# **Litmus**: Running Tests Against Hardware

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

[1] INRIA
[2] University of Cambridge

**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)
```

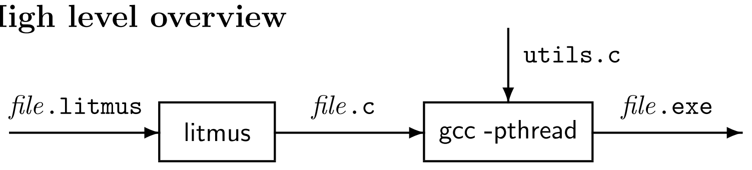
The actual relaxed memory model exposed to the programmer by a particular multiprocessor is often unclear. Many models are described only in informal prose documentation [int09,pow09], which is often ambiguous, usually incomplete [**?**,AMSS], and sometimes unsound (forbidding behaviour that is observable in reality) [**?**]. 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 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 [AMSS].

## 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.

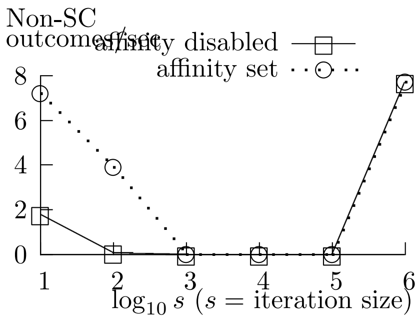
*Affinity* Affinity is a scheduler property binding software (POSIX) threads to given hardware *logical processor*. The latter may be single cores or, on machines with hyper-threading (x86) or simultaneous multi threading (SMT, Power) each core may host several logical processors.

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.

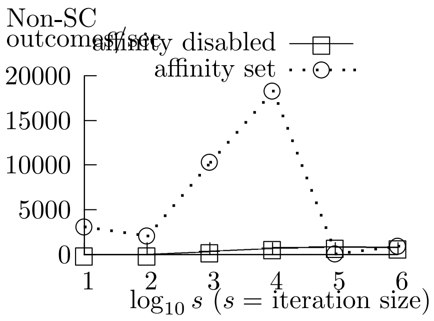
## 4 The impact of test parameters

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

[AMSS] 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.

[SSZN$^+$] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The Semantics of x86-CC Multiprocessor Machine Code. In *POPL 2009.*

# Tool demonstration

## 1 Introduction

The demonstration will show litmus at work. The tool litmus runs *litmus tests* on actual hardware. Litmus tests are small programs designed to highlight the features of a given model in a quick glance and concrete manner.

*Outline of the planned demonstration* We shall first introduce our running example, a classical litmus test designed to illustrate store buffering. Then we shall begin the demonstration itself, running litmus on our demonstration machine, and on a remote Power 6 machine. These runs will illustrate the basic structure and usage of the tool (Sec. **??**). In particular, we shall demonstrate how tests can be compiled on one machine to C source files, and executed on another. A C source file generated by litmus includes the code for the test proper, as inline assembly, within a test harness. The test harness runs the test numerous times and is partly under user control. We shall then show a slightly simplified version of the C source produced from the running example. The focus will be on basic, user-accessible, controls on the test harness (Sec. **??**). We shall pursue the demonstration by introducing and demonstrating more advanced controls: how memory is accessed (either sequentially or randomly), number of identical tests run concurrently, and limited OS scheduler control (Sec. **??**–**??**).

Our presentation will be by examples, running litmus to demonstrate effects and showing C code to describe the test harness. However, we shall also show some pictures, which we include as figures in this document. If the time slot allocated for tool demonstration permits, we might conclude by performing additional experiments on the Power 6 machine. We would then test variations of some classical litmus tests, such as Independent Reads of Independent Writes, focussing on the conclusions that can be drawn from the esxperiments, more than on the experiments themselves.

## 2 Litmus tests

Roughly, litmus tests come in two flavours "Allowed" and "Forbidden". In the former case, one expects some behaviour to show up, while in the second case one expects some behaviour not to show up. Consider for instance the two tests of Fig. 1. In such litmus tests descriptions, `x`, `y` are shared locations (*i.e.* cells of shared memory); while `r0`, `r1` are private locations (*i.e.* registers). By convention, all locations initially hold the value 0, unless otherwise specified. The text of the multi-thread program is followed by the specification of a certain final state for some selected locations, which *outcome* is declared allowed or forbidden. The test **SB** illustrates an effect frequently observed on modern parallel machines, due to buffering stores.

During the demonstration we shall focus on experiments themselves. However, to assert the significance of litmus testing, we shall briefly comment the two tests of Fig. 1. The occurrence of outcome "`r0=0`; `r1=0`" may be surprising if one assumes the simplest memory model of all: *sequential consistency* (SC).

**SB**

| $P_0$ | $P_1$ |
|---|---|
| $(a)$ `y` $\leftarrow$ `1` | $(c)$ `x` $\leftarrow$ `1` |
| $(b)$ `r0` $\leftarrow$ `x` | $(d)$ `r1` $\leftarrow$ `y` |

Allowed: `r0=0`; `r1=0`

**SB+FENCE**

| $P_0$ | $P_1$ |
|---|---|
| $(a)$ `y` $\leftarrow$ `1` | $(c)$ `x` $\leftarrow$ `1` |
| `fence` | `fence` |
| $(b)$ `r0` $\leftarrow$ `x` | $(d)$ `r1` $\leftarrow$ `y` |

Forbidden: `r0=0`; `r1=0`

**Fig. 1.** Two simple litmus tests.

Sequential consistency assumes (1) that memory accesses performed by the concurrent program results from interleaving the accesses performed by each thread; and (2) that writes to memory are visible to all threads instantaneously. As a consequence, assuming SC, **SB** starts by issuing a write to either `x` or `y` and at least one of `r0` or `r1` will hold the value 1 at the end of test. However, test **SB** succeeds on all machines we tested, thereby demonstrating that these machines do not follow the sequential consistency memory model. Those machines provide specialised "fence" instruction, whose purpose may (documentation is often unclear) be to restore sequential consistency, when fence instructions are inserted between memory accesses. The test **SB+FENCE**, of the Forbidden category, is designed to check the effectiveness of fence in that situation. Notice that the occurrence of outcome "`r0=0`; `r1=0`" in **SB** may result from the presence of store buffers that delay the observation of the writes performed by some core by other cores, and that fences may be implemented (naively) by flushing store buffers.

## 3 Tool usage

The tool litmus inputs litmus tests written in the target system assembly language. For instance, here are **SB** and **SB+FENCE** for x86:

```
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)
```

```
X86 SBFENCE
{ x=0; y=0; }
 P0           | P1           ;
 MOV [y],$1   | MOV [x],$1   ;
 MFENCE       | MFENCE       ;
 MOV EAX,[x]  | MOV EAX,[y]  ;
~exists (0:EAX=0 /\ 1:EAX=0)
```

Writing litmus tests in assembly language is a natural choice while testing machines. Namely, assembly is the right language to express what is actually executed, still providing a decent level of abstraction. Additionally, compiler interference is reduced to almost nothing. We shall run the two tests on the presentation machine, `conti`, an Intel Core 2 Duo:

```
con% litmus -mach conti x86/@all | less
```

In the command above, the option `-mach conti` configures litmus appropriately for `conti`. The argument `x86/@all` is a file that lists the tests we want to run:

```
con% cat x86/@all
SB.litmus
SB+FENCE.litmus
```

We shall then describe the output of litmus, going into detail for **SB**. First, the source of the test is reminded, so as to facilitate visual check of test output. Then, we show actual assembly code:

```
Generated assembler
        _litmus_P1_0_: movl $1,(%edx)
        _litmus_P1_1_: movl (%ecx),%eax
        _litmus_P0_0_: movl $1,(%ecx)
        _litmus_P0_1_: movl (%edx),%eax
```

With respect to input assembly code, one notices syntactical changes and the replacement of symbolic addresses `x` and `y` by registers. We argue that those changes are innocuous, in the sense that the results we get apply to the source of the test. Then, the result of the experiment follows:

```
Test SB Allowed
Histogram (4 states)
60246 :>0:EAX=0; 1:EAX=0;
471786:>0:EAX=1; 1:EAX=0;
467953:>0:EAX=0; 1:EAX=1;
15    :>0:EAX=1; 1:EAX=1;
Ok

Witnesses
Positive: 60246, Negative: 939754
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
Time SB 3.65
```

The core information is the list of outcomes with occurrence counts, which list comes first above. The targeted outcome occurred 60246 times (out of $10^6$ outcomes), as highlighted by the "Witnesses" section. One may also notice the presence of all of the 4 possible outcomes. Additional information is provided at the end of output: a hash-code of the test (used for consistency checks during automated analysis of results) and the (wall-clock) time spent by the test.

We shall show the results of **SB+FENCE** more rapidly, observing the outcome `0:EAX=0; 1:EAX=0;` not to show up:

```
Test SB+FENCE Forbidden
Histogram (3 states)
499721:>0:EAX=1; 1:EAX=0;
```

```
499952:>0:EAX=0; 1:EAX=1;
327   :>0:EAX=1; 1:EAX=1;
Ok
```

We shall seize the opportunity to introduce the idea behind our method for tuning testing conditions:

– We perform the tests **SB** and **SB+FENCE** in the same conditions.
– The interesting outcome `0:EAX=0; 1:EAX=0;` shows up easily for **SB** and does not show up for **SB+FENCE**.
– The easier we get the outcome when it is allowed, the more significant is its absence when it is forbidden.

We shall then run similar tests for Power. To that end, we shall need an Internet connection. We shall then log on `abducens`, a 4 cores, 2-ways simultaneous multi-threading (SMT) Power 6 machine. Should the connection be unavailable, we shall present slides. Our intention is:

– to illustrate the cross-compilation feature of litmus;
– to demonstrate that litmus targets the Power architecture.

Cross-compilation exposes the high-level structure of the tool: litmus proper translates its input litmus test(s) in assembly into C source file(s), which are then compiled by gcc — see Fig **??**.

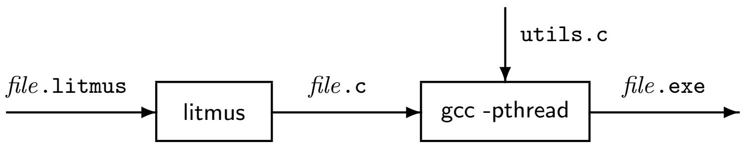


**Fig. 2.** High-level overview of litmus.

In practice, we shall consider three tests, **SB**, **SB+LWSYNC** and **SB+SYNC**, compile them on `conti`:

```
con% ls ppc
@all  SB.litmus  SB+LWSYNC.litmus  SB+SYNC.litmus
con% litmus -mach abducens -o ppc.tar ppc/@all
con% scp ppc.tar abducens-i.cl.cam.ac.uk:ppc
```

In cross-compilation mode (enabled by the option `-o ppc.tar`), litmus output is an archive that contains C source files for the tests. Such C source files contain the tests proper as inline assembly, plus a test harness.

On `abducens` we shall unpack the archive and compile the three tests, using the `Makefile` included in the archive:

```
[maranget@abducens ppc]$ tar xmf ppc.tar && make
gcc -Wall -std=gnu99 -O -pthread -O2 -c outs.c
gcc -Wall -std=gnu99 -O -pthread -O2 -c utils.c
gcc -Wall -std=gnu99 -O -pthread  -o SB.exe outs.o utils.o SB.c
gcc -Wall -std=gnu99 -O -pthread  -o SB+LWSYNC.exe outs.o utils.o SB+LWSYNC.c
gcc -Wall -std=gnu99 -O -pthread  -o SB+SYNC.exe outs.o utils.o SB+SYNC.c
...
```

One may notice that part of the test harness is provided by the additional C source files `utils.c` and `outs.c`. The test is run by the means of a dedicated shell script

```
[maranget@abducens ppc]$ sh run.sh |less
...
```

We shall observe that the targeted outcome shows up for **SB** and **SB+LWSYNC** while it does not for **SB+SYNC**. Fig. **??** shows a screenshot of the cross-compilation demonstration.

## 4 Test harness and parameters

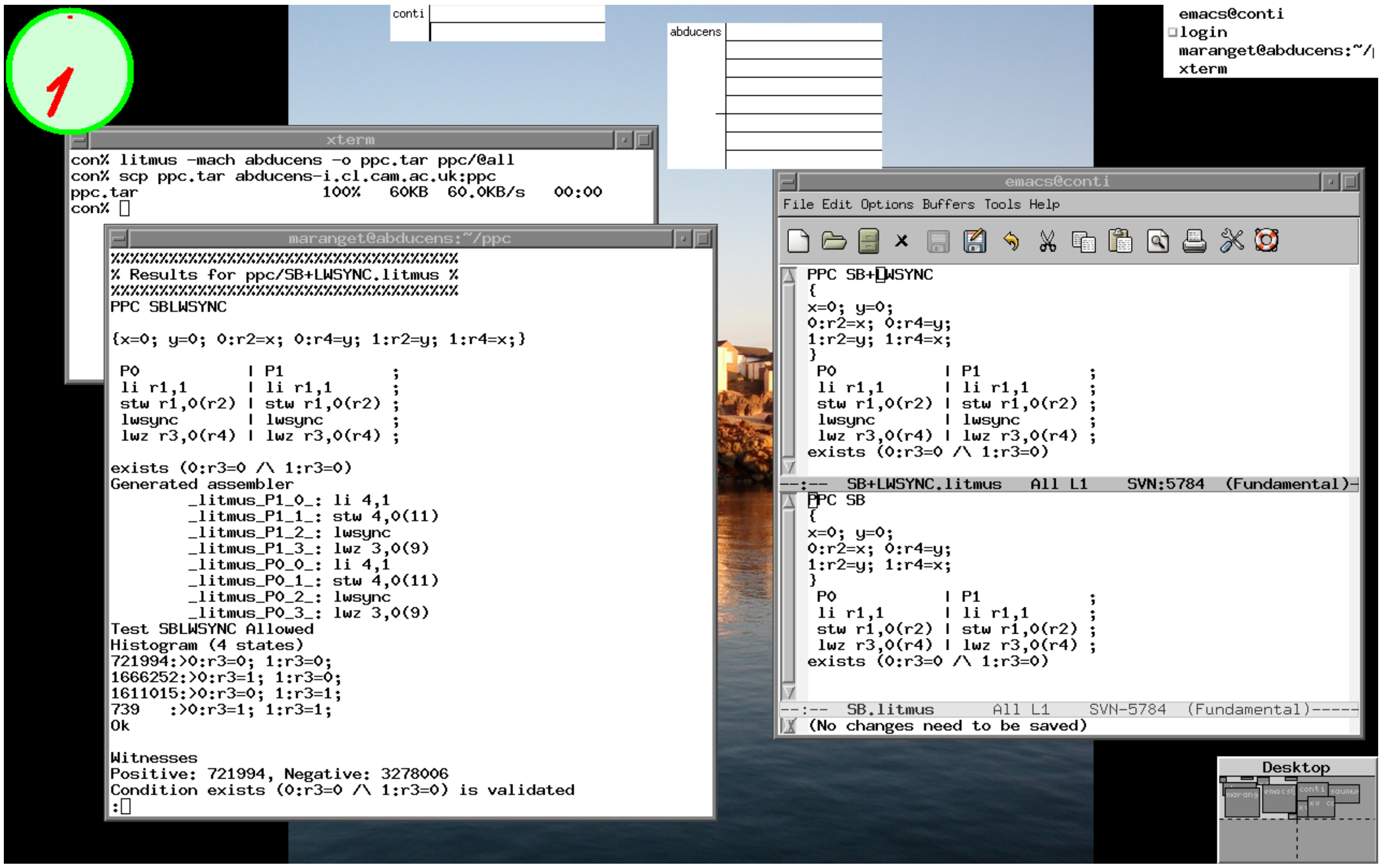
In this part of the demonstration we shall describe our testing techniques. As an introduction we shall first show Fig. **??** that summarises test program structure: we perform $r$ times the sequence of spawning (POSIX) threads that run the test within a loop of size $s$. Each (POSIX) thread does some mild synchronization to ensure that the code of test threads run at the same pace. We shall illustrate our techniques by an example C program. This program is a simplified version of `SB.c`, which we get by compiling `SB.litmus` for x86, by:

```
con% litmus -mach conti -mem direct -o conti/direct/a.tar  x86/SB.litmus
con% cd conti/direct
con% tar xmf a.tar
```

### Size parameters

Fig. 2 depicts (slightly simplified) code for the threads $P_0$ and $P_1$. We shall first point out that the code of $P_0$ and $P_1$ appears as inline assembly and as the body of a loop executed `size_of_test` times (defined as parameter "$s$"). Moreover, loop iterations are synchronised by the means of specific busy-wait lock-free code given as an inline function (a tamed C-macro) `synchro`.

We shall focus on the assembly code for $P_0$, which is a direct translation of the input code: store value 1 into location `x`, and then read the contents of location `y` into register `eax`. Notice that locations are abstracted out (notation `[..]` of `gcc` inline assembly templates). During loop iteration number `i` the shared locations `x` and `y` are in fact the array cells `x[i]` and `y[i]`; while the final contents of the register `eax` is saved into the array cell `r0[i]`. The connection between



```
con% litmus -mach abducens -o ppc.tar ppc/@all
con% scp ppc.tar abducens-i.cl.cam.ac.uk:ppc
ppc.tar                        100%    60KB  60.0KB/s   00:00
con%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Results for ppc/SB+LWSYNC.litmus %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PPC SBLWSYNC

{x=0; y=0; 0:r2=x; 0:r4=y; 1:r2=y; 1:r4=x;}

 P0             | P1             ;
 li r1,1        | li r1,1        ;
 stw r1,0(r2)   | stw r1,0(r2)   ;
 lwsync         | lwsync         ;
 lwz r3,0(r4)   | lwz r3,0(r4)   ;

exists (0:r3=0 /\ 1:r3=0)
Generated assembler
        _litmus_P1_0_: li 4,1
        _litmus_P1_1_: stw 4,0(11)
        _litmus_P1_2_: lwsync
        _litmus_P1_3_: lwz 3,0(9)
        _litmus_P0_0_: li 4,1
        _litmus_P0_1_: stw 4,0(11)
        _litmus_P0_2_: lwsync
        _litmus_P0_3_: lwz 3,0(9)
Test SBLWSYNC Allowed
Histogram (4 states)
721994:>0:r3=0; 1:r3=0;
1666252:>0:r3=1; 1:r3=0;
1611015:>0:r3=0; 1:r3=1;
739    :>0:r3=1; 1:r3=1;
Ok

Witnesses
Positive: 721994, Negative: 3278006
Condition exists (0:r3=0 /\ 1:r3=0) is validated
:
```

```
PPC SB+LWSYNC
{
x=0; y=0;
0:r2=x; 0:r4=y;
1:r2=y; 1:r4=x;
}
 P0             | P1             ;
 li r1,1        | li r1,1        ;
 stw r1,0(r2)   | stw r1,0(r2)   ;
 lwsync         | lwsync         ;
 lwz r3,0(r4)   | lwz r3,0(r4)   ;
exists (0:r3=0 /\ 1:r3=0)
--:--  SB+LWSYNC.litmus    All L1     SVN:5784  (Fundamental)-
PPC SB
{
x=0; y=0;
0:r2=x; 0:r4=y;
1:r2=y; 1:r4=x;
}
 P0             | P1             ;
 li r1,1        | li r1,1        ;
 stw r1,0(r2)   | stw r1,0(r2)   ;
 lwz r3,0(r4)   | lwz r3,0(r4)   ;
exists (0:r3=0 /\ 1:r3=0)
--:--  SB.litmus        All L1     SVN-5784  (Fundamental)-----
(No changes need to be saved)
```

**Fig. 3.** Cross-compilation, compilation in 'xterm', execution in 'abducens'

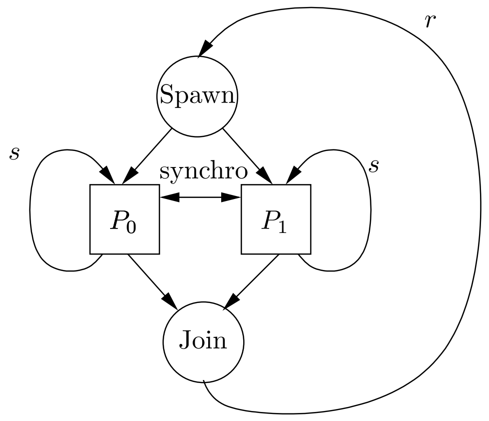


**Fig. 4.** Graphical representation of test program structure.

the abstract shared locations (`[x]` and `[y]`) and the corresponding array cells (`x[i]` and `y[i]`) is implemented by the output declaration of the template (*e.g.* `[x] "=m" (x[i])`). As to the abstract register `[eax]`, `[eax] "=&r" (r0[i])` ensures that its final value will get saved into `r0[i]`. Notice that the actual register is not necessarily "`eax`" as `gcc` performs the allocation of abstract registers.

We shall then detail how synchronisation of loop iterations is achieved. We here use another array of size $s$, `barrier`, whose cells initially hold 0. At loop iteration number `i`, one of the threads writes the value 1 into the flag cell `barrier[i]`, while the other thread loops until it reads a non-zero value in `barrier[i]`. Observe that the thread that writes changes at every iteration.

Figure 3 shows a simplified test harness. The code starts by allocating all arrays (`x`, ..., `barrier`). Notice that dynamic allocation of memory permits the setting of parameter $s$ with the dedicated command line option `-s` of `SB.exe`. The test is then run `nruns` (parameter $r$) times. More precisely, one iteration first initialises the involved arrays, shared locations being initialised as specified by the input file (here 0); while the copies of register final values (*i.e.* the arrays `r0` and `r1`) are initialised to the sentinel value $-1$. Then, the test is run and outcomes counts are collected in the matrix `out`. Once $r$ iterations are completed the matrix `out` is printed.

Notice that the litmus test is run $r \times s$ times, *i.e.* $r \times s$ outcomes are produced and counted. Moreover, for a litmus test involving $t$ threads, $t \times r$ POSIX threads are created.

## User control on the size parameters

Parameters $s$ and $r$ can be given as command line options to litmus (`-s` $s$ `-r` $r$) or in configuration files:

```
con% cat ~/lib/litmus/conti.cfg
```

```
inline static void synchro(int id, int i, int volatile *b) {
  if ((i % 2) == id) {
    *b = 1 ;
  } else {
    while (*b == 0) ;
  }
}

static void *P0(void *unused) {
  for (int i = size_of_test-1 ; i >= 0 ; i--) {
    synchro(0,i,&barrier[i]);
    asm volatile (
      "movl $1,%[y]\n\t"
      "movl %[x],%[eax]\n\t"
      :[x] "=m" (x[i]),[y] "=m" (y[i]),[eax] "=&r" (r0[i])
      :
      :"cc","memory"
    );
  }
  return NULL ;
}

static void *P1(void *unused) {
  for (int i = size_of_test-1 ; i >= 0 ; i--) {
    synchro(1,i,&barrier[i]);
    asm volatile (
      "movl $1,%[x]\n\t"
      "movl %[y],%[eax]\n\t"
      :[x] "=m" (x[i]),[y] "=m" (y[i]),[eax] "=&r" (r1[i])
      :
      :"cc","memory"
    );
  }
  return NULL ;
}
```

**Fig. 5.** Code for $P_0$ and $P_1$ of test **SB**.

```
  /* Allocate */
  x = alloc(size_of_test) ; y = alloc(size_of_test) ;
  r0 = alloc(size_of_test) ; r1 = alloc(size_of_test) ;
  barrier = alloc(size_of_test) ;


  int out[2][2] ; /* Count of outcomes, as count[r0][r1] */
  out[0][0] = out[0][1] = out[1][0] = out [1][1] = 0 ;

  for (int i = 0 ; i < nruns ; i++) {
    /* Initialise */
    for (int k = 0 ; k < size_of_test ; k++) {
      x[k] = y[k] = 0 ;     /* Init */
      r0[k] = r1[k] = -1 ; /* Safety */
      barrier[k] = 0 ;
    }

    /* Run test */
    pthread_t th0, th1;
    pthread_create(&th0, NULL, P0, NULL) ;
    pthread_create(&th1, NULL, P1, NULL) ;
    pthread_join(th0,NULL) ;
    pthread_join(th1,NULL) ;

    /* Count outcomes */
    for (int k = 0 ; k < size_of_test ; k++) {
      assert (r0[k] >= 0 && r1[k] >= 0) ; /* Safety */
      out[r0[k]][r1[k]]++ ;
    }
  }

  /* Print results */
  ...
}
```

**Fig. 6.** Simplified test harness.

```
size_of_test = 5000
number_of_run = 200
...
```

Hence, by default `-mach conti` defines $s = 5,000$ and $r = 200$. Those define the default values of the same controls of `.exe` files.

```
con% ./SB.exe -v -v
n=1, r=200, s=5000
Test SB Allowed
Histogram (2 states)
500000:>0:EAX=1; 1:EAX=0;
500000:>0:EAX=0; 1:EAX=1;
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (0:EAX=0 /\ 1:EAX=0) is NOT validated
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
Time SB 4.23
```

There are 1 million ($5,000 \times 200$) outcomes. Specifying the `-v` option twice commands the repetitive display of iteration numbers as `Run` $i$ `of 200`, illustrating our point on default values more clearly. We notice that the interesting outcome "`0:EAX=0; 1:EAX=0;`" does not show up.

Setting $s = 100$ produces the interesting outcome:

```
con% ./SB.exe -s 100 -r 100
Test SB Allowed
Histogram (3 states)
28    :>0:EAX=0; 1:EAX=0;
5000  :>0:EAX=1; 1:EAX=0;
4972  :>0:EAX=0; 1:EAX=1;
Ok

Witnesses
Positive: 28, Negative: 9972
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
Time SB 0.35
```

However, observing the interesting outcome looks a difficult task:

```
con% ./SB.exe -s 50 -r 200
Test SB Allowed
Histogram (2 states)
5000  :>0:EAX=1; 1:EAX=0;
5000  :>0:EAX=0; 1:EAX=1;
```

```
No

Witnesses
Positive: 0, Negative: 10000
Condition exists (0:EAX=0 /\ 1:EAX=0) is NOT validated
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
Time SB 2.12
con% ./SB.exe -s 200 -r 50
Test SB Allowed
Histogram (2 states)
5000  :>0:EAX=1; 1:EAX=0;
5000  :>0:EAX=0; 1:EAX=1;
No

Witnesses
Positive: 0, Negative: 10000
Condition exists (0:EAX=0 /\ 1:EAX=0) is NOT validated
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
Time SB 0.33
```

Additional, more advanced, controls over the test harness permits getting the interesting outcome more steadily. We examine those now.

## 5 Memory mode

The previous tests were run under *direct* memory mode, *i.e.* the arrays of shared locations are accessed sequentially. As a consequence, the pattern of memory accesses is rather regular and the memory subsystem is exercised in too regular a fashion. In *indirect* memory mode, accesses to memory are more random: array cells are accessed through shuffled arrays of pointers. Here is simplified code for P0 of test **SB** compiled in indirect memory mode:

```
static void *P0(void *unused) {
  for (int i = size_of_test-1 ; i >= 0 ; i--) {
    synchro(0,i,&barrier[i]);
    asm volatile (
      ...
      :[x] "=m" (*xp[i]),[y] "=m" (*yp[i]),[eax] "=&r" (r0[i])
// In direct mode, we had:
//    :[x] "=m" (x[i]),[y] "=m" (y[i]),[eax] "=&r" (r0[i])
      ...
    );
  }
  return NULL ;
}
```

In the code above, xp is the array of pointers to array x. Observe that the only change w.r.t. direct memory mode resides in the output declaration of the assembly template. Changes to the test harness code are more important. In particular, pointer arrays are shuffled at every iteration of the outer loop (of size $r$), at the initialisation stage.

By contrast with size parameters, memory mode is fixed at compile time and cannot be changed later:

```
con% litmus -mach conti -mem indirect -o conti/indirect/a.tar x86/@all
con% cd conti/indirect
con% tar xmf a.tar && make
...
```

We can now run `SB.exe` with default values for $s$ and $r$.

```
con% ./SB.exe
Test SB Allowed
Histogram (4 states)
59920 :>0:EAX=0; 1:EAX=0;
471803:>0:EAX=1; 1:EAX=0;
468258:>0:EAX=0; 1:EAX=1;
19    :>0:EAX=1; 1:EAX=1;
Ok

Witnesses
Positive: 59920, Negative: 940080
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
Time SB 3.47
```

We shall then present a more thorough comparison of the two memory modes for test **SB** on `conti`. We compare *efficiency* $e$ defined as the number of occurrences of the interesting outcome "`0:EAX=0; 1:EAX=1`" produced per second, for various sizes $s$. Fig. **??** gives orders of magnitude of efficiency $e$ as a function of size $s$ for three experiments in indirect mode ($I$) and in direct mode ($D$). For

| $e \backslash s$ | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
|---|---|---|---|---|---|---|
| $I$ | $2 \cdot 10^1$ | $4 \cdot 10^2$ | $9 \cdot 10^3$ | $5 \cdot 10^4$ | $6 \cdot 10^0$ | $1 \cdot 10^3$ |
| $D$ | $2 \cdot 10^0$ | $7 \cdot 10^1$ | | | $1 \cdot 10^0$ | $1 \cdot 10^0$ |
| $I$ | $2 \cdot 10^1$ | $4 \cdot 10^2$ | $1 \cdot 10^4$ | $1 \cdot 10^5$ | $4 \cdot 10^0$ | $7 \cdot 10^2$ |
| $D$ | $2 \cdot 10^0$ | $8 \cdot 10^2$ | | | $4 \cdot 10^0$ | |
| $I$ | $3 \cdot 10^1$ | $7 \cdot 10^2$ | $1 \cdot 10^4$ | $1 \cdot 10^5$ | $8 \cdot 10^1$ | $7 \cdot 10^2$ |
| $D$ | $1 \cdot 10^0$ | $8 \cdot 10^2$ | | | | $2 \cdot 10^0$ |

**Fig. 7.** Comparison of indirect and direct memory mode for test **SB**.

a given setting of parameter $s$, parameter $r$ was chosen so as to get a running time of less than 10 seconds. We observe:

1. In direct mode, the interesting outcome sometimes does not show up, while it always does in indirect mode.
2. For all values of size $s$, efficiency is better in indirect mode.

From these experiment we have a first idea of decent default values for parameters on `conti`: indirect memory mode, $5,000$ for parameter $s$, 200 for parameter $r$. Similar experiments performed on other x86 machines confirm the superiority of indirect mode over direct memory mode. On Power indirect mode also yields better efficiency for test **SB**, but the differrence is less striking.

## 6 Using all processors

Given a machine that features $a$ cores, running one instance of a litmus test that involves $t$ hardware threads is an obvious waste of resource when $a$ exceeds twice the value of $t$ and that the machine is otherwise idle. We ran into the issue on a high-end computer on which we were obliged to reserve (and pay for) 16 cores at a time to run our experiments. We solved the issue by running several instances of the litmus test concurrently, as depicted by Fig. **??**. We define the number of test instances run concurrently as parameter $n$. Notice that outcome counts from the $n$ test instances are summed internally, so that the format of test output is insensitive to parameter $n$.

We routinely compile and run several tests together. All tests in a given series need not involve the same number of threads $t$. Hence, parameter $n$ usually derives from another parameter $a$, the number of available cores, as $n = \lfloor a/t \rfloor$. Namely, parameter $a$ is constant for a given machine and is set in configuration files:

```
con% cat ~/lib/litmus/conti.cfg
size_of_test = 5000
number_of_run = 200
avail = 2
con% cat ~/lib/litmus/saumur.cfg
size_of_test = 5000
number_of_run = 200
avail = 8
...
```

Where `saumur` is a 2 processors $\times$ 2 cores $\times$ 2-ways hyper-threaded Intel Xeon machine.

For some tests and on some machines, using all processors yields a super-linear increase of efficiency. This is the case for `SB` on `saumur`. If the Internet connection is available, we shall demonstrate the effect:

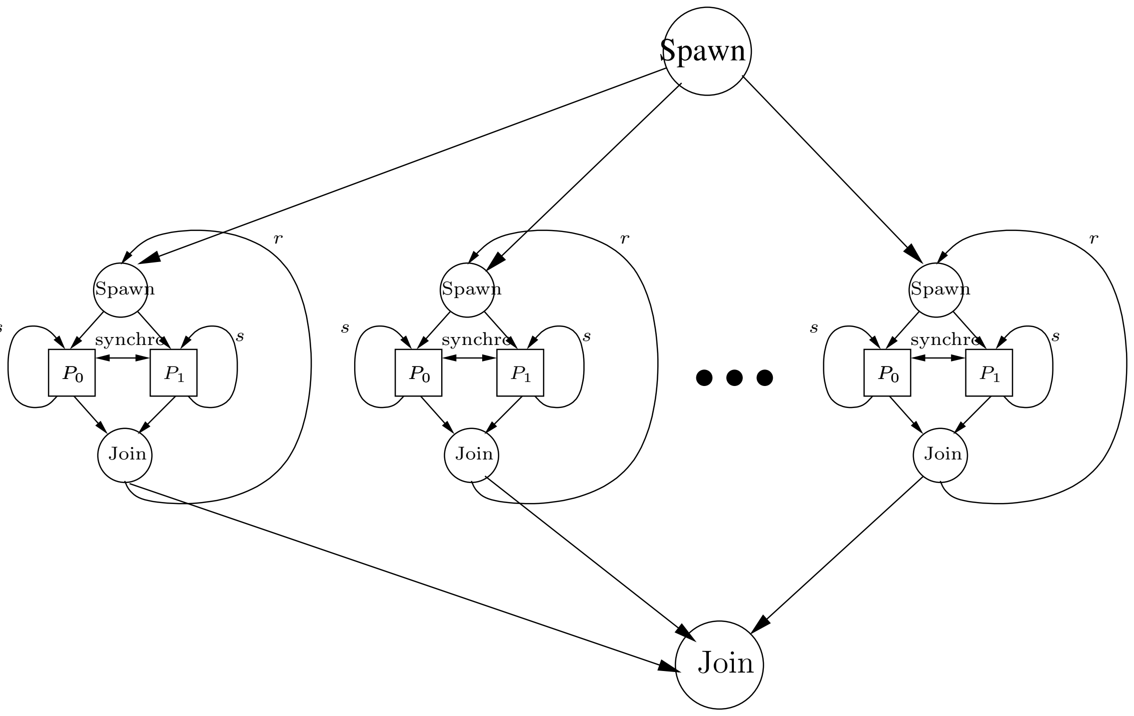


**Fig. 8.** Running $n$ instances of a litmus test.

```
sau% ./SB.exe -n 1
Test SB Allowed
Histogram (4 states)
124   :>0:EAX=0; 1:EAX=0;
499923:>0:EAX=1; 1:EAX=0;
499938:>0:EAX=0; 1:EAX=1;
15    :>0:EAX=1; 1:EAX=1;
...
Time SB 1.54
sau% ./SB.exe -n 4
Test SB Allowed
Histogram (4 states)
4593  :>0:EAX=0; 1:EAX=0;
1997117:>0:EAX=1; 1:EAX=0;
1998121:>0:EAX=0; 1:EAX=1;
169   :>0:EAX=1; 1:EAX=1;
...
Time SB 1.70
```

We observe that for the price of a small increment in running time, using the 8 (logical) cores available results in multiplying interesting outcomes by almost 40, whereas the total number of outcomes only increases by a factor of 4. This desirable effect may be due to increased stress on the scheduler and on the memory sub-system. We also observed it on high-end Power machines.

## 7 Affinity

Linux and AIX offer the possibility to bind a given software, POSIX, thread on a given *logical processor*. In other words, the POSIX thread will be forced to run on the specified logical processor. In the simplest situation, logical processors and cores coincide. For instance, `conti` features two cores, known to the OS as logical processors 0 and 1. However, due to hyper-threading (x86) or simultaneous multi-threading (SMT, Power) a given core can host several logical processors. For instance, `saumur` features 4 cores and 8 logical processors, as depicted by Fig. **??**.

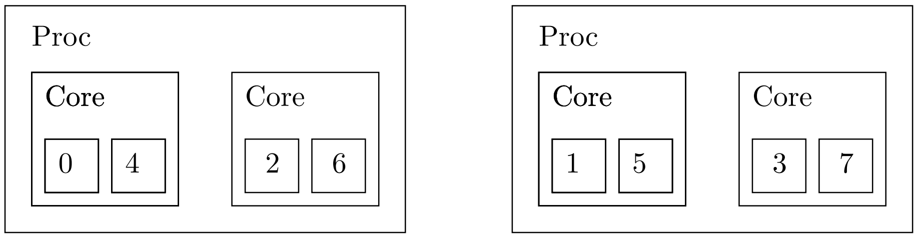


**Fig. 9.** Numbering of logical processors on `saumur`

The litmus tool provides users with two parameters for affinity control, the logical processor sequence $P$ and the affinity increment $i$. Those two parameters can be set both at compile and execution time, as command line options. By default, affinity control is disabled (since some OS's do not offer the feature). Users enable affinity control by specifying a value for the affinity increment. Then, the default logical processor sequence is inferred by `.exe` files, as $0, 1, \ldots A-1$ where $A$ is the number of logical processors available.

A given litmus test involves $t$ threads, written $P_0, P_1, \ldots P_{t-1}$ in its source. Those will run as $t$ software threads, written $T_0, T_1, \ldots, T_{t-1}$. It is worth noticing that the correspondence between test threads and POSIX threads changes at every iteration of the outer loop (the loop of size $r$). The distinction between test thread $P_i$ and POSIX thread $T_j$ becomes significant when affinity control is activated, as logical processors are allocated to POSIX threads, not to test threads.

First consider the simple example of the demonstration machine `conti` ($a = 2$ cores available, thus $P = 0,1$) and of test **SB** ($t = 2$ threads). We first compile the test enabling affinity control by specifying $i = 1$:

```
con% litmus -mach conti -i 1 -o conti/affinity/a.tar x86/@all
con% cd conti/affinity && tar xmf a.tar && make
...
```

The executable `SB.exe` runs one instance of the litmus test **SB** that involves two threads. Those two threads will be forced to run on the logical processors 0 and 1:

```
con% ./SB.exe -v
n=1, r=200, s=5000, i=1, p='0,1'
Test SB Allowed
Histogram (4 states)
2343  :>0:EAX=0; 1:EAX=0;
499920:>0:EAX=1; 1:EAX=0;
497722:>0:EAX=0; 1:EAX=1;
15    :>0:EAX=1; 1:EAX=1;
Ok

Witnesses
Positive: 2343, Negative: 997657
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
Time SB 0.20
```

Option `-v` shows the value of parameters. For comparison, we disable affinity control and run the same test:

```
con% ./SB.exe -i 0
Test SB Allowed
Histogram (4 states)
2519  :>0:EAX=0; 1:EAX=0;
499905:>0:EAX=1; 1:EAX=0;
497552:>0:EAX=0; 1:EAX=1;
24    :>0:EAX=1; 1:EAX=1;
Ok

Witnesses
Positive: 2519, Negative: 997481
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
Time SB 1.91
```

We observe a speedup of about 10 and no significant change as regards outcome counts[3]. We probably witness a scheduler effect similar to raising the priority of the test.

The subtleties of controlling affinity by the means of a single parameter $i$ are better illustrated on a machine more complex than `conti`. We thus turn to `saumur` (2 procs $\times$ 2 cores $\times$ 2-ways hyper-threading = 8 logical processors). Roughly, logical processors are allocated to POSIX threads, test instance by test instance, by steps of the specified increment $i$.

We illustrate the details of the process with an example. Let $P = 0, 1, 2, 3, 4, 5, 6, 7$ be the default logical processor sequence on `saumur`. As **SB** involves two threads ($t = 2$), `SB.exe` runs $n = 4$ instances of the test. Setting $i = 6$ illustrates all the aspects of our allocation procedure. The first test instance gets the logical processors $\{0, 6\}$. Then, the next logical processor is $6 + 6 = 12$, which we reduce modulo 8, yielding 4. The next logical processor is $4 + 6 = 10$, *i.e* 2 after reduction modulo 8. As a result, the second instance gets the logical processors $\{4, 2\}$. Then, the next logical processor is $2 + 6 = 8$, *i.e.* 0. However, the logical processor 0 being already allocated, we allocate the logical processor $0 + 1 = 1$ and the third instance gets the logical processors $\{1, 7\}$. Finally, the last instance gets the remaining two logical processors $\{5, 3\}$ naturally, as 5 is $7 + 6$ modulo 8 and 3 is $7 + 2 \times 6$ modulo 8.

In practice, the following table gives the allocation of logical processors for four settings of interest for $i$.

| $i$ | Allocation | Test threads run on… (cf. Fig **??**) |
|---|---|---|
| 0 |  | Leave scheduler alone |
| 1 | $\{0, 1\}, \{2, 3\}, \{4, 5\}, \{6, 7\}$ | Different processors |
| 2 | $\{0, 2\}, \{4, 6\}, \{4, 5\}, \{6, 7\}$ | Different cores |
| 4 | $\{0, 4\}, \{1, 5\}, \{2, 6\}, \{3, 7\}$ | Same cores |

A few runs of `SB.exe` with default size parameters on `saumur` will then demonstrate that affinity control impacts test output beyond running times.

```
sau% ./SB.exe -i 0
...
Positive: 3596, Negative: 3996404
...
Time SB 1.70
sau% ./SB.exe -i 1
...
Positive: 24533, Negative: 3975467
...
Time SB 1.28
sau% ./SB.exe -i 2
```

[3] In fact, the counts of interesting outcomes vary in the same way for the two settings: from about 100 to about $5,000$.

```
...
Positive: 23350, Negative: 3976650
...
Time SB 0.86
sau% ./SB.exe -i 4
...
Positive: 2171, Negative: 3997829
...
Time SB 0.36
```

We observe increasing speedups in running times as the test threads get closer one to the other. We also observe that interesting outcomes counts are 10 times higher when test threads run on distinct physical cores ($i = 1, i = 2$), than when test threads run on the same physical core ($i = 4$). We here witness an effect related to machine topology.

This effect sometimes makes the difference between observing and not observing a given outcome, as we found for many tests on Power machines.