A don't (diy) tutorial

Version 2.99

October 11, 2010

diy is a tool suite for testing shared memory models. We provide three tools, litmus (Part I) for running tests, diy (Part II) for producing tests from concise specifications, and dont (Part III) for either checking the conformance of a machine to an architecture or exploring the memory model of a given machine automatically. The software is written in Objective Caml[1], and released as sources. The web site of diy is `http://diy.inria.fr/`, authors can be contacted at `diy-devel@inria.fr`

Contents

[1] `http://caml.inria.fr/ocaml/`

Part I
Running tests with litmus

Traditionally, a *litmus test* is a small parallel program designed to exercise the memory model of a parallel, shared-memory, computer. Given a litmus test in assembler (X86 or Power) litmus runs the test.

Using litmus thus requires a parallel machine, which must additionally feature gcc and the pthreads library. At the moment, litmus is a prototype and has numerous limitations (recognised instructions, limited porting). Nevertheless, litmus should accept all tests produced by the companion diy tool and has been successfully used on Linux, MacOS and on AIX.

The authors of litmus are Luc Maranget and Susmit Sarkar. The present litmus is inspired from a prototype by Thomas Braibant (INRIA Rhône-Alpes) and Francesco Zappa Nardelli (INRIA Paris-Rocquencourt).

1 A tour of litmus

1.1 A simple run

Consider the following (rather classical) `classic.litmus` litmus test for X86:

```
X86 classic
"Fre PodWR Fre PodWR"
{ 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)
```

A litmus test source has three main sections:

1. The initial state defines the initial values of registers and memory locations. Initialisation to zero may be omitted.

2. The code section defines the code to be run concurrently — above there are two threads. Yes we know, our X86 assembler syntax is a mistake.

3. The final condition applies to the final values of registers and memory locations.

Run the test by:

```
$ litmus classic.litmus
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Results for classic.litmus %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
X86 classic
"Fre PodWR Fre PodWR"

{ 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)
Generated assembler
```

```
        _litmus_P0_0_: movl $1,(%rcx)
        _litmus_P0_1_: movl (%rsi),%eax
        _litmus_P1_0_: movl $1,(%rsi)
        _litmus_P1_0_: movl $1,(%rsi)
        _litmus_P1_1_: movl (%rcx),%eax

Test classic Allowed
Histogram (4 states)
34    :>0:EAX=0; 1:EAX=0;
499911:>0:EAX=1; 1:EAX=0;
499805:>0:EAX=0; 1:EAX=1;
250   :>0:EAX=1; 1:EAX=1;
Ok

Witnesses
Positive: 34, Negative: 999966
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=eb447b2ffe44de821f49c40caa8e9757
Time classic 0.60
...
```

The litmus test is first reminded, followed by actual assembler — the machine is an AMD64, in-line address references disappeared, registers may change, and assembler syntax is now more familiar. The test has run one million times, producing one million final states, or *outcomes* for the registers `EAX` of threads P_0 and P_1. The test run validates the condition, with 34 positive witnesses.

1.2 Cross compilation

With option `-o <name.tar>`, litmus does not run the test. Instead, it produces a tar archive that contains the C sources for the test.

Consider `ppc-classic.litmus`, a Power version of the previous test:

```
PPC ppc-classic
"Fre PodWR Fre PodWR"
{
0:r2=y; 0:r4=x;
1:r2=x; 1:r4=y;
}
 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)
```

Our target machine (ppc) runs MacOS, wich we specify with the `-os` option:

```
$ litmus -o /tmp/a.tar -os mac ppc-classic.litmus
$ scp /tmp/a.tar ppc:/tmp
```

Then, on the remote machine ppc:

```
ppc$ mkdir classic && cd classic
ppc$ tar xf /tmp/a.tar
ppc$ ls
Makefile comp.sh run.sh ppc-classic.c outs.c utils.c
```

Test is compiled by the shell script `comp.sh` (of by (Gnu) `make`, at user's choice) and run by the shell script `run.sh`:

```
$ sh comp.sh
$ sh run.sh
  ...
Test ppc-classic Allowed
Histogram (3 states)
3947  :>0:r3=0; 1:r3=0;
499357:>0:r3=1; 1:r3=0;
496696:>0:r3=0; 1:r3=1;
Ok

Witnesses
Positive: 3947, Negative: 996053
Condition exists (0:r3=0 /\ 1:r3=0) is validated
  ...
```

As we see, the condition validates also on Power. Notice that compilation produces an executable file, `ppc-classic.exe`, which can be run directly, for a less verbose output.

1.3 Running several tests at once

Consider the additional test `ppc-storefwd.litmus`:

```
PPC ppc-storefwd
"DpdR Fre Rfi DpdR Fre Rfi"
{
0:r2=x; 0:r6=y;
1:r2=y; 1:r6=x;
}
 P0            | P1            ;
 li r1,1       | li r1,1       ;
 stw r1,0(r2)  | stw r1,0(r2)  ;
 lwz r3,0(r2)  | lwz r3,0(r2)  ;
 xor r4,r3,r3  | xor r4,r3,r3  ;
 lwzx r5,r4,r6 | lwzx r5,r4,r6 ;
exists (0:r3=1 /\ 0:r5=0 /\ 1:r3=1 /\ 1:r5=0)
```

To compile the two tests together, we can give two file names as arguments to litmus:

```
$ litmus -o /tmp/a.tar -os mac ppc-classic.litmus ppc-storefwd.litmus
```

Or, more conveniently, list the litmus sources in a file whose name starts with `@`:

```
$ cat @ppc
ppc-classic.litmus
ppc-storefwd.litmus
$ litmus -o /tmp/a.tar -os mac @ppc
```

To run the test on the remote ppc machine, the same sequence of commands as in the one test case applies:

```
ppc$ tar xf /tmp/a.tar && make && sh run.sh
...
Test ppc-classic Allowed
```

```
Histogram (3 states)
4167  :>0:r3=0; 1:r3=0;
499399:>0:r3=1; 1:r3=0;
496434:>0:r3=0; 1:r3=1;
Ok

Witnesses
Positive: 4167, Negative: 995833
Condition exists (0:r3=0 /\ 1:r3=0) is validated
...
Test ppc-storefwd Allowed
Histogram (4 states)
37    :>0:r3=1; 0:r5=0; 1:r3=1; 1:r5=0;
499837:>0:r3=1; 0:r5=1; 1:r3=1; 1:r5=0;
499912:>0:r3=1; 0:r5=0; 1:r3=1; 1:r5=1;
214   :>0:r3=1; 0:r5=1; 1:r3=1; 1:r5=1;
Ok

Witnesses
Positive: 37, Negative: 999963
Condition exists (0:r3=1 /\ 0:r5=0 /\ 1:r3=1 /\ 1:r5=0) is validated
...
```

Now, the output of `run.sh` shows the result of two tests.

2 Controlling test parameters

Users can control some of testing conditions. Those impact efficiency and outcome variability.

Sometimes one looks for a particular outcome — for instance, one may seek to get the outcome `0:r3=1; 1:r3=1;` that is missing in the previous experiment for test `ppc-classical`. To that aim, varying test conditions may help.

2.1 Architecture of tests

Consider a test `a.litmus` designed to run on t threads $P_0,\ldots,\ P_{t-1}$. The structure of the executable `a.exe` that performs the experiment is as follows:

- So as to benefit from parallelism, we run $n = \max(1, a/t)$ (integer division) tests concurrently on a machine where a cores are available.
- Each of these (identical) tests consists in repeating r times the following sequence:
  - Fork t (POSIX) threads $T_0, \ldots T_{t-1}$ for executing $P_0,\ldots,\ P_{t-1}$. Which thread executes which code is either fixed, or changing, controlled by the *launch mode*. In our experience, the launch mode has marginal impact.

    In *cache mode* the T_k threads are re-used. As a consequence, t threads only are forked.
  - Each thread T_k executes a loop of size s. Loop iteration number i executes the code of P_k (in fixed mode) and saves the final contents of its observed registers in some arrays indexed by i. Furthermore, still for iteration i, memory location x is in fact an array cell.

    How this array cell is accessed depends upon the *memory mode*. In *direct mode* the array cell is accessed directly as $x[i]$; as a result, cells are accessed sequentially and false sharing effects are likely. In *indirect mode* the array cell is accessed by the means of a shuffled array of pointers; as a result we observed a much greater variability of outcomes.

If the *preload mode* is enabled, a preliminary loop of size s reads a random subset of the memory locations accessed by P_k. Preload have a noticeable effect.

The iterations performed by the different threads T_k may be unsynchronised, exactly synchronised by a pthread based barrier, or approximately synchronised by specific code. Absence of synchronisation may be interesting when t exceeds a. As a matter of fact, in this situation, any kind of synchronisation leads to prohibitive running times. However, for a large value of parameter s and small t we have observed spontaneous concurrent execution of some iterations amongst many. Pthread based barriers are exact but they are slow and in fact offers poor synchronisation for short code sequences. The approximate synchronisation is thus the preferred technique.

– Wait for the t threads to terminate and collect outcomes in some histogram like structure.

- Wait for the n tests to terminate and sum their histograms.

Hence, running `a.exe` produces $n \times r \times s$ outcomes. Parameters n, a, r and s can first be set directly while invoking `a.exe`, using the appropriate command line options. For instance, assuming $t = 2$, `./a.exe -a 201 -r 10000 -s 1` and `./a.exe -n 1 -r 1 -s 1000000` will both produce one million outcomes, but the latter is probably more efficient. If our machine has 8 cores, `./a.exe -a 8 -r 1 -s 1000000` will yield 4 millions outcomes, in a time that we hope not to exceed too much the one experienced with `./a.exe -n 1`. Also observe that the memory allocated is roughly proportional to $n \times s$, while the number of T_k threads created will be $t \times n \times r$ ($t \times n$ in cache mode). The `run.sh` shell script transmits its command line to all the executable (`.exe`) files it invokes, thereby providing a convenient means to control testing condition for several tests. Satisfactory test parameters are found by experimenting and the control of executable files by command line options is designed for that purpose.

Once satisfactory parameters are found, it is a nuisance to repeat them for every experiment. Thus, parameters a, r and s can also be set while invoking litmus, with the same command line options. In fact those settings command the default values of `.exe` files controls. Additionally, the synchronisation technique for iterations, the memory mode, and several others compile time parameters can be selected by appropriate litmus command line options. Finally, users can record frequently used parameters in configuration files.

2.2 Affinity

We view affinity as a scheduler property that binds a (software, POSIX) thread to a given (hardware) *logical processor*. In the most simple situation a logical processor is a core. However in the presence of hyperthreading (x86) or simultaneous multi threading (SMT, Power) a given core can host several logical processors.

2.2.1 Introduction to affinity

In our experience, binding the threads of test programs to selected logical processors yields sigificant speedups and, more importantly, greater outcome variety. We illustrate the issue by the means of an example.

We consider the test `ppc-iriw-lwsync.litmus`:

```
PPC ppc-iriw-lwsync
{
1:r2=x; 3:r2=y;
0:r2=y; 0:r4=x; 2:r2=x; 2:r4=y;
}
 P0           | P1           | P2           | P3           ;
 lwz r1,0(r2) | li r1,1      | lwz r1,0(r2) | li r1,1      ;
 lwsync       | stw r1,0(r2) | lwsync       | stw r1,0(r2) ;
```

```
 lwz r3,0(r4) |               | lwz r3,0(r4) |               ;
exists (0:r1=1 /\ 0:r3=0 /\ 2:r1=1 /\ 2:r3=0)
```

The test consists of four threads. There are two writers (P1 and P3) that write the value one into two different locations (`x` and `y`), and two readers that read the contents of `x` and `y` in different orders — P0 reads `y` first, while P2 reads `x` first. The load instructions `lwz` in reader threads are separated by a lightweight barrier instruction `lwsync`. The final condition `exists (0:r1=1 /\ 0:r3=0 /\ 2:r1=1 /\ 2:r3=0)` characterises the situation where the reader threads see the writes by P1 and P3 in opposite order. The corresponding outcome `0:r1=1; 0:r3=0; 2:r1=1; 2:r3=0;` is the only non-sequential consistent (non-SC, see Part II) possible outcome. By any reasonable memory model for Power, one expects the condition to validate, *i.e.* the non-SC outcome to show up.

The tested machine `vargas` is a Power 6 featuring 32 cores (*i.e.* 64 logical processors, since SMT is enabled) and running AIX in 64 bits mode. So as not to disturb other users, we run only one instance of the test, thus specifying four available processors. The litmus tool is absent on `vargas`. All these conditions command the following invocation of litmus, performed on our local machine:

```
$ litmus -r 1000 -s 1000 -a 4 -os aix -ws w64 ppc-iriw-lwsync.litmus -o ppc.tar
$ scp ppc.tar vargas:/var/tmp
```

On `vargas` we unpack the archive and compile the test:

```
vargas$ tar xf /var/tmp/ppc.tar && sh comp.sh
```

Then we run the test:

```
vargas$ ./ppc-iriw-lwsync.exe -v
Test ppc-iriw-lwsync Allowed
Histogram (15 states)
152885:>0:r1=0; 0:r3=0; 2:r1=0; 2:r3=0;
35214 :>0:r1=1; 0:r3=0; 2:r1=0; 2:r3=0;
42419 :>0:r1=0; 0:r3=1; 2:r1=0; 2:r3=0;
95457 :>0:r1=1; 0:r3=1; 2:r1=0; 2:r3=0;
35899 :>0:r1=0; 0:r3=0; 2:r1=1; 2:r3=0;
70460 :>0:r1=0; 0:r3=1; 2:r1=1; 2:r3=0;
30449 :>0:r1=1; 0:r3=1; 2:r1=1; 2:r3=0;
42885 :>0:r1=0; 0:r3=0; 2:r1=0; 2:r3=1;
70068 :>0:r1=1; 0:r3=0; 2:r1=0; 2:r3=1;
1     :>0:r1=0; 0:r3=1; 2:r1=0; 2:r3=1;
41722 :>0:r1=1; 0:r3=1; 2:r1=0; 2:r3=1;
95857 :>0:r1=0; 0:r3=0; 2:r1=1; 2:r3=1;
30916 :>0:r1=1; 0:r3=0; 2:r1=1; 2:r3=1;
40818 :>0:r1=0; 0:r3=1; 2:r1=1; 2:r3=1;
214950:>0:r1=1; 0:r3=1; 2:r1=1; 2:r3=1;
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (0:r1=1 /\ 0:r3=0 /\ 2:r1=1 /\ 2:r3=0) is NOT validated
Hash=8ce05c9f86d49b2adfd5546bd471aa44
Time ppc-iriw-lwsync 1.33
```

The non-SC outcome does not show up.

Altering parameters may yield this outcome. In particular, we may try using all the available logical processors with option `-a 64`. Affinity control offers an alternative, which is enabled at compilation time with litmus option `-affinity`:

```
$ litmus ... -affinity incr1 ppc-iriw-lwsync.litmus -o ppc.tar
$ scp ppc.tar vargas:/var/tmp
```

Option `-affinity` takes one argument (`incr1` above) that specifies the increment used while allocating logical processors to test threads. Here, the (POSIX) threads created by the test (named T_0, T_1, T_2 and T_3 in Sec. 2.1) will get bound to logical processors 0, 1, 2, and 3, respectively.

Namely, by default, the logical processors are ordered as the sequence $0, 1, \ldots, A-1$ — where A is the number of available logical processors, which is inferred by the test executable[2]. Furthermore, logical processors are allocated to threads by applying the affinity increment while scanning the logical processor sequence. Observe that since the launch mode is changing (the default) threads T_k correspond to different test threads P_i at each run. The unpack compile and run sequence on `vargas` now yields the non-SC outcome, better outcome variety and a lower running time:

```
vargas$ tar xf /var/tmp/ppc.tar && sh comp.sh
vargas$ ./ppc-iriw-lwsync.exe
Test ppc-iriw-lwsync Allowed
Histogram (16 states)
166595:>0:r1=0; 0:r3=0; 2:r1=0; 2:r3=0;
2841  :>0:r1=1; 0:r3=0; 2:r1=0; 2:r3=0;
19581 :>0:r1=0; 0:r3=1; 2:r1=0; 2:r3=0;
86307 :>0:r1=1; 0:r3=1; 2:r1=0; 2:r3=0;
3268  :>0:r1=0; 0:r3=0; 2:r1=1; 2:r3=0;
9     :>0:r1=1; 0:r3=0; 2:r1=1; 2:r3=0;
21876 :>0:r1=0; 0:r3=1; 2:r1=1; 2:r3=0;
79354 :>0:r1=1; 0:r3=1; 2:r1=1; 2:r3=0;
21406 :>0:r1=0; 0:r3=0; 2:r1=0; 2:r3=1;
26808 :>0:r1=1; 0:r3=0; 2:r1=0; 2:r3=1;
1762  :>0:r1=0; 0:r3=1; 2:r1=0; 2:r3=1;
100381:>0:r1=1; 0:r3=1; 2:r1=0; 2:r3=1;
83005 :>0:r1=0; 0:r3=0; 2:r1=1; 2:r3=1;
72241 :>0:r1=1; 0:r3=0; 2:r1=1; 2:r3=1;
98047 :>0:r1=0; 0:r3=1; 2:r1=1; 2:r3=1;
216519:>0:r1=1; 0:r3=1; 2:r1=1; 2:r3=1;
Ok

Witnesses
Positive: 9, Negative: 999991
Condition exists (0:r1=1 /\ 0:r3=0 /\ 2:r1=1 /\ 2:r3=0) is validated
Hash=8ce05c9f86d49b2adfd5546bd471aa44
Time ppc-iriw-lwsync 0.67
```

One may change the affinity increment with the command line option `-i` of executable files. For instance, one binds the test threads to logical processors 0, 2, 4 and 6 as follows:

```
vargas$ ./ppc-iriw-lwsync.exe -i 2
Test ppc-iriw-lwsync Allowed
Histogram (15 states)
163114:>0:r1=0; 0:r3=0; 2:r1=0; 2:r3=0;
38867 :>0:r1=1; 0:r3=0; 2:r1=0; 2:r3=0;
48395 :>0:r1=0; 0:r3=1; 2:r1=0; 2:r3=0;
81191 :>0:r1=1; 0:r3=1; 2:r1=0; 2:r3=0;
```

[2] Parameter A is not to be confused with a of section 2.1. The former serves to compute logical threads while the latter governs the number of tests that run simultaneously.

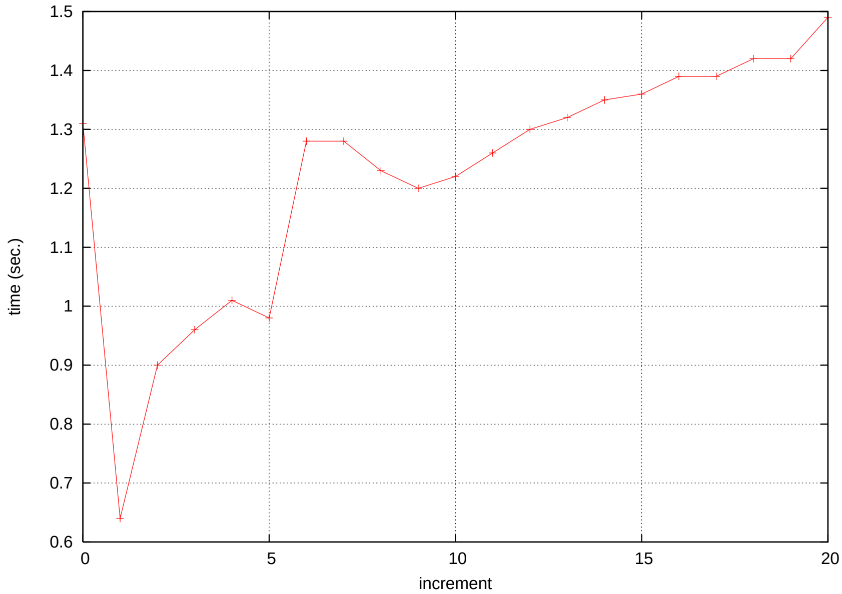
```
38912 :>0:r1=0; 0:r3=0; 2:r1=1; 2:r3=0;
70574 :>0:r1=0; 0:r3=1; 2:r1=1; 2:r3=0;
30918 :>0:r1=1; 0:r3=1; 2:r1=1; 2:r3=0;
47846 :>0:r1=0; 0:r3=0; 2:r1=0; 2:r3=1;
69048 :>0:r1=1; 0:r3=0; 2:r1=0; 2:r3=1;
5     :>0:r1=0; 0:r3=1; 2:r1=0; 2:r3=1;
42675 :>0:r1=1; 0:r3=1; 2:r1=0; 2:r3=1;
82308 :>0:r1=0; 0:r3=0; 2:r1=1; 2:r3=1;
30264 :>0:r1=1; 0:r3=0; 2:r1=1; 2:r3=1;
43796 :>0:r1=0; 0:r3=1; 2:r1=1; 2:r3=1;
212087:>0:r1=1; 0:r3=1; 2:r1=1; 2:r3=1;
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (0:r1=1 /\ 0:r3=0 /\ 2:r1=1 /\ 2:r3=0) is NOT validated
Hash=8ce05c9f86d49b2adfd5546bd471aa44
Time ppc-iriw-lwsync 0.89
```

One observe that the non-SC outcome does not show up with the new affinity setting.

2.2.2 Study of affinity

As illustrated by the previous example, both the running time and the outcomes of a test are sensitive to affinity settings. We measured running time for increasing values of the affinity increment from 0 (which disables affinity control) to 20, producing the following figure:



As regards outcome variety, we get all of the 16 possible outcomes only for an affinity increment of 1.

The differences in running times can be explained by reference to the mapping of logical processors to hardware. The machine `vargas` consists in four MCM's (Multi-Chip-Module), each MCM consists in four "chips", each chip consists in two cores, and each core may support two logical processors. As far as we know, by querying `vargas` with the AIX commands `lsattr`, `bindprocessor` and `llstat`, the MCM's hold the logical processors 0–15, 16–31, 32–47 and 48–63, each chip holds the logical processors $4k, 4k+1, 4k+2, 4k+3$ and each core holds the logical processors $2k, 2k+1$.

The measure of running times for varying increments reveals two noticeable slowdowns: from an increment of 1 to an increment of 2 and from 5 to 6. The gap between 1 and 2 reveals the benefits of SMT for our testing application. An increment of 1 yields both the greatest outcome variety and the minimal running time. The other gap may perhaps be explained by reference to MCM's: for a value of 5 the tests runs on the logical processors $0, 5, 10, 15$, all belonging to the same MCM; while the next affinity increment of 6 results in running the test on two different MCM ($0, 6, 12$ on the one hand and 18 on the other).

As a conclusion, affinity control provides users with a certain level of control over thread placement, which is likely to yield faster tests when threads are constrained to run on logical processors that are "close" one to another. The best results are obtained when SMT is effectively enforced. However, affinity control is no panacea, and the memory system may be stressed by other means, such as, for instance, allocating important chunks of memory (option `-s`).

2.2.3 Advanced control

For specific experiments, the technique of allocating logical processors sequentially by following a fixed increment may be two rigid. `litmus` offers a finer control on affinity by allowing users to supply the logical processors sequence. Notice that most users will probably not need this advanced feature.

Anyhow, so as to confirm that testing `ppc-iriw-lwsync` benefits from not crossing chip boundaries, one may wish to confine its four threads to logical processors 16 to 19, that is to the first chip of the second MCM. This can be done by overriding the default logical processors sequence by an user supplied one given as an argument to command-line option `-p`:

```
vargas$ ./ppc-iriw-lwsync.exe -p 16,17,18,19 -i 1
Test ppc-iriw-lwsync Allowed
Histogram (16 states)
186125:>0:r1=0; 0:r3=0; 2:r1=0; 2:r3=0;
1333  :>0:r1=1; 0:r3=0; 2:r1=0; 2:r3=0;
16334 :>0:r1=0; 0:r3=1; 2:r1=0; 2:r3=0;
83954 :>0:r1=1; 0:r3=1; 2:r1=0; 2:r3=0;
1573  :>0:r1=0; 0:r3=0; 2:r1=1; 2:r3=0;
9     :>0:r1=1; 0:r3=0; 2:r1=1; 2:r3=0;
19822 :>0:r1=0; 0:r3=1; 2:r1=1; 2:r3=0;
72876 :>0:r1=1; 0:r3=1; 2:r1=1; 2:r3=0;
20526 :>0:r1=0; 0:r3=0; 2:r1=0; 2:r3=1;
24835 :>0:r1=1; 0:r3=0; 2:r1=0; 2:r3=1;
1323  :>0:r1=0; 0:r3=1; 2:r1=0; 2:r3=1;
97756 :>0:r1=1; 0:r3=1; 2:r1=0; 2:r3=1;
78809 :>0:r1=0; 0:r3=0; 2:r1=1; 2:r3=1;
67206 :>0:r1=1; 0:r3=0; 2:r1=1; 2:r3=1;
94934 :>0:r1=0; 0:r3=1; 2:r1=1; 2:r3=1;
232585:>0:r1=1; 0:r3=1; 2:r1=1; 2:r3=1;
Ok

Witnesses
Positive: 9, Negative: 999991
Condition exists (0:r1=1 /\ 0:r3=0 /\ 2:r1=1 /\ 2:r3=0) is validated
Hash=8ce05c9f86d49b2adfd5546bd471aa44
Time ppc-iriw-lwsync 0.66
```

Thus we get results similar to the previous experiment on logical processors 0 to 3 (option `-i 1` alone).

We may also run four simultaneous instances (`-n 4`, parameter n of section 2.1) of the test on the four available MCM's:

```
vargas$ ./ppc-iriw-lwsync.exe -p 0,1,2,3,16,17,18,19,32,33,34,35,48,49,50,51 -n 4 -i 1
Test ppc-iriw-lwsync Allowed
Histogram (16 states)
...

Witnesses
Positive: 80, Negative: 3999920
Condition exists (0:r1=1 /\ 0:r3=0 /\ 2:r1=1 /\ 2:r3=0) is validated
Time ppc-iriw-lwsync 0.74
```

Obzserve that, for a negligible penalty in running time, the number of non-SC outcomes increases significantly.

By contrast, binding threads of a given instance of the test to different MCM's results in poor running time and no non-SC outcome.

```
vargas$ ./ppc-iriw-lwsync.exe -p 0,1,2,3,16,17,18,19,32,33,34,35,48,49,50,51 -n 4 -i 4
Test ppc-iriw-lwsync Allowed
Histogram (15 states)
...
Witnesses
Positive: 0, Negative: 4000000
Condition exists (0:r1=1 /\ 0:r3=0 /\ 2:r1=1 /\ 2:r3=0) is NOT validated
Time ppc-iriw-lwsync 1.48
```

In the experiement above, the increment is 4, hence the logical processors allocated to the first instance of the test are $0, 16, 32, 48$, of which indices in the logical processors sequence are $0, 4, 8, 12$, respectively. The next allocated index in the sequence is $12 + 4 = 16$. However, the sequence has 16 items. Wrapping around yields index 0 which happens to be the same as the starting index. Then, so as to allocate fresh processors, the starting index is incremented by one, resulting in allocating processors $1, 17, 33, 49$ (indices $1, 5, 9, 13$) to the second instance — see section 2.3 for the full story. Similarily, the third and fourth instances will get processors $2, 18, 34, 50$ and $3, 19, 35, 51$, respectively. Attentive readers may have noticed that the same experiment can be performed with option `-i 16` and no `-p` option.

Finally, users should probably be aware that at least some versions of linux for x86 feature a less obvious mapping of logical processors to hardware. On a bi-processor, dual-core, 2-ways hyperthreading, linux, AMD64 machine, we have checked that logical processors residing on the same core are k and $k + 4$, where k is an arbitray core number ranging from 0 to 3. As a result, a proper choice for favouring effective hyperthreading on such a machine is `-i 4` (or `-p 0,4,1,5,2,6,3,7 -i 1`). More worthwhile noticing, perhaps, the straightforward choice `-i 1` disfavours effective hyperthreading…

2.3 Controlling executable files

Test conditions Any executable file produced by litmus accepts the following command line options.

`-v` Be verbose, can be repeated to increase verbosity. Specifying `-v` is a convenient way to look at the default of options.

`-q` Be quiet.

`-a <n>` Run maximal number of tests concurrently for n available cores — parameter a in Sec. 2.1.

`-n <n>` Run n tests concurrently — parameter n in Sec. 2.1.

`-r <n>` Perform n runs — parameter r in Sec. 2.1.

`-fr <f>` Multiply r by f (f is a floating point number).

`-s <n>` Size of a run — parameter s in Sec. 2.1.

`-fs <f>` Multiply s by f.

`-f <f>` Multiply s by f and divide r by f.

Affinity If affinity control has been enabled at compilation time (by supplying option `-affinity incr1` to litmus, for instance), the executable file produced by litmus accepts the following two command line options.

`-p <ns>` Logical processors sequence. The sequence `<ns>` is a comma separated list of integers, The default sequence is infered by the executable as $0, 1, \ldots, A-1$, where A is the number of logical processors featured by the tested machine; or is a sequence specified at compile time with litmus option `-p`.

`-i <n>` Increment for allocating logical processors to threads. Default is specified at compile time by litmus option `-affinity incr<n>`. Notice that `-i 0` disable affinity control and that `.exe` files reject the `-i` option when affinity control has not been enabled at compile time.

Logical processors are allocated test instance by test instance (parameter n of Sec. 2.1) and then thread by thread, scanning the logical processor sequence left-to-right by steps of the given increment. More precisely, assume a logical processor sequence $P = p_0, p_1, \ldots, p_{A-1}$ and an increment i. The first processor allocated is p_0, then p_i, then p_{2i} etc, Indices in the sequence P are reduced modulo A so as to wrap around. The starting index of the allocation sequence (initially 0) is recorded, and coincidence with the index of the next processor to be allocated is checked. When coincidence occurs, a new index is computed, as the previous starting index plus one, which also becomes the new starting index. Allocation then proceeds from this new starting index. That way, 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. See section 2.2.3 for an example with $A = 16$ and $i = 4$.

3 Usage of litmus

Arguments

litmus takes file names as command line arguments. Those files are either a single litmus test, when having extension `.litmus`, or a list of file names, when prefixed by `@`. Of course, the file names in `@`files can themselves be `@`files.

Options

There are many command line options. We describe the more useful ones:

General behaviour

`-version` Show version number and exit.

`-libdir` Show installation directory and exit.

`-v` Be verbose, can be repeated to increase verbosity.

`-mach <name>` Read configuration file `name.cfg`. See the next section for the syntax of configuration files.

`-o <name.tar>` Cross compile tests into tar file `name.tar`.

`-index <@name>` Save the source names of compiled files in index file `@name`.

Test conditions The following options set the default values of the options of the executable files produced:

`-a <n>` Run maximal number of tests concurrently for n available cores — set default value for `-a` of Sec. 2.3. Default is 0 (run one test).

`-limit <bool>` Do not process tests with more than n threads, where n is the number of available cores defined above. Default is `false`.

`-r <n>` Perform n runs — set default value for option `-r` of Sec. 2.3. Default is 10.

`-s <n>` Size of a run — set default value for option `-s` of Sec. 2.3. Default is 100000.

The following two options enable affinity control. Affnity control is not implemeted for MacOs.

`-affinity (none|incr<n>)` Step for allocating logical processors to threads — set default value for option `-i` of Sec. 2.3. Default is `none`, *.i.e.* produced code does not feature affinity control. With `-affinity incr0`, produced code features affinity control, which executable files do not exercise by default.

`-i <n>` Alias for `-affinity incr<n>`.

`-p <ns>` Specify the sequence of logical processors, implies `-affinity incr1`. The notation `<ns>` stands for a comma separated list of integers. Set default value for option `-p` of Sec. 2.3.

Default for this `-p` option will let executable files compute the logical processor sequence themselves.

The following additional options control the various modes described in Sec. 2.1. Those cannot be changed without running `litmus` again:

`-barrier (user|pthread|none)` Set synchronisation mode, default `user`.

`-launch (changing|fixed)` Set launch mode, default `changing`.

`-cache <bool>` Enable or disable cache mode, default disabled.

`-mem (indirect|direct)` Set memory mode, default `indirect`.

`-para (self|shell)` Perform several tests concurrently, either by forking POSIX threads (as described in Sec. 2.1), or by forking Unix processes. Only applies for cross compilation. Default is `self`.

`-prealloc <bool>` Enable or disable pre-allocation mode, default disabled. In pre-allocation mode, memory is allocated before forking any thread.

`-preload <bool>` Enable or disable preload, default enabled.

`-safer <bool>` Enable or disable safer mode, default enabled. In safer mode, executable files perform some consistency checks. Those are intended both for debugging and for dynamically checking some assumption on POSIX threads that we rely upon.

`-speedcheck <bool>` Enable or disable quick condition check mode, default enabled. When enabled, stop test as soon as condition is settled.

`-ccopts <flags>` Set additional `gcc` compilation flags (defaults: X86=`"-fomit-frame-pointer -O2"`, PPC=`"-O"`).

Target architecture description Litmus compilation chain may slightly vary depending on the following parameters:

`-os (linux|mac|aix)` Set target operating system. This parameter mostly impacts some of `gcc` options. Default `linux`.

`-ws (w32|w64)` Set word size. This option first selects `gcc` 32 or 64 bits mode, by providing it with the appropriate option (`-m32` or `-m64` on linux, `-maix32` or `-maix64` on AIX). It also slightly impacts code generation in the corner case where memory locations hold other memory locations. Default is a bit contrived: it acts as `w32` as regards code generation, while it provides no 32/64 bits mode selection option to `gcc`.

Configuration files

The syntax of configuration files is minimal: lines "*key* `=` *arg*" are interpreted as setting the value of parameter *key* to *arg*. Each parameter has a corresponding option, usually `-`*key*, except for single-letter options:

option	*key*	*arg*
`-a`	avail	integer
`-s`	size_of_test	integer
`-r`	number_of_run	integer
`-p`	procs	list of integers

As command line option are processed left-to-right, settings from a configuration file (option `-mach`) can be overridden by a later command line option. Some configuration files for the machines we have tested are present in the distribution. As an example here is the configuration file `hpcx.cfg`.

```
size_of_test = 2000
number_of_run = 20000
os = AIX
ws = W32
# A node has 16 cores X2 (SMT)
avail = 32
```

Lines introduced by `#` are comments and are thus ignored.

Configuration files are searched first in the current directory; then in any directory specified by setting the shell environment variable `LITMUSDIR`; and then in litmus installation directory, which is defined while compiling litmus.

Part II
Generating tests

The authors of diy are Jade Alglave and Luc Maranget (INRIA Paris–Rocquencourt).

4 Preamble

We wrote diy as part of our empirical approach to studying relaxed memory models: developing in tandem testing tools and models of multiprocessor behaviour. In this tutorial, we attempt an independent tool presentation. Readers interested by the companion formalism are invited to refer to our CAV 2010 publication [1].

4.1 Relaxation of Sequential Consistency

Relaxation is one of the key concepts behind simple analysis of weak memory models. We define a candidate relaxation by reference to the most natural model of parallel execution in shared memory: Sequential Consistency (SC), as defined by L. Lamport [3]. A parallel program running on a sequentially consistent machine behaves as an interleaving of its sequential threads.

Consider once more the example `classic.litmus`:

```
X86 classic
"Fre PodWR Fre PodWR"
{ x=0; y=0; }
 P0          | P1          ;
 MOV [y],$1  | MOV [x],$1  ; #(a)Wy1  | (c)Wx1
 MOV EAX,[x] | MOV EAX,[y] ; #(b)Rx0  | (d)Ry0
exists (0:EAX=0 /\ 1:EAX=0)
```

To focus on interaction through shared memory, let us consider memory accesses, or *memory events*. A memory event will hold a direction (write, written W, or read, written R), a memory location (written x, y) a value and a unique label. In any run of the simple example above, four memory events occur: two writes $(c)\,\mathrm{Wx}1$ and $(a)\,\mathrm{Wy}1$ and two reads $(b)\,\mathrm{Rx}v_1$ with a certain value v_1 and $(d)\,\mathrm{Ry}v_2$ with a certain value v_2.

If the program's behaviour is modelled by the interleaving of its events, the first event must be a write of value `1` to location `x` or `y` and at least one of the loads must see a `1`. Thus, a SC machine would exhibit only three possible outcomes for this test:

Allowed: 0:EAX = 0 $\wedge$ 1:EAX = 1
Allowed: 0:EAX = 1 $\wedge$ 1:EAX = 0
Allowed: 0:EAX = 1 $\wedge$ 1:EAX = 1

However, running (see Sec. 1.1) this test on a x86 machine yields an additional result:

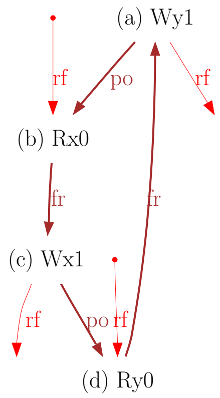
Allowed: 0:EAX = 0 $\wedge$ 1:EAX = 0

And indeed, x86 allows each write-read pair on both processors to be reordered [2]: thus the write-read pair in program order is relaxed on each of these architectures. We cannot use SC as an accurate memory model for modern architectures. Instead we analyse memory models as *relaxing* the ordering constraints of the SC memory model.

4.2 Introduction to candidate relaxations

Consider again our classical example, from a SC perspective. We briefly argued that the outcome "0:EAX = 0 ∧ 1:EAX = 0" is forbidden by SC. We now present a more complete reasoning:

- From the condition on outcome, we get the values in read events: (b) Rx0 and (d) Ry0.
- Because of these values, (b) Rx0 must precede the write (c) Wx1 in the final interleaving of SC. Similarly, (d) Ry0 must precede the write (a) Wy1. This we note $(b) \xrightarrow{\text{fr}} (c)$ and $(d) \xrightarrow{\text{fr}} (a)$.
- Because of sequential execution order on one single processor (a.k.a. *program order*), (a) Wy1 must precede (b) Rx0 (first processor); while (c) Wx1 must precede (d) Ry0 (second processor). This we note $(a) \xrightarrow{\text{po}} (b)$ and $(c) \xrightarrow{\text{po}} (d)$.
- We synthesise the four constraints above as the following graph:

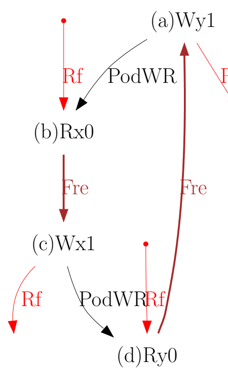


Constraint arrows or *global* arrows are shown in brown colour. As the graph contains a cycle of brown arrows, the events cannot be ordered. Hence the execution presented is not allowed by SC.

The key idea of diy resides in producing programs from similar cycles. To that aim, the edges in cycles must convey additional information:

- For $\xrightarrow{\text{po}}$ edges, we consider whether the locations of the events on both sides of the edge are the same or not ('s' or 'd'); and the direction of these events (W or R). For instance the two $\xrightarrow{\text{po}}$ edges in the example are PodWR. (program order edge between a write and a read whose locations are different).
- For $\xrightarrow{\text{fr}}$ edges, we consider whether the processor of the events on both sides of the edge are the same or not ('i' for internal, or 'e' for external). For instance the two $\xrightarrow{\text{fr}}$ edges in the example are Fre.

So far so good, but our x86 machine produced the outcome 0:EAX=0 ∧ 1:EAX=0. The Intel Memory Ordering White Paper [2] specifies: "Loads may be reordered with older stores to different locations", which we rephrase as: PodWR is relaxed. Considering Fre to be safe, we have the graph:



And the brown sub-graph becomes acyclic.

We shall see later why we choose to relax PodWR and not Fre. At the moment, we observe that we can assume PodWR to be relaxed and Fre not to be (*i.e.* to be *safe*) and test our assumptions, by producing and running more litmus tests. The diy suite precisely provides tools for this approach.

As a first example, `classic.litmus` can be created as follows:

```
% diyone -arch X86 -name classic Fre PodWR Fre PodWR
```

As a second example, we can produce several similar tests as follows:

```
% diy -arch X86 -safe Fre -relax PodWR -name classic
Generator produced 2 tests
Relaxations tested: {PodWR}
```

diy produces two litmus tests, `classical000.litmus` and `classical001.litmus`, plus one index file `@all`. One of the litmus tests generated is the same as above, while the new test is:

```
% cat classic001.litmus
X86 classic001
"Fre PodWR Fre PodWR Fre PodWR"
Cycle=Fre PodWR Fre PodWR Fre PodWR
Relax=PodWR
Safe=Fre
{ }
 P0          | P1          | P2          ;
 MOV [z],$1  | MOV [x],$1  | MOV [y],$1  ;
 MOV EAX,[x] | MOV EAX,[y] | MOV EAX,[z] ;
exists (0:EAX=0 /\ 1:EAX=0 /\ 2:EAX=0)
% cat @all
# diy -arch X86 -safe Fre -relax PodWR -name classic
# Revision: 3333
classic000.litmus
classic001.litmus
```

diy first generates cycles from the candidate relaxations given as arguments, up to a limited size, and then generates litmus tests from these cycles.

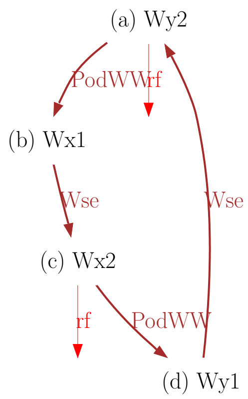
4.3 More candidate relaxations

We assume the memory to be *coherent*. Coherence implies that, in a given execution, the writes to a given location are performed by following a sequence, or *coherence order*, and that all processors see the same sequence.

In diy, the coherence orders are specified indirectly. For instance, the candidate relaxation Wse (resp. Wsi) specifies two writes, performed by different processors (resp. the same processor), to the same location ℓ, the first write preceding the second in the coherence order of ℓ. The condition of the produced test then selects the specified coherence orders. Consider for instance:

```
% diyone -arch X86 -name ws Wse PodWW Wse PodWW
```

The cycle that reveals a violation of the SC memory model is:



So the coherence order is 0 (initial store, not depicted), 1, 2 for both locations x and y. While the produced test is:

```
X86 ws "Wse PodWW Wse PodWW"
{ }
 P0          | P1          ;
 MOV [y],$2 | MOV [x],$2 ;
 MOV [x],$1 | MOV [y],$1 ;
exists (x=2 /\ y=2)
```

By the coherence hypothesis, checking the final value of locations suffices to characterise those two coherence orders, as expressed by the final condition of `ws`:

```
exists (x=2 /\ y=2)
```

See Sec. 7 for alternative means to identify coherence orders.

Candidate relaxations Rfe and Rfi relate writes to reads that load their value. We are now equipped to generate the famous iriw test (independent reads of independent writes):

```
% diyone -arch X86 Rfe PodRR Fre Rfe PodRR Fre -name iriw
```

We generate its internal variation (*i.e.* where all Rfe are replaced by Rfi) as easily:

```
% diyone -arch X86 Rfi PodRR Fre Rfi PodRR Fre -name iriw-internal
```

We get the cycles of Fig. 1, and the litmus tests of Fig. 2.

Candidate relaxations given as arguments really are a "concise specification". As an example, we get iriw for Power, simply by changing `-arch X86` into `-arch PPC`.

```
% diyone -arch PPC Rfe PodRR Fre Rfe PodRR Fre
PPC a
"Rfe PodRR Fre Rfe PodRR Fre"
{
```

Figure 1: Cycles for iriw and iriw-internal

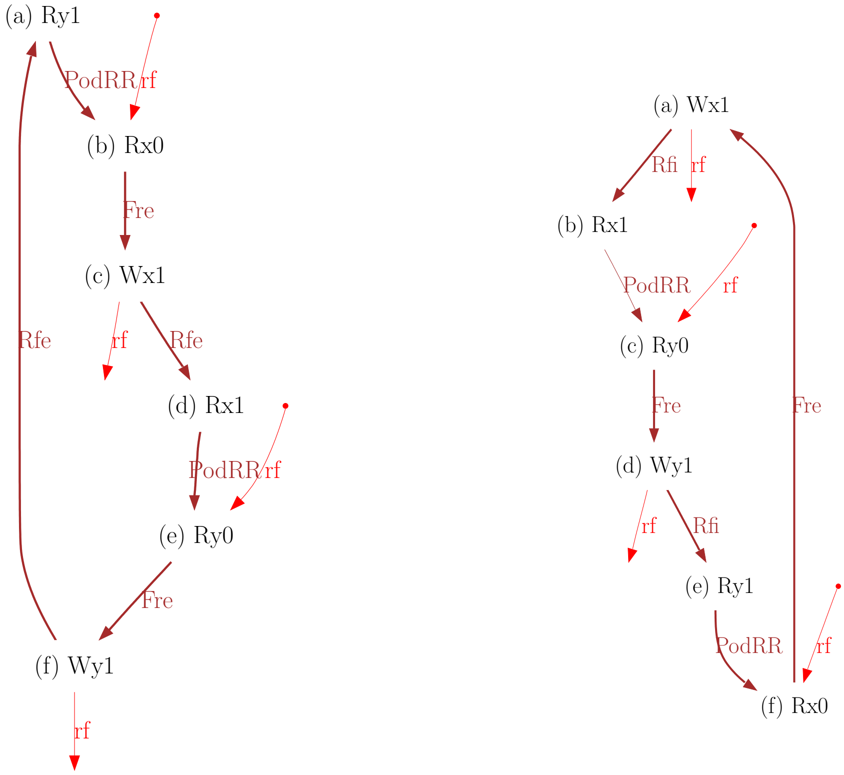


Figure 2: Litmus tests iriw and iriw-internal

```
X86 iriw
"Rfe PodRR Fre Rfe PodRR Fre"
{ }
 P0          | P1          | P2          | P3          ;
 MOV EAX,[y] | MOV [x],$1  | MOV EAX,[x] | MOV [y],$1  ;
 MOV EBX,[x] |             | MOV EBX,[y] |             ;
exists (0:EAX=1 /\ 0:EBX=0 /\ 2:EAX=1 /\ 2:EBX=0)
```

```
X86 iriw-internal
"Rfi PodRR Fre Rfi PodRR Fre"
{ }
 P0          | P1          ;
 MOV [x],$1  | MOV [y],$1  ;
 MOV EAX,[x] | MOV EAX,[y] ;
 MOV EBX,[y] | MOV EBX,[x] ;
exists
(0:EAX=1 /\ 0:EBX=0 /\
 1:EAX=1 /\ 1:EBX=0)
```

```
0:r2=y; 0:r4=x;
1:r2=x;
2:r2=x; 2:r4=y;
3:r2=y;
}
 P0           | P1           | P2           | P3           ;
 lwz r1,0(r2) | li r1,1      | lwz r1,0(r2) | li r1,1      ;
 lwz r3,0(r4) | stw r1,0(r2) | lwz r3,0(r4) | stw r1,0(r2) ;
exists (0:r1=1 /\ 0:r3=0 /\ 2:r1=1 /\ 2:r3=0)
```

Also notice that without the `-name` option, diyone writes its result to standard output.

4.4 Summary of simple candidate relaxations

We summarise the candidate relaxations available on all architectures.

4.4.1 Communication candidate relaxations

We call communication candidate relaxations the relations between two events communicating through memory, though they could belong to the same processor. Thus, these events operate on the same memory location.

diy syntax	Source	Target	Processor	Additional property
`Rfi`	W	R	Same	Target reads its value from source
`Rfe`	W	R	Different	Target reads its value from source
`Wsi`	W	W	Same	Source precedes target in coherence order
`Wse`	W	W	Different	Source precedes target in coherence order
`Fri`	R	W	Same	Source reads a value from a write that precedes target in coherence order
`Fre`	R	W	Different	Source reads a value from a write that precedes target in coherence order

4.4.2 Program order candidate relaxations

We call program order candidate relaxations each relation between two events in the program order. These events are on the same processor, since they are in program order. As regards code output, diy interprets a program order candidate relaxation by generating two memory instructions (load or store) following one another.

Program order candidate relaxations have the following syntax:

$$\mathrm{Po(s|d)(R|W)(R|W)}$$

where:

- s (resp. d) indicates that the two events are to the same (resp. different) location(s);
- R (resp. W) indicates an event to be a read (resp. a write);

In practice, we have:

diy syntax	Source	Target	Location
`PosRR`	R	R	Same
`PodRR`	R	R	Diff
`PosRW`	R	W	Same
`PodRW`	R	W	Diff
`PosWW`	W	W	Same
`PodWW`	W	W	Diff
`PosWR`	W	R	Same
`PodWR`	W	R	Diff

It is to be noticed that PosWR, PosWW and PosRW are similar to Rfi, Wsi and Fri, respectively. More precisely, diy is unable to consider a PosWR (or PosWW, or PosRW) candidate relaxation as not being also a Rfi (or Wsi, or Fri) candidate relaxation. However, litmus tests conditions may be more informative in the case of Rfi and Fri.

4.4.3 Fence candidate relaxations

Relaxed architectures provide specific instructions, namely *barriers* or *fences*, to enforce order of memory accesses. In diy the presence of a fence instruction is specified with fence candidate relaxations, similar to program order candidate relaxations, except that a fence instruction is inserted. Hence we have FencedsRR, FenceddRR. etc. The inserted fence is the strongest fence provided by the architecture — that is, `mfence` for x86 and `sync` for Power.

Fences can also be specified by using specific names. More precisely, we have MFence for x86; while on Power we have Sync and LwSync. Hence, to yield two reads to different locations and separated by the lightweight Power barrier `lwsync`, we specify LwSyncdRR.

5 Testing candidate relaxations with diy

Teh tool diy can probably be used in various, creative, ways; but the tool first stems from our technique for testing relaxed memory models. The `-safe` and `-relax` options are crucial here. We describe our technique by the means of an example: X86-TSO.

5.1 Principle

Before engaging in testing it is important to categorise candidate relaxations as safe or relaxed.

This can done by interpretation of vendor's documentation. For instance, the iriw test of Sec. 4.3 is the example 7.7 of [2] "Stores Are Seen in a Consistent Order by Other Processors", with a Forbid specification. Hence we deduce that Fre, Rfe and PodRR are safe. Then, from test iriw-internal of Sec. 4.3, which is Intel's test 7.5 "Intra-Processor Forwarding Is Allowed" with an allow specification, we deduce that Rfi is relaxed. Namely, the cycle of iriw-internal is "Fre Rfi PodRR Fre Rfi PodRR". Therefore, the only possibility is for Rfi to be relaxed.

Overall, we deduce:

- Candidate relaxations PosWR (Rfi) and PodWR are relaxed
- The remaining candidate relaxations PosRR, PodRR, PosWW (Wsi), PodWW, PosRW (Fri), Fre and Wse are safe. Fence relaxations FencedsWR and FenceddWR are also safe and worth testing.

Of course these remain assumptions to be tested. To do so, we perform one series of tests per relaxed candiate relaxation, and one series of tests for confirming safe candidate relaxations as much as possible. Let S be all safe candidate relaxations.

- Let r be a relaxed candidate relaxation. We produce tests for confirming r being relaxed by `diy -relax` r `-safe` S. We run these tests with `litmus`. If one of the tests yields `Ok`, then r is confirmed to be relaxed, provided the experiments on S below do not fail.
- For confirming the safe set, we produce tests by `diy -safe` S. We run these tests as much as possible and expect never to see `Ok`.

Namely, `diy` builds cycles as follows:

- `diy -relax` r `-safe` S build cycles with at least one r taking other candidate relaxations from S.
- `diy -safe` S build cycles from the candidate relaxations in S.

For the purpose of confirming relaxed candidate relaxations, S can be replaced by a subset.

5.2 Testing x86

Repeating command line options is painful and error prone. Besides, configuration parameters may get lost. Thus, we regroup those in configuration files that simply list the options to be passed to `diy`, one option per line. For instance here is the configuration file for testing the safe relaxations of x86, `x86-safe.conf`.

```
#safe x86 conf file
-arch X86
#Generate tests on four processors or less
-nprocs 4
#From cycles of size at most six
-size 6
#With names safe000, safe0001,...
-name safe
#List  of safe relaxations
-safe PosR* PodR* PodWW PosWW Rfe Wse Fre FencedsWR FenceddWR
```

Observe that the syntax of candidate relaxations allows one shortcut: the wildcard `*` stands for `W` and `R`. Thus `PodR*` gets expanded to the two candidate relaxations `PodRR` and `PodRW`.

We get safe tests by issuing the following command, preferably in a specific directory, say `safe`.

```
% diy -conf x86-safe.conf
Generator produced 38 tests
Relaxations tested: {}
```

Here are the configuration files for confirming that Rfi and PodWR are relaxed, `x86-rfi.conf` and `x86-podwr.conf`.

```
#rfi x86 conf file
-arch X86
-nprocs 4
-size 6
-name rfi
-safe PosR* PodR* PodWW PosWW Rfe Wse Fre FencedsWR FenceddWR
-relax Rfi
```

```
#podrw x86 conf file
-arch X86
-nprocs 4
-size 6
-name podwr
-safe Fre
-relax PodWR
```

Notice that we used the complete safe list in `x86-rfi.conf` and a reduced list in `x86-podwr.conf`. Tests are to be generated in specific directories.

```
% cd rfi
% diy -conf x86-rfi.conf
Generator produced 11 tests
Relaxations tested: {Rfi}
% cd ../podwr
% diy -conf x86-podwr.conf
Generator produced 2 tests
Relaxations tested: {PodWR}
% cd ..
```

Now, let us run all tests at once, with the parameters of machine `saumur` (4 physical cores with hyper-threading):

```
% litmus -mach saumur rfi/@all > rfi/saumur.rfi.00
% litmus -mach saumur podwr/@all > podwr/saumur.podwr.00
% litmus -mach saumur safe/@all > safe/saumur.safe.00
```

If your machine has 2 cores only, try `litmus -a 2 -limit true`...

We now look for the tests that have validated their condition in the result files of litmus. A simple tool, readRelax, does the job:

```
% readRelax rfi/saumur.rfi.00 podwr/saumur.podwr.00 safe/saumur.safe.00
   .
   .
   .
** Relaxation summary **
{Rfi} With {Rfe, Fre, Wse, PodRW, PodRR} {Rfe, Fre, PodRR}\
{Fre, Wse, PodWW, PodRR} {Fre, PosWW, PodRR, MFencedWR}\
{Fre, PodWW, PodRR, MFencedWR} {Fre, PodRR} {Fre, PodRR, MFencedWR}
{PodWR} With {Fre}
```

The tool readRelax first lists the result of all tests (which is omitted above), and then dumps a summary of the relaxations it found. The sets of the candidate relaxations that need to be safe for the tests to indeed reveal a relaxed candidate relaxation are also given. Here, Rfi and PodWR are confirmed to be relaxed, while no candidate relaxation in the safe set is found to be relaxed. Had it been the case, a line `{} With {...}` would have occurred in the relaxation summary. The safe tests need to be run a lot of times, to increase our confidence in the safe set.

6 Additional relaxations

We introduce some additional candidate relaxations that are specific to the Power architecture. We shall not detail here our experiments on Power machines. See our experience report `http://diy.inria.fr/phat/` for more details.

6.1 Intra-processor dependencies

In a very relaxed architecture such as Power, *intra-processor dependencies* becomes significant. Roughly, intra-processor dependencies fall into two categories:

Data dependencies occur when a memory access instruction reads a register whose contents depends upon a previous (in program order) load. In diy we specify such a dependency as:

Dp(s|d)(R|W)

where, as usual, s (resp. d) indicates that the source and target events are to the same (resp. different) location(s); and R (resp. W) indicates that the target event is a read (resp. a write). As a matter of fact, we do not need to specify the direction of the source event, since it always is a read.

Control dependencies occur when the execution of a memory access is conditioned by the contents of a previous load. Their syntax is similar to the one of Dp relaxations, with a Ctrl tag:

$$\mathrm{Ctrl}(\mathrm{s}\,|\,\mathrm{d})(\mathrm{R}\,|\,\mathrm{W})$$

In the produced code, diy expresses a data dependency by a *false dependency* (or *dummy dependency*) that operates on the address of the target memory access. For instance:

```
% diyone DpdW Rfe DpdW Rfe
PPC a "DpdW Rfe DpdW Rfe"
{ 0:r2=x; 0:r5=y; 1:r2=y; 1:r5=x; }
 P0             | P1             ;
 lwz r1,0(r2)   | lwz r1,0(r2)   ;
 xor r3,r1,r1   | xor r3,r1,r1   ;
 li r4,1        | li r4,1        ;
 stwx r4,r3,r5 | stwx r4,r3,r5 ;
exists (0:r1=1 /\ 1:r1=1)
```

On P_0, the effective address of the indexed store `stwx r4,r3,r5` depends on the contents of the index register `r3`, which itself depends on the contents of `r1`. The dependency is a "false" one, since the contents of `r3` always is zero, regardless of the contents of `r1`.

A control dependency is implemented by the means of an useless compare and branch sequence, plus the `isync` instruction when the target event is a load. For instance

```
% diyone CtrldR Fre SyncdWW Rfe
PPC a
"CtrldR Fre SyncdWW Rfe"
{ 0:r2=x; 0:r4=y; 1:r2=y; 1:r4=x; }
 P0            | P1            ;
 li r1,1       | lwz r1,0(r2) ;
 stw r1,0(r2) | cmpw r1,r1    ;
 sync          | beq  LC00     ;
 li r3,1       | LC00:         ;
 stw r3,0(r4) | isync         ;
               | lwz r3,0(r4) ;
exists (1:r1=1 /\ 1:r3=0)
```

Of course, in both cases, we assume that dependencies are not "optimised out" by the assembler or the hardware.

6.2 Composite relaxations and cumulativity

Users may specify a small sequence of single candidate relaxations as behaving as a single candidate relaxation to diy. The syntax is:

$$[r1,\, r2,\, \ldots]$$

The main usage of the feature is to specify *cumulativity candidate relaxations*, that is, the sequence of Rfe and of a fence candidate relaxation (A-cumulativity), the sequence of a fence candidate relaxation and of Rfe (B-cumulativity), or both (AB-cumulativity).

Cumulativity candidate relaxations are best expressed by the following syntactical shortcuts: let r be a fence candidate relaxation, then ACr stands for [Rfe,r], BCr stands for [r,Rfe], while ABCr stands for [Rfe,r,Rfe],

Hence, a simple way to generate iriw-like (see Sec. 4.3) litmus tests with `lwsync` is as follows:

```
% diy -name iriw-lwsync -nprocs 8 -size 8 -relax ACLwSyncdRR -safe Fre
Generator produced 3 tests
Relaxations tested: {ACLwSyncdRR}
```

where we have for instance:

```
% cat iriw-lwsync001.litmus
PPC iriw-lwsync001
"Fre Rfe LwSyncdRR Fre Rfe LwSyncdRR Fre Rfe LwSyncdRR"
Cycle=Fre Rfe LwSyncdRR Fre Rfe LwSyncdRR Fre Rfe LwSyncdRR
Relax=ACLwSyncdRR
Safe=Fre
{
0:r2=z; 0:r4=x; 1:r2=x;
2:r2=x; 2:r4=y; 3:r2=y;
4:r2=y; 4:r4=z; 5:r2=z;
}
 P0           | P1           | P2           | P3           | P4           | P5           ;
 lwz r1,0(r2) | li r1,1      | lwz r1,0(r2) | li r1,1      | lwz r1,0(r2) | li r1,1      ;
 lwsync       | stw r1,0(r2) | lwsync       | stw r1,0(r2) | lwsync       | stw r1,0(r2) ;
 lwz r3,0(r4) |              | lwz r3,0(r4) |              | lwz r3,0(r4) |              ;
exists (0:r1=1 /\ 0:r3=0 /\ 2:r1=1 /\ 2:r3=0 /\ 4:r1=1 /\ 4:r3=0)
```

7 Identifying coherence orders with observers

We first produce the "*four writes*" test `W4` for Power:

```
% diyone -name W4 -arch PPC PodWW Wse PodWW Wse
% cat W4.litmus
PPC W4
"PodWW Wse PodWW Wse"
{ 0:r2=x; 0:r4=y; 1:r2=y; 1:r4=x; }
 P0           | P1           ;
 li r1,2      | li r1,2      ;
 stw r1,0(r2) | stw r1,0(r2) ;
 li r3,1      | li r3,1      ;
 stw r3,0(r4) | stw r3,0(r4) ;
exists (x=2 /\ y=2)
```

Test `W4` is the Power version of X86 test `ws` of Sec. 4.3. In that section, we argued that the final condition `exists (x=2 /\ y=2)` suffices to identify the coherence orders 0, 1, 2 for locations `x` and `y`. As a consequence, a positive final condition reveals the occurrence of the specified cycle: Wse PodWW Wse PodWW.

7.1 Simple observers

Observers provide an alternative, perhaps more intuitive, means to identify coherence orders: an observer simply is an additional thread that performs several loads from the same location in sequence. Here, loading value 1 and then value 2 from location `x` identifies the coherence order 0, 1, 2. The command line switch `-obs force` commands the production of observers (test `W4Obs`):

```
% diyone -name W4Obs -obs force -obstype straight -arch PPC PodWW Wse PodWW Wse
% cat W4Obs.litmus
PPC W4Obs
"PodWW Wse PodWW Wse"
{ 0:r2=x; 1:r2=y; 2:r2=x; 2:r4=y; 3:r2=y; 3:r4=x; }
 P0            | P1            | P2            | P3            ;
 lwz r1,0(r2) | lwz r1,0(r2) | li r1,2       | li r1,2       ;
 lwz r3,0(r2) | lwz r3,0(r2) | stw r1,0(r2) | stw r1,0(r2) ;
               |               | li r3,1       | li r3,1       ;
               |               | stw r3,0(r4) | stw r3,0(r4) ;
exists (0:r1=1 /\ 0:r3=2 /\ 1:r1=1 /\ 1:r3=2)
```

Thread `P0` observes location `x`, while thread `P1` observes location `x`. With respect to `W4`, final condition has changed, the direct observation of the final contents of locations `x` and `y` being replaced by two successive observations of the contents of `x` and `y`.

It should first be noticed that the reasoning above assumes that having the same thread to read 1 from say `x` and then 2 implies that 1 takes place before 2 in the coherence order of `x`. This need not be the case in general — although it holds for Power. Moreover, running `W4` and `W4Obs` yields contrasted results. While a positive conclusion is immediate for `W4`, we were not able to reach a similar conclusion for `W4Obs`. As a matter of fact, `W4Obs` yielding Ok stems from the still-to-be-observed coincidence of several events: *both* observers threads must run at the right pace to observe the change from 1 to 2, while the cycle must indeed occur.

7.2 More observers

A simple observer consisting of loads performed in sequence is a *straight* observer. We define two additional sorts of observers: *fenced* observers, where loads are separated by the strongest fence available, and *loop* observers, which poll on location contents change. Those are selected by the homonymous tags given as arguments to the command line switch `-obstype`. For instance, we get the test `W4ObsFenced` by:

```
% diyone -name W4ObsFenced -obs force -obstype fenced -arch PPC PodWW Wse PodWW Wse
% cat W4ObsFenced.litmus
PPC W4ObsFenced
"PodWW Wse PodWW Wse"
{ 0:r2=x; 1:r2=y; 2:r2=x; 2:r4=y; 3:r2=y; 3:r4=x; }
 P0            | P1            | P2            | P3            ;
 lwz r1,0(r2) | lwz r1,0(r2) | li r1,2       | li r1,2       ;
 sync          | sync          | stw r1,0(r2) | stw r1,0(r2) ;
 lwz r3,0(r2) | lwz r3,0(r2) | li r3,1       | li r3,1       ;
               |               | stw r3,0(r4) | stw r3,0(r4) ;
exists (0:r1=1 /\ 0:r3=2 /\ 1:r1=1 /\ 1:r3=2)
```

Invoking diyone as "`diyone -obs force -obstype loop ...`" yields the additional test `W4ObsLoop`. The HTML version of this document provides details.

As an indication of the performance of the various sorts of observers, the following table summarises a litmus experiment performed on a 4-cores 2-ways SMT Power6 machine.

	W4	W4Obs	W4Fenced	W4Loop
Positive	29k/400M	0/200M	585/200M	347/200M
States	4/4	42/49	49/49	16/16

The first row "Positive" shows the number of observed positive outcomes/total number of outcomes produced. The second row "(Final) States" shows the number of different outcomes observed in practice/theoretical value. For instance, in the case of `W4`, we observed the positive outcome `x=2 /\ y=2` about 29 thousands times out of a total of 400 millions outcomes. Moreover, there are four possible outcomes: `x=1 /\ y=1`, `x=1 /\ y=2`, `x=2 /\ y=1` and `x=2 /\ y=2`, which we all observed at least once. As a conclusion, all techniques achieve decent results, except straight observers.

7.3 Three stores or more

In test `W4` the coherence orders sequence two writes. If there are three writes or more to the same location, it is no longer possible to identify a coherence order by observing the final contents of the memory location involved. In other words, observers are mandatory.

The argument to the `-obs` switch commands the production of observers. It can take three values:

`accept` Produce observers when absolutely needed. More precisely, given memory location `x`, no equality on `x` appears in the final condition for zero or one write to `x`, one such appears for two writes, and observers are produced for three writes or more.

`avoid` Never produce observers, *i.e.* fail when there are three writes to the same location.

`force` Produce observers for two writes or more.

With diyone, one easily build a three writes test as for instance the following `W5`:

```
% diyone -obs accept -obstype fenced -arch PPC -name W5 Wse Wse PodWW Wse PodWW
% cat W5.litmus
PPC W5
"Wse Wse PodWW Wse PodWW"
{ 0:r2=y; 1:r2=y; 1:r4=x; 2:r2=x; 2:r4=y; 3:r2=y; }
 P0           | P1           | P2           | P3           ;
 lwz r1,0(r2) | li r1,3      | li r1,2      | li r1,2      ;
 sync         | stw r1,0(r2) | stw r1,0(r2) | stw r1,0(r2) ;
 lwz r3,0(r2) | li r3,1      | li r3,1      |              ;
 sync         | stw r3,0(r4) | stw r3,0(r4) |              ;
 lwz r4,0(r2) |              |              |              ;
exists (x=2 /\ 0:r1=1 /\ 0:r3=2 /\ 0:r4=3)
```

As apparent from the code above, we have a fenced observer thread on `y` (`P0`), while the final state of `x` is observed directly (`x=2`). The command line switch `-obs force` would yield two observers, while `-obs avoid` would lead to failure.

8 Command usage

The diy suite consists in three tools:

diyone generates one litmus test from the specification of a violation of the sequential consistency memory model as a cycle—see Sec. 4.2.

diy generates several tests, aimed confirming that candidate relaxations are relaxed or safe—see Sec. 5.

readRelax Extract relevant information from the results of tests—see Sec. 5.2.

8.1 Usage of diyone

diyone takes a list of candidate relaxations as arguments and outputs a litmus test. Note that diyone may fail to produce the test, with a message that briefly details the failure.

```
% diyone Rfe Rfe PodRR
Test a [Rfe Rfe PodRR] failed:
Impossible direction PodRR Rfe
```

diyone accepts the following documented options.

`-v` Be verbose, repeat to increase verbosity.

`-version` Show version number and exit.

`-obs <accept|avoid|force>` Management of observers, default is `avoid`. See Sec. 7.3.

`-obstype <fenced|loop|straight>` Style of observers, default is `fenced`. See Sec. 7.2.

`-optcond` Optimise conditions by disregarding the values of loads that are neither the target of Rf, nor the source of Fr. This is the default.

`-nooptcond` Do not optimise conditions.

`-name <name>` Set the name of the test to `<name>` and output it into file `<name>.litmus`. By default, the test name is `a` and output goes to standard output.

`-arch <X86|PPC>` Set architecture. Default is `PPC`.

8.2 Usage of diy

The tool diy accepts the same options as diyone, option `-name <name>` being mandatory and setting the base name of generated litmus tests: *i.e.* diy produces tests `<name>000`, `<name>001`, etc., in files with extension `.litmus`. Moreover, diy produces an index file `@all` that lists file names `<name>000.litmus`, `<name>001.litmus` etc.

diy also accepts the following, additional, documented options.

`-conf <file>` Read configuration file `<file>`. A configuration file consists in a list of options, one option per line. Lines introduced by `#` are comments and are thus ignored.

`-o <name.tar>` Output litmus tests as an archive `<name.tar>`. The default is to output them in the current directory.

`-size <n>` Set the maximal size of cycles. Default is 6.

`-exact` Produce cycles of size exactly `<n>`, in place of size up to `<n>`.

`-nprocs <n>` Reject tests with more than `<n>` threads. Default is 4.

`-eprocs` Produce tests with exactly `<n>` threads, where `<n>` is set above.

`-ins <n>` Reject tests as soon as the code of one thread consists of more than `<n>` instructions. Default is 4.

`-c <bool>` Avoid equivalent cycles. Default is `true`.

`-relax <relax-list>` Set relax list. Default is empty. The syntax of `<relax-list>` is a comma (or space) separated list of candidate relaxations.

`-mix <bool>` Mix the elements of the relax list (see below), default `false`.

`-maxelax <n>` In mix mode, upper bound on the number of different candidate relaxations tested together. Default is 100.

`-safe <relax-list>` Set safe list. Default is empty.

-mode <critical|sc> Control generation of cycles, default sc. Critical mode offers predictive control on cycle generation (see below), and is activated when diy is invoked by the automated front-end dont.

-cumul <bool> Permit implicit cumulativity, *i.e.* authorise building up the sequence Rfe followed by a fence, or the reverse. Default is true.

The relax and safe lists command the generation of cycles as follows:

1. When the relax list is empty, cycles are built from the candidate relaxations of the safe list.
2. When the relax list is of size 1, cycles are built from its single element r and from the elements of the safe list. Additionally, the cycle produced contains r at least once.
3. When the relax list is of size n, with $n > 1$, the behaviour of diy depends on the mix mode:
   (a) By default (-mix false), diy generates n independent sets of cycles, each set being built with one relaxation from the relax list and all the relaxations in the safe list. In other words, diy on a relax list of size n behaves similarly to n runs of diy on each candidate relaxation in the list.
   (b) Otherwise (-mix true), diy generates cycles that contains at least one element from the relax list, including some cycles that contain different relaxations from the relax list. The cycles will contain at most m different elements from the relax list, where m is specified with option "-maxrelax m".

Generally speaking, diy generates "some" cycles and does not generate "all" cycles (up to a certain size *e.g.*). In (default) sc mode, diy performs some optimisation, most of which we leave unspecified. As an exception to this non-specification, diy is guaranteed not to generate redundant elementary communication relaxation in the following sense: let us call Com the union of Ws, Rf and Fr (the e|i specification is irrelevant here). Ws being transitive and by definition of Fr, one easily shows that the transitive closure Com+ of Com is the union of Com plus [Ws,Rf] (Ws followed by Rf) plus [Fr,Rf]. As a consequence, maximal subsequences of communication relaxations in diy cycles are limited to single relaxations (*i.e.* Ws, Rf and Fr) and to the hereabove mentioned two sequences (*i.e.* [Ws,Rf] and [Fr,Rf]). For instance, [Ws,Ws] and [Fr,Ws] should never appear in diy generated cycles. However, such subsequences can be generated on an individual basis with diyone, see the example of W5 in Sec 7.3.

In critical mode (-mode critical), cycles are strictly specified as follows:

1. Communication candidate relaxations sequences are limited to Rf,Fr,Ws,[Ws,Rf] and [Fr,Rf], as in sc mode.
2. No two internal[3] candidate relaxations follow one another.
3. If the option -cumul false is specified, diy will not construct the sequence of Rfe followed by a fence (or B-cumulativity) candidate relaxation or of a fence (or A-cumulativity) candidate relaxation followed by Rfe.
4. None of the rules above applies to the internal sequences of composite candidate relaxations. For instance, if [Rfi,PodRR] is given as a candidate relaxation, the sequence "Rfi,PodRR" appears in cycles.

The cycles described above are the *critical* cycles of [4].

8.3 Usage of readRelax

readRelax is a simple tool to extract relevant information out of litmus run logs. For a given run of a given litmus test, the relevant information is:

- Whether the test yielded Ok or not,

[3] That is, the source and target accesses are by the same processor.

- An optional candidate relaxation, which is the one given as argument to diy option `-relax` at test build time, or none.

- The safe list relevant to the given test, *i.e.* the safe candidate relaxations that appear in the tested cycle.

See Sec. 5.2 for an example.

The tool readRelax takes file names as arguments. If no argument is present, it reads a list of file names on standard input, one name per line.

Part III
Automating the testing process

The authors of dont are Jade Alglave and Luc Maranget (INRIA Paris–Rocquencourt).

9 Preamble

Following Part II, we describe our tests *via* cycles, built from the candidate relaxations they involve. We consider a candidate relaxation to be *relaxed*, or *non-global*, when it corresponds to the weaknesses that can be observed on a system implementing A. We consider a candidate relaxation to be *safe*, or *global*, when it is guaranteed, *e.g.* by the documentation, never to be relaxed.

In the following, we consider an architecture A to be a pair $(\mathrm{Relax}_A, \mathrm{Safe}_A)$, where Relax_A (resp. Safe_A) are the candidate relaxations relaxed (resp. safe) for A.

10 A tour of **dont**

10.1 Checking conformance

We want to check (prove, even) that a given machine M is conform to an architecture A. By conform, we mean that the machine M does not exhibit more behaviours than the architecture A actually allows.

For example, let us consider an x86 machine with 2 processors. Suppose that we have been told that x86 machines are TSO [5], and that we want to check that. As the default values of dont options handle that very situation, we type:

```
$ dont -mode conform
** Step 0 **
Phase 2 in A (6 tests)
...
Phase 2 in A (6 tests)
** Step 5 **
Safe set {Rfe, Fre, Wse, PodWW, PodRW, PodRR, MFencedWR} is conform
```

The automated front-end dont, assumed the TSO safe set (the default for x86), called the diy tool (see Part II) to generate all the tests that are forbidden by TSO — up to 2 processors; ran them (5 times) with our companion litmus tool, (see Part I) against our x86 machine; and observed that the machine does not exhibit any outcome forbidden by TSO. In effect, dont in conformance check mode automates the safe tests of Sec. 5.2.

10.2 Checking non-conformance

Now, we wish to prove that an x86 machine is not sequentially consistent. To that end, we write the following configuration file `x86.sc`:

```
#General behaviour
arch = X86
mode = conform
stabilise = 1
#Cycle control
safe = Rfe,Fre,Wse,Pod**,[Rfi,PodRR]
nprocs = 2
#External tool control
```

```
litmus_opts = -a 2 -i 0
run_opts = -s 100000 -r 10,-s 5000 -r 200 -i 1
build = make -j 2 -s
```

Most of dont controls are set, sometimes to their default values:

- `arch = X86` sets the targeted architecture, `mode = conform` sets conformance check mode, and `stablise = 1` commands performing the check round once (the default is five times, cf. *supra*).
- `safe = Rfe,Fre,Wse,Pod**,[Rfi,PodRR]` defines the set of safe relaxation candidates used to generate litmus tests (up to 2 processors, by `nprocs = 2`).
- The front-end dont calls litmus and runs the tests with the specified options. The setting `litmus_opt = -a 2 -i 0` specifies that two processors are available and enables affinity control (see Sec. 3 for the description of litmus options). Tests will be run twice per check round, once with options `-s 100000 -r 10`, and once with options `-s 5000 -r 200 -i 1` (see Sec. 2.3 for the description of test executable options). Finally, the setting `build = make -j 2 -s` specifies the command to use to compile the C source files that litmus produces.

We run dont configured by `x86.sc` as follows:

```
$ dont x86.sc
** Step 0 **
Phase 2 in A (9 tests)
...
** Step 1 **
Safe set {[Rfi,PodRR], Rfe, Fre, Wse, PodWW, PodWR, PodRW, PodRR} is not conform
++ Invalidating tests ++
A006: 'Fre PodWR Fre PodWR' {Fre, PodWR}
A007: 'Fre PodWW Wse PodWR' {Fre, Wse, PodWW, PodWR}
A001: 'Rfi PodRR Fre PodWR Fre' {[Rfi,PodRR], Fre, PodWR}
A002: 'Rfi PodRR Fre PodWW Wse' {[Rfi,PodRR], Fre, Wse, PodWW}
A000: 'Rfi PodRR Fre Rfi PodRR Fre' {[Rfi,PodRR], Fre}
++++++++
```

The conformance check failed and the tests that invalidate the hypothesis "x86 is sequentially consistent" are listed. The check took place in directory `A`. Directory `A` contains the actual logs of litmus runs as files `A.00`, `A.01` etc., in addition to the sources of the litmus tests:

```
$cat A/A006.litmus
X86 A006
"Fre PodWR Fre PodWR"
Cycle=Fre PodWR Fre PodWR
Relax=
Safe=Fre PodWR
{ }
 P0          | P1          ;
 MOV [x],$1  | MOV [y],$1  ;
 MOV EAX,[y] | MOV EAX,[x] ;
exists (0:EAX=0 /\ 1:EAX=0)
```

Notice that, since tests are described by their cycles, the source of tests can also be reconstructed with diyone:

```
% diyone -arch X86 Fre PodWR Fre PodWR
X86 a
```

```
"Fre PodWR Fre PodWR"
{ }
 P0          | P1          ;
 MOV [y],$1  | MOV [x],$1  ;
 MOV EAX,[x] | MOV EAX,[y] ;
exists (0:EAX=0 /\ 1:EAX=0)
```

10.3 Automatically exploring the memory model exhibited by a machine

Now suppose that we have no idea of the memory model of our 2 processsors x86 machine. Another mode of our dont tool automatically explores a given machine, and outputs an architecture (*i.e.* a pair $(\mathrm{Relax}_A, \mathrm{Safe}_A)$) to which the machine conforms. The following configuration file `x86.explo` instructs dont to perform such an exploration.

```
#General behaviour
arch = X86
mode = explo
#Cycle control
testing = Rfe,Pod**,MFenced**,[Rfi,PodR*]
safe = Fre,Wse
nprocs = 2
#External tool control
litmus_opts = -a 2 -i 0
run_opts = -s 100000 -r 10,-s 5000 -r 200 -i 1
build = make -j 2 -s
```

With respect to conformance check, new or changed settings are the selection of exploration mode by `mode = explo`, the definition of the initial safe set by `safe = Fre,Wse`, and and the definition of the candidate relaxations to be tested (`testing = Rfe,Pod**,MFenced**,[Rfi,PodR*]`).

We lauch the exploration as:

```
$ dont x86.explo
```

The whole process only takes a few minutes, mostly due to the limited number of tests induced by the setting `nprocs = 2`.

We now detail dont output (the html version[4] of this document includes the complete log of the experience). We start by a first exploration round:

```
** Step 0 **
Testing: {[Rfi,PodRW], [Rfi,PodRR], Rfe, PodWW, PodWR, PodRW, PodRR, MFencedWW,
MFencedWR, MFencedRW, MFencedRR}
Relaxed: {}
Safe   : {Fre, Wse}
Phase 1 in A (6 tests)
Actually tested: {[Rfi,PodRW], [Rfi,PodRR], PodWW, PodWR, MFencedWW, MFencedWR}
Added relax: {[Rfi,PodRR], PodWR}
Added safe: {[Rfi,PodRW], PodWW, MFencedWW, MFencedWR}
Phase 2 in B (6 tests)
```

The log above first indicates the current status of exploration as three sets: *testing*, *relaxed* and *safe*. Initially, no candidate relaxation has yet been observed to be relaxed, while the testing and safe sets are as assumed. Each exploration round is divided in two phases. The aim of Phase 1 (performed in directory `A`) is to classify

[4] `http://diy.inria.fr/doc/auto.html`

some candidate relaxations as either relaxed or safe. It here succeeds for 6 candidate relaxations, whose observed status is indicated. Phase 2 (performed in directory B) basically is a conformance check of the current safe set. The conformance check succeeds and all safe candidate relaxations found at phase 1 make it to the next round:

```
** Step 1 **
Testing: {Rfe, PodRW, PodRR, MFencedRW, MFencedRR}
Relaxed: {[Rfi,PodRR], PodWR}
Safe   : {[Rfi,PodRW], Fre, Wse, PodWW, MFencedWW, MFencedWR}
Phase 1 in C (10 tests)
Actually tested: {Rfe, PodRW, PodRR, MFencedRW, MFencedRR}
Added safe: {Rfe, PodRW, PodRR, MFencedRW, MFencedRR}
Phase 2 in D (17 tests)
```

Phase 1 (performed in directory C) can now target new candidate relaxations, because of the increased safe set. All of targeted candidate relaxations are observed to be safe, which is confirmed by phase 2. As a consequence, there does not remain any candidate relaxation to be tested and the next round reduces to a conformance check:

```
** Step 2 **
zzTesting: {}
Relaxed: {[Rfi,PodRR], PodWR}
Safe   : {[Rfi,PodRW], Rfe, Fre, Wse, PodWW, PodRW, PodRR, MFencedWW, MFencedWR, MFencedRW, MFencedR
Phase 1 in E (0 tests)
Phase 2 in D (17 tests)
```

The same check is performed for 4 additional rounds as governed by the default value of 5 for the setting of `stabilise`. Round number 6 then shows the result of exploration, (*i.e.* the pair $(\mathrm{Relax}_A, \mathrm{Safe}_A)$), prefixed by the list of tests that justify observed relaxations:

```
** Step 6 **
...
++ Witness(es) for relaxed [Rfi,PodRR] ++
A001: 'Rfi PodRR Fre Rfi PodRR Fre' {[Rfi,PodRR], Fre}
++++++++
++ Witness(es) for relaxed PodWR ++
A003: 'Fre PodWR Fre PodWR' {Fre, PodWR}
++++++++
Observed relaxed: {Rfi, PodWR}
Observed safe: {Rfe, Fre, Wse, PodWW, PodRW, PodRR, MFencedWW, MFencedWR, MFencedRW, MFencedRR}
```

And we go again for 5 additional rounds of pure conformance check:

```
** Now checking safe set conformance **
** Step 7 **
Phase 2 in F (17 tests)
...
* Step 12 **
Observed relaxed: {Rfi, PodWR}
Observed safe: {Rfe, Fre, Wse, PodWW, PodRW, PodRR, MFencedWW, MFencedWR, MFencedRW, MFencedRR}
```

Once exploration is complete, all litmus tests and logs of `litmus` runs are still present in their directories `A`, `B`, etc. For instance, the directory `F` contain the 10 logs of the final conformance check, as the files `F.01`, …, `F.09`:

```
$ ls F/F.??
F/F.00  F/F.01  F/F.02  F/F.03  F/F.04  F/F.05  F/F.06  F/F.07  F/F.08  F/F.09
```

The tool dont offers a convenient replay feature:

```
$ dont -restart
** Step 0 *
...
* Step 12 **
Observed relaxed: {Rfi, PodWR}
Observed safe: {Rfe, Fre, Wse, PodWW, PodRW, PodRR, MFencedWW, MFencedWR, MFencedRW, MFencedRR}
```

The command above takes a few seconds of time, since experiments are not run again. Instead, the logs of litmus runs are read and their interpretatiosn is re-performed. Notice that the restart feature also permits to pursue interrupted experiments.

11 Usage of dont

In effect, the tool dont automates the complete testing procedure described in the documentation of diy proper (Sec. 5). It is to be noticed that dont requires a fully functional installation of the diy tool suite. In particular, the commands diy and litmus must be installed and runnable as "diy" and "litmus" (*.i.e* installed in path).

11.1 Command-line options

The automated front-end dont is configured mostly by the means of a configuration file, which dont takes as a command-line argument. Nevertheless, dont accepts the following, limited, set of options:

`-v` Be verbose, repeat to increase verbosity.

`-version` Show version number and exit.

`-arch <X86|PPC>` Set architecture. Default is `X86`.

`-mode <conform|explo>` Set main mode, either conformance check or exploration. Default is `explo`.

`-nprocs <n>` Generate tests up to ¡n¿ processors (defaults: X86=2, PPC=4)

`-restart` Restart the experiment in hand in current directory.

Except for `-restart` command lines options are not intended for normal use. In particular, command-line options do not override values defined in configuration files.

Namely, there are many parameters to set and appropriate values for them will depend on the tested machine. In particular, litmus parameters need to be chosen carefully, by the means of preliminary experiments. For instructions on configuring litmus, refer to Sec. 2 of litmus documentation.

11.2 Configuration files

The general syntax of configurations files is a sequence of lines *key* `=` *value*. Comment lines are introduced by `#`. The tool dont recognises the following keys:

General behaviour

mode = <conform|explo> Main operating mode. Default is explo

arch = <X86|PPC> Target architecture. Default is X86.

run = <local|ssh [*user*@]*hostname*> Give access to the tested machine, which can be either the machine where dont runs, or a remote machine which will be accessed by scp and ssh. Default is local.

work_dir = *dir* Directory for temporary files, default is /var/tmp.

stabilise = <*n*> In conformance check mode, dont performs n rounds of conformance testing. In exploration mode, dont ends the exploration after n rounds without state change. Default is 5.

interactive = <true|false> In exploration mode and after n rounds without state change, dont will either assume that the whole current testing set is safe (false), or ask the user (true) to decide for some of the elements of this set to be safe. Default is true, *i.e.* ask user.

Generation of cycles

nprocs = <*n*> Generate cycles up to n processors. Default is 2 for x86 and 4 for Power.

diy_sz = <*m*> Upper limit on the size of cycles of candidate relaxations. Default is $2 \times n$, where n is the number of processors. With decent values of the initial candidate relaxations sets (see below), this default commands the generation of all (critical, see Sec. 8.2) cycles that involve up to n processors.

safe = <*relax-list*> Define the safe set S. In exploration mode, S is the initial value of the safe set (default Fre, Wse). In conformance mode, S is the safe set checked. (default is Rfe, Fre, Wse, PodR*, PodWW, MFencedWR for x86, and Fre, Wse, DpdW, DpdR, CtrldW, Ctrld, Syncd**, ACSyncdR*, BCSyncd*W, ABCSyndRW, LwSyncdWW, LwSyncdR*, ACLwSyncdRW, BCLwSyncdRW, ABCCLwSyncdRW for Power).

testing = <*relax-list*> Define the tested set of candidate relaxations. The tested set is relevant only in exploration mode. Default values are Rfe,Pod**,MFenced**,[Rfi,MFencedR*],[Rfi,PodR*] for x86 and Pod**, Dpd*, Ctrld*, LwSyncd**, Syncd**, ACLwSyncdR*, ACSyncdR*, ABCLwSyncdRW, ABCSyncdRW, BCLwSyncd*W, BCSyncd*W, Rfe,[Rfi,Dpd*],[Rfi,Ctrld*] for Power.

The syntax for *relax-list* above is a comma (or space) separated list of candidate relaxations. Candidate relaxations are introduced by the documentation of diy (see Part II)

Control of external tools

litmus_opts = <*opts*> Define options used by dont when it calls litmus. Default is the empty string, *i.e.* use litmus defaults.

run_opts = <$opts_1$,...,$opts_n$> Define options used for running litmus tests. Any set of litmus tests generated and compiled by dont, will be run n times, with specified options. More concretely, dont will run the litmus tests with commands sh run.sh $opts_1$, ..., sh run.sh $opts_n$. The default is the empty string, *i.e.* run tests once with no option.

build = <*command*> Defines the command issued by dont to compile the C source files produced by litmus. The default is sh comp.sh, *i.e.* runs the compilation script produced by litmus. An intersting alternative is gmake -s -j *n* for concurrent compilation, with up to n concurrent tasks.

References

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

[2] Intel 64 Architecture Memory Ordering White Paper, August 2007.

[3] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1979.

[4] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. In *TPLS 1988*.

[5] Sparc Architecture Manual Versions 8 and 9, 1992 and 1994.