherent and causal. This allows us to use a single representation of the global memory in the state vector, but this limits the completeness of our model. We make this assumption for the simple reason that it limits the state space. It is our assessment that the reduction of the state space this brings forth outweighs covering non-causal behaviour: modeling delayed propagation to a sizable number of threads causes yet another exponential blow-up of the state space. Moreover, we are not aware of software exploiting non-causal cache nor of any programming idiom that relies on causality without also relying on a more strict memory model.

Our model is sound: if we find an error and generate a trace to this error, then this means that the program has a bug. Thus, under certain conditions, running on certain hardware with a specific memory model and using certain memory instruction reorderings, the program can exhibit undesired behaviour. The exploration strategy incrementally relaxes the memory model, from TSO to PSO to LMO. Thus, should we only be interested in TSO, we can stop the exploration when all TSO behaviour is explored and ignore possible future errors on more relaxed memory models. Using this, we can provide a sound answer per model.

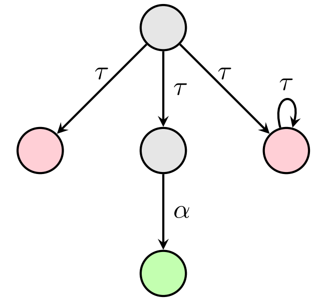
Table 3.2 Costs ($\langle \mathtt{r}, \mathtt{ww}, \mathtt{wr} \rangle$) of transitions

Transition	r	ww	wr
`Write/Read`	0	0	0
`Write/Read [w→r]`	0	0	1
`Write/Read [w→w]`	0	1	0
`Write/Read [r→r/w]`	1	0	0

3.4.2 Deadlock and livelock detection

Detecting a deadlock in the state space is a relatively trivial task: we simple check for states that have no outgoing edges and do not have a status flag indicating proper completion. This will also detect erroneous states, because erroneous states do not have outgoing transitions.

Figure 3.10 Example of a reduced state space



Detecting a livelock can be done by testing for *divergence*. Executions that properly finish are said to *converge*. Infinite executions and executions that end up in an erroneous state are said to *diverge*. LTS_{min} allows us to test for this. We mark every transition except the `Fini` transition as a τ step and then reduce the state space using divergence sensitive weak branching bisimulation. The result is a reduced state space with only a few states. If the original state space has a trace to proper completion, there will be a α or $\tau\alpha$ trace, where α is a proper final `Fini` transition. If the original state space has a deadlock, there will be a τ trace. If the original state space has a livelock, there will be a τ^ω trace.

4 LLMC Implementation

4.1 Implementational Details

In this section we list some details of our implementation

4.1.1 Pointers

We support LLVM pointer types. We achieve this by obtaining a very large slab of virtual memory at the start of LLMC. In this slab, when needed, we allocate memory for the stack and global memory. When we re-instantiate the LLVM Interpreter, we copy the stack and global memory from the state vector to the same place in the slab of virtual memory. Thus, pointer to any of these memory location remain valid.

In the implementation, we use mmap to obtain this virtual memory. To support the multi-core back-ends of LTS_{min}, one process `mmaps` private anonymous memory at some memory location. Then, the other processes try to `mmap` private anonymous memory to the same fixed memory location. This only works for process, not for threads.

4.1.2 Bounded buffer

The execution model supports an unbounded number of instructions in the buffer, but this is not feasible. Therefore, our implementation allows to select the maximum number of buffered instructions per buffer.

4.1.3 Exploration

The current implementation of the exploration strategy does not reflect the design described in Section 3.4. The current implementation of this strategy can only differentiate between SC and a more relaxed memory model: first SC behaviour is explored and then all behaviour that LMO allows is explored.

4.1.4 Features

Here, we list some features of LLMC.

4.1.4.1 Atomic instructions

We provide a few instructions that are executed atomically. Optionally they carry out a memory barrier as well, depending on the argument passed, in accordance with the LLVM memory model. These can be used to implement `std::atomic`. We provide a small version of `std::atomic` as `llmc::atomic` in Appendix C.2.

- `bool __atomic_compare_exchange(size_t bytes, void* target, void* expected, void* desired, std::memory_order model_success, std::memory_order model_failure)` This atomically compares `*target` and `*expected`. If equal (success), `*desired` is assigned to `*target` and `true` is returned. Else (failure), `*target` is assigned to `*expected` and `false` is returned. On success, `model_success` will be used to determine the carried out barrier. For example, if `seq_cst` is used, the function will only be executed when the buffer is empty. On failure, `model_failure` will be used in the same manner. This function is added in addition to the `cmpxchg` instruction to support C++11 atomics.
- `void __atomic_load(size_t bytes, void* source, void* destination, std::memory_order model)` This atomically performs a copy operation from `*source` to `*destination`. The argument passed to `model` determines the carried out barrier.
- `void __atomic_store(size_t bytes, void* source, void* destination, std::memory_order model)` This atomically performs a copy operation from `*source` to `*destination`. The argument passed to `model` determines the carried out barrier.

4.1.4.2 Memory barriers

We support a number of explicit memory barriers as a function call in addition to the LLVM IR `fence` instruction. This is to support LLVM IR generated by clang when compiling C and C++. These can be used by calling llmc_barrier_x.

- `llmc_barrier_acquire` corresponds to `fence acquire`: the call can be executed when there are no `load` instructions in the buffer.
- `llmc_barrier_release` corresponds to `fence release`: the call can be executed when there are no `store` instructions in the buffer.
- `llmc_barrier_seq_cst` corresponds to `fence acquire`: the call can be executed when there are no `load` or `store` instructions in the buffer.

4.1.4.3 Additional memory modification

We provide implementations for commonly used memory modification routines. These calls are only executed if there are no instruction in the buffer that read from or write to any of the memory locations in (`source`,`source+sizeInBytes`) or (`dest`,`dest+sizeInBytes`). Note that `memcmp` and `memset` are implemented as if they were atomic.

- `memcpy(void* dest, void* source, size_t sizeInBytes)` performs a copy from `*source` to `*dest` of `sizeInBytes` bytes.
- `memcmp(void* one, void* other, size_t sizeInBytes)` compares `sizeInBytes` bytes of `*one` and `*other`.
- `memset(void* dest, int val, size_t sizeInBytes)` sets `sizeInBytes` bytes of `*dest` to `val`.

4.1.4.4 C/C++ specific additions

We added support for assert(<condition>) by catching the call to `void __assert_fail()`. This is because `assert()` is a macro that is expanded to code similar to:
if(!<condition>) __assert_fail(...).

To add static constructor support for classes in C++, before setting up the `main()` call when determining the initial state, we perform a call to `_GLOBAL__I_a()`. This adds support to LLVM IR generated from C++ using clang.

4.1.4.5 Exploration guidance flags

In the state vector, each process has `status flags`. These flags contain various bits of information that optimize or guide exploration:

LOCAL If this flag is set, the next instruction is a local transition. This is an optimization flag

to avoid having to re-initialize the LLVM Interpreter to determine whether the next instruction is local. Instead, this flag is set when a state is reported to LTS_{min}.

ERROR If this flag is set, the process reached an erroneous state. No process will execute instructions from this point.

THREADCRT If this flag is set on a process, it means its previous and next instruction are calls to `pthread_create()`. Optionally (Section 4.1.4.6) this means that other processes are not allowed to report a transition, thus culling a part of the state space that a user might deem not interesting. By default this is not used to cull the state space.

DONE If this flag is set, the process has completed execution. Its buffer might still contain instructions.

ATOMIC If this flag is set on a process, no other process may generate a transition. This is used for `llmc_atomic_begin()` and `llmc_atomic_end()`. However, this is an experimental feature in the current implementation of LLMC.

4.1.4.6 Settings file

LLMC supports a few optional features that can be turned on or off in a settings file.

optimize_local_steps This boolean setting controls whether or not local transitions are squashed into a previous transition. By default this is on.

all_tau_except_last This boolean setting controls whether to replace all actions in the generated state space with τ except the `Fini` transition. This is used to test for deadlock and livelocks (Section 3.4.2). By default this is off.

buffer_size This integer setting controls the maximum size of the buffer, i.e. how many instructions the buffer of each process can hold. By default this is 5.

combine_pthread_create This boolean setting controls whether consecutive calls to the function `pthread_create()` are grouped together in order to cull the state space. By default this is off.

5 Results

In this chapter, we first provide a validation of LLMC (Section 5.1), then list multiple experiments that provide further validation (Section 5.2) and finally show some benchmarks of these experiments (Section 5.3).

5.1 Validation

We validated LLMC by implementing various litmus tests [MSS12] in LLVM and determining whether the tool found the missing memory barriers. We used the RMO memory model for these tests to show the similarities and the differences between RMO and LMO. The implementation of these litmus tests can be found in Appendix B. The results of these validations are in Tables 5.1 to 5.3 and 5.5. There is no table for the Dependent Load Litmus Test because memory barriers do not affect the outcome, like expected.

Table 5.1 Store Buffer Litmus Test results for LLMC

SB	t1_fence			
t0_fence	`relaxed`	`acquire`	`release`	`seq_cst`
`relaxed`	✗	✗	✗	✗
`acquire`	✗	✗	✗	✗
`release`	✗	✗	✓	✓
`seq_cst`	✗	✗	✓	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

Table 5.2 Load Buffer Litmus Test results for LLMC

LB	t1_fence			
t0_fence	`relaxed`	`acquire`	`release`	`seq_cst`
`relaxed`	✗	✗	✗	✗
`acquire`	✗	✓	✗	✓
`release`	✗	✗	✗	✗
`seq_cst`	✗	✓	✗	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

Table 5.3 Independent Reads of Independent Writes Litmus Test results for LLMC

IRIW	t2_fence			
t0_fence	relaxed	acquire	release	seq_cst
relaxed	✗	✗	✗	✗
acquire	✗	✓	✗	✓
release	✗	✗	✗	✗
seq_cst	✗	✓	✗	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

Table 5.4 Independent Reads of Independent Writes with Load Dependency Litmus Test results for LLMC

IRIW+addr	t2_fence			
t0_fence	relaxed	acquire	release	seq_cst
relaxed	✓	✓	✓	✓
acquire	✓	✓	✓	✓
release	✓	✓	✓	✓
seq_cst	✓	✓	✓	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

Table 5.5 Message Passing Litmus Test results for LLMC

MP	t1_fence			
t0_fence	relaxed	acquire	release	seq_cst
relaxed	✗	✗	✗	✗
acquire	✗	✗	✗	✗
release	✗	✓	✗	✓
seq_cst	✗	✓	✗	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

Table 5.6 Message Passing Litmus Test with dependency results for LLMC

MP-dep	t1_fence			
t0_fence	relaxed	acquire	release	seq_cst
relaxed	✗	✗	✗	✗
acquire	✗	✗	✗	✗
release	✓	✓	✓	✓
seq_cst	✓	✓	✓	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

The result of running the litmus tests was as expected: the litmus tests are correctly verified or falsified when LMO and a coherent and causal cache is assumed. There are two litmus tests which show the limitations of these assumptions.

The IRIW+addr litmus test (Appendix B.5) shows in Table 5.4 that assuming a coherent and causal cache means that different observers never see a different order of writes. A `store` that is flushed from the buffer (using the `Update` transition) is observed by all observers at the same time.

The MP-dep litmus test (Appendix B.7) shows in Table 5.6 that our LMO model does not allow all `[r→r/w]` memory instruction reorderings. The increment acts as a barrier for that memory location. This confirms that our model does not completely model memory instruction reordering according to what RMO allows.

5.2 Experiments

We performed a series of experiments to further ascertain the soundness and completeness of LLMC. Our experiments contain a simple counting algorithm (Section 5.2.1) and a widely used concurrent queue, the Michael-Scott queue (Section 5.2.2).

5.2.1 Concurrent counting

This experiment is to show that atomically incrementing a single counter by multiple threads running concurrently yields is possible and that a correct version is verified to be correct. It is also used to determine the performance of LLMC, which we will cover in Section 5.3. In the algorithm, we use CAS to indicate an atomic compare-and-swap. Listing 5.1 describes the semantics of this operation.

Listing 5.1 CAS operation used in the concurrent counting experiment.

```
1  CAS(destination: pointer to WORD, expected: pointer to WORD, desired: WORD) -> bool
2    if *destination == *expected;
3      *destination = desired;
4      return true;
5    else
6      *expected = destination;
7      return false;
8    endif
```

5.2.1.1 The algorithm

It is a very simple algorithm: every thread attempts a *compare-and-set* until it succeeds, thus incrementing the atomic counter by one. Every thread does this INCS times, thus after every thread is done, the counter should be equal to THREADS*INCS, where THREADS is the number of threads. The algorithm is shown in Listing 5.2. In Appendix C.1 we list the C++ and LLVM IR implementations. Note that between lines 65 and 66 there is no need for a memory barrier. This is because pthread_join() only returns when the buffer of the joined thread is empty.

5.2.1.2 LLMC applied

LLMC correctly verified the correctness of this algorithm. Upon investigating the state space for two threads (Figure 5.1), each incrementing the counter once, we find what we expected. States 0 through 5 form the setup phase in which the counter is initialized and a second thread is started (pthread_create()). At that point either the main thread performs an increment (to state 6) or the started thread performs its initial setup up until it can also perform an increment (to state 7). From that state, both can perform the increment (to state 8 or to state 9). If one succeeds, the first CAS operation (__atomic_compare_exchange_llmc()) of the other will fail and it will try again and succeed. Notice that the two paths do not merge until the last state. This is because their local variable j differs in the two paths: the first thread that succeeded in incrementing the counter has j equal to 0, the second thread will have j equal to 1. They merge in the last state because the **Fini** transition resets the stack and stack memory.

Listing 5.2 The algorithm of a single thread of the concurrent counting experiment.

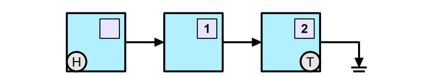
```
1  integer counter;
2
3  increment() {
4    int i=0;
5    while i<INCS
6      int j = i;
7      loop
8        if CAS(&counter,&j,j+1)
9          break;
10       endif
11     endloop
12     i++;
13   endwhile
14 }
```

Figure 5.1 The state space of the concurrent counting experiment for `THREADS`=2 and `INCS`=1

5.2.2 Michael-Scott queue

One example of a software component that is used often in multi-threaded programs is the Michael-Scott queue. This is a concurrent, multiple consumer, multiple producer, lock-free data structure providing a FIFO operation on elements. It can be used as the basics for scheduling work or passing messages. It is available in the Java library as `ConcurrentLinkedQueue`. To understand this experiment, we will first describe the algorithm of the Michael-Scott queue and then explain why the memory barriers are needed. We conclude with the results of `LLMC` regarding this concurrent queue. In the Michael-Scott queue algorithm, we use `CAS(t,e,d)` to indicate an atomic compare-and-set. Listing 5.3 describes the semantics of this operation. The difference with the one used in Section 5.2.1 is that in the event the `CAS` fails, it will not assign the obtained value to `*expected`. Moreover, this version operates on a `DWORD` (double word) instead of a single `WORD`.

Figure 5.2 State after adding `1` and `2`



Listing 5.3 CAS operation used in the Michael-Scott queue.

```
1 CAS(destination: pointer to DWORD, expected: DWORD, desired: DWORD) -> bool
2   if *destination == *expected
3     *destination = desired;
4     return true;
5   else
6     return false;
7   endif
```

Listing 5.4 displays the used structures and initialization of the Michael-Scott queue. The queue works with a linked list, thus the number of elements is not algorithmically bounded. Each node in the linked list has a data member, containing data, and a next member, pointing to the next element or `NULL` in the case of the last node. In the empty state, there is one node in the list: the sentinel node. The value of the data member of this node is not important. There are two shared variables: `Head` and `Tail`; these reflect the global state of the queue. The `Head` pointer points to the oldest node in the linked list, which is the sentinel node. The `Tail` pointer points to the newest node, or the one before that in the case the `Tail` pointer was not yet updated by a `Push()` operation See Figure 5.2 for an illustration of the state of the queue after adding two elements: `1` and `2`.

5.2.2.1 Push

The `push()` operation adds an element by inserting a new node at the tail-end of the queue. When inserting a node, first it is linked in by having the `next` member of the `Tail` node point to the new node. Then, the `Tail` pointer is advanced to reflect the new state of the queue. Of course, at all these stages another thread can interfere. The algorithm takes care of these cases and minimizes the effect of unfortunate context switching by allowing a different thread to finalize the transaction of a thread. Finalizing a transaction means to advance the shared pointer, in this case the `Tail` pointer. Listing 5.5 shows pseudocode of the `push()` operation of the Michael-Scott queue.

Listing 5.4 Structures and initialization of the Michael-Scott queue [MS]

```
1 structure pointer_t {ptr: pointer to node_t, count: unsigned integer}
2 structure node_t {value: data type, next: pointer_t}
3 structure queue_t {Head: pointer_t, Tail: pointer_t}
4
5 initialize(Q: pointer to queue_t)
6   node = new_node()                 # Allocate a free node
7   node->next.ptr = NULL             # Make it the only node in the linked list
8   Q->Head.ptr = Q->Tail.ptr = node # Both Head and Tail point to it
```

Listing 5.5 The `push()` operation of the Michael-Scott queue [MS] with the required memory barriers [BAM07]

```
E1  enqueue(Q: pointer to queue_t, value: data type)
E2    node = new_node()          # Allocate a new node from the free list
E3    node->value = value        # Copy enqueued value into node
E4    node->next.ptr = NULL      # Set next pointer of node to NULL
E5    barrier_release()          # Make sure the data is observed before linking in the node
E6    loop                       # Keep trying until Enqueue is done
E7      tail = Q->Tail           # Read Tail.ptr and Tail.count together
E8      barrier_acquire()        # Obtain the tail before tail.ptr->next
E9      next = tail.ptr->next    # Read next ptr and count fields together
E10     barrier_acquire()        # Obtain the next before rechecking the tail
E11     if tail == Q->Tail       # Are tail and next consistent?
E12       # Was Tail pointing to the last node?
E13       if next.ptr == NULL
E14         # Try to link node at the end of the linked list
E15         if CAS(&tail.ptr->next, next, <node, next.count+1>)
E16           break              # Enqueue is done.  Exit loop
E17         endif
E18       else                   # Tail was not pointing to the last node
E19         # Try to swing Tail to the next node
E20         CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
E21       endif
E22     endif
E23   endloop
E24   # Enqueue is done.  Try to swing Tail to the inserted node
E25   barrier_release()          # Make sure the node is linked in before swinging the Tail
E26   CAS(&Q->Tail, tail, <node, tail.count+1>)
```

5.2.2.2 Pop

When there is data in the queue, the `pop()` operation tries to dequeue an element from the queue and return this element. This is achieved by first advancing the `Head` pointer, thus claiming the data that resides in the new `Head` pointer. After this data is retrieved, the memory of the old sentinel node is freed. As the data in the new sentinel is already obtained, it is no longer important and thus this new sentinel node may be freed without objection by a future `pop()` operation[1].

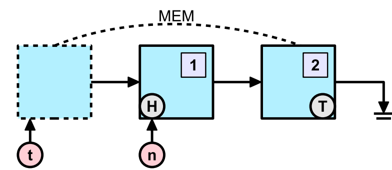
5.2.2.3 Consistent Snapshot

Throughout the descriptions of the algorithms there is the use of the term *consistent snapshot*, but so far we did not make explicit what this exactly entails nor how it is achieved. To explain this, consider the `push()` operation. The first thing that is done in the loop is obtaining the tail and the next of the tail. But what if the tail was advanced before the next of the tail could be obtained? That would not only mean we are using an out of date tail, but also that the tail we are using could potentially be deleted or its memory already reused for a new node. By checking that the tail did not change, we are certain that the `tail` node was not deleted. Because if a `tail` node would have been deleted, it must have been deleted by the `pop()` operation and for that to happen, the head must have caught up with the next of the tail in question. This means the tail should have advanced at least one node, thus the tail changed. So by checking that the `tail` did not change, we can be sure that `tail` was not deleted[1] and that the `next` of the `tail` is consistent with the `tail`.

5.2.2.4 Counted pointers

The pointers mentioned in the algorithm are not normal pointers, but they are *counted pointers*. These are pointers which not only have a normal pointer value, they also have an integer that is increased every time the normal pointer value is altered. They are used to mitigate the ABA problem [IBM83]. This problem has been solved in multiple ways [DPS10, Mic04a, Mic04b] and one of the approaches being the counted pointers [IBM83].

Figure 5.3 Illustration of the ABA problem.



Consider the scenario described in Section 5.2.2.3. There, we check if the `tail` changed, after obtaining the `next` of the

[1] At least, that is what the original algorithm led to believe; in reality the original algorithm only works when *deleted* is replaced with *made available for reuse* and memory of nodes is never given back to the OS.

Listing 5.6 The `pop()` operation of the Michael-Scott queue [MS] with the required memory barriers [BAM07]

```
D1  dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D2    loop                          # Keep trying until Dequeue is done
D3      head = Q->Head              # Read Head
D4      barrier_acquire()           # Obtain the head before the tail
D5      tail = Q->Tail              # Read Tail
D6      barrier_acquire()           # Obtain the tail before the next
D7      next = head.ptr->next       # Read Head.ptr->next
D8      barrier_acquire()           # Obtain the next before rechecking the head
D9      if head == Q->Head          # Are head, tail, and next consistent?
D10       if head.ptr == tail.ptr   # Is queue empty or Tail falling behind?
D11         if next.ptr == NULL     # Is queue empty?
D12           return FALSE          # Queue is empty, couldn't dequeue
D13         endif
D14         # Tail is falling behind.  Try to advance it
D15         CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
D16       else                      # No need to deal with Tail
D17         # Read value before CAS
D18         # Otherwise, another dequeue might free the next node
D19         *pvalue = next.ptr->value
D20         # Try to swing Head to the next node
D21         if CAS(&Q->Head, head, <next.ptr, head.count+1>)
D22           break                 # Dequeue is done.  Exit loop
D23         endif
D24       endif
D25     endif
D26   endloop
D27   free(head.ptr)                # It is safe now to free the old node
D28   return TRUE                   # Queue was not empty, dequeue succeeded
```

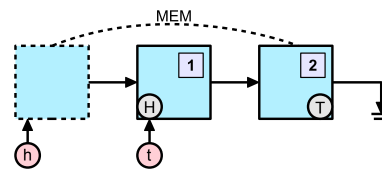
tail. It would be possible for the tail to change twice in a row before it is checked. This would make it possible for the tail pointer to point to the *same* memory location, even though it is pointing to a different node. Without the counted pointers, this would mean that the tail-check would succeed, but this is not intended behaviour, because the next node does not necessarily have to be the node directly after the tail node any more. As can be seen in the illustration in Figure 5.3, the next node could be the node *behind* the tail node. In this scenario, the push() operation will determine that next.ptr != NULL and will then attempt to advance the tail. However, what this thread deems to be advancing the tail is in reality messing up the algorithm, because it will advance the tail in the wrong way.

This can be avoided by using counted pointers. This approach has the disadvantage of requiring a CAS operation on DWORDs, which is not available on all architectures. An alternative is using LL/SC (**l**oad-**l**inked, **s**tore **c**onditional) [Mic04b].

5.2.2.5 The memory barriers

The memory barriers in the Michael-Scott queue are all necessary for correct behaviour. However, they are not all needed on all memory models, because some models already guarantee the memory barriers inherently. Under TSO for example, none of the memory barriers are needed. This is because TSO, for the purpose of this queue, guarantees acquire-release semantics for all memory operations.

Figure 5.4 Illustration of why the D8 memory barrier is needed.



Under RMO, they are in fact needed, with the exception of the two in the enqueue: 1) the **acquire** memory barrier at line 8 is not needed because it protects a dependent load; and 2) the **release** memory barrier at line 25 is not needed, because it protects a data dependent control dependency, the CAS in line 15 and the CAS in line 26. We will discuss the other 5 and why they are needed below. Under LMO, the same 5 barriers are needed. This is essential to understand the result of LLMC.

The E5 memory barrier is needed to make sure the store operations to value and next to the new node are observed by a dequeuer before the enqueuer links in that node. Otherwise, the incorrect data might be read or the queue might end up in an inconsistent state.

The E10 memory barrier is needed to make sure the second read of Q->Tail is done after reading the next of tail. If the reverse would be allowed, it would be possible for a node to be enqueued on a node that might not belong to this queue. This is only a concern when two Michael-Scott queue instances share the same pool of nodes. The trace of such a situation is as follows: operating on an empty queue, an enqueuer obtains the Tail and checks for consistency. Another thread now enqueues and dequeues a node, thus freeing the original node for reuse. This node is then reused by an enqueuer to link in a node on a different Michael-Scott queue instance. The first enqueuer will continue execution, find that next.ptr==NULL and link in the node in the wrong Michael-Scott queue instance.

The D4 memory barrier guarantees the Head is read before the Tail. If the reverse would be allowed, it would be possible to incorrectly determine head.ptr!=tail.ptr while next.ptr is NULL. Thus, the statement in line 19 would produce a segmentation fault. The trace to this situation is easy to describe: operating on an empty queue, a dequeuer first obtains the Tail, then another thread enqueues a node and dequeues a node. Now, the first dequeuer is allowed to obtain the Head and the next of Head. It will then determine head==Q->Head and head.ptr!=tail.ptr.

The D6 memory barrier guarantees the Tail is read before reading the next of the Head. If the reverse would be allowed, it would again be possible to incorrectly determine head.ptr!=tail.ptr while next.ptr is NULL. The trace to this situation is even simpler: a dequeuer obtain the Head and the next of Head, then another thread enqueues a node. Now, the first dequeuer obtains the Tail. It will then determine head==Q->Head and head.ptr!=tail.ptr.

The D8 memory barrier guarantees that checking the Head is done after obtaining the next of Head. If the reverse would be allowed, it would yet again be possible to incorrectly determine

`head.ptr!=tail.ptr` while `next.ptr` is `NULL`. The trace to this situation is more tricky as it requires memory to be reused by one thread while still in use by another. Figure 5.4 illustrates this trace. Starting with a queue with a single element, thus two nodes, a `dequeuer` first obtains the `Head`, `Tail` and checks for consistency (`head==Q->Head`). Then another thread dequeues a node and enqueues a node. The node that it enqueues has the same memory address as the dequeued node, which is possible when allowing nodes to be reused. Because that node is the last node in the linked list, its `next.ptr` field is `NULL`. Thus, when the first dequeuer now obtains the `next.ptr` value and continues, it will determine `head.ptr!=tail.ptr` and dereference `NULL`.

5.2.2.6 LLMC applied

When we applied LLMC to the Michael-Scott queue, we were able to find all 5 required memory barriers for our model. Using the C++ code in Appendix C.2 and simply commenting one of the barriers, one of the asserts would trigger. Sometimes this required modifying `main()` slightly to allow more concurrent threads. The resulting code would be compiled with `clang msq2.cpp -std=c++11 -pthread -S -flto -emit-llvm -omsq2.ll -O3 -g` and then model checked with LLMC. The result is either the state space or a trace to an error.

In the case of the Michael-Scott queue we have the luxury of knowing where the barriers are needed, thus we can modify our test case accordingly. We can describe what test case can be used to find the memory barriers by using an `E` for enqueuer, `D` for dequeuer, `;` for sequential execution and `||` for parallel execution. In our case, `||` will have a stronger binding than `;`. For example, `E;E||D;D` means to first execute an enqueuer, then to parallel execute an enqueuer and a dequeuer and when they are both done to execute a dequeuer. In the tests below we will always use 3 enqueuers and 3 dequeuers, to keep the final assert correct.

- **[`E||D;E;E;D;D`]** Using this, we can find the E5 and D6 memory barriers.
- **[`E||(D;E;C);E;D;D`]** Using this, we can find the E10 memory barrier. `C` is the act of claiming a node for another data structure. Because a node with data is enqueued on a node outside of the queue, the data is lost. This means one of the dequeuers will remain in the while, because there is no data to be dequeued. We can detect this livelock by testing for divergence, as described in Section 3.4.2.
- **[`E;D||(E;D;D;E)`]** Using this, we can find the D4 memory barrier. Note that `D||(E;D)` would have sufficed as well, but would live lock. We can also test for this using divergence.
- **[`E;E;D||(D;E);D`]** Using this, we can find the D8 memory barrier.

Note that if we did not have the luxury of knowing where the memory barriers are required, we would need to try out multiple executions. Starting 6 threads in parallel, i.e. `E||E||E||D||D||D` is not feasible. The largest complete run we have achieved is `E||E||E;D;D;D`, the state space of which is close to 1.4 billion states (see Table 5.8).

5.3 Benchmarks

In this section, we provide benchmarks that provide an indication of what LLMC can handle.

5.3.1 Performance

We ran a series of performance benchmarks using the concurrent counting LLVM IR implementation and the Michael-Scott queue. Tables 5.7 and 5.8 show the results of these experiments. The experiments are executed on machine with 48 cores and 128GiB of memory. To explore the state space in search of error states, we used the multi-core back-end of LTS$_{min}$ with default options and an invariant specifying that the state has no error. This provides us an indication of what LLMC is able to handle.

Comparing X1 to X4 and X2 to X6 we can see that changing the buffer size from 3 to 5 increases the state space by respectively 17% and 27%. This gives a small indication of the influence of the buffer size on the state space.

The decrease in memory usage of using chunk maps can amount to two orders of magnitude: running the recursive Fibonacci LLVM IR program (Appendix C.3), calculating `fib(20)` without this optimization uses 3.3GiB of RAM. When we apply this chunk mapping scheme, the memory usage shrinks to just 30MiB. This is an extreme example because the recursive Fibonacci is intensive for the stack.

Let us look at the influence of the chunk maps on X6: we have 24810 chunks there, taking 17MiB of memory, which is on average 0.7kiB per chunk. If we would not have used chunks for the stack, this would have meant that a single state would need at least 1.4kiB on average, for the stack and globals. This would have amounted to a grand total of 16GiB for all the states. This is only a rough estimate, but it is certainly significantly higher than the 191.8MiB that was actually used.

5.3.2 Implementation bottlenecks

We performed benchmarks showing the bottlenecks of LLMC. In Figure 5.5 a typical call graph for an execution of LLMC is depicted. There are four methods that take the most time:

1. `downloadState()` (`readStateVector()` in the graph), which reads a state vector coming from LTS$_{min}$ and constructs a working LLVM Interpreter from it, initialized to the state in the state vector. This part requires a lot of memory operations.
2. `uploadState()` does the reverse. The state of a working LLVM Interpreter is flattened into

Table 5.7 Results of executed concurrent counting experiments.

THREADS=4, buffer_size=3							
INCS	**States**	**Transitions**	**Time (s)**	**Mem (MiB)**	**States/s**	**Trans/s**	**Chunks**
1	611	1158	<1	<1	20367	38600	206
2	111713	252630	1.120	1.4	99744	225562	1021
3	16075101	37526179	99.540	174.9	161494	376996	7099
4	2868903162	6770299558	19038.061	29882.9	150693	355619	58203

Table 5.8 Results of executed Michael-Scott queue experiments.

<table>
<tr><th colspan="9">buffer_size=3</th></tr>
<tr><th>ID</th><th>Config</th><th>States</th><th>Transitions</th><th>Time (s)</th><th>Mem (MiB)</th><th>States/s</th><th>Trans/s</th><th>Chunks</th></tr>
<tr><td>X1</td><td>E||D;E;D;E;D</td><td>18000</td><td>46046</td><td>3.3</td><td><1</td><td>62069</td><td>158779</td><td>3416, 3MiB</td></tr>
<tr><td>X2</td><td>E;D||(E;D;D;E)</td><td>9749606</td><td>29252651</td><td>180.6</td><td>144.0</td><td>67696</td><td>203115</td><td>23731, 16MiB</td></tr>
<tr><th colspan="9">buffer_size=5</th></tr>
<tr><th>ID</th><th>Config</th><th>States</th><th>Transitions</th><th>Time (s)</th><th>Mem (MiB)</th><th>States/s</th><th>Trans/s</th><th>Chunks</th></tr>
<tr><td>X3</td><td>E||D;E;E;D;D</td><td>20391</td><td>56741</td><td>2.4</td><td><1</td><td>78427</td><td>218235</td><td>2677, 2MiB</td></tr>
<tr><td>X4</td><td>E||D;E;D;E;D</td><td>21036</td><td>57518</td><td>4.4</td><td><1</td><td>67858</td><td>185542</td><td>7099, 3MiB</td></tr>
<tr><td>X5</td><td>E;E;D||(D;E);D</td><td>1027044</td><td>3178352</td><td>27.9</td><td>15.4</td><td>66821</td><td>206789</td><td>9227, 10MiB</td></tr>
<tr><td>X6</td><td>E;D||(E;D;D;E)</td><td>12349674</td><td>39519755</td><td>222.3</td><td>191.8</td><td>64382</td><td>206025</td><td>24810, 17MiB</td></tr>
<tr><td>X7</td><td>E||E||E;D;D;D</td><td>1390462376</td><td>5099266244</td><td>19348.8</td><td>15561.9</td><td>71863</td><td>263545</td><td>663975, 492MiB</td></tr>
<tr><td>X8*</td><td>E||E||D||D</td><td>>3886772182</td><td>>18569898690</td><td>>44406.8</td><td>>60545.2</td><td>∼64196</td><td>∼306711</td><td>>52713, >28MiB</td></tr>
</table>

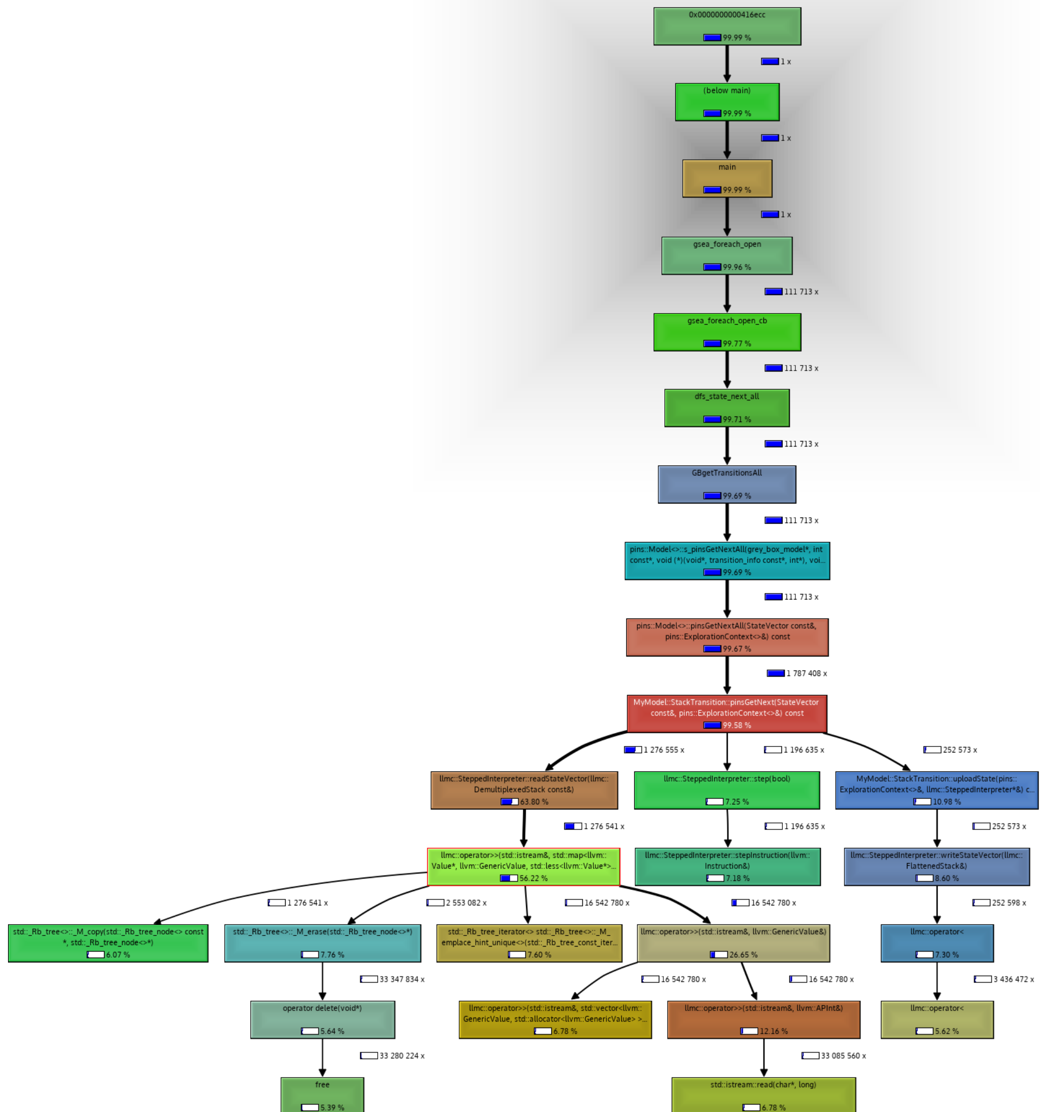
*:incomplete execution

a state vector and is uploaded to LTS_{min}.

3. `step()` performs a single step. This takes roughly a third of the entire time. Ideally this should be significantly higher.

As can be seen in the graphic, uploading and downloading the state from and to LTS_{min} takes the greatest amount of time: almost 75%. The actual calculating of the next state only takes around 7% of time. This was an expected result: in particularly downloading the state and re-instantiating the LLVM Interpreter takes a significant amount of time. We believe there is still a lot of room for optimization, especially in the `readStateVector()` method.

Figure 5.5 A typical call graph for an execution of LLMC



6 Conclusions

In this chapter, we first provide a summary (Section 6.1), then we evaluate our contribution (Section 6.2) and finally we propose future work (Section 6.3).

6.1 Summary

To aid developers of concurrent software, we have developed LLMC, the low-level model checker. We have designed an execution model that captures executing LLVM IR running on a relaxed memory, that completely models the allowed memory instruction reorderings of TSO (`[w→r]`) and PSO (`[w→r/w]`). It also supports a limited relaxed memory model (LMO), which in addition to PSO, allows some load instruction reorderings (`[w→r/w + r→*r/w]`), but not to the same extent as RMO (`[r/w→r/w]`).

We have provided a mapping from LLVM IR to our execution model and to PINS. Using this, we show how we can explore the state space of LLVM IR programs running on LMO. We provide the design of an exploration strategy that explores the state space of an LLVM IR program while incrementally relaxing the memory mode. Using this exploration strategy, we can investigate the behaviour of an LLVM IR program running on SC, TSO, PSO and LMO in one exploration.

We have implemented our design in LLMC by reusing the LLVM Interpreter and modifying it in order to connect it to PINS. This way, newer LLVM IR versions can be supported in the future by merging the new LLVM Interpreter into our LLVM Interpreter.

The implementation limits memory usage by inserting the stacks, stack memory, buffers and globals in a chunk table and using a chunk identifier in the state space. This reduces the footprint of the state space by multiple orders of magnitude.

We applied LLMC to various litmus tests and performed multiple experiments with it. We passed the implemented litmus tests for the LMO memory model and for some of them for RMO, but were incomplete with respect to the RMO memory model in other cases. We applied LLMC to a well-known concurrent queue, the Michael-Scott queue, and were able to confirm the necessity of the required memory barriers for correctness under RMO.

The output of LLMC is not only a boolean answer, it also generates a trace to the erroneous state. While this trace contains mostly chunk map identifiers, the actions clearly indicate the path the program took: when which thread did what and when which buffer was used or partly flushed. The trace uses LLVM IR instructions as actions. This gives a precise path to the erroneous state at the LLVM IR level.

6.2 Evaluation

In this section we evaluate our tool LLMC and the obtained results.

6.2.1 Considerations

By choice, our LMO memory model does not reorder dependent loads. We do not consider this a viable memory model for any hardware as the software running on it would require every dependent load to be separated by a `load-load` memory barrier. Moreover, modern microprocessors do not reorder dependent loads.

One major consideration in our LMO memory model is that it does not allow all `[r→r/w]` that RMO allows. Because instructions are possibly dependent on preceding loads, execution can only happen if these loads are actually performed. Thus, in that scenario, those preceding loads are never reordered with memory operations after such an instruction. A possible solution is to add these instructions to the buffer as well and make an addendum to when an instruction is enabled: an operation is *enabled* iff none of its operands depend on preceding operations in the buffer. This allows operations to overtake each other, but not allowing dependent operations to overtake each other: an operation can only be executed if all the operations it depends on are executed. We have an experimental version of LLMC that uses this technique, but it suffers from related regressions. We leave this for future work.

Moreover, our model assumes a coherent and causal cache. This gave use the benefit of using a single representation of the global memory, but this further limits the completeness of our model. As we saw in the IRIW litmus test, LLMC does not execute behaviour that is not causal.

6.2.2 So where does that leave LLMC?

LLMC can successfully invalidate the Michael-Scott queue when one of 5 memory barriers that do not protect a data dependency is left out. It can correctly validate and invalidate various litmus tests and other test cases. While not complete with respect to RMO, the model is sound: if LLMC claims there is an erroneous state and gives a trace, it means an error in the LLVM IR program.

The bottom line is that it supports model checking LLVM IR programs assuming a limited relaxed memory model (LMO). This in itself is useful for the development and debugging of current and future concurrent data structures. We hope that this will become apparent in the future.

6.3 Future Work

For future work, we first list further future features (Section 6.3.1) both in the supported functionality as well as in LTS$_{min}$, then we suggest future topics of research that can follow in the footsteps of this research (Section 6.3.2) and finally we suggest more test cases to test both performance and validity of both this research and the suggested research (Section 6.3.3).

6.3.1 Future Features

There are a lot of features that did not make it into this version of LLMC. To start, there is only an experimental implementation of `llmc_atomic_begin()` and `llmc_atomic_end()`, which could be used to describe an atomic section. If a thread would be in this section, other threads could not perform a step. It resembles the `d_step` of PROMELA.

Secondly, the implementation does not allow `__atomic_compare_exchange_llmc()` to spuriously fail. In reality, this is possible on LL/SC (load-link/store-conditional) hardware. We expect this is fairly easy to implement, but would require some testing.

Thirdly, we believe that we can optimize the re-instantiation of the LLVM Interpreter by directly operating on the chunk map when dealing with interpreter memory operations. Thus avoiding the need to re-instantiate. We believe this is possible without altering the core implementation of the LLVM Interpreter, but instead writing a thin wrapper around all memory access routines. We can apply the same technique to the access of the registers. By redefining access to the stack frame, we avoid the need to re-instantiate the stack. We suspect that by limiting the overhead of serializing and re-instantiating the LLVM Interpreter we can at least attain a doubling of the performance.

To name a fourth wanted feature, the current implementation does not handle `stdin` or `stdout`. The approach we envision that would be appropriate is to intercept calls to functions like `scanf()` and have them choose an input non-deterministically from a predefined set of possible inputs. This set could be specified on a per call to `scanf()` basis, using for example the static information of file, line number and calling function. Dynamic information such as the number of times `scanf()` had been called before is also possible. Note that this feature is outside of the target audience of LLMC: the primary target is to model check concurrent data structures.

Dynamic memory allocation on the heap did not make it into this version. We can implement this using the chunk mapping technique we used for the global variables, thus pointers are not an issue, as we would keep the memory at the same location.

Lastly, our implementation of PINS does not support distributed model checking. The reason for this is that the LLVM Module is not in the state vector, but resides in memory which is the same for processes on the same machine. However, across multiple machines this will not work. We attempted to put the entire LLVM Module in the state vector, but did not succeed. Having the LLVM Module in the state vector would not only bring distributed model checking to the table, but also self modifying LLVM IR.

6.3.2 Future Research

As mentioned in Section 6.2.1, one of the drawbacks of the current implementation is that local instructions that use a register cause a preceding load to that register to complete before continuing. This acts as a local barrier and does not allow all memory instruction reorderings that should be allowed under RMO. However, adding this may prove a daunting task: even though adding these local instructions to the buffer as well seems like a possible solution, it does yield an enormous growth in the state space. We have some ideas to limit this to a manageable growth: 1) if performing an enabled instruction is flushed from the buffer causes another local instruction (in the buffer or the instruction to be executed) to be enabled, execute that as well; 2) add multiple consecutive instruction to the buffer in one transition, if that instruction uses only local data, thus the observable behaviour of that process is not altered; and 3) instead of generating a new state after a memory barrier, the transition could perform one global instruction afterward, since executing a memory barrier is not observable in our model, its execution is merely conditional. This addition would make the LLMC's exploration more complete.

The current model assumes the cache to be coherent and causal. While this model still allows most reorderings, it does not allow non-causal behaviour and thus makes LLMC less complete. However, modeling such a cache would cause a growth in the state space and more pressingly, the required memory would grow significantly. We expect the chunk mapping will take care of the growth in memory usage, but cannot state this with certainty.

Currently LLMC gives a trace to an erroneous state, or multiple traces if there are more than 1 errors. However, it does not specify how to fix the problem. An interesting topic of research would be to devise a way to use the given traces and the state space to suggest the placement of one or more memory barriers in order to fix the error. Providing that the cause of the error is memory instruction reordering and not an inherent problem in the program.

During this research, we primarily used the multi-core and distributed back-ends of LTS_{min}. We tried the symbolic back-end, but it was significantly slower than the multi-core back-end. We suspect there is a lot of room for improvement in the dependency matrix, because in the current one we have a transition group with a dependency on all state vector variables: the thread management transition group. It has a dependency on all process slots, because any process can create a new process in any process slot. However, by separating this transition group into multiple separate ones, one for each combination of two process slots, we can get rid of the transition group with this large dependency. Then, implementing the PINS interface for short state vectors would be the next step. This would make an interesting follow-up research, to investigate how well symbolic exploration algorithms can handle exploring LLVM IR.

Continuing on this train of thought, it would be of interest if program arguments or even operations using `stdin` could be handled symbolically. However, we suspect this is not practically feasible: if the LLVM IR program would use `argc` to determine the number of threads for example, the exploration would take a tremendous amount of time.

6.3.3 Future test cases

To further test the validity and performance of LLMC, we suggest implementing primarily three new test cases: 1) the Dualqueue [SS04], which is based on the Michael-Scott queue, but provides a way for the dequeuers to wait for an element instead of polling; 2) applying Hazard Pointers to the Michael-Scott queue, to provide a garbage collecting technique customized for concurrent data structures; and 3) applying Hazard Pointers to the Dualqueue. We have an LLVM IR implementation of the Dualqueue, but we were unable to verify it for more than two threads. This is because the algorithm is much more complex than the Michael-Scott queue.

The tool `diy` [AMSS12] is a generator for litmus tests for PowerPC or X86 from concise specifications. It is worth investigating how well LLMC handles the numerous tests this tool can generate, taking into account the various memory models.

In the event that future research is done to model the cache such that non-causal behaviour is allowed, the concurrent counting test case with more than two threads will be of use: the atomic transaction have to make sure all threads agree.

If LLMC is enhanced with support for `stdin` and `stdout`, a useful test case would be the Rigorous Examination of Reactive Systems (RERS) challenge[1]. This is challenge where competitors can attempt to solve white-box, grey-box and black-box problems using any means. These problems are for example implemented in C and use `stdin/stdout` to communicate with the environment. It would be interesting to investigate how well LLMC can handle such problems. While there is no concurreny in them, there is still an explosion of the number of states because of the inputs.

[1] `http://rers-challenge.org/`

Bibliography

[ABBM10] M. F. Atig, A. Bouajjani, S. Burckhardt and M. Musuvathi.
On the verification problem for weak memory models.
In *POPL* (editors M. V. Hermenegildo and J. Palsberg), pages 7–18. ACM, 2010.
ISBN 978-1-60558-479-9

[ABH+04] A. Alexandrescu, H. Boehm, K. Henney, D. Lea and B. Pugh.
Memory model for multithreaded C++.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1680.pdf (checked 2013-10), 2004

[AKT13] J. Alglave, D. Kroening and M. Tautschnig.
Partial Orders for Efficient Bounded Model Checking of Concurrent Software.
In *CAV*, volume 8044 of *Lecture Notes in Computer Science* (editors N. Sharygina and H. Veith), pages 141–157. Springer, 2013.
ISBN 978-3-642-39798-1

[AMSS12] J. Alglave, L. Maranget, S. Sarkar and P. Sewell.
Fences in weak memory models (extended version).
In *Formal Methods in System Design*, volume 40, 2, (2012), pages 170–205

[ANR09] N. H. M. Aan de Brugh, V. Y. Nguyen and T. C. Ruys.
MoonWalker: Verification of .NET Programs.
In *TACAS*, volume 5505 of *Lecture Notes in Computer Science* (editors S. Kowalewski and A. Philippou), pages 170–173. Springer, 2009.
ISBN 978-3-642-00767-5

[ARM10] ARM Limited, 110 Fulbourn Road Cambridge, England CB1 9NJ.
ARM®v7-M Architecture Reference Manual, c_errata_v3 edition, February 2010

[BAM07] S. Burckhardt, R. Alur and M. M. K. Martin.
CheckFence: checking consistency of concurrent data types on relaxed memory models.
In *PLDI* (editors J. Ferrante and K. S. McKinley), pages 12–21. ACM, 2007.
ISBN 978-1-59593-633-2

[BBH+13] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill and J. Weiser.
DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs.
In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013

[BG04] F. Belli and B. Güldali.
Software testing via model checking.
In *Computer and Information Sciences-ISCIS 2004*, pages 907–916. Springer, 2004

[BH08] D. Babic and A. J. Hu.
Calysto: scalable and precise extended static checking.

In *ICSE* (editors W. Schäfer, M. B. Dwyer and V. Gruhn), pages 211–220. ACM, 2008.
ISBN 978-1-60558-079-1

[BL12] F. I. van der Berg and A. W. Laarman.
SpinS: Extending LTSmin with Promela through SpinJa.
In *11th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2012, London, UK* (editors K. Heljanko and W. J. Knottenbelt), Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam, September 2012.
ISSN 1571-0661

[BL13] D. Beyer and S. Löwe.
Explicit-State Software Model Checking Based on CEGAR and Interpolation.
In *FASE*, volume 7793 of *Lecture Notes in Computer Science* (editors V. Cortellessa and D. Varró), pages 146–162. Springer, 2013.
ISBN 978-3-642-37056-4

[BLPW09] S. C. C. Blom, B. Lisser, J. C. van de Pol and M. Weber.
A Database Approach to Distributed State-Space Generation.
In *Journal of Logic and Computation*, volume 21, 1, (2009), pages 45–62.
ISSN 0955-792X

[BLR11] T. Ball, V. Levin and S. K. Rajamani.
A decade of software model checking with SLAM.
In *Commun. ACM*, volume 54, 7, (2011), pages 68–76.
ISSN 0001-0782.
doi:10.1145/1965724.1965743.
URL http://doi.acm.org/10.1145/1965724.1965743

[BM08] S. Burckhardt and M. Musuvathi.
Effective Program Verification for Relaxed Memory Models.
In *CAV*, volume 5123 of *Lecture Notes in Computer Science* (editors A. Gupta and S. Malik), pages 107–120. Springer, 2008.
ISBN 978-3-540-70543-7

[BPW09] S. C. C. Blom, J. C. van de Pol and M. Weber.
Bridging the Gap between Enumerative and Symbolic Model Checkers.
Technical Report TR-CTIT-09-30, Centre for Telematics and Information Technology University of Twente, Enschede, June 2009

[BPW10] S. C. C. Blom, J. C. van de Pol and M. Weber.
LTSmin: Distributed and Symbolic Reachability.
In *Computer Aided Verification, Edinburgh*, volume 6174 of *Lecture Notes in Computer Science* (editors T. Touili, B. Cook and P. Jackson), pages 354–359. Springer Verlag, Berlin, July 2010.
ISSN 0302-9743

[BV05] G. Brat and A. Venet.
Precise and scalable static program analysis of NASA flight software.
In *In Proceedings of the 2005 IEEE Aerospace Conference*, 2005

[Car02] C. Carvalho.
The gap between processor and memory speeds.
In *Proc. of IEEE International Conference on Control and Automation*, 2002

[CDE08] C. Cadar, D. Dunbar and D. R. Engler.
KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.
In *OSDI* (editors R. Draves and R. van Renesse), pages 209–224. USENIX Association, 2008.
ISBN 978-1-931971-65-2

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith.
Counterexample-Guided Abstraction Refinement.
In *CAV*, volume 1855 of *Lecture Notes in Computer Science* (editors E. A. Emerson and A. P. Sistla), pages 154–169. Springer, 2000.
ISBN 3-540-67770-4

[CI10] S. Chaki and J. Ivers.
Software model checking without source code.
In *ISSE*, volume 6, 3, (2010), pages 233–242

[CKL04] E. Clarke, D. Kroening and F. Lerda.
A Tool for Checking ANSI-C Programs.
In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science* (editors K. Jensen and A. Podelski), pages 168–176. Springer, 2004.
ISBN 3-540-21299-X

[DDA08] I. Dillig, T. Dillig and A. Aiken.
Sound, complete and scalable path-sensitive analysis.
In *PLDI* (editors R. Gupta and S. P. Amarasinghe), pages 270–280. ACM, 2008.
ISBN 978-1-59593-860-2

[DH07] W. Damm and H. Hermanns (editors).
Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007.
ISBN 978-3-540-73367-6

[DLP13] T. van Dijk, A. W. Laarman and J. C. van de Pol.
Multi-Core BDD Operations for Symbolic Reachability.
In *Electr. Notes Theor. Comput. Sci.*, volume 296, (2013), pages 127–143

[DPS10] D. Dechev, P. Pirkelbauer and B. Stroustrup.
Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs.
In *ISORC*, pages 185–192. IEEE Computer Society, 2010

[Duc13] G. J. Duck.
Satisfiability Modulo Constraint Handling Rules (Extended Abstract).
In *IJCAI* (editor F. Rossi). IJCAI/AAAI, 2013.
ISBN 978-1-57735-633-2

[Gue04] R. Guerraoui (editor).
Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings, volume 3274 of *Lecture Notes in Computer Science*. Springer, 2004.
ISBN 3-540-23306-7

[GWZ+11] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang and L. Zhang.
Practical software model checking via dynamic interface reduction.
In *SOSP* (editors T. Wobber and P. Druschel), pages 265–278. ACM, 2011.
ISBN 978-1-4503-0977-6

[HKK+13] Y.-L. Hwong, J. J. A. Keiren, V. J. J. Kusters, S. J. J. Leemans and T. A. C. Willemse.
Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider.
In *Sci. Comput. Program.*, volume 78, 12, (2013), pages 2435–2452

[HP06] J. L. Hennessy and D. A. Patterson.
Computer Architecture, Fourth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
ISBN 0123704901

[IBM83] IBM.
IBM System/370 Extended Architecture, Principles of Operation.
In *Publication No. SA22-7085*

[Kur94] R. Kurshan.
Computer-aided verification of coordinating processes: the automata-theoretic approach.
Princeton university press, 1994

[Lara] M. Larabel.
LLVM 3.2 vs GCC.
http://www.phoronix.com/scan.php?page=article&item=llvm_clang32_final (checked 2013-10)

[Larb] M. Larabel.
LLVM 3.3 vs GCC on Intel's Core i7 4770K.
http://www.phoronix.com/scan.php?page=article&item=intel_haswell_llvm33 (checked 2013-10)

[Lat02] C. Lattner.
LLVM: An Infrastructure for Multi-Stage Optimization.
Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.
http://llvm.cs.uiuc.edu.

[LLI] **LLVM IR Language Specification**.
http://llvm.org/docs/LangRef.html (checked 2013-10)

[LLV] **LLVM Project**.
http://llvm.org (checked 2013-10)

[LPPW13] A. W. Laarman, E. Pater, J. C. van de Pol and M. Weber.
Guard-based Partial-Order Reduction.
In *Proceedings of the 20th International SPIN Symposium on Model Checking of Software, SPIN 2013, Stony Brook, NY, USA* (editors E. Bartocci and C. R. Ramakrishnan), Lecture Notes in Computer Science. Springer Verlag, London, July 2013

[LPW10] A. W. Laarman, J. C. van de Pol and M. Weber.
Boosting Multi-Core Reachability Performance with Shared Hash Tables.
In *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Switzerland* (editors N. Sharygina and R. Bloem), pages 247–256. IEEE Computer Society, USA, October 2010

[LPW11a] A. W. Laarman, J. C. van de Pol and M. Weber.
Multi-Core LTSmin: Marrying Modularity and Scalability.
In *Proceedings of the Third International Symposium on NASA Formal Methods, NFM 2011, Pasadena, CA, USA*, volume 6617 of *Lecture Notes in Computer Science* (editors M. Bobaru, K. Havelund, G. Holzmann and R. Joshi), pages 506–511. Springer Verlag, Berlin, July 2011.
ISSN 0302-9743

[LPW11b] A. W. Laarman, J. C. van de Pol and M. Weber.
Parallel Recursive State Compression for Free.
In *Proceedings of the 18th International SPIN Workshop, SPIN 2011, Snow Bird, Utah*, volume 6823 of *Lecture Notes in Computer Science* (editors A. Groce and M. Musuvathi), pages 38–56. Springer Verlag, berlin, July 2011.
ISSN 0302-9743

[LTS] **LTS$_{min}$, a toolset for model checking and manipulating labelled transition systems**.
http://fmt.cs.utwente.nl/tools/ltsmin/ (checked 2013-10)

[M$^+$65] G. E. Moore et al.
Cramming more components onto integrated circuits, 1965

[Mic04a] M. M. Michael.
Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects.
In *IEEE Trans. Parallel Distrib. Syst.*, volume 15, 6, (2004), pages 491–504

[Mic04b] M. M. Michael.
Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS.
In Guerraoui [Gue04], pages 144–158

[Mos09] M. J. Moskal.
Satisfiability Modulo Software.
Ph.D. thesis, University of Wroclaw, 2009

[MQB$^+$07] M. Musuvathi, S. Qadeer, T. Ball, M. Musuvathi, S. Qadeer and T. Ball.
Chess: A systematic testing tool for concurrent software.
In *Microsoft Research*

[MS] M. M. Michael and M. L. Scott.
Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms

[MSS12] L. Maranget, S. Sarkar and P. Sewell.
A Tutorial Introduction to the ARM and POWER Relaxed Memory Models.
http://128.232.0.20/~pes20/ppc-supplemental/test7.pdf (checked 2013-10), October 2012

[Pel96] D. Peled.
Combining Partial Order Reductions with On-the-Fly Model-Checking.
In *Formal Methods in System Design*, volume 8, 1, (1996), pages 39–64

[SFM10] C. Sinz, S. Falke and F. Merz.
A Precise Memory Model for Low-Level Bounded Model Checking.
In *Proceedings of the 5th International Workshop on Systems Software Verification (SSV '10)*, Vancouver, Canada, 2010

[Spa98] Sparc.
The SPARC Architecture Manual Version 9.
Prentice Hall PTR, 1998.
ISBN 0130992275

[SS04] W. N. Scherer III and M. L. Scott.
Nonblocking Concurrent Data Structures with Condition Synchronization.
In Guerraoui [Gue04], pages 174–187

[SST12] K. F. Sagonas, C. Stavrakakis and Y. Tsiouris.
ErLLVM: an LLVM backend for Erlang.
In *Erlang Workshop* (editors T. Hoffman and J. Hughes), pages 21–32. ACM, 2012.
ISBN 978-1-4503-1575-3

[Sut] H. Sutter.
The Free Lunch is Over.
http://www.gotw.ca/publications/concurrency-ddj.htm (checked 2013-10)

[TBS] S. Thompson, G. Brat and K. Schimpf.
The MCP Model Checker.
http://ti.arc.nasa.gov/m/pub-archive/1312h/1312 (Thompson, S).pdf (checked 2013-10)

[TC10] D. A. Terei and M. M. T. Chakravarty.
An llVM backend for GHC.
In *Haskell* (editor J. Gibbons), pages 109–120. ACM, 2010.
ISBN 978-1-4503-0252-4

[Trc] **Therac 25: An investigation of the accidents**.
http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html (checked 2013-10)

[VPP] **Verification and Validation of Software Related to Nuclear Power Plant Instrumentation and Control**.
http://www-pub.iaea.org/mtcd/publications/pdf/trs384_scr.pdf (checked 2013-10)

[WC05] R. V. Welch and Z. N. Cox.
Verification and validation of Mars Exploration Rover surface capabilities.
In *SMC*, pages 456–461. IEEE, 2005.
ISBN 0-7803-9298-1

[WM95] W. A. Wulf and S. A. McKee.
Hitting the memory wall: implications of the obvious.
In *ACM SIGARCH computer architecture news*, volume 23, 1, (1995), pages 20–24

[XA05] Y. Xie and A. Aiken.
Saturn: A SAT-Based Tool for Bug Detection.
In *CAV*, volume 3576 of *Lecture Notes in Computer Science* (editors K. Etessami and S. K. Rajamani), pages 139–143. Springer, 2005.
ISBN 3-540-27231-3

A

Glossary

Glossary

Alpha DEC Alpha is a 64-bit RISC instruction set architecture that has notoriously weak memory ordering. 7

AMD64 AMD64 was created as an alternative to the IA-64 architecture. It was positioned by AMD from the beginning as an evolutionary way to add 64-bit computing capabilities to the existing x86 architecture. This in contrast to Intel's approach of creating an entirely new 64-bit architecture with IA-64. AMD64 uses a TSO memory model. 7

ARMv7 ARMv7 is the 7th version of the ARM instruction set. It is a RISC (Reduced Instruction Set Computer) instruction set. It is widely used in electronic devices, most prominently in smartphones. 7, 8

Bug A bug in this context means a fault in the source code of a problem. The origin of the words comes from the word *debugging*, which was used to describe the activity of removing real bugs from the first computers. These computers were so large that real bugs could enter it and cause it to clog up. 1

CPU A Central Processing Unit (CPU) performs calculations like integer and floating arithmetic. 5

IA-64 IA-64 was Intel's 64-bit successor of the x86 instruction set. It is a 64-bit register-rich explicitly parallel architecture. A key property is that it does not use a TSO memory model like its predecessor, but uses a much more relaxed memory model instead. 7

JIT Just-In-Time compilation (JIT) is a method to improve the run-time performance of computer programs based on virtual machine code. 9

library A library is a collection of subroutines that can be linked to by another binary to use these subroutines. 1

memory barrier A memory barrier is an instruction used to refrain the processor from reordering certain memory operations. The four relevant barriers are `LoadLoad`, `LoadStore`, `StoreLoad`, `StoreStore`. 7

memory model A memory model dictates the conditions under which writes of one processor become observable to another processor and the constraints under which read operations may succeed. 6

mmap `mmap` is a POSIX function that allows mapping a file to addresses in memory. It also allows to extend the heap per process by requesting a range of memory to be used privately, per process. Optionally an address may be specified to request that the range of memory to be allocated is to start at that address. 35

MMX MMX is a single instruction, multiple data (SIMD) instruction set designed by Intel, introduced in 1997 with their P5-based Pentium line of microprocessors. 10

Sparc-V9 SPARC Version 9, the 64-bit SPARC architecture, was released by SPARC International in 1993. It assumes only a weak memory model, allowing various memory instruction reorderings. 8

TDP The thermal design power (TDP), sometimes called thermal design point, refers to the maximum amount of power the cooling system in a computer is required to dissipate. 8

x86 Going back as far as the Intel 8086 CPU, X86 means a backwards compatible instruction set. The term usually implies a binary compatibility with the 32-bit instruction set of the Intel 80368. 7, 8

Litmus Tests

B.1 Store Buffer Litmus Test (SB)

This litmus test investigates whether or not `store` operations can be reordered. They can, thus in an unfixed state, the assert will be triggered.

B.1.1 Summary of inserted barriers

Table B.1 Store Buffer Litmus Test results for LLMC

	t1_fence			
t0_fence	relaxed	acquire	release	seq_cst
relaxed	✗	✗	✗	✗
acquire	✗	✗	✗	✗
release	✗	✗	✓	✓
seq_cst	✗	✗	✓	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

B.1.2 C and LLVM IR implementations

C implementation of SB

```
1  #include <assert.h>
2
3  int x;
4  int y;
5  int R1;
6  int R2;
7
8  int llmc_init() {
9      return 0;
10 }
11 int llmc_proc_1_t0() {
12     x = 1;
13     // t0_fence
14     R1 = y;
15     return 0;
16 }
17 int llmc_proc_1_t1() {
18     y = 1;
19     // t1_fence
20     R2 = x;
21     return 0;
22 }
23 int llmc_fini() {
24     assert(R1||R2);
25     return 0;
26 }
```

LLVM IR generated by `clang SB.c -S -flto -emit-llvm -oSB.ll -O3`

```
; ModuleID = 'SB.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64
    :64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@x = common global i32 0, align 4
@y = common global i32 0, align 4
@R1 = common global i32 0, align 4
@R2 = common global i32 0, align 4
@.str = private unnamed_addr constant [7 x i8] c"R1||R2\00", align 1
@.str1 = private unnamed_addr constant [5 x i8] c"SB.c\00", align 1
@__PRETTY_FUNCTION__.llmc_fini = private unnamed_addr constant [16 x i8] c"int llmc_fini()
    \00", align 1

; Function Attrs: nounwind readnone uwtable
define i32 @llmc_init() #0 {
  ret i32 0
}

; Function Attrs: nounwind uwtable
define i32 @llmc_proc_1_t0() #1 {
  store i32 1, i32* @x, align 4, !tbaa !0
  %1 = load i32* @y, align 4, !tbaa !0
  store i32 %1, i32* @R1, align 4, !tbaa !0
  ret i32 0
}

; Function Attrs: nounwind uwtable
define i32 @llmc_proc_1_t1() #1 {
  store i32 1, i32* @y, align 4, !tbaa !0
  %1 = load i32* @x, align 4, !tbaa !0
  store i32 %1, i32* @R2, align 4, !tbaa !0
  ret i32 0
}

; Function Attrs: nounwind uwtable
define i32 @llmc_fini() #1 {
  %1 = load i32* @R1, align 4, !tbaa !0
  %2 = load i32* @R2, align 4, !tbaa !0
  %3 = or i32 %2, %1
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %5, label %6

; <label>:5                                       ; preds = %0
  tail call void @__assert_fail(i8* getelementptr inbounds ([7 x i8]* @.str, i64 0, i64 0), i8
      * getelementptr inbounds ([5 x i8]* @.str1, i64 0, i64 0), i32 25, i8* getelementptr
      inbounds ([16 x i8]* @__PRETTY_FUNCTION__.llmc_fini, i64 0, i64 0)) #3
  unreachable

; <label>:6                                       ; preds = %0
  ret i32 0
}

; Function Attrs: noreturn nounwind
declare void @__assert_fail(i8*, i8*, i32, i8*) #2

attributes #0 = { nounwind readnone uwtable "less-precise-fpmad"="false" "no-frame-pointer-
    elim"="false" "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-
    fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="false
    " "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="
    false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #2 = { noreturn nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="
    false" "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math
    "="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #3 = { noreturn nounwind }

!0 = metadata !{metadata !"int", metadata !1}
!1 = metadata !{metadata !"omnipotent char", metadata !2}
!2 = metadata !{metadata !"Simple C/C++ TBAA"}
```

C implementation of SB (pthread version)

```
1  #include <assert.h>
2  #include <stdint.h>
3  #include <pthread.h>
4
5  int x;
6  int y;
7
8  void* t0(void* data) {
9      x = 1;
10     // t0_fence
11     return y;
12 }
13
14 void* t1(void* data) {
15     y = 1;
16     // t1_fence
17     return x;
18 }
19
20 int main(int argc, char** argv) {
21     pthread_t th1, th2;
22     int R1;
23     int R2;
24     pthread_create(&th1, 0, &t0, 0);
25     pthread_create(&th2, 0, &t1, 0);
26     pthread_join(th1, &R1);
27     pthread_join(th2, &R2);
28     assert( R1 || R2  );
29     return 0;
30 }
```

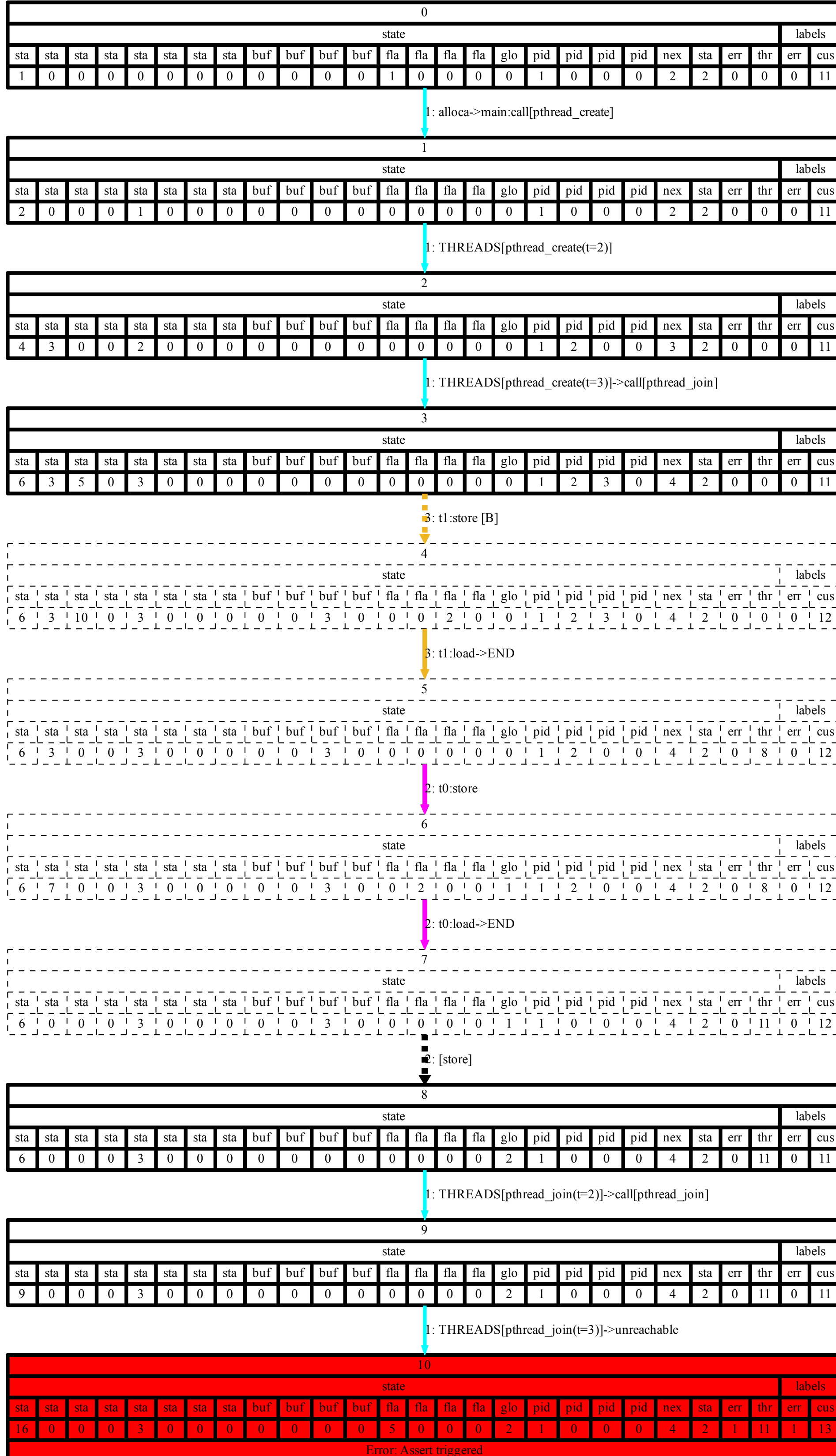
LLVM IR generated by clang SB.c -S -flto -emit-llvm -oSB.ll -O3 (pthread version)

```
1  ; ModuleID = 'pSB.c'
2  target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64
       :64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
3  target triple = "x86_64-unknown-linux-gnu"
4  %union.pthread_attr_t = type { i64, [48 x i8] }
5  @x = common global i32 0, align 4
6  @y = common global i32 0, align 4
7  @.str = private unnamed_addr constant [9 x i8] c"R1 || R2\00", align 1
8  @.str1 = private unnamed_addr constant [6 x i8] c"pSB.c\00", align 1
9  @__PRETTY_FUNCTION__.main = private unnamed_addr constant [23 x i8] c"int main(int, char **)
       \00", align 1
10 ; Function Attrs: nounwind uwtable
11 define i8* @t0(i8* nocapture %data) #0 {
12   store i32 1, i32* @x, align 4, !tbaa !0
13   %1 = load i32* @y, align 4, !tbaa !0
14   %2 = sext i32 %1 to i64
15   %3 = inttoptr i64 %2 to i8*
16   ret i8* %3
17 }
18 ; Function Attrs: nounwind uwtable
19 define i8* @t1(i8* nocapture %data) #0 {
20   store i32 1, i32* @y, align 4, !tbaa !0
21   %1 = load i32* @x, align 4, !tbaa !0
22   %2 = sext i32 %1 to i64
23   %3 = inttoptr i64 %2 to i8*
24   ret i8* %3
25 }
26
27 ; Function Attrs: nounwind uwtable
28 define i32 @main(i32 %argc, i8** nocapture %argv) #0 {
29   %th1 = alloca i64, align 8
30   %th2 = alloca i64, align 8
31   %R1 = alloca i32, align 4
32   %R2 = alloca i32, align 4
33   %1 = call i32 @pthread_create(i64* %th1, %union.pthread_attr_t* null, i8* (i8*)* @t0, i8*
       null) #4
34   %2 = call i32 @pthread_create(i64* %th2, %union.pthread_attr_t* null, i8* (i8*)* @t1, i8*
       null) #4
35   %3 = load i64* %th1, align 8, !tbaa !3
36   %4 = bitcast i32* %R1 to i8**
37   %5 = call i32 @pthread_join(i64 %3, i8** %4) #4
38   %6 = load i64* %th2, align 8, !tbaa !3
39   %7 = bitcast i32* %R2 to i8**
40   %8 = call i32 @pthread_join(i64 %6, i8** %7) #4
41   %9 = load i32* %R1, align 4, !tbaa !0
42   %10 = icmp eq i32 %9, 0
43   br i1 %10, label %11, label %15
44
45 ; <label>:11                                       ; preds = %0
46   %12 = load i32* %R2, align 4, !tbaa !0
47   %13 = icmp eq i32 %12, 0
48   br i1 %13, label %14, label %15
49
50 ; <label>:14                                       ; preds = %11
51   call void @__assert_fail(i8* getelementptr inbounds ([9 x i8]* @.str, i64 0, i64 0), i8*
         getelementptr inbounds ([6 x i8]* @.str1, i64 0, i64 0), i32 26, i8* getelementptr
         inbounds ([23 x i8]* @__PRETTY_FUNCTION__.main, i64 0, i64 0)) #5
52   unreachable
53
54 ; <label>:15                                       ; preds = %11, %0
55   ret i32 0
56 }
57
58 ; Function Attrs: nounwind
59 declare i32 @pthread_create(i64*, %union.pthread_attr_t*, i8* (i8*)*, i8*) #1
60 declare i32 @pthread_join(i64, i8**) #2
61
62 ; Function Attrs: noreturn nounwind
63 declare void @__assert_fail(i8*, i8*, i32, i8*) #3
64 attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="false
       " "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="
       false" "unsafe-fp-math"="false" "use-soft-float"="false" }
65 attributes #1 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-
       frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "
       unsafe-fp-math"="false" "use-soft-float"="false" }
66 attributes #2 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-frame-
       pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe
       -fp-math"="false" "use-soft-float"="false" }
67 attributes #3 = { noreturn nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="
       false" "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math
       "="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
68 attributes #4 = { nounwind }
69 attributes #5 = { noreturn nounwind }
70 !0 = metadata !{metadata !"int", metadata !1}
71 !1 = metadata !{metadata !"omnipotent char", metadata !2}
72 !2 = metadata !{metadata !"Simple C/C++ TBAA"}
73 !3 = metadata !{metadata !"long", metadata !1}
```

B.1.3 Traces to error

The following are traces to the error states, i.e. states where the assertion is triggered.

Figure B.1 Traces to the error states.



B.2 Load Buffer Litmus Test (LB)

This litmus test investigates whether or not `load` operations can be reordered. They can, thus in an unfixed state, the assert will be triggered.

B.2.1 Summary of inserted barriers

Table B.2 Load Buffer Litmus Test results for LLMC

	t1_fence			
t0_fence	relaxed	acquire	release	seq_cst
relaxed	✗	✗	✗	✗
acquire	✗	✓	✗	✓
release	✗	✗	✗	✗
seq_cst	✗	✓	✗	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

B.2.2 C and LLVM IR implementations

C implementation of LB

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

int x;
int y;

void* t0(void* data) {
  register int lx;
  lx = x;
  // t0_fence
  y = 1;
  return lx;
}

void* t1(void* data) {
  register int ly;
  ly = y;
  // t1_fence
  x = 1;
  return ly;
}

int main(int argc, char** argv) {
  pthread_t th1, th2;
  int R1;
  int R2;
  pthread_create(&th1, 0, &t0, 0);
  pthread_create(&th2, 0, &t1, 0);
  pthread_join(th1, &R1);
  pthread_join(th2, &R2);
  assert( !R1 || !R2  );
  return 0;
}
```

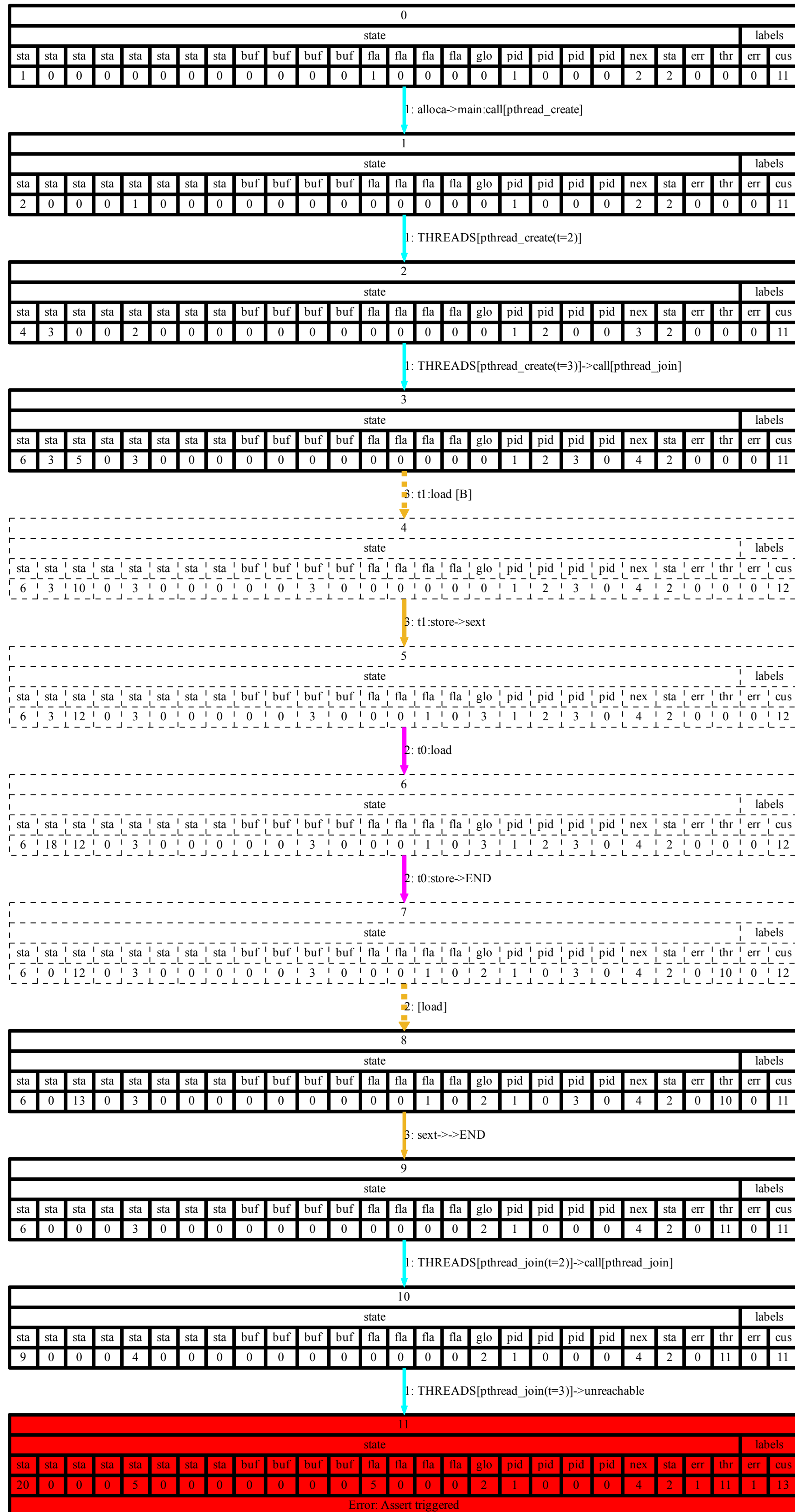
LLVM IR generated by `clang pLB.c -S -flto -emit-llvm -opLB.ll -O3`

```
; ModuleID = 'pLB.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64
     :64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
%union.pthread_attr_t = type { i64, [48 x i8] }
@x = common global i32 0, align 4
@y = common global i32 0, align 4
@.str = private unnamed_addr constant [11 x i8] c"!R1 || !R2\00", align 1
@.str1 = private unnamed_addr constant [6 x i8] c"pLB.c\00", align 1
@__PRETTY_FUNCTION__.main = private unnamed_addr constant [23 x i8] c"int main(int, char **)
     \00", align 1
; Function Attrs: nounwind uwtable
define i8* @t0(i8* nocapture %data) #0 {
  %1 = load i32* @x, align 4, !tbaa !0
  store i32 1, i32* @y, align 4, !tbaa !0
  %2 = sext i32 %1 to i64
  %3 = inttoptr i64 %2 to i8*
  ret i8* %3
}
; Function Attrs: nounwind uwtable
define i8* @t1(i8* nocapture %data) #0 {
  %1 = load i32* @y, align 4, !tbaa !0
  store i32 1, i32* @x, align 4, !tbaa !0
  %2 = sext i32 %1 to i64
  %3 = inttoptr i64 %2 to i8*
  ret i8* %3
}

; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** nocapture %argv) #0 {
  %th1 = alloca i64, align 8
  %th2 = alloca i64, align 8
  %R1 = alloca i32, align 4
  %R2 = alloca i32, align 4
  %1 = call i32 @pthread_create(i64* %th1, %union.pthread_attr_t* null, i8* (i8*)* @t0, i8*
       null) #4
  %2 = call i32 @pthread_create(i64* %th2, %union.pthread_attr_t* null, i8* (i8*)* @t1, i8*
       null) #4
  %3 = load i64* %th1, align 8, !tbaa !3
  %4 = bitcast i32* %R1 to i8**
  %5 = call i32 @pthread_join(i64 %3, i8** %4) #4
  %6 = load i64* %th2, align 8, !tbaa !3
  %7 = bitcast i32* %R2 to i8**
  %8 = call i32 @pthread_join(i64 %6, i8** %7) #4
  %9 = load i32* %R1, align 4, !tbaa !0
  %10 = icmp eq i32 %9, 0
  br i1 %10, label %15, label %11

; <label>:11                                      ; preds = %0
  %12 = load i32* %R2, align 4, !tbaa !0
  %13 = icmp eq i32 %12, 0
  br i1 %13, label %15, label %14

; <label>:14                                      ; preds = %11
  call void @__assert_fail(i8* getelementptr inbounds ([11 x i8]* @.str, i64 0, i64 0), i8*
       getelementptr inbounds ([6 x i8]* @.str1, i64 0, i64 0), i32 30, i8* getelementptr
       inbounds ([23 x i8]* @__PRETTY_FUNCTION__.main, i64 0, i64 0)) #5
  unreachable

; <label>:15                                      ; preds = %11, %0
  ret i32 0
}

; Function Attrs: nounwind
declare i32 @pthread_create(i64*, %union.pthread_attr_t*, i8* (i8*)*, i8*) #1
declare i32 @pthread_join(i64, i8**) #2

; Function Attrs: noreturn nounwind
declare void @__assert_fail(i8*, i8*, i32, i8*) #3
attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="false
     " "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="
     false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-
     frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "
     unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #2 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-frame-
     pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe
     -fp-math"="false" "use-soft-float"="false" }
attributes #3 = { noreturn nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="
     false" "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math
     "="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #4 = { nounwind }
attributes #5 = { noreturn nounwind }
!0 = metadata !{metadata !"int", metadata !1}
!1 = metadata !{metadata !"omnipotent char", metadata !2}
!2 = metadata !{metadata !"Simple C/C++ TBAA"}
!3 = metadata !{metadata !"long", metadata !1}
```

B.2.3 Traces to error

The following are traces to the error states, i.e. states where the assertion is triggered.

Figure B.2 Traces to the error states.



B.3 Dependent Load Litmus Test (DL)

This litmus test investigates whether or not dependent loads are reordered. The dependent load is in line 14: `*p`. First the value of `p` is loaded and then the value at the address `p` points to is loaded. They should never be reordered in the model we use, thus under no circumstance should the assert fail.

B.3.1 C and LLVM IR implementations

C implementation of DL

```
1  #include <assert.h>
2
3  int x;
4  int y;
5  int* p;
6
7  int llmc_init() {
8    y = 1234;
9    p = &y;
10   return 0;
11 }
12 int llmc_proc_1_t0() {
13   assert(*p==y);
14   return 0;
15 }
16 int llmc_fini() {
17   return 0;
18 }
```

LLVM IR generated by `clang DL.c -S -flto -emit-llvm -oDL.ll -O3`

```
1  ; ModuleID = 'counter.c'
2  target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0
       :64:64-f80:128:128-n8:16:32:64-S128"
3  target triple = "x86_64-unknown-linux-gnu"
4
5  @x = common global i32 0, align 4
6  @.str = private unnamed_addr constant [6 x i8] c"x==20\00", align 1
7  @.str1 = private unnamed_addr constant [10 x i8] c"counter.c\00", align 1
8  @__PRETTY_FUNCTION__.llmc_fini = private unnamed_addr constant [16 x i8] c"int llmc_fini()\00", align 1
9
10 ; Function Attrs: nounwind uwtable
11 define i32 @llmc_init() #0 {
12   store i32 0, i32* @x, align 4
13   ret i32 0
14 }
15
16 ; Function Attrs: nounwind uwtable
17 define i32 @llmc_proc_2_t0() #0 {
18   %i = alloca i32, align 4
19   store i32 0, i32* %i, align 4
20   br label %1
21
22 ; <label>:1                                       ; preds = %4, %0
23   %2 = load i32* %i, align 4
24   %3 = icmp slt i32 %2, 10
25   br i1 %3, label %4, label %7
26
27 ; <label>:4                                       ; preds = %1
28   %5 = load i32* @x, align 4
29   %6 = add nsw i32 %5, 1
30   store i32 %6, i32* @x, align 4
31   br label %1
32
33 ; <label>:7                                       ; preds = %1
34   ret i32 0
35 }
36
37 ; Function Attrs: nounwind uwtable
38 define i32 @llmc_fini() #0 {
39   %1 = load i32* @x, align 4
40   %2 = icmp eq i32 %1, 20
41   br i1 %2, label %3, label %4
42
43 ; <label>:3                                       ; preds = %0
44   br label %6
45
46 ; <label>:4                                       ; preds = %0
47   call void @__assert_fail(i8* getelementptr inbounds ([6 x i8]* @.str, i32 0, i32 0), i8* getelementptr inbounds ([10 x i8]* @.str1
         , i32 0, i32 0), i32 19, i8* getelementptr inbounds ([16 x i8]* @__PRETTY_FUNCTION__.llmc_fini, i32 0, i32 0)) #2
48   unreachable
49                                                    ; No predecessors!
50   br label %6
51
52 ; <label>:6                                       ; preds = %5, %3
53   ret i32 0
54 }
55
56 ; Function Attrs: noreturn nounwind
57 declare void @__assert_fail(i8*, i8*, i32, i8*) #1
58
59 attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"="
       true" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
60 attributes #1 = { noreturn nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"="
       true" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
61 attributes #2 = { noreturn nounwind }
```

B.4 Store propagation litmus test (IRIW)

This litmus test investigates whether or not `stores` can propagate to different processes in different orders. IRIW stands for Independent Reads of Independent Writes. Because in this version (plain IRIW) the `loads` can be reordered, the assert can be triggered. A different version with load dependency removes this and is handled differently (IRIW+addr), which is shown in Appendix B.5.

B.4.1 Summary of inserted barriers

Table B.3 Independent Reads of Independent Writes Litmus Test results for LLMC

t0_fence	t2_fence relaxed	acquire	release	seq_cst
relaxed	✗	✗	✗	✗
acquire	✗	✓	✗	✓
release	✗	✗	✗	✗
seq_cst	✗	✓	✗	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

B.4.2 C and LLVM IR implementations

C implementation of IRIW

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

int x;
int y;

void* t0(void* data) {
  int register lx,ly;
  lx = x;
  // t0_fence
  ly = y;
  return lx*10+ly;
}

void* t1(void* data) {
  y = 1;
}

void* t2(void* data) {
  int register lx,ly;
  ly = y;
  // t2_fence
  lx = x;
  return lx*10+ly;
}

int main(int argc, char** argv) {
  pthread_t th1, th2, th3;
  int R1,R2;
  pthread_create(&th1, 0, &t0, 0);
  pthread_create(&th2, 0, &t1, 0);
  pthread_create(&th3, 0, &t2, 0);
  x = 1;
  pthread_join(th1, &R1);
  pthread_join(th2, NULL);
  pthread_join(th3, &R2);
  assert(R1!=10 || R2!=1);
  return 0;
}
```

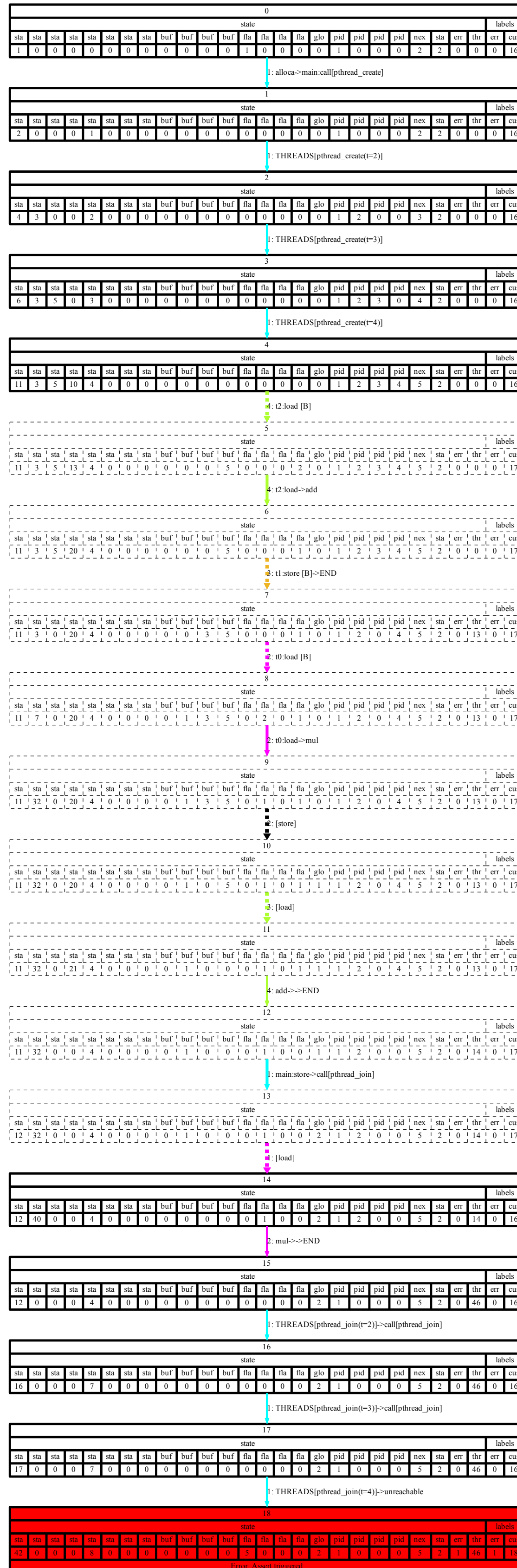
LLVM IR generated by `clang pIRIW.c -S -flto -emit-llvm -opIRIW.ll -O3`

```
1  target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64
       :64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
2  target triple = "x86_64-unknown-linux-gnu"
3  %union.pthread_attr_t = type { i64, [48 x i8] }
4  @x = common global i32 0, align 4
5  @y = common global i32 0, align 4
6  @.str = private unnamed_addr constant [16 x i8] c"R1!=10 || R2!=1\00", align 1
7  @.str1 = private unnamed_addr constant [8 x i8] c"pIRIW.c\00", align 1
8  @__PRETTY_FUNCTION__.main = private unnamed_addr constant [23 x i8] c"int main(int, char **)
       \00", align 1
9  define i8* @t0(i8* nocapture %data) #0 {
10   %1 = load i32* @x, align 4, !tbaa !0
11   %2 = load i32* @y, align 4, !tbaa !0
12   %3 = mul nsw i32 %1, 10
13   %4 = add nsw i32 %3, %2
14   %5 = sext i32 %4 to i64
15   %6 = inttoptr i64 %5 to i8*
16   ret i8* %6
17 }
18 define noalias i8* @t1(i8* nocapture %data) #1 {
19   store i32 1, i32* @y, align 4, !tbaa !0
20   ret i8* undef
21 }
22 define i8* @t2(i8* nocapture %data) #0 {
23   %1 = load i32* @y, align 4, !tbaa !0
24   %2 = load i32* @x, align 4, !tbaa !0
25   %3 = mul nsw i32 %2, 10
26   %4 = add nsw i32 %3, %1
27   %5 = sext i32 %4 to i64
28   %6 = inttoptr i64 %5 to i8*
29   ret i8* %6
30 }
31 define i32 @main(i32 %argc, i8** nocapture %argv) #1 {
32   %th1 = alloca i64, align 8
33   %th2 = alloca i64, align 8
34   %th3 = alloca i64, align 8
35   %R1 = alloca i32, align 4
36   %R2 = alloca i32, align 4
37   %1 = call i32 @pthread_create(i64* %th1, %union.pthread_attr_t* null, i8* (i8*)* @t0, i8*
         null) #5
38   %2 = call i32 @pthread_create(i64* %th2, %union.pthread_attr_t* null, i8* (i8*)* @t1, i8*
         null) #5
39   %3 = call i32 @pthread_create(i64* %th3, %union.pthread_attr_t* null, i8* (i8*)* @t2, i8*
         null) #5
40   store i32 1, i32* @x, align 4, !tbaa !0
41   %4 = load i64* %th1, align 8, !tbaa !3
42   %5 = bitcast i32* %R1 to i8**
43   %6 = call i32 @pthread_join(i64 %4, i8** %5) #5
44   %7 = load i64* %th2, align 8, !tbaa !3
45   %8 = call i32 @pthread_join(i64 %7, i8** null) #5
46   %9 = load i64* %th3, align 8, !tbaa !3
47   %10 = bitcast i32* %R2 to i8**
48   %11 = call i32 @pthread_join(i64 %9, i8** %10) #5
49   %12 = load i32* %R1, align 4, !tbaa !0
50   %13 = icmp eq i32 %12, 10
51   br i1 %13, label %14, label %18
52 ; <label>:14                                       ; preds = %0
53   %15 = load i32* %R2, align 4, !tbaa !0
54   %16 = icmp eq i32 %15, 1
55   br i1 %16, label %17, label %18
56 ; <label>:17                                       ; preds = %14
57   call void @__assert_fail(i8* getelementptr inbounds ([16 x i8]* @.str, i64 0, i64 0), i8*
         getelementptr inbounds ([8 x i8]* @.str1, i64 0, i64 0), i32 36, i8* getelementptr
         inbounds ([23 x i8]* @__PRETTY_FUNCTION__.main, i64 0, i64 0)) #6
58   unreachable
59 ; <label>:18                                       ; preds = %14, %0
60   ret i32 0
61 }
62 declare i32 @pthread_create(i64*, %union.pthread_attr_t*, i8* (i8*)*, i8*) #2
63 declare i32 @pthread_join(i64, i8**) #3
64 declare void @__assert_fail(i8*, i8*, i32, i8*) #4
65 attributes #0 = { nounwind readonly uwtable "less-precise-fpmad"="false" "no-frame-pointer-
       elim"="false" "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-
       fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
66 attributes #1 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="false
       " "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="
       false" "unsafe-fp-math"="false" "use-soft-float"="false" }
67 attributes #2 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-
       frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "
       unsafe-fp-math"="false" "use-soft-float"="false" }
68 attributes #3 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-frame-
       pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe
       -fp-math"="false" "use-soft-float"="false" }
69 attributes #4 = { noreturn nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="
       false" "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math
       "="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
70 attributes #5 = { nounwind }
71 attributes #6 = { noreturn nounwind }
72 !0 = metadata !{metadata !"int", metadata !1}
73 !1 = metadata !{metadata !"omnipotent char", metadata !2}
74 !2 = metadata !{metadata !"Simple C/C++ TBAA"}
75 !3 = metadata !{metadata !"long", metadata !1}
```

B.4.3 Traces to error

The following are traces to the error states, i.e. states where the assertion is triggered.

Figure B.3 Traces to the error states.



B.5 Store propagation and Load Dependency litmus test (IRIW+addr)

This litmus test investigates whether or not `stores` can propagate to different processes in different orders. IRIW stands for Independent Reads of Independent Writes. We assume a coherent and causal cache, thus the assert will not trigger.

B.5.1 Summary of inserted barriers

Table B.4 Independent Reads of Independent Writes with Load Dependency Litmus Test results for LLMC

	t2_fence			
t0_fence	relaxed	acquire	release	seq_cst
relaxed	✓	✓	✓	✓
acquire	✓	✓	✓	✓
release	✓	✓	✓	✓
seq_cst	✓	✓	✓	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

B.5.2 C and LLVM IR implementations

C implementation of IRIW+addr

```
1  #include <assert.h>
2  #include <stdint.h>
3  #include <pthread.h>
4
5  int x;
6  int y;
7
8  void* t0(void* data) {
9    int register lx,ly;
10   lx = x;
11   // t0_fence
12   ly = *(&y+lx-lx);
13   return lx*10+ly;
14 }
15
16 void* t1(void* data) {
17   y = 1;
18 }
19
20 void* t2(void* data) {
21   int register lx,ly;
22   ly = y;
23   // t1_fence
24   lx = *(&x+ly-ly);
25   return lx*10+ly;
26 }
27
28 int main(int argc, char** argv) {
29   pthread_t th1, th2, th3;
30   int R1,R2;
31   pthread_create(&th1, 0, &t0, 0);
32   pthread_create(&th2, 0, &t1, 0);
33   pthread_create(&th3, 0, &t2, 0);
34   x = 1;
35   pthread_join(th1, &R1);
36   pthread_join(th2, NULL);
37   pthread_join(th3, &R2);
38   assert(R1!=10 || R2!=1);
39   return 0;
40 }
```

LLVM IR generated by `clang pIRIWaddr.c -S -flto -emit-llvm -opIRIWaddr.ll -O3`

```
; ModuleID = 'pIRIWaddr.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
%union.pthread_attr_t = type { i64, [48 x i8] }
@x = common global i32 0, align 4
@y = common global i32 0, align 4
@.str = private unnamed_addr constant [16 x i8] c"R1!=10 || R2!=1\00", align 1
@.str1 = private unnamed_addr constant [12 x i8] c"pIRIWaddr.c\00", align 1
@__PRETTY_FUNCTION__.main = private unnamed_addr constant [23 x i8] c"int main(int, char **)\00", align 1
; Function Attrs: nounwind uwtable
define i8* @t0(i8* %data) #0 {
  %1 = alloca i8*, align 8
  %lx = alloca i32, align 4
  %ly = alloca i32, align 4
  store i8* %data, i8** %1, align 8
  %2 = load i32* @x, align 4
  store i32 %2, i32* %lx, align 4
  %3 = load i32* %lx, align 4
  %4 = sext i32 %3 to i64
  %5 = getelementptr inbounds i32* @y, i64 %4
  %6 = load i32* %lx, align 4
  %7 = sext i32 %6 to i64
  %8 = sub i64 0, %7
  %9 = getelementptr inbounds i32* %5, i64 %8
  %10 = load i32* %9, align 4
  store i32 %10, i32* %ly, align 4
  %11 = load i32* %lx, align 4
  %12 = mul nsw i32 %11, 10
  %13 = load i32* %ly, align 4
  %14 = add nsw i32 %12, %13
  %15 = sext i32 %14 to i64
  %16 = inttoptr i64 %15 to i8*
  ret i8* %16
}
; Function Attrs: nounwind uwtable
define i8* @t1(i8* %data) #0 {
  %1 = alloca i8*, align 8
  %2 = alloca i8*, align 8
  store i8* %data, i8** %2, align 8
  store i32 1, i32* @y, align 4
  %3 = load i8** %1
  ret i8* %3
}
; Function Attrs: nounwind uwtable
define i8* @t2(i8* %data) #0 {
  %1 = alloca i8*, align 8
  %lx = alloca i32, align 4
  %ly = alloca i32, align 4
  store i8* %data, i8** %1, align 8
  %2 = load i32* @y, align 4
  store i32 %2, i32* %ly, align 4
  %3 = load i32* %ly, align 4
  %4 = sext i32 %3 to i64
  %5 = getelementptr inbounds i32* @x, i64 %4
  %6 = load i32* %ly, align 4
  %7 = sext i32 %6 to i64
  %8 = sub i64 0, %7
  %9 = getelementptr inbounds i32* %5, i64 %8
  %10 = load i32* %9, align 4
  store i32 %10, i32* %lx, align 4
  %11 = load i32* %lx, align 4
  %12 = mul nsw i32 %11, 10
  %13 = load i32* %ly, align 4
  %14 = add nsw i32 %12, %13
  %15 = sext i32 %14 to i64
  %16 = inttoptr i64 %15 to i8*
  ret i8* %16
}
; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** %argv) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
  %th1 = alloca i64, align 8
  %th2 = alloca i64, align 8
  %th3 = alloca i64, align 8
  %R1 = alloca i32, align 4
  %R2 = alloca i32, align 4
  store i32 0, i32* %1
  store i32 %argc, i32* %2, align 4
  store i8** %argv, i8*** %3, align 8
  %4 = call i32 @pthread_create(i64* %th1, %union.pthread_attr_t* null, i8* (i8*)* @t0, i8* null) #4
  %5 = call i32 @pthread_create(i64* %th2, %union.pthread_attr_t* null, i8* (i8*)* @t1, i8* null) #4
  %6 = call i32 @pthread_create(i64* %th3, %union.pthread_attr_t* null, i8* (i8*)* @t2, i8* null) #4
  store i32 1, i32* @x, align 4
  %7 = load i64* %th1, align 8
  %8 = bitcast i32* %R1 to i8**
  %9 = call i32 @pthread_join(i64 %7, i8** %8)
  %10 = load i64* %th2, align 8
  %11 = call i32 @pthread_join(i64 %10, i8** null)
  %12 = load i64* %th3, align 8
  %13 = bitcast i32* %R2 to i8**
  %14 = call i32 @pthread_join(i64 %12, i8** %13)
  %15 = load i32* %R1, align 4
  %16 = icmp ne i32 %15, 10
  br i1 %16, label %20, label %17
  %18 = load i32* %R2, align 4
  %19 = icmp ne i32 %18, 1
  br i1 %19, label %20, label %21
  br label %23
  call void @__assert_fail(i8* getelementptr inbounds ([16 x i8]* @.str, i32 0, i32 0), i8* getelementptr inbounds ([12 x i8]* @.str1, i32 0, i32 0), i32 36, i8* getelementptr inbounds ([23 x i8]* @__PRETTY_FUNCTION__.main, i32 0, i32 0)) #5
  unreachable
  br label %23
  ret i32 0
}
; Function Attrs: nounwind
declare i32 @pthread_create(i64*, %union.pthread_attr_t*, i8* (i8*)*, i8*) #1
declare i32 @pthread_join(i64, i8**) #2
; Function Attrs: noreturn nounwind
declare void @__assert_fail(i8*, i8*, i32, i8*) #3
attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"="true" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"="true" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #2 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"="true" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #3 = { noreturn nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"="true" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #4 = { nounwind }
attributes #5 = { noreturn nounwind }
```

B.6 Message Passing Litmus Test (MP)

This litmus test investigates the order of propagation of `store` operations to matching `load` operations.

B.6.1 Summary of inserted barriers

Table B.5 Message Passing Litmus Test results for LLMC

	t1_fence			
t0_fence	relaxed	acquire	release	seq_cst
relaxed	✗	✗	✗	✗
acquire	✗	✗	✗	✗
release	✗	✓	✗	✓
seq_cst	✗	✓	✗	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

B.6.2 C and LLVM IR implementations

C implementation of MP

```
1  #include <assert.h>
2  #include <stdint.h>
3  #include <pthread.h>
4
5  int x;
6  int y;
7
8  void* t0(void* data) {
9      x = 1;
10     // t0_fence
11     y = 1;
12     return 0;
13 }
14
15 void* t1(void* data) {
16     register int lx,ly;
17     ly = y;
18     // t1_fence
19     lx = x;
20     assert(!ly || lx);
21     return 0;
22 }
23
24 int main(int argc, char** argv) {
25     pthread_t th1, th2;
26     int R;
27     pthread_create(&th1, 0, &t0, 0);
28     pthread_create(&th2, 0, &t1, 0);
29     pthread_join(th1, NULL);
30     pthread_join(th2, &R);
31     return 0;
32 }
```

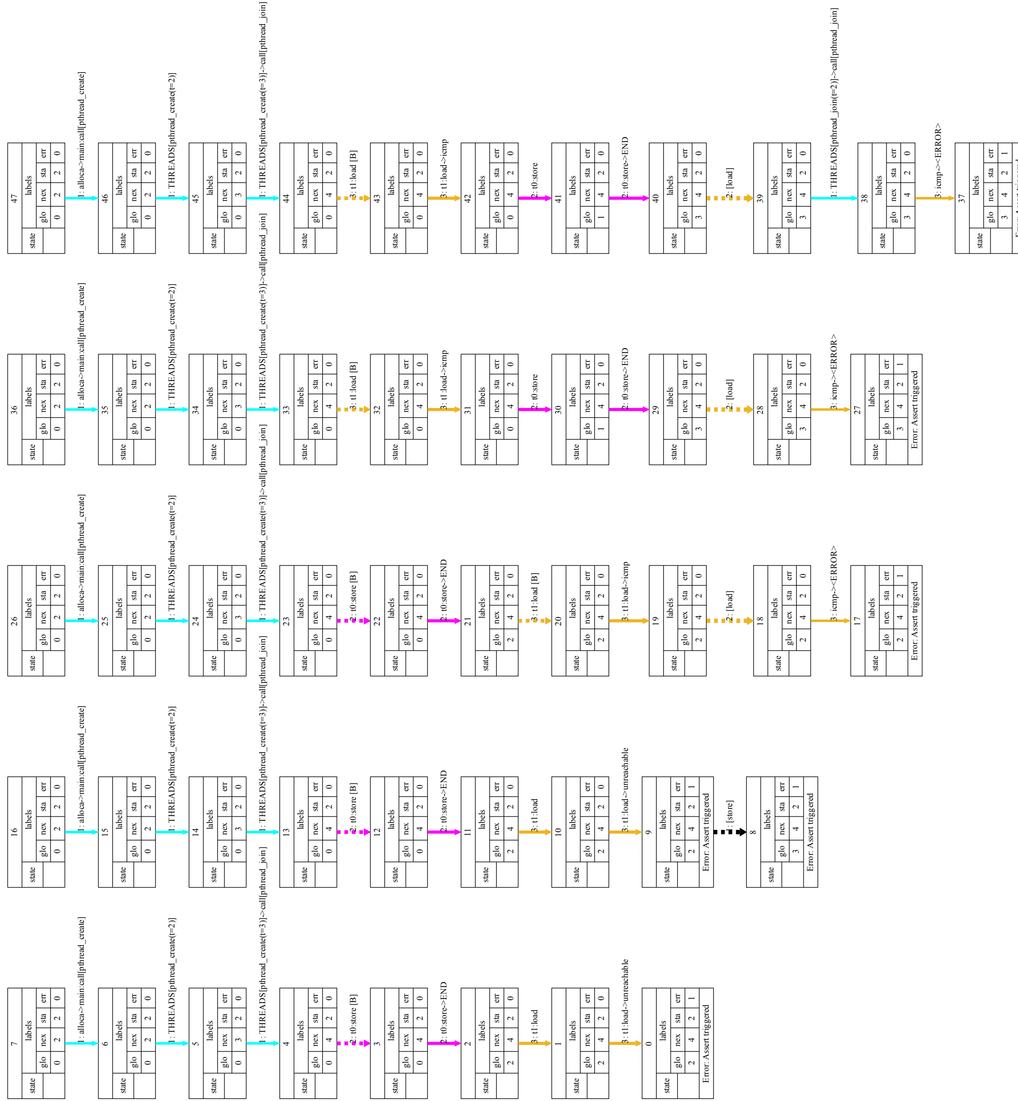
LLVM IR generated by `clang MP.c -S -flto -emit-llvm -oMP.ll -O3`

```
1  ; ModuleID = 'pMP.c'
2  target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64
       :64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
3  target triple = "x86_64-unknown-linux-gnu"
4
5  %union.pthread_attr_t = type { i64, [48 x i8] }
6
7  @x = common global i32 0, align 4
8  @y = common global i32 0, align 4
9  @.str = private unnamed_addr constant [10 x i8] c"!1y || 1x\00", align 1
10 @.str1 = private unnamed_addr constant [6 x i8] c"pMP.c\00", align 1
11 @__PRETTY_FUNCTION__.t1 = private unnamed_addr constant [17 x i8] c"void *t1(void *)\00",
       align 1
12
13 ; Function Attrs: nounwind uwtable
14 define noalias i8* @t0(i8* nocapture %data) #0 {
15   store i32 1, i32* @x, align 4, !tbaa !0
16   store i32 1, i32* @y, align 4, !tbaa !0
17   ret i8* null
18 }
19
20 ; Function Attrs: nounwind uwtable
21 define noalias i8* @t1(i8* nocapture %data) #0 {
22   %1 = load i32* @y, align 4, !tbaa !0
23   %2 = load i32* @x, align 4, !tbaa !0 ; manual intervention was required
24   %3 = icmp ne i32 %1, 0               ; to swap these two operations
25   %4 = icmp eq i32 %2, 0
26   %or.cond = and i1 %3, %4
27   br i1 %or.cond, label %5, label %6
28
29 ; <label>:5                                       ; preds = %0
30   tail call void @__assert_fail(i8* getelementptr inbounds ([10 x i8]* @.str, i64 0, i64 0),
         i8* getelementptr inbounds ([6 x i8]* @.str1, i64 0, i64 0), i32 19, i8* getelementptr
         inbounds ([17 x i8]* @__PRETTY_FUNCTION__.t1, i64 0, i64 0)) #5
31   unreachable
32
33 ; <label>:6                                       ; preds = %0
34   ret i8* null
35 }
36
37 ; Function Attrs: noreturn nounwind
38 declare void @__assert_fail(i8*, i8*, i32, i8*) #2
39
40 ; Function Attrs: nounwind uwtable
41 define i32 @main(i32 %argc, i8** nocapture %argv) #0 {
42   %th1 = alloca i64, align 8
43   %th2 = alloca i64, align 8
44   %R = alloca i32, align 4
45   %1 = call i32 @pthread_create(i64* %th1, %union.pthread_attr_t* null, i8* (i8*)* @t0, i8*
         null) #4
46   %2 = call i32 @pthread_create(i64* %th2, %union.pthread_attr_t* null, i8* (i8*)* @t1, i8*
         null) #4
47   %3 = load i64* %th1, align 8, !tbaa !3
48   %4 = call i32 @pthread_join(i64 %3, i8** null) #4
49   %5 = load i64* %th2, align 8, !tbaa !3
50   %6 = bitcast i32* %R to i8**
51   %7 = call i32 @pthread_join(i64 %5, i8** %6) #4
52   ret i32 0
53 }
54
55 ; Function Attrs: nounwind
56 declare i32 @pthread_create(i64*, %union.pthread_attr_t*, i8* (i8*)*, i8*) #3
57
58 declare i32 @pthread_join(i64, i8**) #1
59
60 attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="false
       " "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="
       false" "unsafe-fp-math"="false" "use-soft-float"="false" }
61 attributes #1 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-frame-
       pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe
       -fp-math"="false" "use-soft-float"="false" }
62 attributes #2 = { noreturn nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="
       false" "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math
       "="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
63 attributes #3 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-
       frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "
       unsafe-fp-math"="false" "use-soft-float"="false" }
64 attributes #4 = { nounwind }
65 attributes #5 = { noreturn nounwind }
66
67 !0 = metadata !{metadata !"int", metadata !1}
68 !1 = metadata !{metadata !"omnipotent char", metadata !2}
69 !2 = metadata !{metadata !"Simple C/C++ TBAA"}
70 !3 = metadata !{metadata !"long", metadata !1}
```

B.6.3 Traces to error

The following are traces to the error states, i.e. states where the assertion is triggered.

Figure B.4 Traces to the error states.



B.7 Message Passing Litmus Test with dependency (MP-dep)

This litmus test investigates the order of propagation of `store` operations to matching `load` operations. This illustrates the difference between LMO and RMO: The increment in line 21 has a dependent load, which is always executed before the load after the increment.

B.7.1 Summary of inserted barriers

Table B.6 Message Passing Litmus Test with dependency results for LLMC

	t1_fence			
t0_fence	relaxed	acquire	release	seq_cst
relaxed	✗	✗	✗	✗
acquire	✗	✗	✗	✗
release	✓	✓	✓	✓
seq_cst	✓	✓	✓	✓

✗: assertion triggered, ✓: assertion *not* triggered
✗✓: correct answer, ✗✓: incorrect answer

B.7.2 C and LLVM IR implementations

C implementation of MP-dep

```
1  #include <assert.h>
2  #include <stdint.h>
3  #include <pthread.h>
4
5  volatile int x;
6  volatile int y;
7  volatile void llmc_barrier_ss();
8  volatile void llmc_barrier_ll();
9
10 void* t0(void* data) {
11   x = 1;
12   // t0_fence
13   y = 1;
14   return 0;
15 }
16
17 void* t1(void* data) {
18   volatile int lx,ly;
19   ly = y;
20   // t1_fence
21   ly++;
22   lx = x;
23   ly--;
24   assert(!ly || lx);
25   return 0;
26 }
27
28 int main(int argc, char** argv) {
29   pthread_t th1, th2;
30   int R;
31   pthread_create(&th1, 0, &t0, 0);
32   pthread_create(&th2, 0, &t1, 0);
33   pthread_join(th1, NULL);
34   pthread_join(th2, &R);
35   return 0;
36 }
```

LLVM IR generated by `clang MP-dep.c -S -flto -emit-llvm -oMP-dep.ll -O3`

```
1  ; ModuleID = 'pMP-dep.c'
2  target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64
       :64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
3  target triple = "x86_64-unknown-linux-gnu"
4
5  %union.pthread_attr_t = type { i64, [48 x i8] }
6
7  @x = common global i32 0, align 4
8  @y = common global i32 0, align 4
9  @.str = private unnamed_addr constant [10 x i8] c"!ly || lx\00", align 1
10 @.str1 = private unnamed_addr constant [10 x i8] c"pMP-dep.c\00", align 1
11 @__PRETTY_FUNCTION__.t1 = private unnamed_addr constant [17 x i8] c"void *t1(void *)\00",
       align 1
12
13 ; Function Attrs: nounwind uwtable
14 define noalias i8* @t0(i8* nocapture %data) #0 {
15   store volatile i32 1, i32* @x, align 4, !tbaa !0
16   tail call void (...)* @llmc_barrier_ss() #4
17   store volatile i32 1, i32* @y, align 4, !tbaa !0
18   ret i8* null
19 }
20
21 declare void @llmc_barrier_ss(...) #1
22
23 ; Function Attrs: nounwind uwtable
24 define noalias i8* @t1(i8* nocapture %data) #0 {
25   %lx = alloca i32, align 4
26   %ly = alloca i32, align 4
27   %1 = load volatile i32* @y, align 4, !tbaa !0
28   store volatile i32 %1, i32* %ly, align 4
29   %2 = load volatile i32* @x, align 4, !tbaa !0
30   store volatile i32 %2, i32* %lx, align 4
31   %3 = load volatile i32* %ly, align 4
32   %4 = icmp eq i32 %3, 0
33   br i1 %4, label %9, label %5
34 ; <label>:5                                       ; preds = %0
35   %6 = load volatile i32* %lx, align 4
36   %7 = icmp eq i32 %6, 0
37   br i1 %7, label %8, label %9
38 ; <label>:8                                       ; preds = %5
39   call void @__assert_fail(i8* getelementptr inbounds ([10 x i8]* @.str, i64 0, i64 0), i8*
         getelementptr inbounds ([10 x i8]* @.str1, i64 0, i64 0), i32 23, i8* getelementptr
         inbounds ([17 x i8]* @__PRETTY_FUNCTION__.t1, i64 0, i64 0)) #5
40   unreachable
41 ; <label>:9                                       ; preds = %5, %0
42   ret i8* null
43 }
44
45 ; Function Attrs: noreturn nounwind
46 declare void @__assert_fail(i8*, i8*, i32, i8*) #2
47 ; Function Attrs: nounwind uwtable
48 define i32 @main(i32 %argc, i8** nocapture %argv) #0 {
49   %th1 = alloca i64, align 8
50   %th2 = alloca i64, align 8
51   %R = alloca i32, align 4
52   %1 = call i32 @pthread_create(i64* %th1, %union.pthread_attr_t* null, i8* (i8*)* @t0, i8*
         null) #4
53   %2 = call i32 @pthread_create(i64* %th2, %union.pthread_attr_t* null, i8* (i8*)* @t1, i8*
         null) #4
54   %3 = load i64* %th1, align 8, !tbaa !3
55   %4 = call i32 @pthread_join(i64 %3, i8** null) #4
56   %5 = load i64* %th2, align 8, !tbaa !3
57   %6 = bitcast i32* %R to i8**
58   %7 = call i32 @pthread_join(i64 %5, i8** %6) #4
59   ret i32 0
60 }
61 ; Function Attrs: nounwind
62 declare i32 @pthread_create(i64*, %union.pthread_attr_t*, i8* (i8*)*, i8*) #3
63 declare i32 @pthread_join(i64, i8**) #1
64 attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="false
       " "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="
       false" "unsafe-fp-math"="false" "use-soft-float"="false" }
65 attributes #1 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-frame-
       pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe
       -fp-math"="false" "use-soft-float"="false" }
66 attributes #2 = { noreturn nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="
       false" "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math
       "="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
67 attributes #3 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-
       frame-pointer-elim-non-leaf"="false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "
       unsafe-fp-math"="false" "use-soft-float"="false" }
68 attributes #4 = { nounwind }
69 attributes #5 = { noreturn nounwind }
70 !0 = metadata !{metadata !"int", metadata !1}
71 !1 = metadata !{metadata !"omnipotent char", metadata !2}
72 !2 = metadata !{metadata !"Simple C/C++ TBAA"}
73 !3 = metadata !{metadata !"long", metadata !1}
```

Implementations of Experiments

C.1 Concurrent counting

These are C++ and LLVM IR implementation of multiple threads concurrently incrementing a single atomic integer. They are discussed in Section 5.2.1.

C++ implementation

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

#define THREADS 3
#define INCS 2

extern "C" {
volatile void llmc_barrier_release();
volatile void llmc_barrier_acquire();
volatile bool __atomic_compare_exchange_llmc(int, void*, void*, void const*, std::memory_order ms, std::memory_order mf);
}

namespace llmc {
template<typename T>
class atomic {
private:
  T data;
public:
  void operator=(T const& d) volatile {
    llmc_barrier_release();
    data = d;
  }
  void store(T const& d, std::memory_order model = std::memory_order_seq_cst) volatile {
    if(model==std::memory_order_release || model==std::memory_order_seq_cst)
      llmc_barrier_release();
    data = d;
  }
  T load(std::memory_order model = std::memory_order_seq_cst) volatile {
    T d = data;
    if(model==std::memory_order_acquire || model==std::memory_order_seq_cst)
      llmc_barrier_acquire();
    return d;
  }
  bool compare_exchange_strong(T& expected, T const& desired) volatile {
    return __atomic_compare_exchange_llmc(sizeof(T), (void*)&data, (void*)&expected, (void*)&desired, std::memory_order_relaxed, std
        ::memory_order_relaxed);
  }
  bool compare_exchange_weak(T& expected, T const& desired) volatile {
    return __atomic_compare_exchange_llmc(sizeof(T), (void*)&data, (void*)&expected, (void*)&desired, std::memory_order_relaxed, std
        ::memory_order_relaxed);
  }
};
}

volatile llmc::atomic<int> atomicInt;

void* tadd(void* data) {
  int i=0;
  while(i<INCS) {
    int j = i;
    while(!atomicInt.compare_exchange_weak(j,j+1));
    i++;
  }
  return NULL;
}

int main(int argc, char** argv) {
  atomicInt = 0;
  pthread_t t[4];
  for(int i=THREADS-1; i--;) {
    pthread_create(&t[i], 0, &tadd, 0);
  }
  tadd(NULL);
  for(int i=THREADS-1; i--;) {
    pthread_join(t[i], nullptr);
  }
  assert(atomicInt.load()==THREADS*INCS);
}
```

LLVM IR generated by `clang tadd.cpp -std=c++11 -pthread -S -flto -emit-llvm -otadd.ll -O3`

```
; ModuleID = 'tadd.cpp'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0
    :64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

%"class.llmc::atomic" = type { i32 }
%union.pthread_attr_t = type { i64, [48 x i8] }

@atomicInt = global %"class.llmc::atomic" zeroinitializer, align 4
@.str = private unnamed_addr constant [22 x i8] c"atomicInt.load()==2*1\00", align 1
@.str1 = private unnamed_addr constant [9 x i8] c"tadd.cpp\00", align 1
@__PRETTY_FUNCTION__.main = private unnamed_addr constant [23 x i8] c"int main(int, char **)\00", align 1

; Function Attrs: uwtable
define noalias i8* @_Z4taddPv(i8* nocapture %data) #0 {
  %j = alloca i32, align 4
  %1 = alloca i32, align 4
  %2 = bitcast i32* %j to i8*
  %3 = bitcast i32* %1 to i8*
  store i32 0, i32* %j, align 4, !tbaa !0
  br label %4

; <label>:4                                       ; preds = %._crit_edge, %0
  %5 = phi i32 [ %phitmp, %._crit_edge ], [ 1, %0 ]
  store i32 %5, i32* %1, align 4, !tbaa !0
  %6 = call zeroext i1 @__atomic_compare_exchange_llmc(i32 4, i8* bitcast (%"class.llmc::atomic"* @atomicInt to i8*), i8* %2, i8*
      %3, i32 0, i32 0)
  br i1 %6, label %7, label %._crit_edge

._crit_edge:                                      ; preds = %4
  %.pre = load i32* %j, align 4, !tbaa !0
  %phitmp = add i32 %.pre, 1
  br label %4

; <label>:7                                       ; preds = %4
  ret i8* null
}

; Function Attrs: uwtable
define i32 @main(i32 %argc, i8** nocapture %argv) #0 {
  %j.i = alloca i32, align 4
  %1 = alloca i32, align 4
  %t = alloca [4 x i64], align 16
  call void @llmc_barrier_release()
  store volatile i32 0, i32* getelementptr inbounds (%"class.llmc::atomic"* @atomicInt, i64 0, i32 0), align 4, !tbaa !0
  %2 = getelementptr inbounds [4 x i64]* %t, i64 0, i64 0
  %3 = call i32 @pthread_create(i64* %2, %union.pthread_attr_t* null, i8* (i8*)* @_Z4taddPv, i8* null) #4
  %4 = bitcast i32* %j.i to i8*
  call void @llvm.lifetime.start(i64 4, i8* %4)
  %5 = bitcast i32* %1 to i8*
  call void @llvm.lifetime.start(i64 4, i8* %5)
  store i32 0, i32* %j.i, align 4, !tbaa !0
  store i32 1, i32* %1, align 4, !tbaa !0
  %6 = call zeroext i1 @__atomic_compare_exchange_llmc(i32 4, i8* bitcast (%"class.llmc::atomic"* @atomicInt to i8*), i8* %4, i8*
      %5, i32 0, i32 0)
  br i1 %6, label %_Z4taddPv.exit.preheader, label %._crit_edge.i

_Z4taddPv.exit.preheader:                         ; preds = %._crit_edge.i, %0
  %7 = load i64* %2, align 16, !tbaa !3
  %8 = call i32 @pthread_join(i64 %7, i8** null)
  %9 = load volatile i32* getelementptr inbounds (%"class.llmc::atomic"* @atomicInt, i64 0, i32 0), align 4, !tbaa !0
  call void @llmc_barrier_acquire()
  %10 = icmp eq i32 %9, 2
  br i1 %10, label %13, label %12

._crit_edge.i:                                    ; preds = %0, %._crit_edge.i
  %.pre.i = load i32* %j.i, align 4, !tbaa !0
  %phitmp.i = add i32 %.pre.i, 1
  store i32 %phitmp.i, i32* %1, align 4, !tbaa !0
  %11 = call zeroext i1 @__atomic_compare_exchange_llmc(i32 4, i8* bitcast (%"class.llmc::atomic"* @atomicInt to i8*), i8* %4, i8*
      %5, i32 0, i32 0)
  br i1 %11, label %_Z4taddPv.exit.preheader, label %._crit_edge.i

; <label>:12                                      ; preds = %_Z4taddPv.exit.preheader
  call void @__assert_fail(i8* getelementptr inbounds ([22 x i8]* @.str, i64 0, i64 0), i8* getelementptr inbounds ([9 x i8]* @.str1
      , i64 0, i64 0), i32 73, i8* getelementptr inbounds ([23 x i8]* @__PRETTY_FUNCTION__.main, i64 0, i64 0)) #5
  unreachable

; <label>:13                                      ; preds = %_Z4taddPv.exit.preheader
  ret i32 0
}

; Function Attrs: nounwind
declare i32 @pthread_create(i64*, %union.pthread_attr_t*, i8* (i8*)*, i8*) #1

declare i32 @pthread_join(i64, i8**) #2

; Function Attrs: noreturn nounwind
declare void @__assert_fail(i8*, i8*, i32, i8*) #3

declare void @llmc_barrier_acquire() #2

declare void @llmc_barrier_release() #2

declare zeroext i1 @__atomic_compare_exchange_llmc(i32, i8*, i8*, i8*, i32, i32) #2

; Function Attrs: nounwind
declare void @llvm.lifetime.start(i64, i8* nocapture) #4

attributes #0 = { uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-frame-pointer-elim-non-leaf"="false" "no-
    infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-frame-pointer-elim-non-leaf"="false" "no
    -infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #2 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-frame-pointer-elim-non-leaf"="false" "no-infs-fp-
    math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #3 = { noreturn nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="false" "no-frame-pointer-elim-non-leaf"="
    false" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #4 = { nounwind }
attributes #5 = { noreturn nounwind }

!0 = metadata !{metadata !"int", metadata !1}
!1 = metadata !{metadata !"omnipotent char", metadata !2}
!2 = metadata !{metadata !"Simple C/C++ TBAA"}
!3 = metadata !{metadata !"long", metadata !1}
```

C.2 Michael-Scott queue

This is a C++ implementation of the Michael-Scott queue. Due to its size, we omit the LLVM IR implementation. They are discussed in Section 5.2.2.

C++ implementation: headers, atomics, node (part 1/3)

```
1   #include <pthread.h>
2   #include <atomic>
3   #include <assert.h>
4
5   extern "C" {
6
7   volatile void llmc_assume(bool b);
8   volatile void llmc_barrier_seq_cst();
9   volatile void llmc_barrier_release();
10  volatile void llmc_barrier_acquire();
11  volatile void llmc_atomic_begin();
12  volatile void llmc_atomic_end();
13  void memcpy(void*, void*, int);
14  __uint32_t memcmp(void*, void*, int);
15  volatile bool __atomic_compare_exchange_llmc(int s, void* t, void* e, void const* d, std::memory_order ms, std::memory_order mf);
16
17  }
18
19  struct Node;
20
21  struct NodePtr {
22  public:
23    Node* ptr;
24    intptr_t count;
25
26    bool operator==(NodePtr const& other) {
27      return this->ptr == other.ptr
28        && this->count == other.count;
29    }
30
31    NodePtr(Node* volatile const& ptr, intptr_t const& count)
32      : ptr(ptr)
33      , count(count)
34      {
35    }
36
37    NodePtr(Node* const& ptr, intptr_t const& count)
38      : ptr(ptr)
39      , count(count)
40      {
41    }
42
43    NodePtr()
44      : ptr(nullptr)
45      , count(0)
46      {
47    }
48
49    volatile NodePtr& operator=(NodePtr const& d) volatile {
50      memcpy((void*)this,(void*)&d,sizeof(d));
51      return *this;
52    }
53
54    void store(NodePtr const& d, std::memory_order model = std::memory_order_seq_cst) volatile  {
55      if(model==std::memory_order_release || model==std::memory_order_seq_cst)
56        llmc_barrier_release();
57      memcpy((void*)this,(void*)&d,sizeof(d));
58    }
59    NodePtr load(std::memory_order model = std::memory_order_seq_cst) volatile {
60      NodePtr d;
61      load(d,model);
62      return d;
63    }
64    void load(NodePtr& d, std::memory_order model = std::memory_order_seq_cst) volatile {
65      memcpy((void*)&d,(void*)this,sizeof(d));
66      if(model==std::memory_order_acquire || model==std::memory_order_seq_cst)
67        llmc_barrier_acquire();
68    }
69    bool compare_exchange_strong(NodePtr volatile& expected, NodePtr const& desired, std::memory_order const& mo = std::
          memory_order_seq_cst) volatile {
70      return __atomic_compare_exchange_llmc(sizeof(NodePtr), (void*)this, (void*)&expected, (void*)&desired, mo, mo);
71    }
72    bool compare_exchange_weak(NodePtr volatile& expected, NodePtr const& desired, std::memory_order const& mo = std::
          memory_order_seq_cst) volatile {
73      return __atomic_compare_exchange_llmc(sizeof(NodePtr), (void*)this, (void*)&expected, (void*)&desired, mo, mo);
74    }
75
76  };
77
78  struct Node {
79  public:
80    NodePtr next __attribute__ ((aligned (16)));
81    int data;
82    Node(): next(), data() {}
83  } __attribute__ ((aligned (16)));
84  int const MAX_NODES = 10;
85  Node globalMemory[MAX_NODES] __attribute__ ((aligned (16)));
86  bool inUse[MAX_NODES] __attribute__ ((aligned (16)));
87
88  Node* newNode() {
89    int i=0;
90    bool F = false;
91    bool T = true;
92    do {
93      i++;
94      assert(i<MAX_NODES);
95      F = false;
96    } while(!__atomic_compare_exchange_llmc(sizeof(bool), &inUse[i], &F, &T, std::memory_order_relaxed, std::memory_order_relaxed));
97    return &globalMemory[i];
98  }
99  void freeNode(Node* node) {
100   inUse[node-globalMemory] = false;
101 }
```

C++ implementation: Michael-Scott queue (part 2/3)

```
struct MSQ {
public:
  volatile NodePtr Head;
  volatile NodePtr Tail;

public:
  MSQ() {
    Node* node = &globalMemory[1];
    inUse[1] = true;
    node->next = NodePtr(nullptr, 0);
    Head = NodePtr(node, 0);
    Tail = NodePtr(node, 0);
    assert(Head.ptr==Tail.ptr);
  }

  void enqueue(int const& data, int nid) {
    volatile NodePtr tail;
    volatile NodePtr next;
    volatile NodePtr tail2;
    Node* node = newNode();//&globalMemory[nid];
    node->data = data;
    node->next.ptr = nullptr;
    int i=0;
    llmc_barrier_release(); // Can find this one: E || D; E; E; D; D (others should be ACQUIRE!)
    while(true) {

      tail.count = Tail.count;
      tail.ptr = Tail.ptr;
      llmc_barrier_seq_cst(); // not needed because of data dependency
      next.ptr = tail.ptr->next.ptr;
      next.count = tail.ptr->next.count;
      llmc_barrier_seq_cst(); // Can find this one
      tail2.ptr = Tail.ptr;
      tail2.count = Tail.count;

      if(tail.ptr==tail2.ptr && tail.count==tail2.count)  {
        if(!next.ptr) {
          if(tail.ptr->next.compare_exchange_weak(next, NodePtr(node, next.count+1), std::memory_order_relaxed)) {
            break;
          }
        } else {
          Tail.compare_exchange_strong(tail, NodePtr(next.ptr, tail.count+1), std::memory_order_relaxed);
        }
      }
    }
    llmc_barrier_release(); // not needed because of data dependency
    Tail.compare_exchange_strong(tail, NodePtr(node, tail.count+1), std::memory_order_relaxed);
  }

  bool dequeue(int& data) {
    NodePtr head;
    NodePtr head2;
    NodePtr tail;
    NodePtr next;
    int i=0;
    while(true) {

      head.count = Head.count;
      head.ptr = Head.ptr;

      llmc_barrier_seq_cst(); // Verified. Can find this one: E; D || (E; D; D; E)   or   E; D || (E; D; D) (can block!) needs a buffer size of 5

      tail.ptr = Tail.ptr;
      tail.count = Tail.count;

      llmc_barrier_seq_cst(); // Verified. Can find this one: E || D; E; E; D; D needs a buffer size of 5

      next.ptr = head.ptr->next.ptr;
      next.count = head.ptr->next.count;

      llmc_barrier_seq_cst(); // Verified. Can find this one: E; E; D || (D; E); D

      head2.ptr = Head.ptr;
      head2.count = Head.count;

      if(head.ptr==head2.ptr && head.count==head2.count) {  // Count is needed to fix: E; E; D || (D; E); D
        if(head.ptr == tail.ptr) {
          if(!next.ptr) return false;
          Tail.compare_exchange_strong(tail, NodePtr(next.ptr, tail.count+1), std::memory_order_relaxed);
        } else {
          assert(next.ptr);
          data = next.ptr->data;
          assert(data);
          if(Head.compare_exchange_weak(head, NodePtr(next.ptr, head.count+1), std::memory_order_relaxed)) {
            assert(data);
            break;
          }
        }
      }
    }
    freeNode(head.ptr);
    return true;
  }

};
```

C++ implementation: main (part 3/3)

```
197 MSQ msq;
198
199 void* E(intptr_t data) {
200   msq.enqueue(data, data+1);
201   return nullptr;
202 }
203
204 void* E1(void*) {
205   msq.enqueue(1, 2);
206   return nullptr;
207 }
208
209 void* E2(void*) {
210   msq.enqueue(2, 3);
211   return nullptr;
212 }
213
214 void* E3(void*) {
215   msq.enqueue(3, 4);
216   return nullptr;
217 }
218
219 void* D(void*) {
220   int data;
221   while(!msq.dequeue(data));
222   return (void*)(intptr_t)data;
223 }
224
225 int main(int argc, char** argv) {
226   pthread_t e1, d1;
227   pthread_t e2, d2;
228   pthread_t e3, d3;
229   intptr_t data[3];
230
231   data[0] = 0;
232   data[1] = 0;
233   data[2] = 0;
234
235   llmc_barrier_seq_cst();
236
237   E(1);
238   E(2);
239   E(3);
240   data[0] = (intptr_t)D(nullptr);
241   data[1] = (intptr_t)D(nullptr);
242   data[2] = (intptr_t)D(nullptr);
243
244   assert( (data[0]==1 && data[1]==2 && data[2]==3)
245        || (data[0]==1 && data[1]==3 && data[2]==2)
246        || (data[0]==2 && data[1]==1 && data[2]==3)
247        || (data[0]==2 && data[1]==3 && data[2]==1)
248        || (data[0]==3 && data[1]==1 && data[2]==2)
249        || (data[0]==3 && data[1]==2 && data[2]==1)
250   );
251 }
```

C.3 Recursive Fibonacci algorithm

This is an LLVM implementation of a recursive Fibonacci algorithm.

LLVM IR generated by `clang tadd.cpp -std=c++11 -pthread -S -flto -emit-llvm -otadd.ll -O3`

```
define i32 @fib(i32 %n) {
entry:
  %n1 = alloca i32
  store i32 %n, i32* %n1
  %deref = load i32* %n1
  %_eq_ = icmp slt i32 %deref, 2
  br i1 %_eq_, label %then, label %else

then:                                              ; preds = %entry
  %deref2 = load i32* %n1
  ret i32 %deref2

else:                                              ; preds = %entry
  %deref3 = load i32* %n1
  %_sub_ = sub i32 %deref3, 1
  %_call_ = call i32 @fib(i32 %_sub_)
  %deref4 = load i32* %n1
  %_sub_5 = sub i32 %deref4, 2
  %_call_6 = call i32 @fib(i32 %_sub_5)
  %_add_ = add i32 %_call_, %_call_6
  ret i32 %_add_
}

define i32 @main(i32 %argc, i8** %argv) {
entry:
  %_call_ = call i32 @fib(i32 1)
  ret i32 %_call_
}
```