Litmus: Running Tests Against Hardware

Jade Alglave[1,3] Luc Maranget[1] Susmit Sarkar[2] Peter Sewell[2]

[1] INRIA
[2] University of Cambridge
[3] Oxford University

Abstract. Shared memory multiprocessors typically expose subtle, poorly understood and poorly specified relaxed-memory semantics to programmers. To understand them, and to develop formal models to use in program verification, we find it essential to take an empirical approach, testing what results parallel programs can actually produce when executed on the hardware. We describe a key ingredient of our approach, our litmus tool, which takes small 'litmus test' programs and runs them for many iterations to find interesting behaviour. It embodies various techniques for making such interesting behaviour appear more frequently.

1 Introduction

Modern shared memory multiprocessors do not actually provide the sequentially consistent (SC) memory semantics [Lam79] typically assumed in concurrent program verification. Instead, they provide a *relaxed memory model*, arising from optimisations in multiprocessor hardware, such as store buffering and instruction reordering (relaxed-memory behaviour can also arise from compiler optimisations). For example, in hardware with store buffers, the program below (in pseudo-code on the left and x86 assembly on the right) can end with 0 in both r_0 and r_1 on x86, a result not possible under SC:

Shared: x, y, initially zero
Thread-local: r_0, r_1

Proc 0	Proc 1
$y \leftarrow 1$	$x \leftarrow 1$
$r_0 \leftarrow x$	$r_1 \leftarrow y$

Finally: is $r_0 = 0$ and $r_1 = 0$ possible?

```
X86 SB (* Store Buffer test *)
{ x=0; y=0; }
 P0           | P1           ;
 MOV [y],$1   | MOV [x],$1   ;
 MOV EAX,[x]  | MOV EAX,[y]  ;
exists (0:EAX=0 /\ 1:EAX=0)
```

The actual relaxed memory model exposed to the programmer by a particular multiprocessor is often unclear. Many models are described only in informal prose documentation [int09,pow09], which is often ambiguous, usually incomplete [SSS+10,AMSS10], and sometimes unsound (forbidding behaviour that is observable in reality) [SSS+10]. Meanwhile, researchers have specified various formal models for relaxed memory, but whether they accurately capture the subtleties of actual processor implementations is usually left unexamined. In

We acknowledge funding from EPSRC grants EP/F036345, EP/H005633, and EP/H027351, from ANR project parsec (ANR-06-SETIN-010), and from INRIA associated team MM.

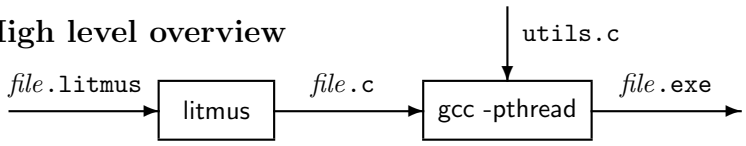
contrast, we take a firmly empirical approach: testing what current implementations actually provide, and use the test results to inform the building of models. This is in the spirit of Collier's early work on ARCHTEST [Col92], which explores various violations of SC, but which does not deal with many complexities of modern processors, and also does not easily support testing new tests.

Much interesting memory model behaviour already shows up in small, but carefully crafted, concurrent programs operating on shared memory locations, "litmus tests". Given a specified initial state, the question for each test is what final values of registers and memory locations are permitted by actual hardware. Our litmus tool takes as input a litmus file, as on the right above, and runs the program within a test harness many times. On one such run of a million executions, it produced the result below, indicating that the result of interest occurred 34 times.

```
Positive: 34, Negative: 999966
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
```

The observable behaviour of a typical multiprocessor arises from an extremely complex (and commercially confidential) internal structure, and is highly non-deterministic, dependent on details of timing and the processors' internal state. Black-box testing cannot be guaranteed to produce all permitted results in such a setting, but with careful design the tool does generate interesting results with reasonable frequency.

2 High level overview



Our litmus tool takes as input small concurrent programs in x86 or Power assembly code (*file*`.litmus`). It accepts symbolic locations (such as `x` and `y` in our example), and symbolic registers. The tool then translates the program *file*`.litmus` into a C source file, encapsulating the program as inline assembly in a test harness. The C file is then compiled by gcc into executables which can be run on the machine to perform checks. The translation process performs some simple liveness analysis (to properly identify registers read and trashed by inline assembly), and some macro expansions (macros for lock acquire and release are translated to packaged assembly code).

The test harness initialises the shared locations, and then spawns threads (using the POSIX pthread library) to run the various threads within a loop. Each thread does some mild synchronization to ensure the programs run roughly at the same time, but with some variability so that interesting behaviour can show up. In the next section we describe various ways in which the harness can be adjusted, so that results of interest show up more often.

The entire program consists of about 10,000 lines of Objective Caml, plus about 1,000 lines of C. The two phases can be separated, allowing translated C files to be transferred to many machines. It is publicly distributed as a

part of the diy tool suite, available at `http://diy.inria.fr`, with companion user documentation. litmus has been run successfully on Linux, Mac OS and AIX [AMSS10].

3 Test infrastructure and parameters

Users can control various parameters of the tool, which impact efficiency and outcome variability, sometimes dramatically.

Test repetition To benefit from parallelism and stress the memory subsystem, given a test consisting of t threads $P_0,\ldots,P_{t-1}$, we run $n = \max(1, a/t)$ identical test *instances* concurrently on a machine with a cores. Each of these tests consists in repeating r times the sequence of creating t threads, collectively running the litmus test s times, then summing the produced outcomes in an histogram.

Thread assignment We first fork t POSIX threads $T_0,\ldots T_{t-1}$ for executing $P_0,\ldots,P_{t-1}$. We can control which thread executes which code with the *launch mode*: if *fixed* then T_k executes P_k; if *changing* (the default) the association between POSIX and test threads is random. In our experience, the launch mode has a marginal impact, except when affinity is enabled—see *Affinity* below.

Accessing memory cells Each thread executes a loop of size s. Loop iteration number i executes the code of one test thread and saves the final contents of its observed registers in arrays indexed by i; a memory location x in the `.litmus` source corresponds to an array cell. The access to this array cell depends on the *memory mode*. In *direct mode* the array cell is accessed directly as $x[i]$; hence cells are accessed sequentially and false sharing effects are likely. In *indirect mode* (the default) the array cell is accessed by a shuffled array of pointers, giving a much greater variability of outcomes. If the (default) *preload mode* is enabled, a preliminary loop of size s reads a random subset of the memory locations accessed by P_k, also leading to a greater outcome variability.

Thread synchronisation The iterations performed by the different threads T_k may be unsynchronised, synchronised by a pthread-based barrier, or synchronised by busy-wait loops. Absence of synchronisation is of marginal interest when t exceeds a or when $t = 2$. Pthread-based barriers are slow and in fact offer poor synchronisation for short code sequences. Busy-waiting synchronisation is thus the preferred technique and the default.

Affinity Affinity is a scheduler property binding software (POSIX) threads to given hardware *logical processor*. The latter may be single cores or, on machines with hyper-threading (x86) or simultaneous multi threading (SMT, Power) each core may host several logical processors.

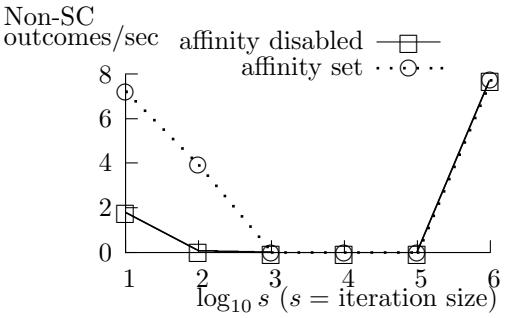
We allocate logical processors test instance by test instance (parameter n) and then POSIX thread by POSIX thread, scanning the logical processors sequence left-to-right by steps of the specified *affinity increment*. Suppose a logical processors sequence $P = 0, 1, \ldots, A-1$ (the default on a machine with A logical processors available) and an increment i: we allocate (modulo A) first the

processor 0, then i, then $2i$, *etc.* If we reach 0 again, we allocate the processor 1 and then increment again. Thereby, all the processors in the sequence will get allocated to different threads naturally, provided of course that less than A threads are scheduled to run.

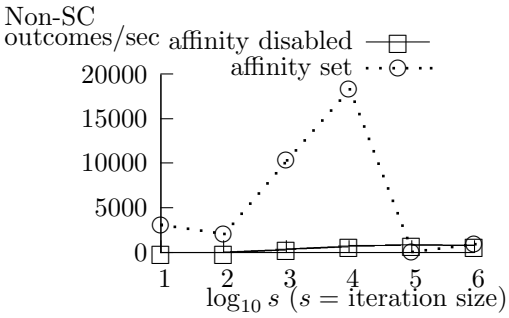
4 The impact of test parameters

Test parameters can have a large impact on the frequency of interesting results. Our tests are non-deterministic and parallel, and the behaviours of interest arise from specific microarchitectural actions at specific times. Thus the observed frequency is quite sensitive to the machine in question and to its operating system, in addition to the specific test itself.

Let us run the SB test from the introduction with various combinations of parameters on a lightly loaded Intel Core 2 Duo. There is one interesting outcome here, and we graph the frequency of that outcome arising per second below against the logarithm of the iteration size s. Note that only the orders of magnitude are significant, not the precise numbers, for a test of this nature.



Test SB: direct memory mode



Test SB: indirect memory mode

We obtain the best results with indirect memory mode and affinity control, and 10^4 iterations per thread creation. These settings depend on the characteristics of the machine and scheduler, and we generally find such combinations of parameters remain good on the same testbed, even for different tests.

References

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

[Col92] W. W. Collier. *Reasoning About Parallel Architectures.* Prentice-Hall, 1992.

[int09] Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3A, rev. 30, March 2009.

[Lam79] L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1979.

[pow09] *Power ISA Version 2.06.* 2009.

[SSS+10] P. Sewell, S.Sarkar, S.Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. (Research Highlights).