A don't (diy) tutorial

Version 5.99-D

April 16, 2014

diy is a tool suite for testing shared memory models. We provide several tools, litmus (Part I) for running tests, diy generators (Part II) for producing tests from concise specifications, and herd (Part III) for simulating memory models. In Part IV we describe a few concrete experiments, illustrating frequent usage patterns of diy generators and of litmus. Finally (Part V), we briefly describe our experimental dont tool 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`. This software is released under the terms of the Lesser GNU Public License.

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, Power or ARM) 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 and Francesco Zappa Nardelli.

1 A tour of litmus

1.1 A simple run

Consider the following (rather classical, store buffering) `SB.litmus` litmus test for X86:

```
X86 SB
"Fre PodWR Fre PodWR"
{ x=0; y=0; }
 P0          | P1          ;
 MOV [x],$1  | MOV [y],$1  ;
 MOV EAX,[y] | MOV EAX,[x] ;
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 SB.litmus
%%%%%%%%%%%%%%%%%%%%%%%%%%
% Results for SB.litmus %
%%%%%%%%%%%%%%%%%%%%%%%%%%
X86 SB
"Fre PodWR Fre PodWR"

{x=0; y=0;}

 P0          | P1          ;
 MOV [x],$1  | MOV [y],$1  ;
 MOV EAX,[y] | MOV EAX,[x] ;

exists (0:EAX=0 /\ 1:EAX=0)
Generated assembler
```

```
#START _litmus_P1
movl $1,(%r10)
movl (%r9),%eax
#START _litmus_P0
movl $1,(%r9)
movl (%r10),%eax

Test SB Allowed
Histogram (4 states)
40    *>0:EAX=0; 1:EAX=0;
499923:>0:EAX=1; 1:EAX=0;
500009:>0:EAX=0; 1:EAX=1;
28    :>0:EAX=1; 1:EAX=1;
Ok

Witnesses
Positive: 40, Negative: 999960
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
Observation SB Sometimes 40 999960
Time SB 0.44
...
```

The litmus test is first reminded, followed by actual assembler — the machine is a 64 bits one, 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 40 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 `SB-PPC.litmus`, a Power version of the previous test:

```
PPC SB-PPC
"Fre PodWR Fre PodWR"
{
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)
```

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

```
% litmus -o /tmp/a.tar -os mac SB-PPC.litmus
% scp /tmp/a.tar ppc:/tmp
```

Then, on the remote machine ppc:

```
ppc% mkdir SB && cd SB
ppc% tar xf /tmp/a.tar
ppc% ls
comp.sh  Makefile  outs.c  outs.h  README.txt  run.sh  SB-PPC.c  show.awk  utils.c  utils.h
```

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

```
ppc% sh comp.sh
ppc% sh run.sh
...
Test SB-PPC Allowed
Histogram (3 states)
1784  *>0:r3=0; 1:r3=0;
498564:>0:r3=1; 1:r3=0;
499652:>0:r3=0; 1:r3=1;
Ok

Witnesses
Positive: 1784, Negative: 998216
Condition exists (0:r3=0 /\ 1:r3=0) is validated
Hash=4edecf6abc507611612efaecc1c4a9bc
Observation SB-PPC Sometimes 1784 998216
Time SB-PPC 0.55
...
```

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

1.3 Running several tests at once

Consider the additional test `STFW-PPC.litmus`:

```
PPC STFW-PPC
"Rfi PodRR Fre Rfi PodRR Fre"
{
0:r2=x; 0:r5=y;
1:r2=y; 1:r5=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) ;
 lwz r4,0(r5) | lwz r4,0(r5) ;
exists
(0:r3=1 /\ 0:r4=0 /\ 1:r3=1 /\ 1:r4=0)
```

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

```
$ litmus -o /tmp/a.tar -os mac SB-PPC.litmus STFW-PPC.litmus
```

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

```
$ cat @ppc
SB-PPC.litmus
```

```
STFW-PPC.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 SB-PPC Allowed
Histogram (3 states)
1765  *>0:r3=0; 1:r3=0;
498741:>0:r3=1; 1:r3=0;
499494:>0:r3=0; 1:r3=1;
Ok

Witnesses
Positive: 1765, Negative: 998235
Condition exists (0:r3=0 /\ 1:r3=0) is validated
Hash=4edecf6abc507611612efaecc1c4a9bc
Observation SB-PPC Sometimes 1765 998235
Time SB-PPC 0.57
...
Test STFW-PPC Allowed
Histogram (4 states)
480   *>0:r3=1; 0:r4=0; 1:r3=1; 1:r4=0;
499560:>0:r3=1; 0:r4=1; 1:r3=1; 1:r4=0;
499827:>0:r3=1; 0:r4=0; 1:r3=1; 1:r4=1;
133   :>0:r3=1; 0:r4=1; 1:r3=1; 1:r4=1;
Ok

Witnesses
Positive: 480, Negative: 999520
Condition exists (0:r3=1 /\ 0:r4=0 /\ 1:r3=1 /\ 1:r4=0) is validated
Hash=92b2c3f63323093250006 56d0632131e
Observation STFW-PPC Sometimes 480 999520
Time STFW-PPC 0.56
...
```

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 **SB-PPC**. 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 logical processors 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. Additionally, the increment of the main loop (of size s) can be set to a value or *stride* different from the default of one. Running a test several times with changing the stride value also proved quite effective in favouring outcome variability.

    If the *random 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 and teh random preload mode is enabled by default. Starting from version 5.0, we provide a more precise control over preloading memory locations — See Sec. 3.2.

    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.

    Starting from version 5.0, we provide a slightly altered user synchronisation mode: *userfence*, which alters user mode by executing memory fences to speedup write propagation. The new mode features overall better synchronisation, yielding dramatic improvements on some examples. However, outcome variability may suffer from this more accurate synchronisation, hence user mode remains the default.

    More importantly, we provide an additional exact, *timebase* synchronisation technique: test threads will first synchronise using polling synchronisation barrier code, agree on a target timebase[2] value and then loop reading the timebase until it exceeds the target value. This technique yields very good synchronisation and allows fine synchronisation tuning by assigning different starting delays to different threads — see Sec. 3.1. As ARM does not provide timebase counters, notice that "timebase" synchronisation for ARM silently degrades to synchronisation by the means of the polling synchronisation barrier.
  - 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 10k -s 1` and `./a.exe -n 1 -r 1 -s 1M` 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 1M` 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

[2]Power and x86-based systems provide a user accessible timebase counter that should provide consistent times to all cores and processors.

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

The test consists of four threads. There are two writers (P0 and P2) 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 — P1 reads `x` first, while P3 reads `y` first. The load instructions `lwz` in reader threads are separated by a lightweight barrier instruction `lwsync`. The final condition `exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0)` characterises the situation where the reader threads see the writes by P0 and P2 in opposite order. The corresponding outcome `1:r1=1; 1:r3=0; 3:r1=1; 3: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
Test ppc-iriw-lwsync Allowed
Histogram (15 states)
163674:>1:r1=0; 1:r3=0; 3:r1=0; 3:r3=0;
34045 :>1:r1=1; 1:r3=0; 3:r1=0; 3:r3=0;
40283 :>1:r1=0; 1:r3=1; 3:r1=0; 3:r3=0;
95079 :>1:r1=1; 1:r3=1; 3:r1=0; 3:r3=0;
33848 :>1:r1=0; 1:r3=0; 3:r1=1; 3:r3=0;
72201 :>1:r1=0; 1:r3=1; 3:r1=1; 3:r3=0;
32452 :>1:r1=1; 1:r3=1; 3:r1=1; 3:r3=0;
43031 :>1:r1=0; 1:r3=0; 3:r1=0; 3:r3=1;
73052 :>1:r1=1; 1:r3=0; 3:r1=0; 3:r3=1;
1     :>1:r1=0; 1:r3=1; 3:r1=0; 3:r3=1;
42482 :>1:r1=1; 1:r3=1; 3:r1=0; 3:r3=1;
90470 :>1:r1=0; 1:r3=0; 3:r1=1; 3:r3=1;
30306 :>1:r1=1; 1:r3=0; 3:r1=1; 3:r3=1;
43239 :>1:r1=0; 1:r3=1; 3:r1=1; 3:r3=1;
205837:>1:r1=1; 1:r3=1; 3:r1=1; 3:r3=1;
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0) is NOT validated
Hash=4fbfaafa51f6784d699e9bdaf5ba047d
Observation ppc-iriw-lwsync Never 0 1000000
Time ppc-iriw-lwsync 1.32
```

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[3]. 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
```

[3]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. However parameters a will be set to A when affinity control is enabled and when a value is 0.

```
Histogram (16 states)
180600:>1:r1=0; 1:r3=0; 3:r1=0; 3:r3=0;
3656  :>1:r1=1; 1:r3=0; 3:r1=0; 3:r3=0;
18812 :>1:r1=0; 1:r3=1; 3:r1=0; 3:r3=0;
77692 :>1:r1=1; 1:r3=1; 3:r1=0; 3:r3=0;
2973  :>1:r1=0; 1:r3=0; 3:r1=1; 3:r3=0;
9     *>1:r1=1; 1:r3=0; 3:r1=1; 3:r3=0;
28881 :>1:r1=0; 1:r3=1; 3:r1=1; 3:r3=0;
75126 :>1:r1=1; 1:r3=1; 3:r1=1; 3:r3=0;
20939 :>1:r1=0; 1:r3=0; 3:r1=0; 3:r3=1;
30498 :>1:r1=1; 1:r3=0; 3:r1=0; 3:r3=1;
1234  :>1:r1=0; 1:r3=1; 3:r1=0; 3:r3=1;
89993 :>1:r1=1; 1:r3=1; 3:r1=0; 3:r3=1;
75769 :>1:r1=0; 1:r3=0; 3:r1=1; 3:r3=1;
76361 :>1:r1=1; 1:r3=0; 3:r1=1; 3:r3=1;
87864 :>1:r1=0; 1:r3=1; 3:r1=1; 3:r3=1;
229593:>1:r1=1; 1:r3=1; 3:r1=1; 3:r3=1;
Ok

Witnesses
Positive: 9, Negative: 999991
Condition exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0) is validated
Hash=4fbfaafa51f6784d699e9bdaf5ba047d
Observation ppc-iriw-lwsync Sometimes 9 999991
Time ppc-iriw-lwsync 0.68
```

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)
160629:>1:r1=0; 1:r3=0; 3:r1=0; 3:r3=0;
33389 :>1:r1=1; 1:r3=0; 3:r1=0; 3:r3=0;
43725 :>1:r1=0; 1:r3=1; 3:r1=0; 3:r3=0;
93114 :>1:r1=1; 1:r3=1; 3:r1=0; 3:r3=0;
33556 :>1:r1=0; 1:r3=0; 3:r1=1; 3:r3=0;
64875 :>1:r1=0; 1:r3=1; 3:r1=1; 3:r3=0;
34908 :>1:r1=1; 1:r3=1; 3:r1=1; 3:r3=0;
43770 :>1:r1=0; 1:r3=0; 3:r1=0; 3:r3=1;
64544 :>1:r1=1; 1:r3=0; 3:r1=0; 3:r3=1;
4     :>1:r1=0; 1:r3=1; 3:r1=0; 3:r3=1;
54633 :>1:r1=1; 1:r3=1; 3:r1=0; 3:r3=1;
92617 :>1:r1=0; 1:r3=0; 3:r1=1; 3:r3=1;
34754 :>1:r1=1; 1:r3=0; 3:r1=1; 3:r3=1;
54027 :>1:r1=0; 1:r3=1; 3:r1=1; 3:r3=1;
191455:>1:r1=1; 1:r3=1; 3:r1=1; 3:r3=1;
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0) is NOT validated
Hash=4fbfaafa51f6784d699e9bdaf5ba047d
```

```
Observation ppc-iriw-lwsync Never 0 1000000
Time ppc-iriw-lwsync 0.92
```

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

One may also bind test thread to logical processors randomly with executable option `+ra`.

```
vargas% ./ppc-iriw-lwsync.exe +ra
Test ppc-iriw-lwsync Allowed
Histogram (15 states)
...
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0) is NOT validated
Hash=4fbfaafa51f6784d699e9bdaf5ba047d
Observation ppc-iriw-lwsync Never 0 1000000
Time ppc-iriw-lwsync 1.85
```

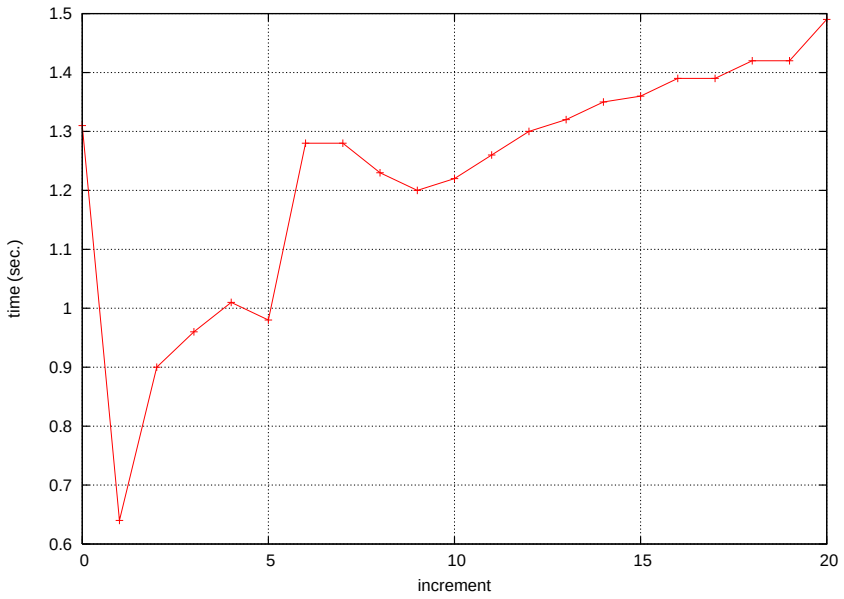
As we see, the condition does not validate either with random affinity. As a matter of fact, logical processors are taken at random in the sequence 0, 1, ..., 63; while the successful run with `-i 1` took them in the sequence 0, 1, 2, 3. One can limit the sequence of logical processor with option `-p`, which takes a sequence of logical processors numbers as argument:

```
vargas% ./ppc-iriw-lwsync.exe +ra -p 0,1,2,3
Test ppc-iriw-lwsync Allowed
Histogram (16 states)
...
8     *>1:r1=1; 1:r3=0; 3:r1=1; 3:r3=0;
...
Ok

Witnesses
Positive: 8, Negative: 999992
Condition exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0) is validated
Hash=4fbfaafa51f6784d699e9bdaf5ba047d
Observation ppc-iriw-lwsync Sometimes 8 999992
Time ppc-iriw-lwsync 0.70
```

The condition now validates.

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)
```

```
169420:>1:r1=0; 1:r3=0; 3:r1=0; 3:r3=0;
1287  :>1:r1=1; 1:r3=0; 3:r1=0; 3:r3=0;
17344 :>1:r1=0; 1:r3=1; 3:r1=0; 3:r3=0;
85329 :>1:r1=1; 1:r3=1; 3:r1=0; 3:r3=0;
1548  :>1:r1=0; 1:r3=0; 3:r1=1; 3:r3=0;
3     *>1:r1=1; 1:r3=0; 3:r1=1; 3:r3=0;
27014 :>1:r1=0; 1:r3=1; 3:r1=1; 3:r3=0;
75160 :>1:r1=1; 1:r3=1; 3:r1=1; 3:r3=0;
19828 :>1:r1=0; 1:r3=0; 3:r1=0; 3:r3=1;
29521 :>1:r1=1; 1:r3=0; 3:r1=0; 3:r3=1;
441   :>1:r1=0; 1:r3=1; 3:r1=0; 3:r3=1;
93878 :>1:r1=1; 1:r3=1; 3:r1=0; 3:r3=1;
81081 :>1:r1=0; 1:r3=0; 3:r1=1; 3:r3=1;
76701 :>1:r1=1; 1:r3=0; 3:r1=1; 3:r3=1;
93623 :>1:r1=0; 1:r3=1; 3:r1=1; 3:r3=1;
227822:>1:r1=1; 1:r3=1; 3:r1=1; 3:r3=1;
Ok

Witnesses
Positive: 3, Negative: 999997
Condition exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0) is validated
Hash=4fbfaafa51f6784d699e9bdaf5ba047d
Observation ppc-iriw-lwsync Sometimes 3 999997
Time ppc-iriw-lwsync 0.63
```

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)
...
57    *>1:r1=1; 1:r3=0; 3:r1=1; 3:r3=0;
...
Ok

Witnesses
Positive: 57, Negative: 3999943
Condition exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0) is validated
Hash=4fbfaafa51f6784d699e9bdaf5ba047d
Observation ppc-iriw-lwsync Sometimes 57 3999943
Time ppc-iriw-lwsync 0.75
```

Observe 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 experiment 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. Similarly, 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 hyper-threading, Linux, AMD64 machine, we have checked that logical processors residing on the same core are k and $k + 4$, where k is an arbitrary core number ranging from 0 to 3. As a result, a proper choice for favouring effective hyper-threading 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 hyper-threading...

2.2.4 Custom control

Most tests run by litmus are produced by the litmus test generators described in Part II. Those tests include meta-information that may direct affinity control. For instance we generate one test with the diyone tool, see Sec. 5.2. More specifically we generate **IRIW+lwsyncs** for Power (**ppc-iriw-lwsync** in the previous section) as follows:

```
% diyone -arch PPC -name IRIW+lwsyncs Rfe LwSyncdRR Fre Rfe LwSyncdRR Fre
```

We get the new source file `IRIW+lwsyncs.litmus`:

```
PPC IRIW+lwsyncs
"Rfe LwSyncdRR Fre Rfe LwSyncdRR Fre"
Prefetch=0:x=T,1:x=F,1:y=T,2:y=T,3:y=F,3:x=T
Com=Rf Fr Rf Fr
Orig=Rfe LwSyncdRR Fre Rfe LwSyncdRR Fre
{
0:r2=x;
1:r2=x; 1:r4=y;
2:r2=y;
3:r2=y; 3:r4=x;
}
 P0           | P1           | P2           | P3           ;
 li r1,1      | lwz r1,0(r2) | li r1,1      | lwz r1,0(r2) ;
 stw r1,0(r2) | lwsync       | stw r1,0(r2) | lwsync       ;
              | lwz r3,0(r4) |              | lwz r3,0(r4) ;
exists
(1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0)
```

The relevant meta-information is the "`Com`" line that describes how test threads are related — for instance, thread 0 stores a value to memory that is read by thread 1, written "`Rf`" (see Part II for more details). Custom affinity control will tend to run threads related by "`Rf`" on "close" logical processors, where we can for instance consider that close logical processors belong to the same physical core (SMT for Power). This minimal logical processor topology is described by two litmus command-line option: `-smt <n>` that specifies n-way SMT; and `-smt_mode (seq|end)` that specifies how logical processors from the same core are numbered. For a 8-cores 4-ways SMT power7 machine we invoke litmus as follows:

```
% litmus -mem direct -smt 4 -smt_mode seq -affinity custom -o a.tar IRIW+lwsyncs.litmus
```

Notice that memory mode is direct and that the number of available logical processors is unspecified, resulting in running one instance of the test. More importantly, notice that affinity control is enabled `-affinity custom`, additionally specifying custom affinity mode.

We then upload the archive `a.tar` to our Power7 machine, unpack, compile and run the test:

```
power7% tar xmf a.tar
power7% make
...
power7% ./IRIW+lwsyncs.exe -v
./IRIW+lwsyncs.exe  -v
IRIW+lwsyncs: n=1, r=1000, s=1000, +rm, +ca, p='0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21
thread allocation:
[23,22,3,2] {5,5,0,0}
```

Option `-v` instructs the executable to show settings of the test harness: we see that one instance of the test is run (`n=1`), size parameters are reminded (`r=1000, s=1000`) and shuffling of indirect memory mode is performed (`+rm`). Affinity settings are also given: mode is custom (`+ca`) and the logical processor sequence inferred is given (`-p 0,1,...,31`). Additionally, the allocation of test threads to logical processors is given, as `[...]`, as well as the allocation of test threads to physical cores, as `{...}`.

Here is the run output proper:

```
Test IRIW+lwsyncs Allowed
Histogram (15 states)
2700  :>1:r1=0; 1:r3=0; 3:r1=0; 3:r3=0;
142   :>1:r1=1; 1:r3=0; 3:r1=0; 3:r3=0;
37110 :>1:r1=0; 1:r3=1; 3:r1=0; 3:r3=0;
181257:>1:r1=1; 1:r3=1; 3:r1=0; 3:r3=0;
78    :>1:r1=0; 1:r3=0; 3:r1=1; 3:r3=0;
15    *>1:r1=1; 1:r3=0; 3:r1=1; 3:r3=0;
103459:>1:r1=0; 1:r3=1; 3:r1=1; 3:r3=0;
149486:>1:r1=1; 1:r3=1; 3:r1=1; 3:r3=0;
30820 :>1:r1=0; 1:r3=0; 3:r1=0; 3:r3=1;
9837  :>1:r1=1; 1:r3=0; 3:r1=0; 3:r3=1;
2399  :>1:r1=1; 1:r3=1; 3:r1=0; 3:r3=1;
204629:>1:r1=0; 1:r3=0; 3:r1=1; 3:r3=1;
214700:>1:r1=1; 1:r3=0; 3:r1=1; 3:r3=1;
5186  :>1:r1=0; 1:r3=1; 3:r1=1; 3:r3=1;
58182 :>1:r1=1; 1:r3=1; 3:r1=1; 3:r3=1;
Ok

Witnesses
Positive: 15, Negative: 999985
Condition exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0) is validated
Hash=836eb3085132d3cb06973469a08098df
Com=Rf Fr Rf Fr
Orig=Rfe LwSyncdRR Fre Rfe LwSyncdRR Fre
Affinity=[2, 3] [0, 1] ; (1,2) (3,0)
Observation IRIW+lwsyncs Sometimes 15 999985
Time IRIW+lwsyncs 0.70
```

As we see, the test validates. Namely we observe the non-SC behaviour of **IRIW** in spite of the presence of two `lwsync` barriers. We may also notice, in the executable output some meta-information related to

affinity: it reads that threads 2 and 3 on the one hand and threads 0 and 1 on the other are considered "close" (*i.e.* will run on the same physical core); while threads 1 and 2 on the one hand and threads 3 and 0 on the other are considered "far" (*i.e.* will run on different cores).

Custom affinity can be disabled by enabling another affinity mode. For instance with `-i 0` we specify an affinity increment of zero. That is, affinity control is disabled altogether:

```
power7% ./IRIW+lwsyncs.exe -i 0 -v
./IRIW+lwsyncs.exe -i 0  -v
IRIW+lwsyncs: n=1, r=1000, s=1000, +rm, i=0, p='0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21
Test IRIW+lwsyncs Allowed
Histogram (15 states)
...
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r3=0) is NOT validated
Hash=836eb3085132d3cb06973469a08098df
Com=Rf Fr Rf Fr
Orig=Rfe LwSyncdRR Fre Rfe LwSyncdRR Fre
Observation IRIW+lwsyncs Never 0 1000000
Time IRIW+lwsyncs 0.90
```

As we see, the test does not validate under those conditions.

Notice that section 17 describes a complete experiment on affinity control.

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 logical processors — parameter a in Sec. 2.1. Notice that if affinity control is enabled (see below), `-a 0` will set parameter a to the number of logical processors effectively available.

`-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.

Notice that options `-s` and `-r` accept a generalised syntax for their integer argument: when suffixed by `k` (resp. `M`) the integer gets multiplied by 10^3 (resp. 10^6).

The following options are accepted only for tests compiled in indirect memory mode (see Sec. 2.1):

`-rm` Do not shuffle pointer arrays, resulting a behaviour similar do direct mode, without recompilation.

`+rm` Shuffle pointer arrays, provided for regularity.

The following option is accepted only for tests compiled with a specified stride value (see Sec. 2.1).

`-st <n>` Change stride to `<n>`. The default stride is specified at compile time by litmus option `-stride`.

The following option is accepted when enabled at compile time:

`-l <n>` Insert the assembly code of each thread in a loop of size `<n>`.

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

`-p <ns>` Logical processors sequence. The sequence `<ns>` is a comma separated list of integers, The default sequence is inferred 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 and that `.exe` files reject the `-i` option when affinity control has not been enabled at compile time.

`+ra` Perform random allocation of affinity at each test round.

`+ca` Perform custom affinity.

Notice that when custom affinity is not available, would it be that the test source lacked meta-information or that logical processor topology was not specified at compile-time, then `+ca` behaves as `+ra`.

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 Advanced control of test parameters

3.1 Timebase synchronisation mode

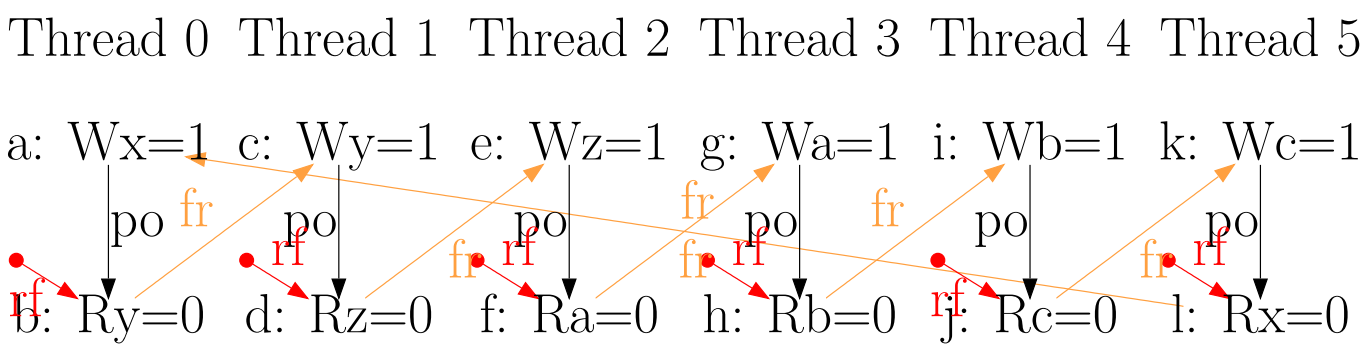
Timebase synchronisation of the testing loop iterations (see Sec. 2.1) is selected by litmus command line option `-barrier timebase`. In that mode, test threads will first synchronise using polling synchronisation barrier code, agree on a target timebase value and then loop reading the timebase until it exceeds the target value. Some tests demonstrate that timebase synchronisation is more precise than user synchronisation (`-barrier user` and default).

For instance, consider the x86 test **6.SB**, a 6-thread analog of the **SB** test:

```
X86 6.SB
"Fre PodWR Fre PodWR Fre PodWR Fre PodWR Fre PodWR Fre PodWR"
{
}
 P0          | P1          | P2          | P3          | P4          | P5          ;
 MOV [x],$1  | MOV [y],$1  | MOV [z],$1  | MOV [a],$1  | MOV [b],$1  | MOV [c],$1  ;
```

```
 MOV EAX,[y] | MOV EAX,[z] | MOV EAX,[a] | MOV EAX,[b] | MOV EAX,[c] | MOV EAX,[x] ;
exists
(0:EAX=0 /\ 1:EAX=0 /\ 2:EAX=0 /\ 3:EAX=0 /\ 4:EAX=0 /\ 5:EAX=0)
```

As for **SB**, the final condition of **6.SB** identifies executions where each thread loads the initial value 0 of a location that is writtent into by another thread.



We first compile the test in user synchronisation mode, saving litmus output files into the directory `R`:

```
% mkdir -p R
% litmus -barrier user -vb true -o R 6.SB.litmus
% cd R
% make
```

The additional command line option `-vb true` activates the printing of some timing information on synchronisations.

We then directly run the test executable `6.SB.exe`:

```
% ./6.SB.exe
Test 6.SB Allowed
Histogram (62 states)
7569  :>0:EAX=1; 1:EAX=0; 2:EAX=0; 3:EAX=0; 4:EAX=0; 5:EAX=0;
8672  :>0:EAX=0; 1:EAX=1; 2:EAX=0; 3:EAX=0; 4:EAX=0; 5:EAX=0;
...
326   :>0:EAX=1; 1:EAX=0; 2:EAX=1; 3:EAX=1; 4:EAX=1; 5:EAX=1;
907   :>0:EAX=0; 1:EAX=1; 2:EAX=1; 3:EAX=1; 4:EAX=1; 5:EAX=1;
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (0:EAX=0 /\ 1:EAX=0 /\ 2:EAX=0 /\ 3:EAX=0 /\ 4:EAX=0 /\ 5:EAX=0) is NOT validated
Hash=107f1303932972b3abace3ee4027408e
Observation 6.SB Never 0 1000000
Time 6.SB 0.85
```

The targeted outcome — reading zero in the `EAX` registers of the 6 threads — is not observed. We can observe synchronisation times for all tests runs with the executable command line option `+vb`:

```
% ./6.SB.exe  +vb
99999: 162768 420978 564546   -894 669468
99998:    -93      3     81   -174   -651
```

```
99997:   -975    -30    -33     93   -192
99996:    990   1098    852   1176    774
...
```

We see five columns of numbers that list, for each test run, the starting delays of P_1, P_2 etc. with respect to P_0, expressed in timebase ticks. Obviously, synchronisation is rather loose, there are always two threads whose starting delays differ of about 1000 ticks.

We now compile the same test in timebase synchronisation mode, saving litmus output files into the pre-existing directory `RT`:

```
% mkdir -p RT
% litmus -barrier timebase -vb true -o RT 6.SB.litmus
% cd RT
% make
```

And we run the test directly (option `-vb` disable the printing of any synchronisation timing information):

```
% ./6.SB.exe -vb
Test 6.SB Allowed
Histogram (64 states)
60922 *>0:EAX=0; 1:EAX=0; 2:EAX=0; 3:EAX=0; 4:EAX=0; 5:EAX=0;
38299 :>0:EAX=1; 1:EAX=0; 2:EAX=0; 3:EAX=0; 4:EAX=0; 5:EAX=0;
...
598   :>0:EAX=0; 1:EAX=1; 2:EAX=1; 3:EAX=1; 4:EAX=1; 5:EAX=1;
142   :>0:EAX=1; 1:EAX=1; 2:EAX=1; 3:EAX=1; 4:EAX=1; 5:EAX=1;
Ok

Witnesses
Positive: 60922, Negative: 939078
Condition exists (0:EAX=0 /\ 1:EAX=0 /\ 2:EAX=0 /\ 3:EAX=0 /\ 4:EAX=0 /\ 5:EAX=0) is validated
Hash=107f1303932972b3abace3ee4027408e
Observation 6.SB Sometimes 60922 939078
Time 6.SB 1.62
```

We now see that the test validates. Moreover all of the 64 possible outcomes are observed.

Timebase synchronisation works as follows: at every iteration,

1. one of the threads reads timebase T;
2. all threads synchronise by the means of a polling synchronisation barrier;
3. each thread computes $T_i = T + \delta_i$, where δ_i is *the timebase delay*, a thread specific constant;
4. each thread loops, reading the timebase until the read value exceeds T_i.

By default the timebase delay δ_i is $2^{11} = 2048$ for all threads.

The precision of timebase synchronisation can be illustrated by enabling the printing of all synchronisation timings:

```
% ./6.SB.exe +vb
99999: 672294[1] 671973[1] 672375[1] 672144[1] 672303[1] 672222[1]
99998:  4524[1]  4332[1]  4446[1]  2052[65]  2064[73]  4095[1]
...
99983:  4314[1]  3036[1]  3141[1]  2769[1]  4551[1]  3243[1]
99982:* 2061[36]  2064[33]  2067[11]  2079[12]  2064[14]  2064[24]
99981:  2121[1]  2382[1]  2586[1]  2643[1]  2502[1]  2592[1]
...
```

For each test iteration and each thread, two numbers are shown (1) the last timebase value read by and (2) (in brackets `[...]`) how many iterations of loop 4. were performed. Additionally a star "`*`" indicates the occurrence of the targeted outcome. Here, we see that a nearly perfect synchronisation can be achieved (cf. line `99982:` above).

Once timebase synchronisation have been selected (litmus option `-barrier timebase`), test executable behaviour can be altered by the following two command line options:

`-ta <n>` Change the timebase delay δ_i of all threads.

`-tb <0:n0;1:n1;···>` Change the timebase delay δ_i of individual threads.

The litmus command line option `-vb true` (verbose barrier) governs the printing of synchronisation timings. It comes handy when choosing values for the `-ta` and `-tb` options. When set, the executable show synchronisation timings for outcomes that validate the test final condition. This default behaviour can be altered with the following two command line options:

`-vb` Do not show synchronisation timings.

`+vb` Show synchronisation timings for all outcomes.

Synchronisation timings are expressed in timebase ticks. The format depends on the synchronisation mode (litmus option `-barrier`). This section just gave two examples for user mode (timings are show as differences from thread P_0); and for timebase mode (timings are shown as differences from a commonly agreed by all thread timebase value). Notice that, when affinity control is enabled, the running logical processors of threads are also shown.

3.2 Advanced prefetch control

Supplying the tags `custom`, `static`, `static1` or `static2` to litmus command line option `-preload` commands the insertion of cache prefetch or flush instructions before every test instance.

In custom mode the execution of such cache management instruction is under total user control, the other, "static", modes offer less control to the user, for the sake of not altering test code proper.

3.2.1 Custom prefetch

Custom prefetch mode offers complete control over cache management instructions. Users enable this mode by supplying the command line option `-preload custom` to litmus. For instance one may compile the x86 test `6.SB.litmus` as follows:

```
% mkdir -p R
% litmus -mem indirect -preload custom -o R 6.SB.litmus
% cd R
% make
```

Notice the test is compiled in indirect memory mode, in order to reduce false sharing effects.

The executable `6.SB.exe` accepts two new command line options: `-prf` and `-pra`. Those options takes arguments that describe cache management instructions. The option `-pra` takes one letter that stands for a cache management instruction as we here describe:

`I`: do nothing, `F`: cache flush, `T`: cache touch, `W`: cache touch for a write.

All those cache management instructions are not provided by all architectures, in case some instruction is missing, the letters behave as follows:

`F`: do nothing, `T`: do nothing, `W`: behave as `T`.

With `-pra` X the commanded action applies to all threads and all variables, for instance:

```
% ./6.SB.exe -pra T
```

will perform a run where every test thread touches the test locations that it refers to (*i.e.* `x` and `y` for Thread 0, `y` and `z` for Thread 1, etc.) before executing test code proper. Although one may achieve interesting results by using this `-pra` option, the more selective `-prf` option should prove more useful. The `-prf` option takes a comma separated list of cache managment directives. A cache management directive is n:*loc*=X, where n is a thread number, *loc* is a program variable, and X is a cache management controle letter. For instance, `-prf 0:y=T` instructs thread 0 to touch location `y`. More generally, having each thread of the test **6.SB** to touch the memory location it reads with its second instruction would favor reading the initial value of these locations, and thus validating the final condition of the test "`(0:EAX=0 /\ 1:EAX=0 /\ 2:EAX=0 /\ 3:EAX=0 /\ 4:EAX=0 /\ 5:EAX=0)`".

Notice that those locations can be found by looking at the test code or at the diagram of the target execution. Let us have a try:

```
./6.SB.exe -prf 0:y=T,1:z=T,2:a=T,3:b=T,4:c=T,5:x=T
Test 6.SB Allowed
Histogram (63 states)
10    *>0:EAX=0; 1:EAX=0; 2:EAX=0; 3:EAX=0; 4:EAX=0; 5:EAX=0;
...
Witnesses
Positive: 10, Negative: 999990
...
Prefetch=0:y=T,1:z=T,2:a=T,3:b=T,4:c=T,5:x=T
...
```

As can be seen, the final condition is validated. Also notice that the prefetch directives used during the run are reminded. If given several times, `-prf` options cumulate, the rightmost directives taking precedence in case of ambiguity. As a consequence, one may achieve the same prefetching effect as above with:

```
% ./6.SB.exe -prf 0:y=T -prf 1:z=T -prf 2:a=T -prf 3:b=T -prf 4:c=T -prf 5:x=T
```

3.2.2 Prefetch metadata

The source code of tests may include prefetch directives as metadata prefixed with "`Prefetch=`". In particular, the generators of the diy suite (see Part II) produce such metadata. For instance in the case of the **6.SB** test (generated source `6.SB+Prefetch.litmus`), this metadata reads:

```
Prefetch=0:x=F,0:y=T,1:y=F,1:z=T,2:z=F,2:a=T,3:a=F,3:b=T,4:b=F,4:c=T,5:c=F,5:x=T
```

That is, each thread flushes the location it stores to and touches each location it reads from. Notice that each thread starts with a memory location access (here a store) and ends with another (here a load). The idea simply is to accelerate the exit access (with a cache touch) while delaying the entry access (with a cache flush).

When prefetch metadata is available, it acts as the default of prefetch directives:

```
% litmus -mem indirect -preload custom -o R 6.SB+Prefetch.litmus
% cd R
% make
```

Then we run the test by:

```
% ./6.SB+Prefetch.exe
Test 6.SB Allowed
Histogram (63 states)
674   *>0:EAX=0; 1:EAX=0; 2:EAX=0; 3:EAX=0; 4:EAX=0; 5:EAX=0;
```

```
...
Witnesses
Positive: 674, Negative: 999326
...
Prefetch=0:x=F,0:y=T,1:y=F,1:z=T,2:a=T,2:z=F,3:a=F,3:b=T,4:b=F,4:c=T,5:c=F,5:x=T
...
```

One may notice that the prefetch directives from the source file medata found its way to the test executable.

As with any kind of metadata, one can change the prefetch metadata by editing the litmus source file, or better by using the `-hints` command line option. The `-hints` command line option takes a filename as argument. This file is a mapping that associates new metadata to test names. As an example, we reverse diy scheme for cache management directives: accelerating entry accesses and delaying exit accesses:

```
% cat map.txt
6.SB Prefetch=0:x=W,0:y=F,1:y=W,1:z=F,2:a=F,2:z=W,3:a=W,3:b=F,4:b=W,4:c=F,5:c=W,5:x=F
% litmus -mem indirect -preload custom -hints map.txt -o R 6.SB.litmus
% cd R
% make
...
% ./6.SB.exe
Test 6.SB Allowed
Histogram (63 states)
24    *>0:EAX=0; 1:EAX=0; 2:EAX=0; 3:EAX=0; 4:EAX=0; 5:EAX=0;
...
Prefetch=0:x=W,0:y=F,1:y=W,1:z=F,2:a=F,2:z=W,3:a=W,3:b=F,4:b=W,4:c=F,5:c=W,5:x=F
...
```

As we see above, the final condition validates. It does so in spite of the apparently unfavourable cache management directives.

We can experiment further without recompilation, by using the `-pra` and `-prf` command line options of the test executable. Those are parsed left-to-right, so that we can (1) cancel any default cache management directive with `-pra I` and (2) enable cache touch for the stores:

```
% ./6.SB.exe -pra I -prf 0:x=W -prf 1:y=W -prf 2:z=W -prf 3:a=W -prf 4:b=W -prf 5:c=W
Test 6.SB Allowed
...
Witnesses
Positive: 0, Negative: 1000000
...
Prefetch=0:x=W,1:y=W,2:z=W,3:a=W,4:b=W,5:c=W
```

As we see, the final condition does not validate.

By contrast, flushing or touching the locations that the threads load permit to repetitively achieve validation:

```
chi% ./6.SB.exe -pra I -prf 0:y=F -prf 1:z=F -prf 2:a=F -prf 3:b=F -prf 4:c=F -prf 5:x=F
Test 6.SB Allowed
Histogram (63 states)
211   *>0:EAX=0; 1:EAX=0; 2:EAX=0; 3:EAX=0; 4:EAX=0; 5:EAX=0;
...
% ./6.SB.exe -pra I -prf 0:y=T -prf 1:z=T -prf 2:a=T -prf 3:b=T -prf 4:c=T -prf 5:x=T
Test 6.SB Allowed
Histogram (63 states)
10    *>0:EAX=0; 1:EAX=0; 2:EAX=0; 3:EAX=0; 4:EAX=0; 5:EAX=0;
...
```

As a conclusion, interpreting the impact of cache management directives is not easy. However, custom preload mode (litmus command line option `-preload custom`) and test executable options `-pra` and `-prf` allow experimentation on specific tests.

3.2.3 "Static" prefetch control

Custom prefetch mode comes handy when one wants to tailor cache management directives for a particular test. In practice, we run batches of tests using source metadata for prefetch directives. In such a setting, the code that interprets the prefetch directives is useless, as we do not use the `-prf` option of the test executables. As this code get executed before each test thread code, it may impact test results. It is desirable to supress this code from test executables, still performing cache management instructions. To that aim, litmus provides some "static" preload modes, enabled with command line options `-preload static`, `-preload static1` and `-preload static2`.

In the former mode `-preload static` and without any further user intervention, each test thread executes the cache management instructions commanded by the `Prefetch` metadata:

```
% mkdir -p S
% litmus -mem indirect -preload static -o R 6.SB+Prefetch.litmus
% make -C S
% S/6.SB+Prefetch.exe
Test 6.SB Allowed
Histogram (63 states)
804   *>0:EAX=0; 1:EAX=0; 2:EAX=0; 3:EAX=0; 4:EAX=0; 5:EAX=0;
...
Observation 804 999196
...
```

As we can see above, the effect of the cache management instructions looks more favorable than in custom preload mode.

Users still have a limited control on the execution of cache management instructions: produced executable accept a new `-prs <n>` option, which take a positive or null integer as argument. Then, each test thread executes the cache management instructions commanded by source metadata with probability $1/n$, the special value $n = 0$ disabling prefetch altogether. The default for the `-prs` options is "`1`" (always execute the cache management instructions). Let us try:

```
% S/6.SB+Prefetch.exe -prs 0 | grep Observation
Observation 6.SB Never 0 1000000
% S/6.SB+Prefetch.exe -prs 1 | grep Observation
Observation 6.SB Sometimes 901 999099
% S/6.SB+Prefetch.exe -prs 2 | grep Observation
Observation 6.SB Sometimes 29 999971
% S/6.SB+Prefetch.exe -prs 3 | grep Observation
Observation 6.SB Sometimes 16 999984
```

In those experiments we show the "`Observation`" field of litmus output: this field gives the count of outcomes that validate the final condition, followed by the count of outcomes that do not validate the final condition. The above counts confirm that cache management instructions favor validation.

The remaining preload modes `static1` and `static2` are similar, except that they produce executable files that do not accept the `-prs` option. Furthermore, in the former mode `-preload static1` cache management instructions are always executed, while in the latter mode `-preload static2` cache management instructions are executed with probability 1/2. Those modes thus act as pure static mode (litmus option `-preload static`), with runtime options `-prs 1` and `-prs 2` respectively. Moreover, as the test scaffold includes no code to interpret the `-prs <n>` switch, the test code is less perturbed. In practice and for the **6.SB** example, there is little difference:

```
% mkdir -p S1 S2
% litmus -mem indirect -preload static1 -o S1 6.SB+Prefetch.litmus
% litmus -mem indirect -preload static2 -o S2 6.SB+Prefetch.litmus
% make -C S1 && make -C S2
...
% S1/6.SB+Prefetch.exe | grep Observation
Observation 6.SB Sometimes 1119 998881
% S2/6.SB+Prefetch.exe | grep Observation
Observation 6.SB Sometimes 16 999984
```

4 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 <dest>` Save C-source of test files into `<dest>` instead of running them. If argument `<dest>` is an archive (extension `.tar`) or a compressed archive (extension `.tgz`), litmus builds an archive: this is the "cross compilation feature" demonstrated in Sec. 1.2. Otherwise, `<dest>` is interpreted as the name of an existing directory and tests are saved in it.

`-driver (shell|C|XCode)` Choose the driver that will run the tests. In the "`shell`" (and default) mode, each test will be compiled into an executable. A dedicated shell script `run.sh` will launch the test executables. In the "`C`" mode, one executable `run.exe` is produced, which will launch the tests (see Sec. `??` for an example). Finally, the `XCode` mode is for inclusion of the tests into a dedicated iOS App, which we do not distribute at the moment.

`-crossrun <(user@)?host(:port)?|adb>` When the shell driver is used (`-driver shell` above), instruct the `run.sh` script to run individual tests on a remote machine. The remote host can be contacted by the means of `ssh` or the Android Debug Bridge.

`ssh` `user` is a login name on the the remote host, `<host>` is the name of the remote host, and `port` is a port-number which can be omitted when standard (22).

`adb` Tests will be run in the remote directory `/data/tmp`.

This option may be useful when the tested machine has little disk space or a crippled installation. Default is disabled — *i.e.* run tests on the machine where the `run.sh` script runs.

`-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 logical processors — set default value for `-a` of Sec. 2.3. Default is 1 (run one test). When affinity control is enabled, the value 0 has the special meaning of having executables to set the number of available logical processors according to how many are actually present.

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

`-r <n>` Perform n runs — set default value for option `-r` of Sec. 2.3. The option accepts generalised syntax for integers and default is 10.

`-s <n>` Size of a run — set default value for option `-s` of Sec. 2.3. The option accepts generalised syntax for integers and default is 100000 (or `100k`).

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

`-barrier (user|userfence|pthread|none|timebase)` Set synchronisation mode, default `user`. Synchronisation modes are described in Sec. 2.1

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

`-mem (indirect|direct)` Set memory mode, default `indirect`. It is possible to instruct executables compiled in indirect mode to behave almost as if compiled in direct mode, see Sec. 2.3.

`-stride <n>` Specify a stride value of `<n>` — set default value for option `-st` of Sec. 2.3. See Sec. 2.1 for details on the stride parameter. If ¡n¿ is negative or zero, restore the default, which is stride feature disabled.

`-st <n>` Alias for `-stride <n>`.

`-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 (no|random|custom|static|static1|static2)` Specify preload mode (see Sec. 2.1), default is `random`. Starting from version 5.0 we provide additional "custom" and "static" modes for a finer control of prefetching and flushing of some memory locations by some threads. See Sec 3.2.

`-safer (no|all|write)` Specify safer mode, default is `write`. When instructed to do so, executable files perform some consistency checks. Those are intended both for debugging and for dynamically checking some assumptions on POSIX threads that we rely upon. More specifically the test harness checks for the stabilisation of memory locations after a test round in the "`all`" and "`write`" mode, while the initial values of memory locations are checked in "`all`" mode.

`-speedcheck (no|some|all)` Quick condition check mode, default is "`no`". In mode "`some`", test executable will stop as soon as its condition is settled. In mode "`all`", the `run.sh` script will additionally not run the test if invoked once more later.

The following optiondra commands affinity control:

`-affinity (none|incr<n>|random|custom)` Enable (of disable with tag `none`) affinity control, specifying default affinity mode of executables. Default is `none`, *i.e.* executables do not include affinity control code. The various tags are interpreted as follows:

1. `incr<n>`: integer `<n>` is the increment for allocating logical processors to threads — see Sec. 2.2. Notice that with `-affinity incr0` the produced code features affinity control, which executable files do not exercise by default.
2. `random`: executables perform random allocation of test threads to logical processors.
3. `custom`: executables perform custom allocation of test threads to logical processors.

Notice that the default for executables can be overridden using options `-i`,`+ra` and `+ca` of Sec. 2.3.

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

Notice that affinity control is not implemented for MacOs.

The following options are significant when affinity control is enabled. Otherwise they are silent no-ops.

`-p <ns>` Specify the sequence of logical processors. 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.

`-force_affinity <bool>` Code that sets affinity will spin until all specified cores (as given with option `-avail <n>`) processors are up. This option is necessary on devices that let core sleep when the computing load is low. Default is false.

Custom affinity control (see Sec. 2.2.4) is enabled, first by enabling affinity control (*e.g.* with `-affinity ...`), and then by specifying a logical processor topology with options `-smt` and `-smt_mode`.

`-smt <n>` Specify that logical processors are close by groups of n, default is `1`.

`-smt_mode (none|seq|end)` Specify how "close" logical processors are numbered, default is `none`. In mode "`end`", logical processors of the same core are numbered as c, $c + A_c$ etc. where c is a physical core number and A_c is the number of physical cores available. In mode "`seq`", logical processors of the same core are numbered in sequence.

Notice that custom affinity works only for those tests that include the proper meta-information. Otherwise, custom affinity silently degrades to random affinity.

Finally, a few miscellaneous options are documented:

`-l <n>` Insert the assembly code of each thread in test in a loop of size `<n>`. Accepts generalised integer syntax, disabled by default. Sets default value for option `-l` of Sec. 2.3.

This feature may prove useful for measuring running times that are not too much perturbed by the test harness, in combination with options `-s 1 -r 1`.

`-vb <bool>` Disable/enable the printing of synchronisation timings, default is `false`.

This feature may prove useful for analysing the synchronisation behaviour of a specific test, see Sec. 3.1.

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

`-gcc <name>` Change the name of C compiler, default `gcc`.

`-linkopt <flags>` Set `gcc` linking flags. (default: void).

`-gas <bool>` Emit Gnu as extensions (default Linux/Mac=`true`, AIX=`false`)

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`.

Change input Some items in the source of tests can be changed at the very last moment. The new items are defined in mapping files whose names are arguments to the appropriate command line options. Mapping files simply are lists of pairs, with one line starting with a test name, and the rest of line defining the changed item. The changed item may also contains several lines: in that case it should be included in double quotes "`"`".

`-names <file>` Run litmus only on tests whose names are listed in `<file>`.

`-rename <file>` Change test names.

`-kinds <file>` Change test kinds. This amonts to changing the quantifier of final conditions, with kind `Allow` being `exists`, kind `Forbid` being `~exists` and kind `Require` being `forall`.

`-conds <file>` Change the final condition of tests.

`-hints <file>` Change meta-data, or hints. Hints command avanced features such as custom affinity (option `-affinity custom` and Sec. 2.2.4) and prefech control (option `-preload custom` and Sec. 3.2).

Observe that the rename mapping is applied first. As a result kind or condition change must refer to new names. For instance, we can highlight that a X86 machine is not sequentially consistent by first renaming **SB** into **SB+SC**, and then changing the final condition. The new condition expresses that the first instruction (a store) of one of the threads must come first:

```
          rename.txt                                              cond.txt
```

```
SB SB+SC                                          SB+SC "forall (0:EAX=1 \/ 1:EAX=1)"
```

Then, we run litmus:

```
% litmus -mach x86 -rename rename.txt -conds cond.txt SB.litmus
%%%%%%%%%%%%%%%%%%%%%%%%%%
% Results for SB.litmus %
%%%%%%%%%%%%%%%%%%%%%%%%%%
X86 SB+SC
"Fre PodWR Fre PodWR"

{x=0; y=0;}

 P0          | P1          ;
 MOV [x],$1  | MOV [y],$1  ;
 MOV EAX,[y] | MOV EAX,[x] ;
```

```
forall (0:EAX=1 \/ 1:EAX=1)
Generated assembler
#START _litmus_P1
        movl $1,(%r8,%rdx)
        movl (%rdx),%eax
#START _litmus_P0
        movl $1,(%rdx)
        movl (%r8,%rdx),%eax

Test SB+SC Required
Histogram (4 states)
39954 *>0:EAX=0; 1:EAX=0;
3979407:>0:EAX=1; 1:EAX=0;
3980444:>0:EAX=0; 1:EAX=1;
195   :>0:EAX=1; 1:EAX=1;
No

Witnesses
Positive: 7960046, Negative: 39954
Condition forall (0:EAX=1 \/ 1:EAX=1) is NOT validated
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
Observation SB+SC Sometimes 7960046 39954
Time SB+SC 0.48
```

One sees that the test name and final condition have changed.

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
`-l`	loop	integer

Notice that litmus in fact accepts long versions of options (*e.g.* `-avail` for `-a`).

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`. 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).

5 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].

The distribution includes additional test generators: diyone for generating one test and diycross for generating simple variations on one test.

5.1 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 `SB.litmus`:

```
X86 SB
"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) Wx1 and (a) Wy1 and two reads (b) Rxv_1 with a certain value v_1 and (d) Ryv_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 ∧ 1:EAX = 1
Allowed: 0:EAX = 1 ∧ 1:EAX = 0
Allowed: 0:EAX = 1 ∧ 1:EAX = 1

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

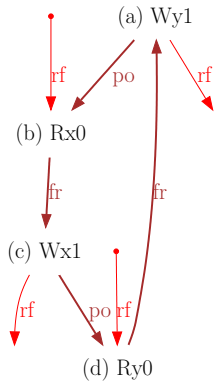
Allowed: 0:EAX = 0 ∧ 1:EAX = 0

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.

5.2 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:

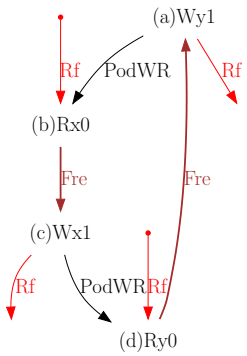


  Constraint arrows or *global* arrows are shown in brown colour. As the graph contains a cycle of brown arrows, the events cannot be ordered. Hence the execution presented is not allowed by SC.

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, `SB.litmus` can be created as follows:

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

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

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

diy produces two litmus tests, `SB000.litmus` and `SB001.litmus`, plus one index file `@all`. One of the litmus tests generated is the same as above, while the new test is:

```
% cat SB001.litmus
X86 SB001
"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 SB
# Revision: 3333
SB000.litmus
SB001.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.

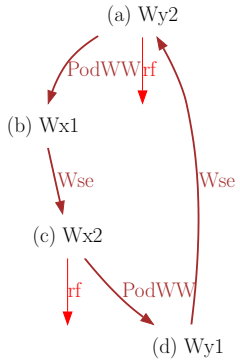
5.3 More candidate relaxations

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 x86-2+2W 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 x86-2+2W
"Wse PodWW Wse PodWW"
Prefetch=0:x=F,0:y=W,1:y=F,1:x=W
Com=Ws Ws
Orig=Wse PodWW Wse PodWW
{
}
 P0          | P1          ;
 MOV [x],$2 | MOV [y],$2 ;
 MOV [y],$1 | MOV [x],$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 **x86-2+2W**:

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

See Sec. 9 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`.

Figure 1: Cycles for iriw and iriw-internal

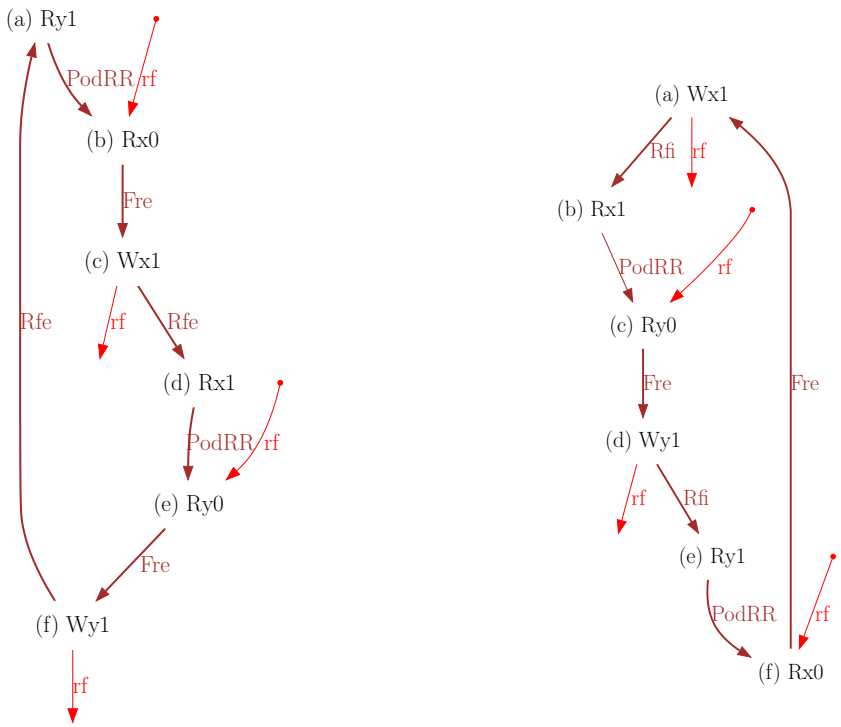


Figure 2: Litmus tests iriw and iriw-internal

```
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)
```

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

5.4 Summary of simple candidate relaxations

We summarise the candidate relaxations available on all architectures.

5.4.1 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

5.4.2 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.

5.4.3 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, LwSync, Eieio and Isync. Hence, to yield two reads to different locations and separated by the lightweight Power barrier `lwsync`, we specify LwSyncdRR. On ARM we have DMB, DSB and ISB.

6 Testing candidate relaxations with diy

The 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.

Notice that this style of model exploration is mechanised by dont (diy) — see Part V.

6.1 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. 5.3 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. 5.3, 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 candidate 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.

6.2 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.

7 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.

7.1 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:

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

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.

Finally, one may control the nature of the dependency: address dependency (DpAddr(s|d)(R|W) or data dependency (DpData(s|d)W).

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:

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

This default syntax expands to control dependencies as guaranteed by the Power documentation. For read to write, conditioning execution is enough (expanded syntax, DpCtrl(s|d)W). But for read to read, an extra instruction, `isync`, is needed (expanded syntax DpCtrlIsync(s|d)R, see below). The syntax DpCtrl(s|d)R also exists, it expresses the conditional execution of a load instruction and does *not* create ordering.

ARM has similar candidate relaxations, Isync being replaced by ISB.

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
"DpAddrdW Rfe DpAddrdW Rfe"
{
0:r2=y; 0:r5=x;
1:r2=x; 1:r5=y;
}
 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`. One may observe that DpdW is changed into DpAddrdW in the comment field of the test. As a matter of fact, DpdW is a macro for the address dependency DpAddrW. We could have specified data dependency instead:

```
% diyone DpDatadW Rfe DpAddrdW Rfe
PPC a
"DpDatadW Rfe DpAddrdW Rfe"
{
0:r2=y; 0:r4=x;
1:r2=x; 1:r5=y;
}
 P0           | P1            ;
 lwz r1,0(r2) | lwz r1,0(r2)  ;
 xor r3,r1,r1 | xor r3,r1,r1  ;
 addi r3,r3,1 | li r4,1       ;
 stw r3,0(r4) | stwx r4,r3,r5 ;
```

```
exists
(0:r1=1 /\ 1:r1=1)
```

On P_0, the value stored by the last (store) instruction `stw r3,0(r4)` is now computed from the value read by the first (load) instruction `lwz r1,0(r2)`. Again, this is a "false" dependency.

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

Also notice that CtrldR is interpreted as DpCtrlIsyncR in the comment field of the test.

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

7.2 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 `AC`r stands for `[Rfe,`r`]`, `BC`r stands for `[`r`,Rfe]`, while `ABC`r stands for `[Rfe,`r`,Rfe]`,

Hence, a simple way to generate iriw-like (see Sec. 5.3) 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)
```

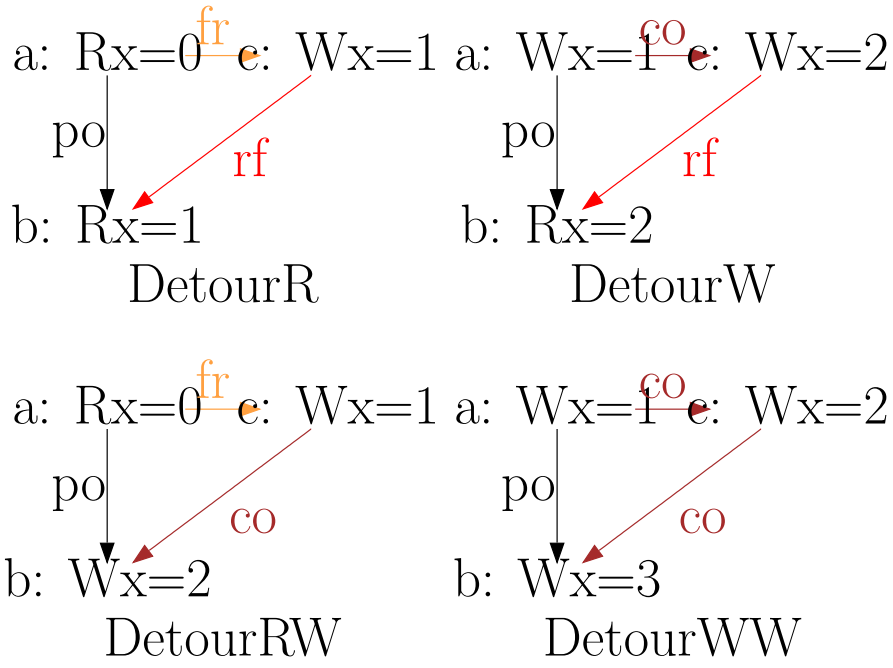
7.3 Detour candidate relaxations

Detours combine a Pos candidate relaxation and a sequence of two *external* communication candidate relaxations. More precisely detours are some constrained Pos candidate relaxations: the source and target events must be related by a sequence of two communication candidate relaxations, whose target and source are a common event whose processor is new.

diy syntax	Source	Target	Detour
`DetourR`	R	R	Fre; Rfe
`DetourW`	W	R	Wse; Rfe
`DetourRW`	R	W	Fre;Wse
`DetourWW`	W	W	Wse;Wse

DetourRR and DetourWR are accepted as synonyms for DetourR and DetourW respectively.

Graphically, we have:



Finally notice that "internal" detours need no special treatement as they can be expressed by the sequences "Fri; Rfi", "Wsi;Rfi", etc.

8 Test variations with diycross

The tool diycross has an interface similar to diyone, except it accepts list of candidate relaxations where diyone accepts single candidate relaxations. The new tool produces the test resulting by "cross producing" the lists. For instance, one can generate all variations on the IRIW test (see Sec. 5.3) that involve data dependencies and the lightweight barrier `lwsync` as follows:

```
% diycross -arch PPC -name IRIW Rfe DpdR,LwSyncdRR Fre Rfe DpdR,LwSyncdRR Fre
Generator produced 3 tests
% ls
@all  IRIW+addrs.litmus  IRIW+lwsync+addr.litmus  IRIW+lwsyncs.litmus
```

diycross outputs the index file `@all` that lists the test source files, and three tests, with names we believe to be self-explanatory:

```
% cat IRIW+lwsync+addr.litmus
PPC IRIW+lwsync+addr
"Rfe LwSyncdRR Fre Rfe DpAddrdR Fre"
Cycle=Rfe LwSyncdRR Fre Rfe DpAddrdR Fre
{
0:r2=y;
1:r2=y; 1:r4=x;
2:r2=x;
3:r2=x; 3:r5=y;
}
 P0           | P1           | P2           | P3            ;
 li r1,1      | lwz r1,0(r2) | li r1,1      | lwz r1,0(r2)  ;
 stw r1,0(r2) | lwsync       | stw r1,0(r2) | xor r3,r1,r1  ;
              | lwz r3,0(r4) |              | lwzx r4,r3,r5 ;
exists (1:r1=1 /\ 1:r3=0 /\ 3:r1=1 /\ 3:r4=0)
```

Users may use the special keywords allRR, allRW, allWR and allWW to specify the set of all existing program order candidate relaxations between the specified "R" or "W". For instance, we get the complete variations on IRIW by:

```
% diycross -arch PPC -name IRIW Rfe allRR Fre Rfe allRR Fre
Generator produced 28 tests
% ls
@all
IRIW.litmus
IRIW+addr+po.litmus
IRIW+lwsync+addr.litmus
...
IRIW+isyncs.litmus
```

9 Identifying coherence orders with observers

We first produce the "*four writes*" test 2+2W for Power:

```
% diyone -name 2+2W -arch PPC PodWW Wse PodWW Wse
% cat 2+2W.litmus
PPC 2+2W
"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 **2+2W** is the Power version of the x86 test **x86-2+2W** of Sec. 5.3. 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.

9.1 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 `2+2WObs`):

```
% diyone -name 2+2WObs -obs force -obstype straight -arch PPC PodWW Wse PodWW Wse
% cat 2+2WObs.litmus
PPC 2+2WObs
"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 `y`. With respect to `2+2W`, 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 may not be the case in general — although it holds for Power. Moreover, running `2+2W` and `2+2WObs` yields contrasted results. While a positive conclusion is immediate for `2+2W`, we were not able to reach a similar conclusion for `2+2WObs`. As a matter of fact, `2+2WObs` 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.

9.2 More observers

9.2.1 Fences and loops in 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 `2+2WObsFenced` by:

```
% diyone -name 2+2WObsFenced -obs force -obstype fenced -arch PPC PodWW Wse PodWW Wse
% cat 2+2WObsFenced.litmus
```

```
PPC 2+2WObsFenced
"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 `2+2WObsLoop`. The HTML version of this document provides details.

9.2.2 Local observers

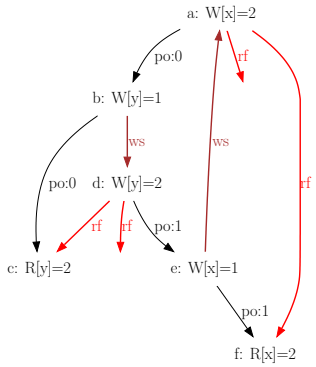
With local observers, coherence order is observed by the test threads. This implies changing the tests, and some care must be exercised when interpreting results.

The idea is as follows: when two threads are connected by a Wse candidate relaxation, meaning that the first thread ends by writing v to some location ℓ and that the second threads starts by writing $v+1$ to the same location ℓ, we add an observing read of location ℓ at the end of the first thread. Then, reading $v+1$ means that the write by the first thread precedes the write by the second thread in ℓ coherence order. More concretely, we instruct diy generators to emit such local observers with option `-obs local`:

```
% diyone -name 2+2WLocal -obs local -obstype straight -arch PPC PodWW Wse PodWW Wse
% cat 2+2WLocal.litmus
PPC 2+2WLocal
"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) ;
 lwz r5,0(r4) | lwz r5,0(r4) ;
exists
(0:r5=2 /\ 1:r5=2)
```

With respect to `2+2W`, final condition has changed, the direct observation of the final contents of locations `y` and `x` being replaced local observation of `y` by thread 0 and local observation of `x` by thread 1.

Based for instance on the test execution witness, whose only SC-violation cycle is the same as as for **2+2W**,



one may argue that tests **2+2W** and **2+2WLocal** are equivalent, in the sense that both are allowed or both are forbidden by a model or machine.

Local observers can also be fenced or looping. For instance, one produces **2+2WLocalFenced**, the fenced local observer version of **2+2W** as follows:

```
% diyone -name 2+2WLocalFenced -obs local -obstype fenced -arch PPC PodWW Wse PodWW Wse
% cat 2+2WLocalFenced.litmus
PPC 2+2WLocalFenced
"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) ;
 sync         | sync         ;
 lwz r5,0(r4) | lwz r5,0(r4) ;
exists
(0:r5=2 /\ 1:r5=2)
```

While one produces **2+2WLocalLoop**, the looping local observer version of **2+2W** as follows:

```
% diyone -name 2+2WLocalLoop -obs local -obstype loop -arch PPC PodWW Wse PodWW Wse
% cat 2+2WLocalLoop.litmus
PPC 2+2WLocalLoop
"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) ;
 li r6,200    | li r6,200    ;
 L00:         | L02:         ;
```

```
 lwz r5,0(r4)  | lwz r5,0(r4)  ;
 cmpwi r5,1    | cmpwi r5,1    ;
 bne  L01      | bne  L03      ;
 addi r6,r6,-1 | addi r6,r6,-1 ;
 cmpwi r6,0    | cmpwi r6,0    ;
 bne  L00      | bne  L02      ;
 L01:          | L03:          ;
exists (0:r5=2 /\ 1:r5=2)
```

In the code above, observing loads are attempted at most 200 time or until a value different from 1 is read.

9.2.3 Performance of observers

As an indication of the performance of the various sorts of observers, the following table summarises a litmus experiment performed on a 8-cores 4-ways SMT Power7 machine machine.

	2+2W	2+2WObs	2+2WObsFenced	2+2WObsLoop	2+2WLocal	2+2WLocalFenced	2+2W
Positive	2.2M/160M	0/80M	326/80M	25k/80M	2/160M	34k/160M	111

The row "Positive" shows the number of observed positive outcomes/total number of outcomes produced. For instance, in the case of **2+2W**, we observed the positive outcome `x=2 /\ y=2` more than 2 millions times out of a total of 160 millions outcomes. As a conclusion, all techniques achieve decent results, except straight observers.

9.3 Three stores or more

In test **2+2W** 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 four 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.

`local` Always produce local observers.

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.

With command line switch `-obs local` we get three local observations of coherence, which suffice to reconstruct the complete coherence orders:

```
% diyone -obs local -obstype fenced -arch PPC -name W5Local Wse Wse PodWW Wse PodWW
chi% cat W5Local.litmus
PPC W5Local
"Wse Wse PodWW Wse PodWW"
{
0:r2=x; 0:r4=y;
1:r2=y; 1:r4=x;
2:r2=x;
}
 P0           | P1           | P2           ;
 li r1,3      | li r1,2      | li r1,2      ;
 stw r1,0(r2) | stw r1,0(r2) | stw r1,0(r2) ;
 li r3,1      | li r3,1      | sync         ;
 stw r3,0(r4) | stw r3,0(r4) | lwz r3,0(r2) ;
 sync         | sync         |              ;
 lwz r5,0(r4) | lwz r5,0(r4) |              ;
exists (0:r5=2 /\ 1:r5=2 /\ 2:r3=3)
```

10 Command usage

The diy suite consists in four main tools:

diyone generates one litmus test from the specification of a violation of the sequential consistency memory model as a cycle — see Sec. 5.2.

diycross generates variations of diyone style tests — see Sec. 8.

diy generates several tests, aimed at confirming that candidate relaxations are relaxed or safe—see Sec. 6.

readRelax Extract relevant information from the results of tests—see Sec. 6.2.

10.1 A note on test names

We have designed a simple naming scheme for tests. A normalised test name decomposes first as a family name, and second as a description of program-order (or internal) candidate relaxations.

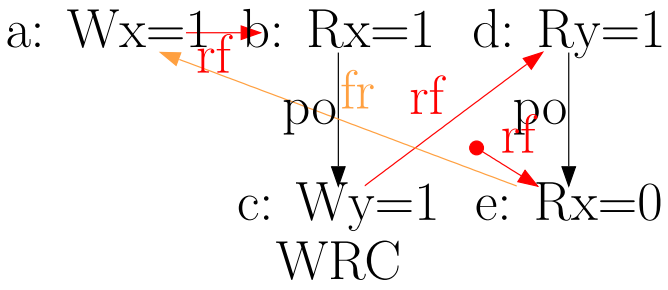
10.1.1 Family names

Cycles (and thus tests) are first grouped by families. Family names describe test structure, based upon external communication candidates relaxations. More specifically, external communication candidates relaxations suffice to settle the directions (`W` or `R`) of first and last events of threads, considering the case when those two events are the same. For instance, consider the cycle "PodWW Rfe PodRR Fre": there are two threads in the corresponding test (as there are two external communication candidate relaxations), one thread starts and ends with a write (written `WW`), while the other thread starts and ends with a read (written `RR`). The family name is thus `WW+RR`, (or `RR+WW`, but we choose the former). For reference, a normalised family name is the minimal amongst the representations of a given cycle, following the lexical order derived from the order $\texttt{W} < \texttt{WW} < \texttt{RR} < \texttt{RW} < \texttt{WR} < \texttt{R}$.

The most common families have nicknames, which are defined by this document[4]. For instance, consider the test whose cycle is "PodWR Fre PodWR Fre". The family name is `WR+WR`, as this is a two-thread test, both threads starting with a write and ending with a read. The nickname for this family is, as we already know, SB (store-buffering). Here is the list of nicknames and family names for two thread tests:

2+2W	`WW+WW`	PodWW Wse PodWW Wse
LB	`RW+RW`	PodRW Rfe PodRW Rfe
MP	`WW+RR`	PodWW Rfe PodRR Fre
R	`WW+WR`	PodWW Wse PodWR Fre
S	`WW+RW`	PodWW Rfe PodRW Wse
SB	`WR+WR`	PodWR Fre PodWR Fre

Isolated writes (and reads) originate from the combinations of communication relaxations, for instance [Fre,Rfe]. They appear as "`W`" (and `R`) in family names. For instance, "Rfe PodRR Fre Rfe PodRR Fre" contains two such isolated writes, its name is thus `W+RR+W+RR` and its nickname is, as we know, IRIW (Independent reads of independent writes). The test "Rfe PodRW Rfe PodRR Fre" contains one isolated write, as apparent from this diagram:



The family name is thus `W+RW+RR` and the nickname is WRC (Write to Read Causality).

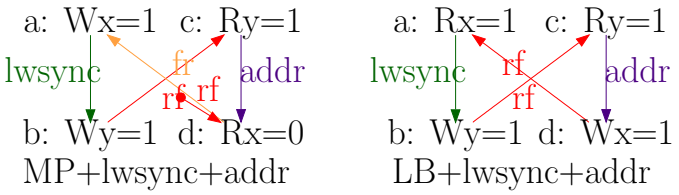
10.1.2 Descriptive names for variants

Every family has a prototype, homonymous test where every thread code consists in one (for `W` or `R`) or two memory accesses to different locations (for `WW`, `WR` etc.). For instance, the **MP** test is derived from the cycle "PodWW Rfe PodRR Fre". Variants are described by tags that illustrates the various program-order relaxations: they appear after the family name, still with "`+`" as a separation. For instance the test derived from "LwSyncdWW Rfe DpAddrdR Fre" is named **MP+lwsync+addr**.

When all threads have the same tag *tag*, the test name is abbreviated as *`Family+tags`*. For instance, the test **MP+lwsync+lwsync** ("LwSyncdWW Rfe LwSyncdRR Fre") is in fact **MP+lwsyncs**. Additionally, the tag `pos` (all `po`'s) is omitted, in order to yield family names for the prototype tests — cf. **MP** whose name would have been `MP+pos` otherwise.

For the sake of terseness, tags do not describe program-order relaxations completely. For instance both DpAddrdR and DpAddrdW (address dependency to read and write, respectively) have the same tag, `addr`. It does not harm for simple tests, as the missing direction can be inferred from the family name. Consider for instance **MP+lwsync+addr** and **LB+lwsync+addr**.

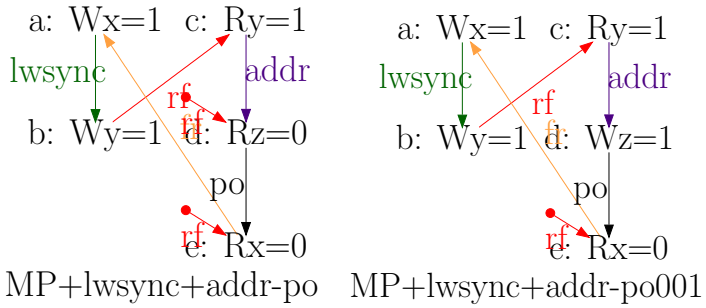
[4] `http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test6.pdf`



The naming scheme extends to cycles with consecutive program-order relaxations, by separating tags with "-" when they follow one another: for instance "LwSyncdWW Rfe DpAddrdR PodRR Fre" is named **MP+lwsync+addr-po**. Unfortunately, the current naming scheme falls short in supplying non-ambiguous names to all tests. For instance, "LwSyncdWW Rfe DpAddrdW PodWR Fre" is also named **MP+lwsync+addr-po**. In that situation tools will either fail or silently add a numeric suffix, depending on the boolean `-addnum` option.

```
% diycross -addnum false LwSyncdWW Rfe [DpAddrdR,PodRR],[DpAddrdW,PodWR] Fre
Fatal error: Duplicate name MP+lwsync+addr-po
% diycross -addnum true LwSyncdWW Rfe [DpAddrdR,PodRR],[DpAddrdW,PodWR] Fre
Generator produced 2 tests
% cat @all
# diycross -addnum true LwSyncdWW Rfe [DpAddrdR,PodRR],[DpAddrdW,PodWR] Fre
MP+lwsync+addr-po.litmus
MP+lwsync+addr-po001.litmus
```

As a result, we get the two tests: **MP+lwsync+addr-po** and **MP+lwsync+addr-po001**.



Future versions of diy may solve this issue in a more satisfying manner. At the moment, users are advised not to rely too much on the automatic naming scheme. Users may name tests in a non-ambiguous fashion by (1) specifying an explicit family name (`-name` *name*) and (2) selecting the numeric scheme (`-num true`):

```
% diycross -name MP+X -num true LwSyncdWW Rfe [DpAddrdR,PodRR],[DpAddrdW,PodWR] Fre
Generator produced 2 tests
```

The diycross generator outputs the same tests as above, with names **MP+X000** and **MP+X001**.

10.2 Common options

All diy test generators accept the following documented command-line options:

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

`-version` Show version number and exit.

`-arch (X86|PPC|ARM)` Set architecture. Default is `PPC`. ARM support is experimental.

`-o <dest>` Redirect output to `<dest>`. This option applies when tools generate a set of tests and an index file `@all`, *.i.e.* in all situations except for diyone simplest operating mode.

If argument `<dest>` is an archive (extension `.tar`) or a compressed archive (extension `.tgz`), the tool builds an archive. Otherwise, `<dest>` is interpreted as the name of an existing directory. Default is "`.`", that is tool output goes into the current directory.

`-obs (accept|avoid|force|local)` Management of observers, default is `avoid`. See Sec. 9.3.

`-obstype (fenced|loop|straight)` Style of observers, default is `fenced`. See Sec. 9.2.

`-cond (cycle|uni|observe)` Control final condition of tests, default is `cycle`. In mode `cycle`, the final condition identifies executions that correspond to the generating cycle. In mode `unicond`, the final condition identifies executions that are valid w.r.t. the uniproc model (see Sec. 12.2). In mode `observe` there is no final condition: the litmus and herd tools will simply list the final values of locations.

`-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.

`-optcoherence` Optimise conditions assuming that the tested system (at least) follows the uniproc model (see Sec. 12.2).

`-nooptcoherence` Do not optimise conditions assuming that the tested system (at least) follows the uniproc model. This is the default.

`-neg <bool>` Negate final condition, default is `false`.

`-c <bool>` Avoid equivalent cycles. Default is `true`. Setting `-c true` is intended for debug.

The naming of tests is controlled by the following options:

`-name <name>` Use name for naming tests, the exact consequences depend on the generator. By default the generator has no name available.

`-num <bool>` Use numeric names, *i.e.* from a base name ¡base¿ the generator will name tests as `<base>000`, `<base>001` etc. The default depends upon the generator.

`-addnum <bool>` If true, when faced with tests whose name `<name>` has already been given, use names `<name>001`, `<name>002`, etc. Otherwise fail in the same situation. The default depends upon the generator.

`-fmt <n>` Size of numerical suffixes, default is 3.

10.3 Usage of diyone

The tool diyone has two operating modes. The selected mode depends on the presence of command-line arguments,

In the first operating mode, diyone takes a non-empty 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
```

In this mode, `-name <name>` sets the name of the test to `<name>` and output it into file `<name>.litmus`. If absent, the test name is `A` and output goes to standard output.

Otherwise, *i.e.* when there are no command-line arguments, diyone reads the standard input and generates the tests described by the lines it reads. Each line starts with a test name *name*, followed by ":", followed by a list of candidate relaxations *RS*. Then, diyone acts as if invoked as `diyone` *opts* `-name` *name RS*.

The tool diyone accepts the following documented option:

`-norm` Normalise tests and give them normalised names. In the first operating mode (when a cycle is explicitly given) the test will be named with a family name and a descriptive name. In the second operating mode, numeric names are used, base being either given explicitly (with option `-name <base>`) or being a normalised family name.

10.4 Usage of diycross

diycross produces several tests by "cross producing" lists of candidate relaxations given as arguments, see Sec 8. diycross also produces an index file `@all` that lists all produced litmus source files.

If option `-name <name>` is given, it sets the family name of generated tests, otherwise standard family names are used (cf. Sec. 10.1). By default descriptive names are used (*i.e.* `-num false`) and diycross will fail if two different tests have the same name (*i.e.* `-addnum false`):

```
% diycross PodWW Rfe [DpAddrdR,PodRR],[DpAddrdW,PodWR] Fre
Fatal error: Duplicate name MP+po+addr-po
```

Should this happen users can resort either to numeric names,

```
%diycross -num true PodWW Rfe [DpAddrdR,PodRR],[DpAddrdW,PodWR] Fre
Generator produced 2 tests
con% ls
@all  MP000.litmus  MP001.litmus
```

or to disambiguating numeric suffixes.

```
%diycross -addnum true PodWW Rfe [DpAddrdR,PodRR],[DpAddrdW,PodWR] Fre
Generator produced 2 tests
con% ls
@all  MP+po+addr-po001.litmus  MP+po+addr-po.litmus
```

10.5 Usage of diy

As diycross, diy produce several files, hence naming issues are critical. By default, diy uses family names and the numeric naming scheme (`-num true`). Users can specify a family name *family* for all tests with `-name` *family*, or attempt using the descriptive names of Sec 10.1 with `-num false`. Moreover, diy produces an index file `@all` that lists the file names of all tests produced.

The tool 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.

`-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 originates from `<n>` edges or more. Default is 4.

`-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`.

`-maxrelax <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|free|uni)` Control generation of cycles, default `sc`. Those tags command the activation of some constraints over cycle generation, see below.

`-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 in `sc` (default) mode 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 above 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 9.3.

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[5] 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. Cycles that access one single memory location are rejected.

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

5. 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 [5].

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

1. Communication candidate relaxations sequences are limited to Rf,Fr,Ws,[Ws,Rf] and [Fr,Rf]. However, arbitrary sequences of communication candidates are accepted when they are internal and external or external and internal.

2. Cycles that access one single memory location are rejected.

Finally, the `uni` mode enforces the following constraints on cycles:

1. Sequences of communication candidate relaxations are restricted in the same manner as for `free` mode (see above).

2. Sequences of Po candidate relaxation are rejected.

10.6 Usage of readRelax

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

- Whether the test yielded `Ok` or not,
- 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. 6.2 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.

11 Additional tools: extracting cycles and classification

When non-standard family names or numeric names are used, it proves convenient to rename tests with the standard naming scheme. We provide two tools to do so: mcycles that extracts cycles from litmus source files and classify that normalises and renames cycles.

For instance, one can use diy to generate all simple, critical, tests up to three threads for X86 with the following configuration file `X.conf`

```
-arch X86
-name X
-nprocs 3
-size 6
-safe Pod**,Fre,Rfe,Wse
-mode critical
```

```
% diy -conf X.conf
Generator produced 23 tests
% ls
```

```
@all         X003.litmus  X007.litmus  X011.litmus  X015.litmus  X019.litmus  X.conf
X000.litmus  X004.litmus  X008.litmus  X012.litmus  X016.litmus  X020.litmus
X001.litmus  X005.litmus  X009.litmus  X013.litmus  X017.litmus  X021.litmus
X002.litmus  X006.litmus  X010.litmus  X014.litmus  X018.litmus  X022.litmus
```

Cycles are extracted with mcycles, which takes the index file `@all` as argument:

```
% mcycles @all
X000: Wse PodWR Fre PodWR Fre PodWW
X001: Rfe PodRR Fre PodWR Fre PodWW
X002: Wse PodWR Fre PodWW
X003: Wse PodWW Wse PodWR Fre PodWW
X004: Rfe PodRW Wse PodWR Fre PodWW
X005: Rfe PodRR Fre PodWW
X006: Wse PodWW Rfe PodRR Fre PodWW
X007: Rfe PodRW Rfe PodRR Fre PodWW
X008: Wse Rfe PodRR Fre PodWW
X009: Wse PodWW Wse PodWW
...
```

The output of mcycles can be piped into classify for family classification:

```
% mcycles @all | classify -arch X86
2+2W
  X009 -> 2+2W : PodWW Wse PodWW Wse
3.2W
  X010 -> 3.2W : PodWW Wse PodWW Wse PodWW Wse
3.LB
  X020 -> 3.LB : PodRW Rfe PodRW Rfe PodRW Rfe
3.SB
  X016 -> 3.SB : PodWR Fre PodWR Fre PodWR Fre
ISA2
  X007 -> ISA2 : PodWW Rfe PodRW Rfe PodRR Fre
LB
  X019 -> LB : PodRW Rfe PodRW Rfe
MP
  X005 -> MP : PodWW Rfe PodRR Fre
...
```

Notice that classify accepts the `arch` option, as it needs to parse cycles.

Finally, one can normalise tests, using normalised names by piping mcycles output into diyone with options `-norm -num false`:

```
% mkdir src
% mcycles @all | diyone -arch X86 -norm -num false -o src
Generator produced 23 tests
% ls src
2+2W.litmus  @all         R.litmus    WRC.litmus     WRW+WR.litmus  Z6.2.litmus
3.2W.litmus  ISA2.litmus  RWC.litmus  WRR+2W.litmus  WWC.litmus     Z6.3.litmus
3.LB.litmus  LB.litmus    SB.litmus   WRW+2W.litmus  Z6.0.litmus    Z6.4.litmus
3.SB.litmus  MP.litmus    S.litmus    W+RWC.litmus   Z6.1.litmus    Z6.5.litmus
```

Alternatively, one may instruct classify to produce output for diyone. In that case one should pass option `-diyone` to classify so as to instruct it to produce output that is parsable by diyone:

```
% rm -rf src && mkdir src
% mcycles @all | classify -arch X86 -diyone | diyone -arch X86 -o src
Generator produced 23 tests
% ls src
2+2W.litmus  @all         R.litmus    WRC.litmus     WRW+WR.litmus  Z6.2.litmus
3.2W.litmus  ISA2.litmus  RWC.litmus  WRR+2W.litmus  WWC.litmus     Z6.3.litmus
3.LB.litmus  LB.litmus    SB.litmus   WRW+2W.litmus  Z6.0.litmus    Z6.4.litmus
3.SB.litmus  MP.litmus    S.litmus    W+RWC.litmus   Z6.1.litmus    Z6.5.litmus
```

11.1 Usage of mcycles

The tool mcycles has no options and takes litmus source files or index files as arguments. It outputs a list of lines to standard output. Each line starts with a test name, suffixed by ":", then the cycle of the named test. Notice that this format is the input format to diyone in its second operating mode — see Sec. 10.3.

It is important to notice that, for mcycles to extract cycles, those must be present as meta-information in source files. In practice, this means that mcycles operates normally on sources produced by diyone, diycross and diy. Moreover only one instance of a given cycle will be output.

11.2 Usage of classify

The tool classify reads its standard input, interpreting is as a list of cycles in the output format of mcycles. It normalises and classifies those cycles. The tool classify accepts the following documented options:

`-arch (X86|PPC|ARM)` Set architecture. Default is `PPC`. ARM support is experimental.

`-u` Instruct classify to fail when two tests have the same normalised name. Otherwise classify will output one line per test, regardless of duplicate names.

`-diyone` Output a normalised list of names and cycles, which is legal input for diyone.

Part III
Simulating memory models with herd

The tool herd is a memory model simulator. Users may write simple, single events, axiomatic models of their own and run litmus tests on top of their model. The herd distribution already includes some models.

The authors of herd are Jade Alglave and Luc Maranget.

12 Writing simple models

12.1 Sequential consistency

The simulator herd accepts models written in text files. For instance here is `sc.cat`, the definition of the sequentially consistent (SC) model:

```
SC
(* Sequential consistency *)
acyclic po | fr | rf | co
```

The model above illustrate two features of model definitions:

1. The computation of new relations from other relations, and their binding to a name with the `let` construct. Here, a new relation `com` is the union "`|`" of the three pre-defined communication relations.

2. The peformance of some checks. Here the relation "`po|com`" (*i.e.* the union of program order `po` and of communication relations) is required to be acyclic.

We postpone the discussion of the `show` instruction, see Sec. 13.

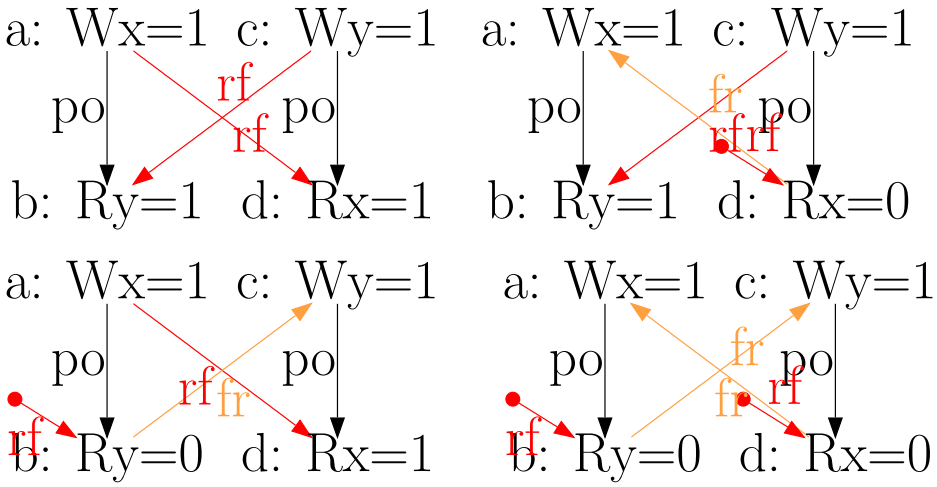
One can then run some litmus test, for instance **SB** (for *Store Buffering*, see also Sec. 1.1), on top of the SC model:

```
% herd -model ./sc.cat SB.litmus
Test SB Allowed
States 3
0:EAX=0; 1:EAX=1;
0:EAX=1; 1:EAX=0;
0:EAX=1; 1:EAX=1;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (0:EAX=0 /\ 1:EAX=0)
Observation SB Never 0 3
Hash=7dbd6b8e6dd4abc2ef3d48b0376fb2e3
```

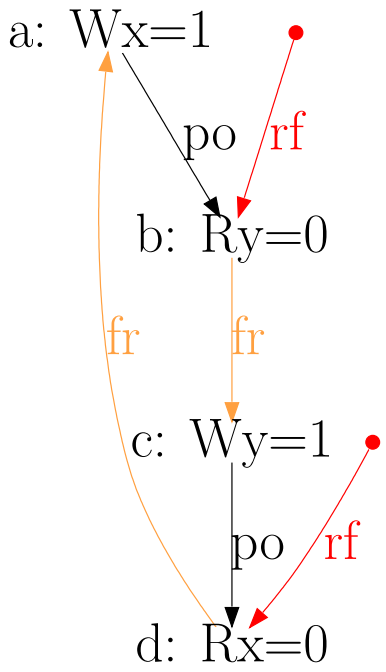
The output of herd mainly consists in the list of final states that are allowed by the simulated model. Additional output relates to the test condition. One sees that the test condition does not validate on top of SC, as "`No`" appears just after the list of final states and as there is no "Positive" witness. Namely, the condition "`exists (0:EAX=0 /\ 1:EAX=0)`" reflects a non-SC behaviour, see Sec. 5.2.

The simulator herd works by generating all executions of a given test. By "execution" we mean a choice of events, program order po, read-from rf, and coherence orders co[6]. In the case of the **SB** example, we get the following four executions:

[6]The last communication relation from-read fr, derives from rf and co. A read event r is fr-before a write event w when r takes its value from a write w_0 that is co-before w.



Indeed, there is no choice for the program order po, as there are no conditional jumps in this example; and no choice for the coherence orders co either, as there is only one store per location, which must be co-after the initial stores (not pictured for clarity). Then, there are two read events from locations x and y respectively, which take their values either from the initial stores or from the stores in program. As a result, there are four possible executions. The model `sc.cat` gets executed on each of the four (candidate) executions. The three first executions are accepted and the last one is rejected, as it presents a cycle in $\mathsf{po} \cup \mathsf{fr}$. The following diagram pictures the ghb relation. The cycle is obvious:



12.2 Total Store Order (TSO)

However, the non-SC execution shows up on x86 machines, whose memory model is TSO. As TSO relaxes the write-to-read order, we attempt to write a TSO model `tso-00.cat`, by simply removing write-to-read pairs from the acyclicity check:

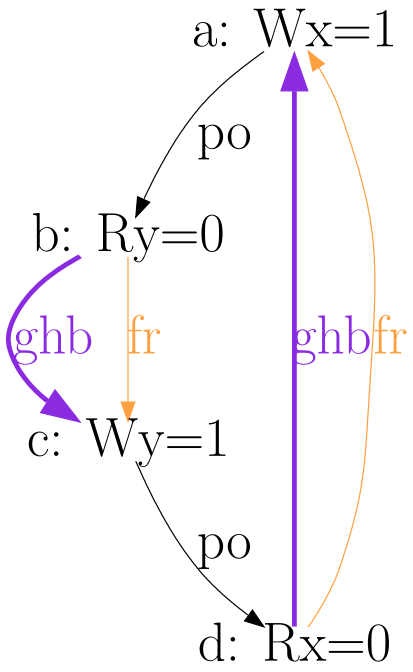
```
"A first attempt for TSO"

(* Communication relations that order events*)
let com-tso = rf | co | fr
(* Program order that orders events *)
let po-tso = WW(po) | RM(po)

(* TSP global-happens-before *)
let ghb = po-tso | com-tso
acyclic ghb
show ghb
```

This model illustrates another feature of model definitions: filters such as `WW` and `RM` restrict their argument by selecting some sort of events. As `W` stands for write events, `R` for read events and `M` for all memory events, the effect of `let po-tso = WW(po)|RM(po)` is to define `po-tso` as the program order minus write-to-read pairs.

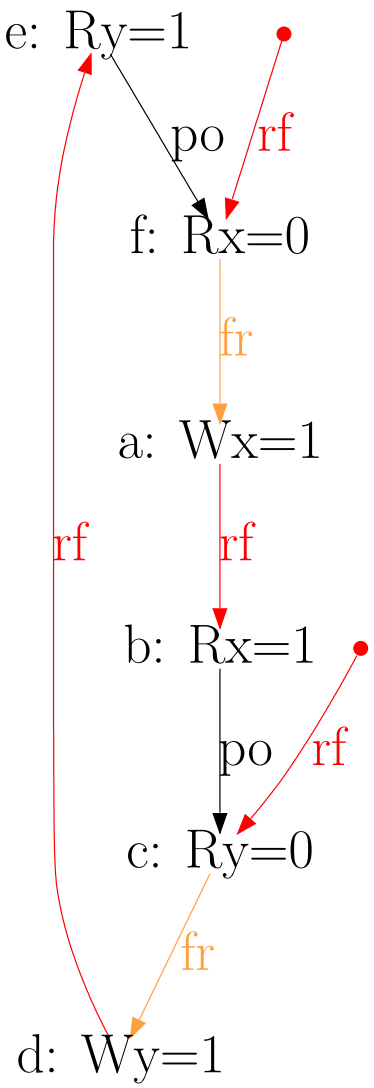
We run **SB** on top of the tentative TSO model:

```
% herd -model tso-00.cat SB.litmus
Test SB Allowed
States 4
0:EAX=0; 1:EAX=0;
0:EAX=0; 1:EAX=1;
0:EAX=1; 1:EAX=0;
0:EAX=1; 1:EAX=1;
Ok
Witnesses
Positive: 1 Negative: 3
...
```

The non-SC behaviour is now accepted, as write-to-read po-pairs do not participate to the acyclicity check any more. In effect, this allows the last execution above, as ghb (*i.e.* po-tso $\cup$ com-tso) is acyclic.



However, our model `tso-00.cat` is flawed: it is still to strict, forbidding some behaviours that the TSO model should accept. Consider the test **SB+rfi-pos**, which is test **STFW-PPC** for X86 from Sec. 1.3 with a normalised name (see Sec. 10.1). This test targets the following execution:



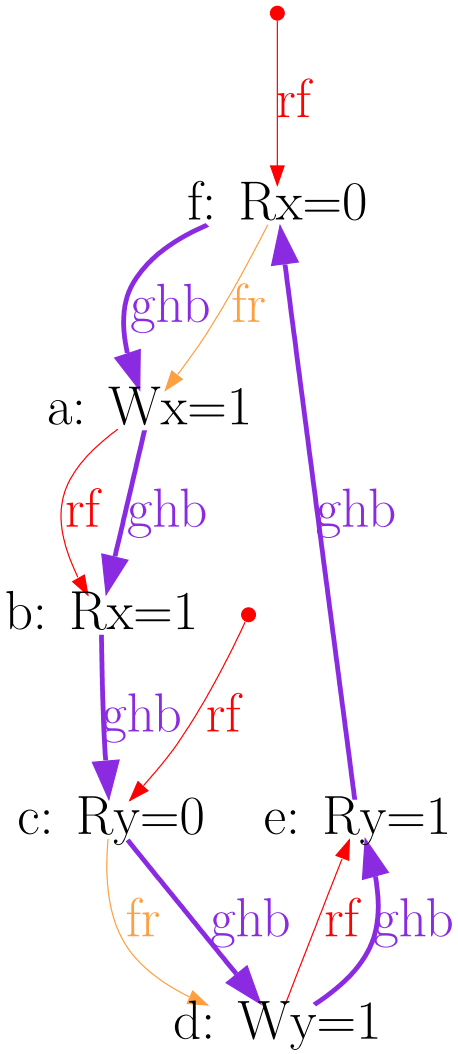
Namely the test condition `exists (0:EAX=1 /\ 0:EBX=0 /\ 1:EAX=1 /\ 1:EBX=0)` specifies that Thread 0 writes 1 into location x, reads the value 1 from the location x (possibly by store forwarding) and then reads the value 0 from the location y; while Thread 1 writes 1 into y, reads 1 from y and then reads 0 from x. Hence, this test derives from the previour **SB** by adding loads in the middle, those loads being satisfied from local stores. As can be seen by running the test on top of the `tso-00.cat` model, the target execution is forbidden:

```
% herd -model tso-00.cat SB+rfi-pos.litmus
Test SB+rfi-pos Allowed
States 15
0:EAX=0; 0:EBX=0; 1:EAX=0; 1:EBX=0;
...
0:EAX=1; 0:EBX=1; 1:EAX=1; 1:EBX=1;
No
Witnesses
Positive: 0 Negative: 15
..
```

However, running the test with litmus demonstrates that the behaviour is observed on some X86 machine:

```
% arch
x86_64
% litmus -mach x86 SB+rfi-pos.litmus
...
Test SB+rfi-pos Allowed
Histogram (4 states)
11589 *>0:EAX=1; 0:EBX=0; 1:EAX=1; 1:EBX=0;
3993715:>0:EAX=1; 0:EBX=1; 1:EAX=1; 1:EBX=0;
3994308:>0:EAX=1; 0:EBX=0; 1:EAX=1; 1:EBX=1;
388   :>0:EAX=1; 0:EBX=1; 1:EAX=1; 1:EBX=1;
Ok

Witnesses
Positive: 11589, Negative: 7988411
Condition exists (0:EAX=1 /\ 0:EBX=0 /\ 1:EAX=1 /\ 1:EBX=0) is validated
...
```

As a conclusion, our tentative TSO model is too strong. The following diagram pictures its ghb relation:



One easily sees that ghb is cyclic, wheras it should not. Namely, the internal read-from relation rfi does not create global order in the TSO model. Hence, rfi is not included in ghb. We rephrase our tentative TSO model, resulting into the new model `tso-01.cat`:

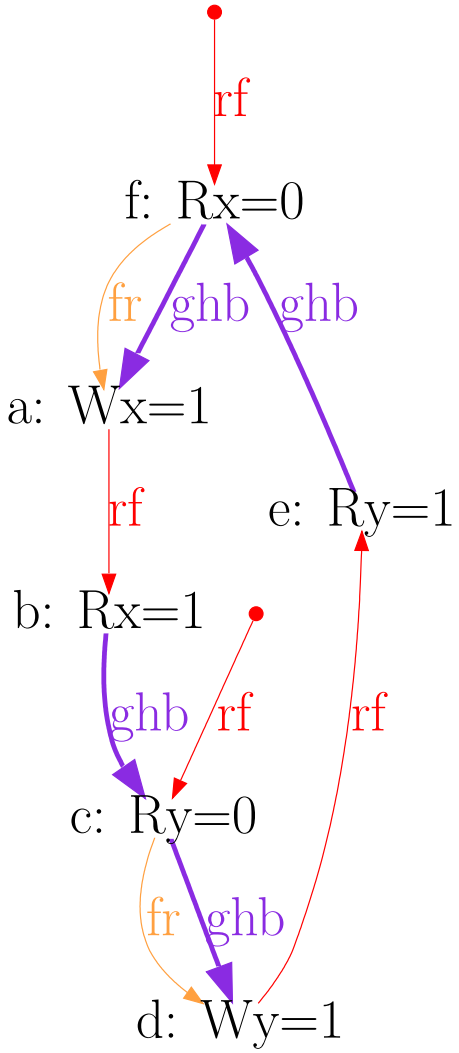
```
"A second attempt for TSO"

(* Communication relations that order events*)
let com-tso = rfe | co | fr
(* Program order that orders events *)
let po-tso = WW(po) | RM(po)

(* TSP global-happens-before *)
let ghb = po-tso | com-tso
acyclic ghb
show ghb
```

As can be observed above rfi (internal read-from) is no longer included in ghb. However, rfe (external read-from) still is.

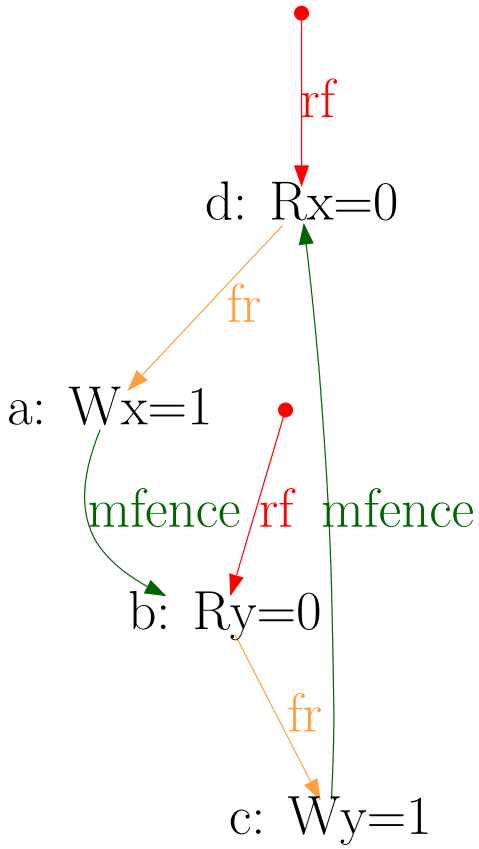
As intended, this new tentative TSO model allows the behaviour of test **SB+rfi-pos**:

```
%  herd -model tso-01.cat SB+rfi-pos.litmus
Test SB+rfi-pos Allowed
States 16
...
0:EAX=1; 0:EBX=1; 1:EAX=1; 1:EBX=0;
...
Ok
Witnesses
Positive: 1 Negative: 15
...
```

And indeed, the global-happens-before relation is no-longer cyclic:



We are not done yet, as our model is too weak in two aspects. First, it has no semantics for fences. As a result the test **SB+mfences** is allowed, whereas it should be forbidden, as this is the very purpose of the fence `mfence`.

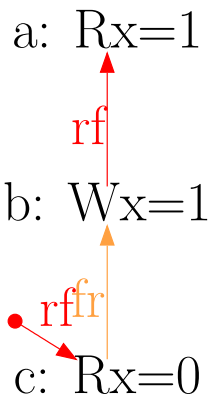


One easily solves this issue by adding the `mfence` relation to the definition of `po-tso`:

```
let po-tso = WW(po) | RM(po) | mfence
```

But the resulting model is still too weak, as it allows some behaviours that any model must reject for the sake of single thread correctness. The following test **CoRWR** illustrates the issue:

```
X86 CoRWR
{ }
 P0           ;
 MOV EAX,[x] ;
 MOV [x],$1  ;
 MOV EBX,[x] ;
exists (0:EAX=1 /\ 0:EBX=0)
```



The TSO check "`acyclic po-tso|com-tso`" does not suffice to reject two absurd behaviours pictured in the execution diagram above: (1) the read a is allowed to read from the po-after write b, as rfi is not included in com-tso; and (2) the read c is allowed to read the initial value of location x although this (unpictured) write is co-before the write b, as `WR(po)` is not in po-tso.

For any model, we rule out those very untimely behaviours by the so-called UNIPROC check that states that executions projected on events that access one variable only are SC. In practice, having defined `po-loc` as `po` restricted to events that touch the same address, we further require the acyclicity of the relation `po-loc|fr|rf|co`. In the TSO case, the UNIPROC check can be somehow simplified by considering only the cycles in `po-loc|fr|rf|co` that are not already rejected by the main check of the model. This amounts to design specific checks for the two relations that are not global in TSO: `rfi` and `WR(po)`. Doing so, we finally produce a correct model for TSO `tso-02.cat`:

```
"A third attempt for TSO"

(* Uniproc check specialized for TSO *)
irreflexive RW(po-loc);rfi as uniprocRW
irreflexive WR(po-loc);fri as uniprocWR

(* Communication relations that order events*)
let com-tso = rfe | co | fr
(* Program order that orders events *)
let po-tso = WW(po) | RM(po) | mfence

(* TSP global-happens-before *)
let ghb = po-tso | com-tso
show mfence ghb
acyclic ghb as tso
```

This last model illustrates other features of herd: herd may also performs irreflexivity checks with the keyword "`irreflexive`"; and the checks can be given names by suffixing them with "`as` *name*". This last feature will be used in Sec. 13.2

13 Producing pictures of executions

The simulator herd can be instructed to produce pictures of executions. Those pictures are instrumental in understanding and debugging models. It is important to understand that herd does not produce pictures

by default. To get pictures one must instruct herd to produce pictures of some executions with the `-show` option. This option accepts specific keywords, its default being "`none`", instructing herd not to produce any picture.

A frequentlty used keyword is "`prop`" that means "show the executions that validate the proposition in the final condition". Namely, the final condition in litmus test is a quantified boolean proposition as for instance "`exists (0:EAX=0 /\ 1:EAX=0)`" at the end of test **SB**.

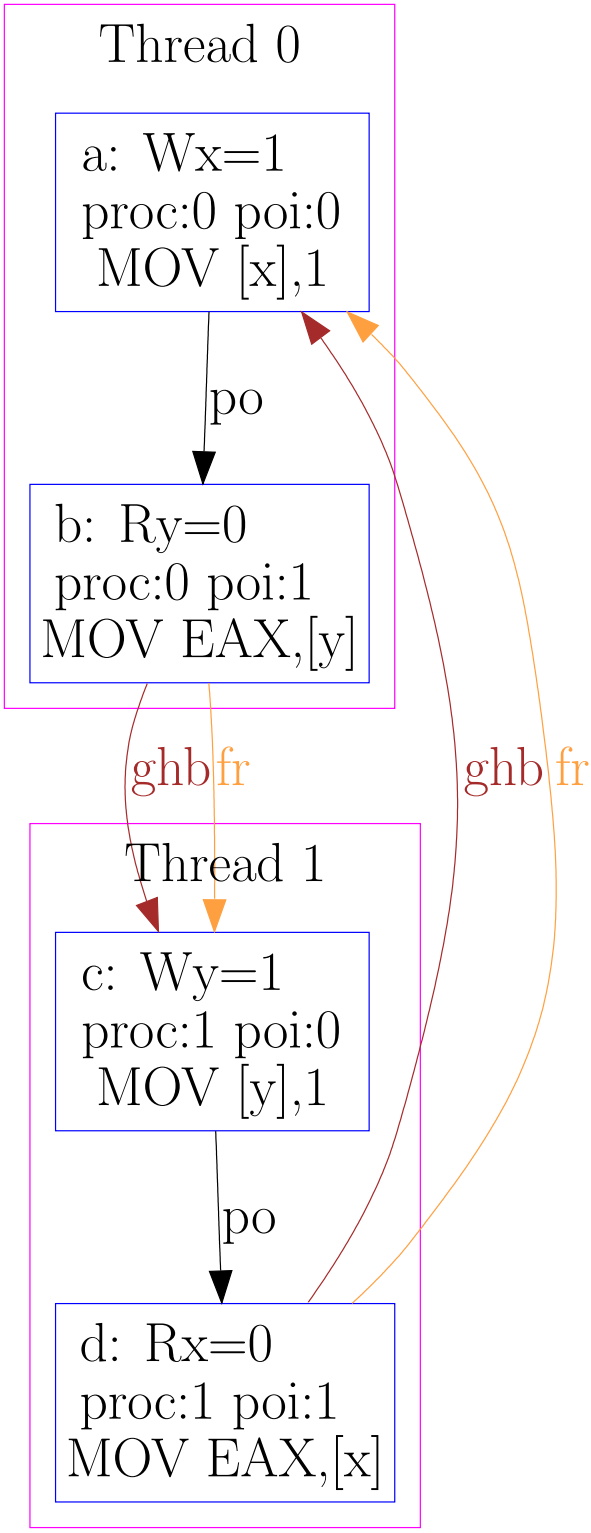
But this is not enough, users also have to specify what to do with the picture: save it in file in the DOT format of the graphviz graph visualization software, or display the image,[7] or both. One instructs herd to save images with the `-o` *dirname* option, where *dirname* is the name of a directory, which must exists. Then, when processing the file *name*`.litmus`, herd will create a file *name*`.dot` into the directory *dirname*. For displaying images, one uses the `-gv` option.

As an example, so as to display the image of the non-SC behaviour of **SB**, one should invoke herd as:

```
% herd -model tso-02.cat -show prop -gv SB.litmus
```

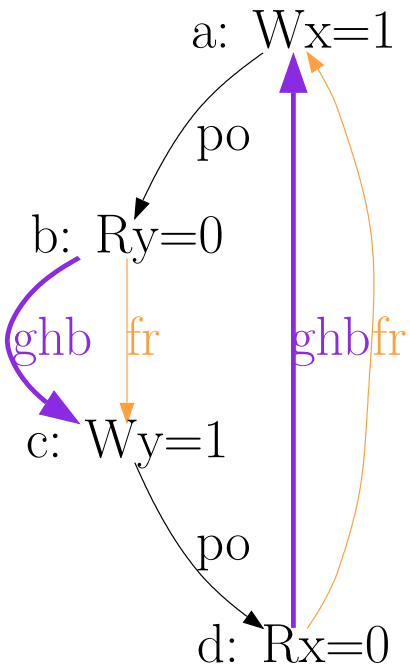
As a result, users should see a window popping and displaying this image:

[7] This option requires the Postscript visualiser gv.



Test SB, Generic(A third attempt for TSO)

Figure 3: The non-SC behaviour of **SB** is allowed by TSO



Notice that we got the PNG version of this image as follows:

```
% herd -model tso-02.cat -show prop -o /tmp SB.litmus
% dot -Tpng /tmp/SB.dot -o SB+CLUSTER.png
```

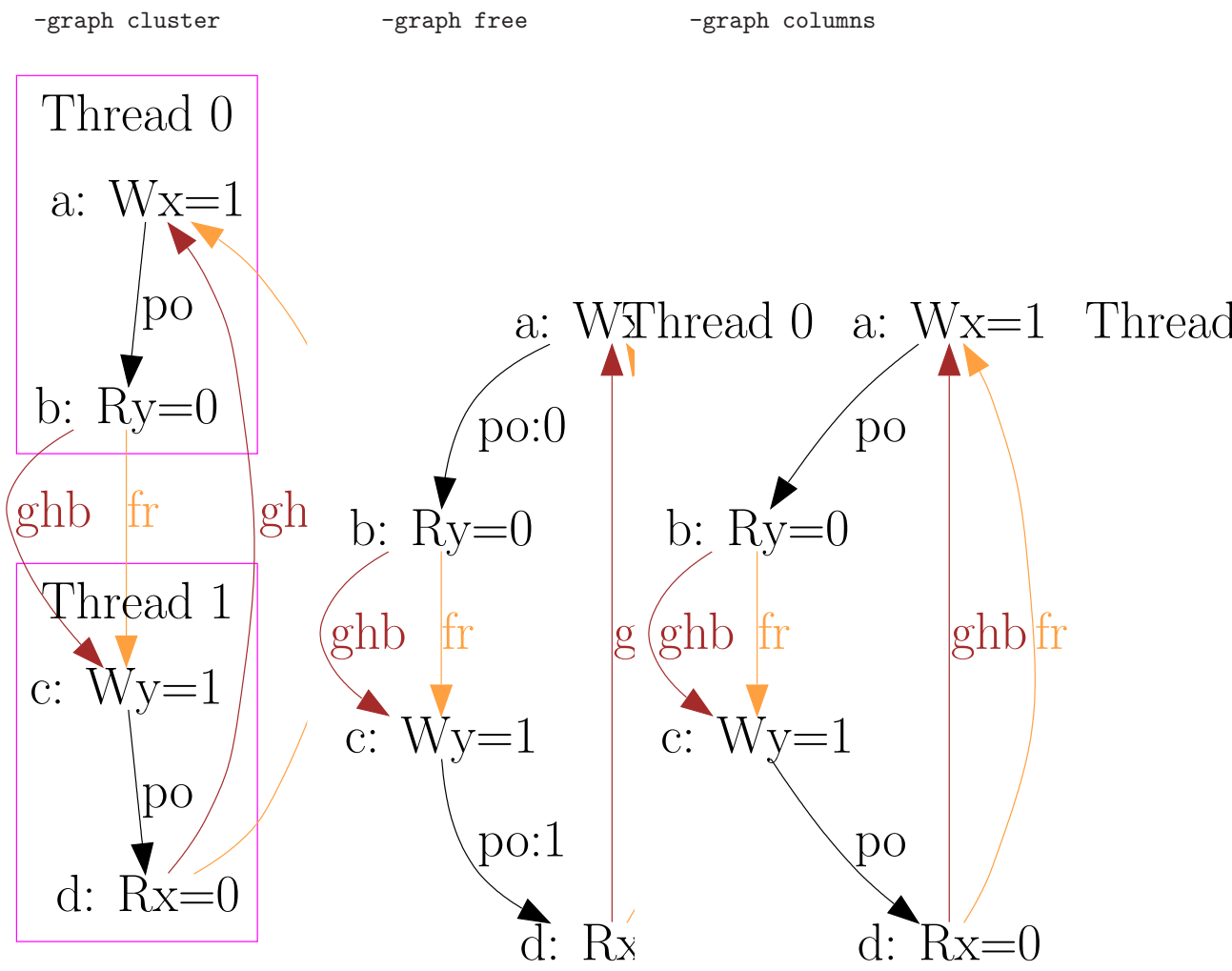
That is, we applied the dot tool from the graphviz package, using the appropriate option to produce a PNG image.

One may observe that there are `ghb` arrows in the diagram. This results from the `show ghb` instruction at the end of the model file `tso-02.cat`.

13.1 Graph modes

The image above much differs from the one in Sec. 12.2 that describes the same execution and that is reproduced in Fig. 3

In effect, herd can produce three styles of pictures, dot clustered pictures, dot free pictures, and neato pictures with explicit placement of the events of one thread as a colum. The style is commanded by the `-graph` option that accepts three possible arguments: `cluster` (default), `free` and `columns`. The following pictures show the effect of graph styles on the **SB** example:



Notice that we used another option `-squished true` that much reduces the information displayed in nodes. Also notice that the first two pictures are formatted by dot, while the rightmost picture is formatted by neato.

One may also observe that the "`-graph columns`" picture does not look exactly like Fig. 3. For instance the ghb arrows are thicker in the figure. There are many parameters to control neato (and dot), many of which are accessible to herd users by the means of appropriate options. We do not intend to describe them all. However, users can reproduce the style of the diagram of this manual using yet another feature of herd: configuration files that contains settings for herd options and that are loaded with the `-conf` *name* option. In this manual we mostly used the `doc.cfg` configuration file. As this file is present in herd distribution, users

can use the diagram style of this manual:

```
% herd -conf doc.cfg ...
```

13.2 Showing forbidden executions

Images are produced or displayed once the model has been executed. As a consequence, forbidden executions won't appear by default. Consider for instance the test **SB+mfences**, where the `mfence` instruction is used to forbid **SB** non-SC execution. Runing herd as

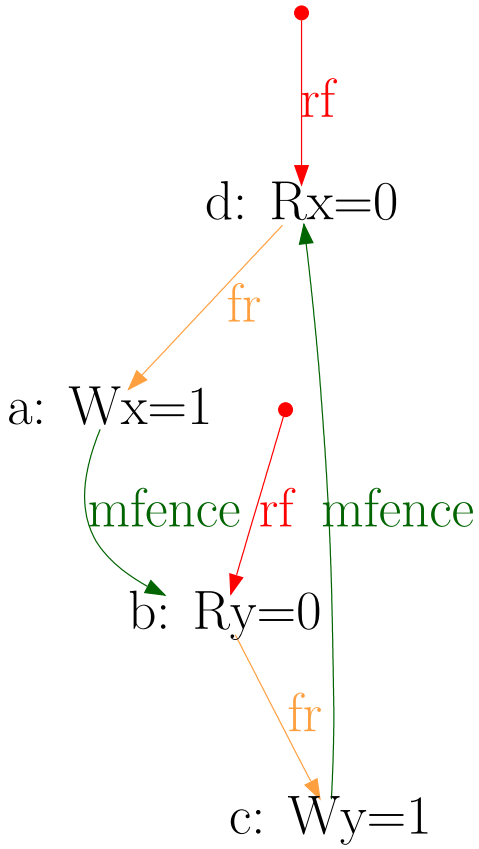
```
% herd -model tso-02.cat -conf doc.cfg -show prop -gv SB+mfences.litmus
```

will produce no picture, as the TSO model forbids the target execution of SB+mfences.

To get a picture, we can run SB+mfences on top of the mininal model, a pre-defined model that allows all executions:

```
% herd -model minimal -conf doc.cfg -show prop -gv SB+mfences.litmus
```

And we get the picture:

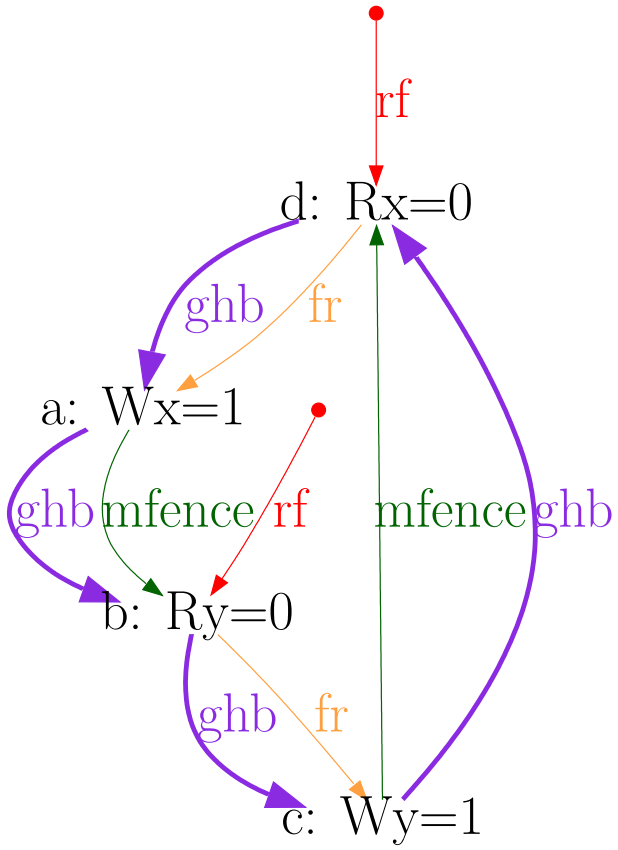


It is worth mentioning again that although the minimal model allows all executions, the final condition selects the displayed picture, as we have specified the `-show prop` option.

The picture above shows `mfence` arrows, as all fence relations are displayed by the minimal model. However, it does not show the `ghb` relation, as the minimal model knows nothing of it. To display `ghb` we could write another model file that would be just as `tso-02.cat`, with checks erased. The simulator herd provides a simpler technique: one can instruct herd to ignore either all checks (`-through invalid`), or a selection of checks (`-skipcheck` $name_1 \ldots$ $name_n$). Thus, either of the following two commands

```
% herd -through invalid -model tso-02.cat -conf doc.cfg -show prop -gv SB+mfences.litmus
% herd -skipcheck tso -model tso-02.cat -conf doc.cfg -show prop -gv SB+mfences.litmus
```

will produce the picture we wish:

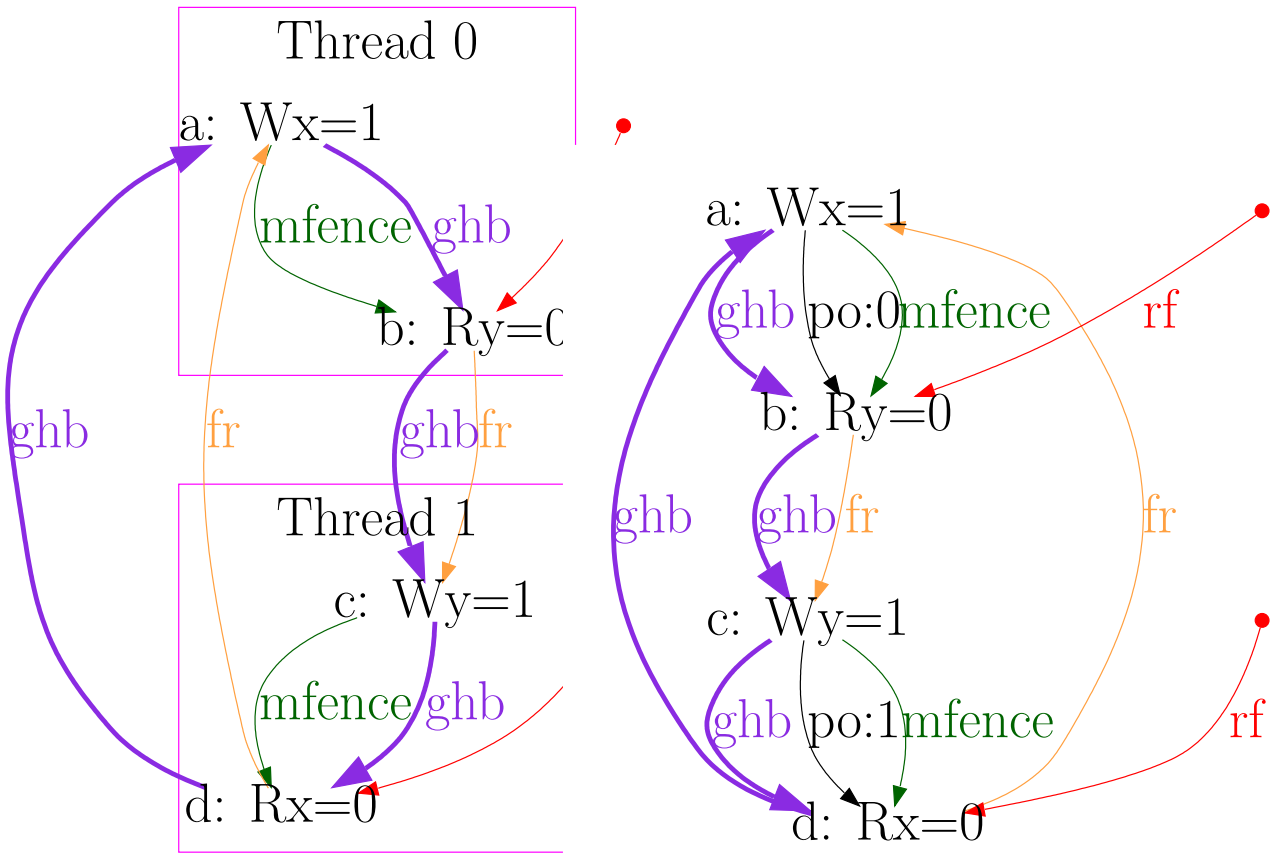


Notice that `mfence` and `ghb` are displayed because of the instruction "`show mfence ghb`" (fence relation are not shown by default); while `-skipcheck tso` works because the `tso-02.cat` model names its main check with "`as tso`".

The image above is barely readable. For such graphs with many relations, the `cluster` and `free` modes are worth a try. The commands:

```
% herd -skipcheck tso -model tso-02.cat -conf doc.cfg -show prop -graph cluster -gv SB+mfences.litmus
% herd -skipcheck tso -model tso-02.cat -conf doc.cfg -show prop -graph free -gv SB+mfences.litmus
```

will produce the images:



Namely, command line options are scanned left-to-right, so that most of the settings of `doc.cfg` are kept[8] (for instance thick `ghb` arrows), while the graph mode is overriden.

14 Model definitions

We describe our langage for defining models. The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter font (`like this`). Non-terminal symbols are set in italic font (*like that*). An unformatted vertical bar ... | ... denotes alternative. Square brackets [...] denote optional components. Curly brackets {...} denotes zero, one or several repetitions of the enclosed components.

Model source files may contain comments of the OCaml type (`(*...*)`, can be nested), or line comments starting with "`#`" and running until end of line.

14.1 Identifiers

letter ::= `a...z` | `A...Z`

digit ::= `0...9`

id ::= *letter* {*letter* | *digit* | `_` | `.` | `-`}

[8] The setting of `showthread` is also changed, by the omitted `-showthread true` command line option

Identifiers are rather standard: they are a sequence of letters, digits, "_" (the underscore character), "." (the dot character) and "-" (the minus character), starting with a letter. Using the minus character inside identifiers may look a bit surprising. We did so as to allow identifiers such as `po-loc`.

At startup, pre-defined identifiers are bound to relations between memory events. Those relations describe a candidate execution. Executing the model means allowing or forbiding that candidate execution.

A first pre-defined identifier is `id`, the identity. Other pre-defined identifiers are the program order `po` and its refinements:

identifier	name	description
`po`	program order	instruction order lifted to events
`po-loc`	`po` restricted to the same address	events are in `po` and touch the same address
`addr`	address dependency	the address of the second event depends on the value loaded by the first (read) event
`data`	data dependency	the value stored by the second (write) event depends on the value loaded by the first (read) event
`ctrl`	control dependency	the second event is in a branch controled by the value loaded by the commfirst (read) event
`ctrlisync/ctrlisb`	control dependency + `isync/isb`	the branch additionally contains a `isync/isb` barrier before the second event

Other pre-defined relations denote the presence of a specific fence (or barrier) in-between two events, those are `mfence`, `sfence`, `lfence` (X86); `sync`, `lwsync`, `lwsync`, `isync` (Power); and `dsb`, `dmb`, `dsb.st`, `dmb.st`, `isb` (ARM).

Finally, pre-defined identifiers are bound the communication relations:

identifier	name	description
`rf`	read-from	links a write w to a read r taking its value from w
`co`	coherence	total order over writes to the same address
`fr`	from-read	links a read r to a write w' `co`-after the write w from which r takes its value
`rfi`, `fri`, `coi`	internal communications	communication between events of the same thread
`rfe`, `fre`, `coe`	external communications	communication between events of different threads

14.2 Expressions

Expressions are evaluated by herd, yielding a relation over memory events as a value. A relation can be seen as a set of pairs of memory events.

```
expr       ::= 0
             | id
             | expr * | expr + | expr ?
             | expr | expr | expr ; expr | expr \ expr | expr & expr
             | selector ( expr )
             | id ( args )
             | fun ( params ) -> expr
             | let binding {and binding} in expr
             | let rec valbinding {and valbinding} in expr
             | ( expr )
selector   ::= MM | MR | RM | MW | WM | RR | WR | RW | WW
             | AA | AP | PA | PP
args       ::= ϵ
             | expr {, expr}
params     ::= ϵ
             | id {, id}
binding    ::= valbinding | funbinding
valbinding ::= id = expr
funbinding ::= id ( params ) = expr
```

Simple expressions

Simple expressions are the empty relation (keyword 0) and identifiers *id*. Identifiers are bound to values, either before the execution (see pre-defined identifiers in Sec. 14.1), or by the model itself.

Operator expressions

The transitive and reflexive-transitive closure of an expression are performed by the postfix operators + and * . The construct *expr* ? (option) evaluates to the union of *expr* value and of the identity relation.

Infix operators are | (union), ; (sequence), & (intersection) and \ (set difference). The precedence of operators is as follows: postfix operators bind tighter than infix operators. Infix operators from higher precedence to lower precedence are: & , \ , ; and | . All infix operators are right-associative, except set difference which is left-associative.

For the record, given two relations r_1 and r_2, the sequence $r_1; r_2$ is defined as $\{(x, y) \mid \exists z, (x, z) \in r_1 \wedge (z, y) \in r_2\}$.

Selectors

Selectors are filters that operate on relations. The value of *selector* (*expr*) is some subrelation of the value of *expr*. Which subrelation depends upon *selector*. Selectors are two-letters keywords with the first letter operating on the first component of pairs and the second letter operating on the second component of pairs.

More precisely, we define the semantics of selector letters as predicates over memory events: M always yield true, R yields true on read events, W yields true on write events, A yields true on atomic events

(produced by X86 locked instructions, or ARM/Power load reserve and store conditional instructions), and `P` yields true on ordinary (plain) events. Then, we define:

$$XY(r) = \{(x, y) \in r \mid X(x) \wedge Y(y)\}$$

For instance, the expression `RW(po)` yields the read-to-write pairs in program order.

Function calls

Functions calls are written *id* `(` $expr_1$ `,` ... `,` $expr_n$ `)`. That is, functions have an arity (n above, which can be zero) and arguments are given as a comma separated list of expressions. Our language have call-by-value semantics. That is, the effective parameters $expr_1$ `,` ... `,` $expr_n$ are evaluated before being bound to formal parameters.

Functions

Functions are first class values, as reflected by the anonymous function construct `fun (` *params* `) ->` *expr*. Function arity is defined by the length of its formal parameter list *params*, which can be empty. Notice that functions have the usual static scoping semantics: variables that appear free in function bodies (*expr* above) are bound to the value of such free variable at function creation time.

Local bindings

The local binding construct `let` [`rec`]*bindings* `in` *expr* binds the names defined by *bindings* for evaluating the expression *expr* .

Both non-recursive and recursive bindings are allowed. The function binding *id* `(` *params* `)` `=` *expr* is syntactic sugar for *id* `=` `fun (` *params* `) ->` *expr*. It is allowed in non-recursive bindings only as there is little point in defining recursive function in our setting.

In the following we only consider value bindings *id* `=` *expr*. The construct

$$\texttt{let}\ id_1 = expr_1\ \texttt{and} \ldots \texttt{and}\ id_n = expr_n\quad \texttt{in}\quad expr$$

evaluates $expr_1, \ldots, expr_n$, and binds the names $id_1, \ldots, id_n$ to the resulting values.

The construct

$$\texttt{let rec}\ id_1 = expr_1\ \texttt{and} \ldots \texttt{and}\ id_n = expr_n\quad \texttt{in}\quad expr$$

computes the least fixpoint of the equations $id_1 = expr_1, \ldots, id_n = expr_n$. It then binds the names $id_1, \ldots, id_n$ to the resulting values. The least fixpoint computation applies only to relations, using inclusion for ordering. Notice that `let rec` *id* `=` `fun (` *params* `) ->` *expr* `in` ... is legal syntax. Such a definition will yield a runtime error.

Parenthesized expressions

The expression `(` *expr* `)` has the same value as *expr*.

14.3 Instructions

Instruction are executed for their effect. There are three kind of effects: adding new bindings, checking a condition, and specifying relations that are shown in pictures.

instruction ::= `let` *binding* {`and` *binding*}
| `let rec` *valbinding* {`and` *valbinding*}
| *check expr* [`as` *id*]
| `show` *expr* `as` *id*
| `show` *id* {`,` *id*}
| `unshow` *id* {`,` *id*}

check ::= `acyclic` | `irreflexive` | `empty`

Bindings

The `let` and `let rec` constructs bind value names for the rest of model execution. See the subsection on bindings in Section 14.2 for additional information on the syntax and semantics of bindings.

Recursive definitions computes fixpoints of relations. For instance, the following fragment computes the transitive closure of all communication relations:

```
let com = rf | co | fr
let rec complus = com | (complus ; complus)
```

Notice that the instruction `let complus = (rf|co|fr)+` is equivalent.

There are no recursive functions, as those would not be very useful in our limited language. Nevertheless, one may for instance write a generic transitive closure function by using a local recursive binding:

```
let tr(r) = let rec t = r | (t;t) in t
let complus = tr(rf|co|fr)
```

Again notice that the instruction `let complus = (rf|co|fr)+` is equivalent.

Checks

The construct

check expr

evaluates *expr* and applies *check*. If the check succeeds, that is if the relation is acyclic, irreflexive or empty; depending on *check* being `acyclic`, `irreflexive` or `empty`, execution goes on. Otherwise, execution stops.

The performance of a check can optionally be named by appending `as` *id* after it. The feature permits not to perform some checks at user's will, thanks to the `-skipcheck` *id* command line option.

Show (and unshow) directives

The constructs:

`show` *id* {`,` *id*} and `unshow` *id* {`,` *id*}

take (non-empty, comma separated) lists of identifiers as arguments. The `show` construct adds the present values of identifiers for being shown in pictures. The `unshow` construct removes the identifiers from shown relations.

The more sophisticated construct

`show` *expr* `as` *id*

evaluates *expr* to a relation, which will be shown in pictures with label *id*. Hence `show`*id* can be viewed as a shorthand for `show`*id* `as`*id*

14.4 Models

$$\begin{array}{rcl} \textit{model} & ::= & \textit{model-comment}\ \{\textit{instruction}\} \\ \textit{model-comment} & ::= & \textit{id} \\ & | & \textit{string} \end{array}$$

A model is a a list of instruction preceded by a small comment, which can be either a name that follows herd conventions for identifiers, or a string enclosed in double quotes "".

Models operate on candidate executions (see Sec. 14.1), instructions are executed in sequence, until one instruction stops, or until the end of the instruction list. In that latter case, the model accepts the execution. The accepted execution is then passed over to the rest of herd engine, in order to collect final states of locations and to display pictures.

15 Usage of herd

Arguments

The command herd handles its arguments like litmus. That is, herd interprets its argument as file names. Those files are either a single litmus test when having extension `.litmus`, or a list of file names when prefixed by `@`.

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.

`-q` Be quiet, suppress any diagnostic message.

`-conf <name>` Read configuration file `name`. Configuration files have a very simple syntax: a line "*opt arg*" has the same effect as the command-line option "`-`*opt arg*".

`-o <dest>` Output files into directory `<dest>`. Notice that `<dest>` must exist. At the moment herd may output one `.dot` file per processed test: the file for test *base*`.litmus` is named *base*`.dot`. By default herd does not generate `.dot` files.

`-suffix <suf>` Change the name of `.dot` files into *base suff* `.dot`. Useful when several `.dot` files derive from the same test. Default is the empty string (no suffix).

`-gv` Fork the gv Postscript viewer to display execution diagrams.

`-dumpes <bool>` Dump genererated event structures and exit. Default is `false`. Event structures will be dumped in a `.dot` file whose name is determined as usual — See options `-o` and `-suffix` above. Optionally the event structures can be displayed with the `-gv` option.

`-unroll <int>` The setting `-unroll` n performs backwards jumps n times. This is a workaround for one of herd main limitation: herd does not really handle loops. Default is 2.

`-hexa <bool>` Print numbers in hexadecimal. Default is `false` (numbers are printed in decimal).

Engine control The main purpose of herd is to run tests on top of memory models. For a given test, herd performs a three stage process:

1. Generate candidate executions.
2. For each exection, run the model. The model may reject or accept the execution.
3. For each candidate that the model accepts, record observed locations and a diagram of the execution (if instructed so).

We now describe options that control those three stages.

`-model (herd|cav12|minimal|uniproc|x86tso|<filename>.cat)` Select model, this option accept one tag or one file name with extension `.cat`. Tags instruct herd to select an internal model, while file names are read for a model definition. By default, herd run tests on top of an advanced (`herd`) model for Power and ARM, and on top of `x86tso` for X86. Other (documented) model tags are:

- `cav12`, the model of [4] (Power);
- `minimal`, the minimal model that allows all executions;
- `uniproc`, the uniproc model that checks single-thread correctness.

In fact, herd accepts potentially infinitely many models, as models can given in text files in an ad-hoc language described in Sec. 14. The herd distribution includes several such models: `herd.cat`, `minimal.cat`, `uniproc.cat` and `x86tso.cat` are text file versions of the homonymous internal models, but may produce pictures that show different relations. Model files are searched according to the same rules as configuration files.

`-through (all|invalid|none)` Let additional executions reach the final stage of herd engine. This option permits users to generate pictures of forbidden executions, which are otherwise rejected at an early stage of herd engine — see Sec. 13.2. Namely, the default "`none`" let only valid (according to the active model) executions through. The behaviour of this option differs between internal and text file models:

- For internal models: the tag `all` let all executions go through; while the tag `invalid` will reject executions that violate uniproc, while letting other forbidden execution go through.
- Text file models: the tags `all` and `invalid` let all executions go through. For such models, a more precise control over executions that reach herd final phase can be achieved with the option `-skipcheck` — see next option.

Default is `none`.

`-skipcheck <`$name_1$`,...,`$name_n$`>` This option applies to text file models. It instructs herd to ignore the outcomes of the given checks. For the option to operate, checks must be named in the model file with the `as` *name* construct – see Sec. 14.3. Notice that the arguments to `-skipcheck` options cumulate. That is, "`-skipcheck` $name_1$ `-skipcheck` $name_2$" acts like "`-skipcheck` $name_1$`,`$name_2$".

`-strictskip <bool>` Setting this option (`-strictskip true`), will change the behaviour of the previous option `-skipcheck`: it will let executions go through when the skipped checks yield false and the unskipped checks yield true. This option comes handy when one want to observe the executions that fail one (or several) checks while passing others. Default is `false`.

`-optace <bool>` Optimise the axiomatic candidate execution stage. When enabled by `-optace true`, herd does not generate candidate executions that fail the uniproc test. The default is "`true`" for internal models (except the minimal model), and "`false`" for text file models. Notice that `-model uniproc.cat` and `-model minimal.cat -optace true` should yield identical results, the second being faster. Setting `-optace true` can lower the execution time significantly, but one should pay attention not to design models that forget the uniproc condition.

`-show (prop|neg|all|cond|wit|none)` Select execution diagrams for picture display and generation. Execution diagrams are shown according to the final condition of test. The final condition is a quantified boolean proposition `exists` p, `~exists` p, or `forall` p. The semantics of recognised tags is as follows:

- `prop` Picture executions for which p is true.
- `neg` Picture executions for which p is false.
- `all` Picture all executions.
- `cond` Picture executions that validate the condition, *i.e.* p is true for `exists` and `forall`, and false for `~exists`.
- `wit` Picture "*interesting*" executions, *i.e.* p is true for `exists` and `~exists`, and false for `forall`.
- `none` Picture no execution.

Default is `none`.

Discard some observations Those options intentionally omit some of the final states that herd would normally generate.

`-speedcheck (false|true|fast)` When enabled by `-speedcheck true` or `-speedcheck fast`, attempt to settle the test condition. That is, herd will generate a subset of executions (those named "*interesting*" above) in place of all executions. With setting `-speedcheck fast`, herd will additionally stop as soon as a condition `exists` p is validated, and as soon as a condition `~exists` p or `forall` p is invalidated. Default is `false`.

`-nshow <int>` Stop once `<int>` pictures have been collected. Default is to collect all (specified, see option `-show`) pictures.

Control `dot` pictures These options control the content of DOT images.
We first describe options that act at the general level.

`-graph (cluster|free|columns)` Select main mode for graphs. See Sec. 13.1. The default is `cluster`.

`-dotmode (plain|fig)` The setting `-dotmode fig` produces output that includes the proper escape sequence for translating `.dot` files to `.fig` files (*e.g.* with `dot -Tfig...`). Default is `plain`.

`-dotcom (dot|neato|circo)` Select the command that formats graphs displayed by the `-gv` option. The default is `dot` for the `cluster` and `free` graph modes, and `neato` for the `columns` graph mode.

`-showevents (all|mem|noregs)` Control which events are pictured:

- `all` Picture all events.
- `mem` Picture memory events.
- `noregs` Picture all events except register events, *i.e.* memory, fences and branch events.

Default is `noregs`.

`-mono <bool>` The setting `-mono true` commands monochrome pictures. This option acts upon default color selection. Thus, it has no effect on colors given explicitely with the `-edgeattr` option.

`-scale <float>` Global scale factor for graphs in `columns` mode. Default is `1.0`.

`-xscale <float>` Global scale factor for graphs in `columns` mode, x direction. Default is `1.0`.

`-yscale <float>` Global scale factor for graphs in `columns` mode, y direction. Default is `1.0`.

`-showthread <bool>` Show thread numbers in figures. In `cluster` mode where the events of a thread are clustered, thread cluster have a label. In `free` mode `po` edges are suffixed by a thread number. In `columns` mode, columhs have a header node that shows the thread number. Default is `true`.

`-texmacros <bool>` Use latex commands in some text of pictures. If activated (`-showthread true`), thread numbers are shown as `\myth{`n`}`. Assembler instructions are locations in nodes are argument to an `\asm` command. It user responsability to define those commands in their LaTeX documents that include the pictures. Possible definitions are `\newcommand{\myth}[1]{Thread~#1}` and `\newcommand{\asm}[1]{\texttt{#1}}`. Default is `false`.

A few options control picture legends.

`-showlegend <bool>` Add a legend to pictures. By default legends show the test name and a comment from the executed model. This comment is the first item of model syntax — see Sec 14.4. Default is `true`.

`-showkind <bool>` Show test kind in legend. The kind derive from the quantifier of test final condition, kind `Allow` being `exists`, kind `Forbid` being `~exists`, and kind `Require` being `forall`. Default is `false`.

`-shortlegend <bool>` Limit legend to test name. Default is `false`.

A few options control what is shown in nodes and on their sizes, *i.e.* on how events are pictured.

`-squished <bool>` The setting `-squished true` drastically limits the information displayed in graph nodes. This is usually what is wanted in modes `free` and `columns`. Default is `false`.

`-fixedsize <bool>` This setting is meaningfull in `columns` graph mode and for squished nodes. When set by `-fixedsize true` it forces node width to be 65% of the space between columns. This may sometime yield a nice edge routing. Default is `false`

`-extrachars <float>` This setting is meaningful in `columns` graph mode and for squished nodes. When the size of nodes is not fixed (*i.e.* `-fixedsize false` and default), herd computes the width of nodes by counting caracters in node labels and scaling the result by the font size. The setting `-extrachars` v commands adding the value v before scaling. Negative values are of course accepted. Default is `0.0`.

`-showobserved <bool>` Highlight observed memory read events with stars "`*`". A memory read is observed when the value it reads is stored in a register that appears in final states. Default is `false`.

`-brackets <bool>` Show brackets around locations. Default is `false`.

Then we list options that offer some control on which edges are shown. We recall that the main controls over the shown and unshown edges are the `show` and `unshow` directives in model definitions — see Sec. 14.3. However, some edges can be controled only with options (or configuration files) and the `-unshow` option proves convenient.

`-showpo <bool>` Show program order (`po`) edges. Default is `true`. Default is `false`.

`-showinitrf <bool>` Show read-from edges from initial state. Default is `false`.

`-showfinalrf <bool>` Show read-from edges to the final state, *i.e* show the last store to locations. Default is `false`.

`-showfr <bool>` Show from-read edges. Default is `true`.

`-doshow <`$name_1$`,...,`$name_n$`>` Do show edges labelled with $name_1$,...,$name_n$. This setting applies when names are bound in model definition.

`-unshow <`$name_1$`,...,`$name_n$`>` Do not show edges labelled with $name_1$,...,$name_n$. This setting applies at the very last momement and thus cancels any `show` directive in model definition and any `-doshow` command line option.

Other options offer some control over some of the attributes defined in Graphviz software documentation. Notice that the controlled attributes are omitted from DOT files when no setting is present. For instance in the absence of a `-spline <tag>` option, herd will generate no definition for the `splines` attribute thus resorting to DOT tools defaults. Most of the following options accept the `none` argument that restores their default behaviour.

`-splines (spline|true|line|false|polyline|ortho|curved|none)` Define the value of the `splines` attribute. Tags are replicated in output files as the value of the attribute, except for `none`.

`-margin <float|none>` Specifies the `margin` attribute of graphs.

`-pad <float|none>` Specifies the `pad` attribute of graphs.

`-sep <string|none>` Specifies the `sep` attribute of graphs. Notice that the argument is an arbitray string, so as to allow DOT general syntax for this attribute.

`-fontname <string|none>` Specifies the graph fontname attribute.

`-fontsize <int|none>` Specifies the fontsize attribute n of all text in the graph.

`-edgefontsizedelta <int>` option `-edgefontsizedelta` m sets the fontsize attributes of edges to $n+m$, where n is the argument to the `-fontsize` option. Default is `0`. This option has no effect if fontsize is unset.

`-penwidth <float|none>` Specifies the `penwidth` attribute of edges.

`-arrowsize <float|none>` Specifies the `arrowsize` attribute of edges.

`-edgeattr <label,attribute,value>` Give value `value` to attribute `attribute` of all edges labelled `label`. This powerful option permits alternative styles for edges. For instance, the `ghb` edges of the diagrams of this document are thick purple (blueviolet) arrows thanks to the settings: `-edgeattr ghb,color,blueviolet -edgeattr ghb,penwidth,3.0 -edgeattr ghb,arrowsize,1.2`. Notice that the settings performed by the `-edgeattr` option override other settings. This option has no default.

Change input Those options are the same as the ones of litmus — see Sec. 4.

`-names <file>` Run herd only on tests whose names are listed in `<file>`.

`-rename <file>` Change test names.

`-kinds <file>` Change test kinds. This amonts to changing the quantifier of final conditions, with kind `Allow` being `exists`, kind `Forbid` being `~exists` and kind `Require` being `forall`.

`-conds <file>` Change the final condition of tests. This is by far the most useful of these options: in combination with option `-show prop` it permits a fine grain selection of execution pictures — see Sec. 19.

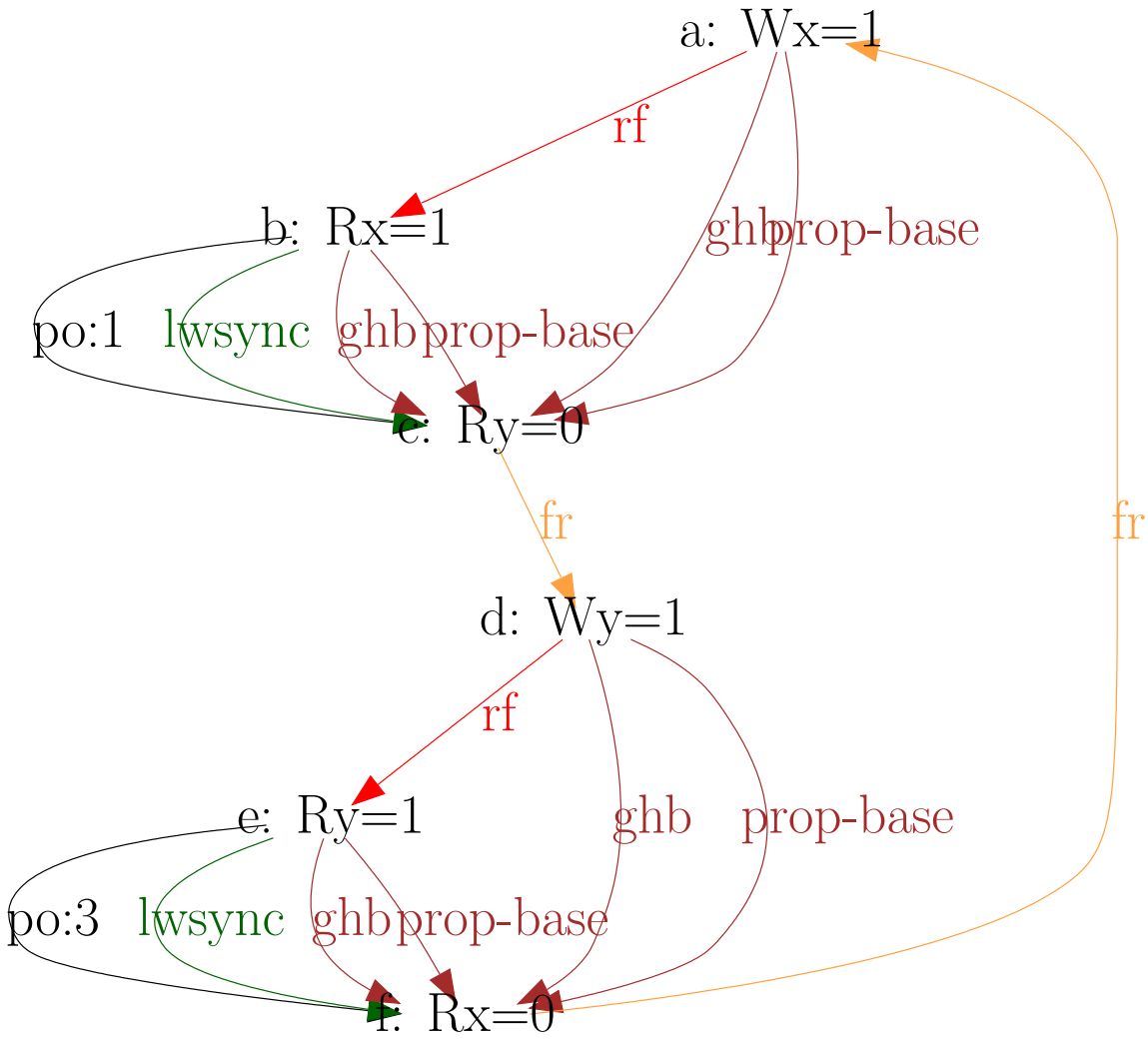
Configuration files

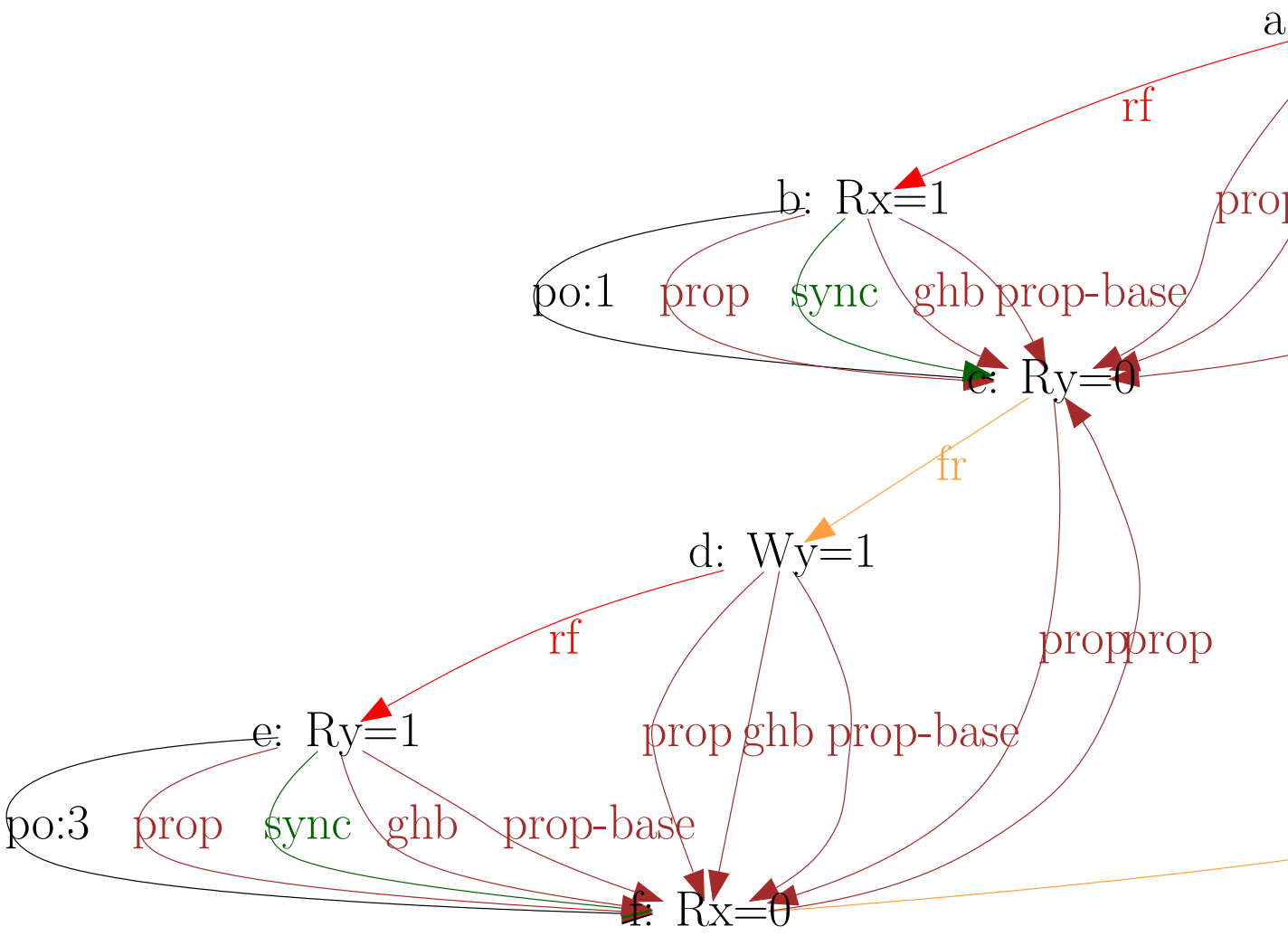
The syntax of configuration files is minimal: lines "*key arg*" are interpreted as setting the value of parameter *key* to *arg*. Each parameter has a corresponding option, usually `-`*key*, except for the single letter option `-v` whose parameter is `verbose`.

As command line option are processed left-to-right, settings from a configuration file (option `-conf`) can be overridden by a later command line option. Configuration files will be used mostly for controling pictures. Some configuration files are are present in the distribution. As an example, here is the configuration file `apoil.cfg`, which can be used to display images in `free` mode. The configuration above is commented with line comments that starts with "`#`". The above configuration file comes handy to eye-proof model output, even for relatively complex tests, such as **IRIW+lwsyncs** and **IRIW+syncs**:

```
% herd -conf apoil.cfg -show prop -gv IRIW+lwsyncs.litmus
% herd -through invalid -conf apoil.cfg -show prop -gv IRIW+syncs.litmus
```

We run the two tests on top of the default model that computes, amongst others, a `prop` relation. The model rejects executions with a cyclic prop. One can then see that the relation prop is acyclic for **IRIW+lwsyncs** and cyclic for **IRIW+syncs**:





Notice that we used the option `-through invalid` in the case of **IRIW+syncs** as we would otherwise have no image.

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

Part IV
Some examples

In the following experiment reports we describe both how we generate tests and how we run them on various machines under various conditions.

16 Running several tests at once, changing critical parameters

In this section we describe an experiment on changing the stride (cf Sec. 2.1). This usage pattern applies to many situations, where a series of test is compiled once and run many times under changing conditions.

We assume a directory `tst-x86`, that contains a series of litmus tests and an index file `@all`. Those tests where produced by the diy tool (see Sec. 6). They are two thread tests that exercise various relaxed behaviour of x86 machines. More specifically, diy is run as "`diy -conf X.conf`", where `X.conf` is the following configuration file

```
-arch  X86
-name  X
-safe  Rfe,Fre,Wse,PodR*,PodWW,MFencedWR
-relax PodWR,[Rfi,PodRR]
-mix   true
-mode  critical
-size  5
-nprocs 2
```

As described in Sec. 10.5, diy will generate all *critical* cycles of size at most 5, built from the given lists of candidate relaxations, spanning other two threads, and including at least one occurrence of PodWR, [Rfi,PodRR] or both. In effect, as x86 machines follow the TSO model that relaxes write to read pairs, all produced tests should *a priori* validate.

We test some `x86-64` machine, using the following `x86-64.cfg` litmus configuration file:

```
#Machine/OS specification
os = linux
word = w64
#Test parameters
size_of_test = 1000
number_of_run = 10
memory = direct
stride = 1
```

The number of available logical processors is unspecified, it thus defaults to 1, leading to running one instance of the test only (cf parameter a in Sec. 2.1)

We invoke litmus as follows, where `run` is a pre-existing empty directory:

```
% litmus -mach x86-64 -o run tst-x86/@all
```

The directory `run` now contains C-source files for the tests, as well as some additional files:

```
% ls run
comp.sh   outs.c  README.txt  utils.c  X000.c  X002.c  X004.c  X006.c
Makefile  outs.h  run.sh      utils.h  X001.c  X003.c  X005.c
```

One notices a short `README.txt` file, two scripts to compile (`com.sh`) and run the tests (`run.sh`), and a `Makefile`. We use the latter to build test executables:

```
% cd run
% make -j 8
gcc -Wall -std=gnu99 -fomit-frame-pointer -O2 -m64 -pthread -O2 -c outs.c
gcc -Wall -std=gnu99 -fomit-frame-pointer -O2 -m64 -pthread -O2 -c utils.c
gcc -Wall -std=gnu99 -fomit-frame-pointer -O2 -m64 -pthread -S X000.c
...
gcc -Wall -std=gnu99 -fomit-frame-pointer -O2 -m64 -pthread  -o X005.exe outs.o utils.o X005.s
gcc -Wall -std=gnu99 -fomit-frame-pointer -O2 -m64 -pthread  -o X006.exe outs.o utils.o X006.s
rm X005.s X004.s X006.s X000.s X001.s X002.s X003.s
```

This builds the seven tests `X000.exe` to `X006.exe`. The size parameters (`size_of_test = 1000` and `number_of_run = 10`) are rather small, leading to fast tests:

```
% ./X000.exe
Test X000 Allowed
Histogram (2 states)
5000  :>0:EAX=1; 0:EBX=1; 1:EAX=1; 1:EBX=0;
5000  :>0:EAX=1; 0:EBX=0; 1:EAX=1; 1:EBX=1;
No
...
Condition exists (0:EAX=1 /\ 0:EBX=0 /\ 1:EAX=1 /\ 1:EBX=0) is NOT validated
...
Observation X000 Never 0 10000
Time X000 0.01
```

However, the test fails, in the sense that the relaxed outcome targeted by `X000.exe` is not observed, as can be seen quite easily from the "`Observation Never...`" line above .

To observe the relaxed outcome, it happens it suffices to change the stride value to 2:

```
% ./X000.exe -st 2
Test X000 Allowed
Histogram (3 states)
21    *>0:EAX=1; 0:EBX=0; 1:EAX=1; 1:EBX=0;
4996  :>0:EAX=1; 0:EBX=1; 1:EAX=1; 1:EBX=0;
4983  :>0:EAX=1; 0:EBX=0; 1:EAX=1; 1:EBX=1;
Ok
...
Condition exists (0:EAX=1 /\ 0:EBX=0 /\ 1:EAX=1 /\ 1:EBX=0) is validated
...
Observation X000 Sometimes 21 9979
Time X000 0.00
```

We easily perform a more complete experiment with the stride changing from 1 to 8, by running the `run.sh` script, which transmits its command line options to all test executables:

```
% for i in $(seq 1 8)
> do
> sh run.sh -st $i > X.0$i
> done
```

Run logs are thus saved into files `X.01` to `X.08`. The following table summarises the results:

	X.01	X.02	X.03	X.04	X.05	X.06	X.07	X.08
X000	0/10k	21/10k	0/10k	17/10k	0/10k	19/10k	2/10k	40/10k
X001	0/10k	108/10k	0/10k	77/10k	2/10k	29/10k	0/10k	29/10k
X002	0/10k	2/10k	0/10k	6/10k	0/10k	7/10k	0/10k	5/10k
X003	0/10k	4/10k	2/10k	1/10k	0/10k	5/10k	0/10k	11/10k
X004	0/10k	4/10k	0/10k	33/10k	0/10k	10/10k	0/10k	8/10k
X005	0/10k	1/10k	0/10k	0/10k	0/10k	5/10k	0/10k	4/10k
X006	0/10k	8/10k	0/10k	9/10k	0/10k	11/10k	1/10k	12/10k

For every test and stride value cells show how many times the targeted relaxed outcome was observed/total number of outcomes. One sees that even stride value perfom better — noticeably 2, 6 and 8. Moreover variation of the stride parameters permits the observation of the relaxed outcomes targeted by all tests.

We can perform another, similar, experiment changing the s (`size_of_test`) and r (`number_of_run`) parameters. Notice that the respective default values of s and r are 1000 and 10, as specified in the `x86-64.cfg` configuration file. We now try the following settings:

```
% sh run.sh -a 16 -s 10 -r 10000 > Y.01
% sh run.sh -a 16 -s 100 -r 1000 > Y.02
% sh run.sh -a 16 -s 1000 -r 100 > Y.03
% sh run.sh -a 16 -s 10000 -r 10 > Y.04
% sh run.sh -a 16 -s 100000 -r 1 > Y.05
```

The additional `-a 16` command line option informs test executable to use 16 logical processors, hence running 8 instances of the "**X**" tests concurrently, as those tests all are two thread tests. This technique of flooding the tested machine obviously yields better ressource usage and, according to our experience, favours outcome variability.

The following table summarises the results:

	Y.01	Y.02	Y.03	Y.04	Y.05
X000	2.3k/800k	602/800k	465/800k	551/800k	297/800k
X001	2.9k/800k	632/800k	774/800k	667/800k	315/800k
X002	633/800k	55/800k	5/800k	7/800k	0/800k
X003	1.2k/800k	182/800k	152/800k	390/800k	57/800k
X004	2.4k/800k	974/800k	1.5k/800k	2.4k/800k	1.6k/800k
X005	239/800k	21/800k	8/800k	0/800k	1/800k
X006	912/800k	129/800k	102/800k	143/800k	14/800k

Again, we observe all targeted relaxed outcomes. In fact, x86 relaxations are relatively easy to observe on our 16 logical core machine.

Another test statistic of interest is *efficiency*, that is the number of targeted outcomes observed per second:

	Y.01	Y.02	Y.03	Y.04	Y.05
X000	285	2.2k	6.6k	9.2k	4.2k
X001	366	2.4k	13k	11k	5.2k
X002	78	212	71	140	
X003	150	650	2.5k	7.8k	950
X004	288	3.7k	25k	59k	32k
X005	28	72	114		17
X006	118	461	1.7k	2.9k	280

As we can see, although the setting `-s 10 -r 10000` yields the most relaxed outcomes, it may not be considered as the most efficient. Moreover, we see that tests **X002** and **X005** look more challenging than others.

Finally, it may be interesting to classify the "`X`" tests:

```
% mcycles @all | classify -arch X86
R
  X003 -> R+po+rfi-po : PodWW Wse Rfi PodRR Fre
  X006 -> R : PodWW Wse PodWR Fre
SB
  X000 -> SB+rfi-pos : Rfi PodRR Fre Rfi PodRR Fre
  X001 -> SB+rfi-po+po : Rfi PodRR Fre PodWR Fre
  X002 -> SB+mfence+rfi-po : MFencedWR Fre Rfi PodRR Fre
  X004 -> SB : PodWR Fre PodWR Fre
  X005 -> SB+mfence+po : MFencedWR Fre PodWR Fre
```

One sees that two thread non-SC tests for x86 are basically of two kinds.

17 Cross compiling, affinity experiment

In this section we describe how to produce the C sources of tests on a machine, while running the tests on another. We also describe a sophisticated affinity experiment.

We assume a directory `tst-ppc`, that contains a series of litmus tests and an index file `@all`. Those tests where produced by the diycross tool. They illustrate variations of the classical **IRIW** test. More specifically, the **IRIW** variations are produced as follows (see also Sec. 8):

```
% mkdir tst-ppc
% diycross -name IRIW -o tst-ppc Rfe PodRR,DpAddrdR,LwSyncdRR,EieiodRR,SyncdRR Fre Rfe PodRR,DpAddrdR,L
Generator produced 15 tests
```

We target a Power7 machine described by the configuration file `power7.cfg`:

```
#Machine/OS specification
os = linux
word = w64
smt = 4
smt_mode = seq
#Test parameters
size_of_test = 1000
number_of_run = 10
avail = 0
memory = direct
stride = 1
affinity = incr0
```

One may notice the SMT (*Simultaneaous Multi-Threading*) specification: 4-ways SMT (`smt=4`), logical processors pertaining to the same core being numbered in sequence (`smt_mode = seq`) — that is, logical processors from the first core are 0, 1 ,2 and 3; logical processors from the second core are 4, 5 ,6 and 7; etc. The SMT specification is necessary to enable custom affinity mode (see Sec. 2.2.4).

One may also notice the specification of 0 available logical processors (`avail=0`). As affinity support is enabled (`affinity=incr0`), test executables will find themselves the number of logical processors available on the target machine.

We compile tests to C-sources packed in archive `a.tar` and upload the archive to the target power7 machine as follows:

```
% litmus -mach power7 -o a.tar tst-ppc/@all
% scp a.tar power7:
```

Then, on `power7` we unpack the archive and produce executable tests as follows:

```
power7% tar xmf a.tar
power7% make -j 8
gcc -D_GNU_SOURCE -Wall -std=gnu99 -O -m64 -pthread -O2 -c affinity.c
gcc -D_GNU_SOURCE -Wall -std=gnu99 -O -m64 -pthread -O2 -c outs.c
gcc -D_GNU_SOURCE -Wall -std=gnu99 -O -m64 -pthread -S IRIW+eieios.c
...
```

As a starter, we can check the effect of available logical processor detection and custom affinity control (option `+ca`) by passing the command line option `-v` to one test executable, for instance `IRIW.exe`:

```
power7% ./IRIW.exe -v +ca
./IRIW.exe -v +ca
IRIW: n=8, r=10, s=1000, st=1, +ca, p='0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24
thread allocation:
[23,22,3,2] {5,5,0,0}
[7,6,15,14] {1,1,3,3}
[11,10,5,4] {2,2,1,1}
[21,20,27,26] {5,5,6,6}
[9,8,25,24] {2,2,6,6}
[31,30,13,12] {7,7,3,3}
[19,18,29,28] {4,4,7,7}
[1,0,17,16] {0,0,4,4}
...
```

We see that our machine `power7` features 32 logical processors numbered from 0 to 31 (cf `p=...` above) and will thus run `n=8` concurrent instances of the 4 thread IRIW test. Additionally allocation of threads to logical processors is shown: here, the four threads of the test are partitioned into two groups, which are scheduled to run on different cores. For example, threads 0 and 1 of the first instance of the test will run on logical processors 23 and 22 (core 5); while threads 2 and 3 will run on logical processors 3 and 2 (core 0).

Our experiment consists in running all tests with affinity increment (see Sec. 2.2.1) being from 0 and then 1 to 8 (option `-i` i), as well as in random and custom affinity mode (options `+ra` and `+ca`):

```
power7% for i in $(seq 0 8)
> do
> sh run.sh -i $i > Z.0$i
> done
power7% sh run.sh +ra > Z.0R
power7% sh run.sh +ca > Z.0C
```

The following table summarises the results, with X meaning that the targeted relaxed outcome is observed:

	Z.00	Z.01	Z.02	Z.03	Z.04	Z.05	Z.06	Z.07	Z.08	Z.0C	Z.0R
IRIW	X		X	X		X	X	X	X	X	X
IRIW+addr+po	X		X	X					X	X	
IRIW+addrs			X							X	X
IRIW+eieio+addr			X	X						X	
IRIW+eieio+po			X	X						X	
IRIW+eieios			X	X						X	X
IRIW+lwsync+addr			X	X						X	
IRIW+lwsync+eieio			X	X						X	
IRIW+lwsync+po	X		X	X				X		X	
IRIW+lwsyncs			X							X	
IRIW+sync+addr			X							X	
IRIW+sync+eieio			X							X	
IRIW+sync+lwsync			X							X	
IRIW+sync+po	X		X	X	X					X	X
IRIW+syncs											

On sees that all possible relaxed outcomes shows up with proper affinity control. More precisely, setting the affinity increment to 2 or resorting to custom affinity result in the same effect: the first two threads of the test run on one core, while the last two threads of the test run on a different core. As demonstrated by the experiment, this allocation of test threads to cores suffices to favour relaxed outcomes for all tests except for **IRIW+syncs**, where the `sync` fences forbid them.

18 Cross running, testing low-end devices

Together litmus options `-gcc` and `-linkopt` permit using a C cross compiler. For instance, assume that litmus runs on machine A and that `crossgcc`, a cross compiler for machine C, is available on machine B. Then, the following sequence of commands can be used to test machine C:

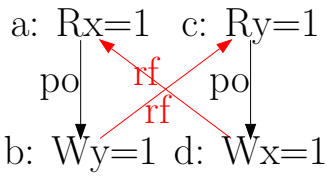
```
A% litmus -gcc crossgcc -linkopt -static -o C-files.tar ...
A% scp C-files.tar B:

B% tar xf C-files.tar
B% make
B% tar cf /tmp/C-compiled.tar .
B% scp /tmp/C-compiled.tar C:

C% tar xf C-compiled.tar
C% sh run.sh
```

Alternatively, using option `-crossrun` C, one can avoid copying the archive `C-compiled.tar` to machine C:

```
A% litmus -crossrun C -gcc crossgcc -linkopt -static -o C-files.tar ...
A% scp C-files.tar B:

B% tar xf C-files.tar
B% make
B% sh run.sh
```

More specifically, option `-crossrun` C instructs the `run.sh` script to upload executables individually to machine C, just before running them. Notice that executables are removed from C once run.

We illustrate the crossrun feature by testing **LB** variations on an ARM-based Tegra3 (4 cores) tablet. Test **LB** (load-buffering) exercises the following "causality" loop:



That is, thread 0 reads the values stored to location `x` by thread 1, thread 1 reads the values stored to location `y` by thread 0, and both threads read "before" they write.

We shall consider tests with varying interpretations of "before": the write may simply follow the read in program order (`po` in test names), may depend on the read (`data` and `addr`), or they may be some fence in-betweeen (`isb` and `dmb`). We first generate tests `tst-arm`with diycross:

```
% mkdir tst-arm
% diycross -arch ARM -name LB -o tst-arm PodRW,DpDatadW,DpCtrldW,ISBdRW,DMBdRW Rfe PodRW,DpDatadW,DpCtr
Generator produced 15 tests
```

We use the following, `tegra3.cfg`, configuration file:

```
#Tegra 3
size_of_test = 5k
number_of_run = 200
avail = 4
memory = direct
#Cross compilation
gcc = arm-linux-gnueabi-gcc
ccopts = -march=armv7-a -O2
linkopt = -static
```

Notice the "cross-compilation" section: the name of the gcc cross-compiler is `arm-linux-gnueabi-gcc`, while the adequate version of the target ARM variant and static linking are specified.

We compile the tests from litmus source files to C source files in directory `TST` as follows:

```
% mkdir TST
% litmus -mach tegra3 -crossrun app_81@wifi-auth-188153:2222 tst-arm/@all -o TST
```

The extra option `-crossrun app_81@wifi-auth-188153:2222` specifies the address to log onto the tablet by `ssh`, which is connected on a local WiFi network and runs a `ssh` daemon that listens on port 2222.

We compile to executables and run them as as follows:

```
% cd TST
% make
arm-linux-gnueabi-gcc -Wall -std=gnu99 -march=armv7-a -O2 -pthread -O2 -c outs.c
arm-linux-gnueabi-gcc -Wall -std=gnu99 -march=armv7-a -O2 -pthread -O2 -c utils.c
arm-linux-gnueabi-gcc -Wall -std=gnu99 -march=armv7-a -O2 -pthread -S LB.c
...
% sh run.sh > ARM-LB.log
```

It is important to notice that the shell script `run.sh` runs on the local machine, not on the remote tablet. Each test executable is copied (by using `scp`) to the tablet, runs there and is deleted (by using `ssh`), as can be seen with `sh` "`-x`" option:

```
% sh -x run.sh 2>&1 >ARM-LB.log | grep -e scp -e ssh
+ scp -P 2222 -q ./LB.exe app_81@wifi-auth-188153:
+ ssh -p 2222 -q -n app_81@wifi-auth-188153 ./LB.exe -q  && rm ./LB.exe
```

```
+ scp -P 2222 -q ./LB+data+po.exe app_81@wifi-auth-188153:
+ ssh -p 2222 -q -n app_81@wifi-auth-188153 ./LB+data+po.exe -q  && rm ./LB+data+po.exe
...
```

Experiment results can be extracted from the log file quite easily, by reading the "Observation" information from test output:

```
% grep Observation ARM-LB.log
Observation LB Sometimes 1395 1998605
Observation LB+data+po Sometimes 360 1999640
Observation LB+ctrl+po Sometimes 645 1999355
Observation LB+isb+po Sometimes 1676 1998324
Observation LB+dmb+po Sometimes 18 1999982
Observation LB+datas Never 0 2000000
Observation LB+ctrl+data Never 0 2000000
Observation LB+isb+data Sometimes 654 1999346
Observation LB+dmb+data Never 0 2000000
Observation LB+ctrls Never 0 2000000
Observation LB+isb+ctrl Sometimes 1143 1998857
Observation LB+dmb+ctrl Never 0 2000000
Observation LB+isbs Sometimes 2169 1997831
Observation LB+dmb+isb Sometimes 178 1999822
Observation LB+dmbs Never 0 2000000
```

What is observed (`Sometimes`) or not (`Never`) is the occurence of the non-SC behaviour of tests. All tests have the same structure and the observation of the non-SC behaviour can be interpreted as some read not being "before" the write by the same thread. This situation occurs for plain program order (plain test **LB** and `po` variations) and for the `isb` fence.

19 Finding and showing invalid executions

Part V
Automating the testing process

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

20 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. The automated front-end dont mechanises the task of checking that a machine or executable model conforms to such an architecture, and of exploring architectures. We provide some experiment reports elsewhere[9]. This document is intended to be a gentle introduction to dont and a partial reference.

21 A tour of dont

21.1 Checking conformance

We want to check 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, 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. 6.2.

21.2 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
```

[9] `http://diy.inria.fr/dont/dont/index.html`

```
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. 4 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)
```

21.3 Automatically exploring the memory model exhibited by a machine

Now suppose that we have no idea of the memory model of our 2 processors 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 launch 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[10] 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)
```

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

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 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 **
Testing: {}
Relaxed: {[Rfi,PodRR], PodWR}
Safe   : {[Rfi,PodRW], Rfe, Fre, Wse, PodWW, PodRW, PodRR, MFencedWW, MFencedWR, MFencedRW, MFencedRR}
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 interpretation is re-performed. Notice that the restart feature also permits to pursue interrupted experiments.

22 Usage of dont

In effect, the tool dont automates the complete testing procedure described in the documentation of diy proper (Sec. 6). 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).

22.1 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|ARM)` Set architecture. Default is `X86`. ARM is untested.

`-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.

22.2 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|ARM)` Target architecture. Default is `X86`.

`run = (local|ssh <addr>|cross <addr1> <addr2>)` Give access to the tested machine, which can be either the machine where dont runs, or remote machine `<addr>`, or compile C files on remote machine `addr1` and execute on tests on remote machine `addr2`. Machine addresses are `[user@]machine[:port]` expressing connection elements for both `ssh` and `scp`. 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 = <bool>` 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.

Controlling Cycle Generation

`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. 10.5) 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. Ddefault is `Rfe, Fre, Wse, PodR*, PodWW, MFencedWR` for x86, and unspecified for other architectures.

`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 unspecified for other architectures.

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 interesting alternative is `make -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] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for Power multiprocessors. In *CAV*, 2012.

[5] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.