RUNNING TESTS WITH Litmus7

Litmus tests

- Litmus tests are small, focused programs designed to test specific behaviors of a memory model. In the context of memory consistency models, litmus tests help illustrate how different memory operations might interact under various conditions.

- 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.

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

- Run the test using litmus7 tool
  - **% litmus7 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] ;
locations [x;y;]
exists (0:EAX=0 /\ 1:EAX=0)
```

Cross compilation

- With option `-o <name.tar>`, litmus7 does not run the test. Instead, it produces a tar archive that contains the C sources for the test.

```
$ litmus7 -o power.tar power.litmus
```

- Extract the contents of the tar file on required machine where we need to actually run the test

```
$ tar xf ../power.tar
$ ls
comp.sh        litmus_rand.h  outs.c  power.c     run.sh    utils.c
litmus_rand.c  Makefile       outs.h  README.txt  show.awk  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:

```
$ sh comp.sh
$ sh run.sh
$ ls
comp.sh        litmus_rand.o  outs.h  power.exe   run.sh    utils.h
litmus_rand.c  Makefile       outs.o  power.t     show.awk  utils.o
litmus_rand.h  outs.c         power.c README.txt  utils.c
```

- As we execute the run.sh we obtain a executable file **(.exe).**Notice that compilation produces an executable, which can be run directly, for a less verbose output.

Architecture Of Tests

- Consider a sample test **“a.litmus”** designed to run t threads **T0,....,Tt-1**
- The structure of a.exe performs as following :

1. **Parallel Execution of Tests**
   - ➢ To leverage parallelism, the system runs multiple tests concurrently on available logical processors.
   - ➢ We run **n = max(1,a/t)** tests concurrently on a machine
   - ➢ Where,

     n = number of tests to run concurrently.

     a = number of available logical processors.

     t = number of threads per test.

1. **Loop Execution in Threads:**
   - ➢ Each thread (Tk) runs a loop of size **s**. This means that the thread will repeatedly execute **s** times.
   - ➢ After each iteration of loop, the results (the final values of the thread's registers) are saved into an array. This array is indexed by the loop iteration number **i**, so each iteration's result is stored separately.

3. Memory Location as an Array Cell

- For each iteration i, the memory location x that the thread accesses is treated as an array cell (x[i]). This means that instead of accessing a single memory location, the thread accesses different locations in memory for each iteration.
- How this array cell is accessed depends on the **memory mode**.
- **Memory modes :**
  - **Direct Mode**:
    - The array is accessed in a straightforward, sequential manner. If x is an array, then the memory access would be something like x[i], where i is the current iteration number.
  - **Indirect Mode**:
    - Instead of accessing the array sequentially, the array is accessed through a shuffled array of pointers. This means , the thread might access the array x through a pointer that points to x but in a shuffled order

4. Loop Stride

- ➢ The **stride** refers to how much the loop index increases with each iteration. By default, its value is 1(i.e., i goes from 0 to 1 to 2, etc.), but it can be set to a different value.
- ➢ Larger stride might skip over some memory locations, reducing false sharing.
- ➢ **False sharing** occurs when multiple threads in a parallel program access different variables that happen to reside on the same cache line.

5. Collecting Outcomes from Threads

- ➢ After each test execution, the program waits for all t threads to complete their work.
- ➢ As each thread completes its execution, it records its outcomes in a structure, often a **histogram-like data structure.**
- ➢ **Histogram :** the test is evaluating different memory states or results, each thread might record how often each result occurs, which can then be stored into a histogram
- ➢ **Summing Histograms**: Once all tests are complete, the histograms from each test are combined. By summing the results from all the individual histograms to produce a final histogram that represents the final outcomes across all tests.

Command line options of Litmus7

Parameters	Description
-a <n>	Run maximal number of tests concurrently for *n* available logical processors. 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>	Number of concurrent tests
-r <n>	Number of iteration per test
-fr <f>	Multiply *r* by *f* (*f* is a floating point number).
-s <n>	Size of a loop iteration (r)
-fs <f>	Multiply *s* by *f*.
-f <f>	Multiply *s* by *f* and divide *r* by *f*.

Controlling Test Parameters

- The total number of outcomes produced by running a.exe is given by:
    - **Total Outcomes = n×r×s**
    - This is because:
        - n tests are run concurrently.
        - Each test runs r iterations.
        - Each iteration processes s (size of loop of each iteration) memory locations or operations.
- For example :
    - For instance, assuming *t*=2, ./a.exe -a 20 -r 10k -s 1 and ./a.exe -n 1 -r 1 -s 1M will both produce one million outcomes

      n*r*s = (a/t)*r*s = (20/2)*10k*1 = 1M

run.sh

Definition : The script that automates the execution of multiple test cases or executables providing consistent and controlled command line parameters

1. **Create or Modify the Script**:
   - Write or edit **run.sh** to include the appropriate parameters and list of executables you want to run.
2. **Make the Script Executable**:
   - Run the command **chmod +x run.sh** to make the script executable.
3. **Execute the Script**:
   - Run the script using **./run.sh** to start the test execution with the specified parameters.

```
#!/bin/bash
# Define command-line parameters
param_a=8
param_r=1
param_s=1M

# List of executable files
executables=("a.exe" "b.exe" "c.exe")

# Loop through each executable and run
with the parameters
for exe in "${executables[@]}"; do
    echo "Running $exe with parameters
-a $param_a -r $param_r -s $param_s"
    ./$exe -a $param_a -r $param_r -s
$param_s
done
```

Affinity

affinity refers to the property that binds a software thread to a specific hardware logical processor.This can be used to ensure that a thread runs on particular CPUs,

Benefits of setting affinity:

1. **Performance Optimization:**
   - By binding a thread to a specific core or set of cores, you can reduce the overhead associated with context switching between cores and ensure that the thread’s data remains in the cache of the assigned core, which can improve performance.
2. **Cache Utilization:**
   - Threads running on the same core can benefit from cache sharing, reducing cache misses and improving data locality.
3. **Load Balancing:**
   - In systems with multiple threads and cores, setting affinity can help in balancing the load by distributing threads across cores according to their processing requirements.

-affinity option

The **-affinity** option with the arguments **<none|random|custom|incr<n>|scan>** is used to control how threads are attached to logical processors

arguments	description
none	No specific CPU affinity is set for the threads. Threads are allowed to run on any available logical processor
random	Threads are assigned to logical processors in a random manner.
custom	the specifics of the custom affinity configuration can depend on additional arguments or configuration details provided by the user.
incr<n>	Threads are assigned to logical processors incrementally with a step size of <n>
scan	Threads are assigned to logical processors using a scanning approach, possibly covering all available processors in a sequence.

More affinity options

option	description	Example
-i <n>	alias for -affinity incr<n>	% ./ppc-iriw-lwsync.exe -i 2 binds the test threads to logical processors 0, 2, 4 and 6 etc
-procs <int list>	Specifies the exact order of logical processors to be used	%litmus7 -procs 2,0,3,1 mytest.litmus -o mytest_output.tar
-p <int list>	alias for -procs <int list>	%litmus7 -p 2,0,3,1 mytest.litmus -o mytest_output.tar
+ra / +rm	Perform random allocation of affinity at each test round.	% ./ppc-iriw-lwsync.exe +ra

Affinity custom control

-affinity custom : Provides flexibility to define a custom mapping of threads to CPUs according to specific needs or testing scenarios.

It usually requires additional parameters or configuration files that specify how threads should be mapped to CPUs.

This minimal logical processor topology is described by two litmus7 command-line option:

- **-smt <n> :** specify <n>-ways SMT, default 2 (Simultaneous Multi-Threading , where a single core can handle multiple threads.)
- **-smtmode <none|seq|end> :** specifies how logical processors from the same core are numbered, default none
    - **-smtmode none :** No specific numbering scheme is applied to logical processors.
    - **-smtmode seq :** Logical processors within each physical core are numbered sequentially.
    - **-smtmode end :** Logical processors may be numbered sequentially across all cores rather than within each core.

Example :

Let us consider a sample test : **test.litmus**

Enable affinity on test :

```
$ litmus7 -affinity random test.litmus -o test.tar

$ mkdir test_aff && cd test_aff

$ tar -xvf ../test.tar

$ ls

affinity.c  litmus_rand.c  outs.c    run.sh    test.t

affinity.h  litmus_rand.h  outs.h    show.awk  utils.c

affinity.o  litmus_rand.o  outs.o    test.c    utils.h

comp.sh     Makefile       README.txt  test.exe  utils.o

$ make
```

Contd…

```
vlab@HYVLAB8:~/Desktop/test_aff$ ./test.exe --help
usage: ./test.exe (options)*
  -v      be verbose
  -q      be quiet
  -a <n>  run maximal number of tests for n available processors (default 1)
  -n <n>  run n tests concurrently
  -r <n>  perform n runs (default 10)
  -fr <f> multiply run number per f
  -s <n>  outcomes per run (default 100000)
  -fs <f> multiply outcomes per f
  -f <f>  multiply outcomes per f, divide run number by f
  -i <n>  increment for allocating logical processors, -i 0 disables affinity mode
  -p <ns> specify logical processors (default '0,1,2,3')
  +ra     randomise affinity (default)
  +ca     alias for +ra
  +sa     alias for +ra
  +fix    fix thread launch order
```

Options that are enabled due to use of affinity

Usage of affinity enabled options

To limit the sequence of logical processor with option -p, which takes a sequence of logical processors numbers as argument

```
$ ./test.exe -p 0,1,2
```

To 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 :

```
$ ./test.exe -i 2
```

To bind test thread to logical processors randomly with executable option +ra use :

```
$ ./test.exe +ra (or) +ca (or) +sa
```

Use of -smt and -smtmode options

$ litmus7 **-smt 4 -smtmode seq -affinity custom** LB+addr+popx.litmus -o lb_smt.tar

- **-smt 4:** Specifies a 4-way SMT, meaning each core can handle four threads simultaneously.
- **-smtmode seq:** Specifies that logical processors on the same core are numbered sequentially (e.g., 0, 1, 2, 3 for a 4-way SMT core).
- **-affinity custom:** Enables custom affinity control, which allows the user to influence how threads are mapped to logical processors

```
$ mkdir lb_smt &&cd lb_smt

$ tar -xvf ../lb_smt.tar

$ make GCC=riscv64-linux-gnu-gcc

$ qemu-riscv64 -L /usr/riscv64-linux-gnu/ ./LB+addr+popx.exe --help

$ qemu-riscv64 -L /usr/riscv64-linux-gnu/ ./LB+addr+popx.exe -v     (-v  :    be verbose)
$ qemu-riscv64 -L /usr/riscv64-linux-gnu/
./LB+addr+popx.exe -v
LB+addr+popx: n=1, r=10, s=100000, +ca, p='0,1,2,3'
thread allocation:
[2,3] {0,0}
Test LB+addr+popx Allowed
Histogram (3 states)
19    :>0:x5=0; 1:x5=0; 1:x9=0; 1:x10=0; x=1;
499986:>0:x5=1; 1:x5=0; 1:x9=0; 1:x10=0; x=1;
499995:>0:x5=0; 1:x5=1; 1:x9=0; 1:x10=0; x=1;
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (x=1 /\ 0:x5=1 /\ 1:x10=0 /\ 1:x5=1 /\
1:x9=0) is NOT validated
Affinity=[1, 0] ;
Observation LB+addr+popx Never 0 1000000
Time LB+addr+popx 0.29
```

- The output indicates that one instance of the test is **run (n=1)**, with **r=1000 (iterations)** and **s=1000 (size).**
- The mode is **custom (+ca)**, meaning that specific logical processor sequences are chosen for running the threads.
- The test assigns threads to logical processors in the order **-p 0,1,2,3** (due to available of 4 cores)
- **Thread Allocation : [2,3] {0,0}**
  - The **square brackets [2,3]** indicate the allocation of test threads to **logical processors.**
  - The **curly braces {0,0}** indicate the allocation of test threads to **physical core 0** .
  - As **-smt 4** means each core handles 4 threads and number of cores we have are 4 ⇒ 4*4 = 16 logical processors
    - **0-3** logical processors allotted to **core 0** similarly **4-8** to **core 1** , **9-12** to **core 2** , **13-16** to **core 3**
  - This means that threads 2 and 3 are running on core 0

Affinity output

- In the affinity output :
    - The square brackets [..] indicate that the threads are allocated on the same core and near to each other
    - The parentheses(..) indicate that the threads are allocated on the different core and far from each other
- From the previous output **Affinity = [1,0]**:
- Both threads are allocated to same core therefore they are mentioned in square brackets

Timebase synchronisation mode

- It is a technique used to ensure that multiple threads in a concurrent test start their execution at roughly the same time

How it works

1. **Polling barrier :** This barrier ensures that all threads reach this point before proceeding
2. Once all threads have reached the barrier, they collectively agree on a target **timebase value.** The timebase is a reference point in time that the threads will use to determine when to start their main work
3. Threads compare the current timebase value with the target timebase value. They continue to check this condition in a loop until the current timebase exceeds the target value.
4. Once the current timebase value exceeds the target timebase, threads exit the loop and proceed with their main execution tasks

Contd…

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 $\boldsymbol{T_i = T + \delta_i}$, 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 $\mathbf{2^{11} = 2048}$ for all threads.

Command line options of timebase synchronization

Once timebase synchronisation have been selected (litmus7 option -barrier timebase), test executable behaviour can be altered by the following two command line options:

1. **-ta <n> :** Change the timebase delay δ_i of all threads.
2. **-tb <0:n$_0$;1:n_1;···> :** Change the timebase delay δ_i of individual threads.
3. **-vb true (verbose barrier) :** governs the printing of synchronisation timings , used along above options with litmus7

The command line option use along the executable file are :

1. **-vb :** Do not show synchronisation timings.
2. **+vb :** Show synchronisation timings for all outcomes.

Timebase vs user synchronization

Timebase synchronization : (-barrier timebase)

- It is done by polling a shared hardware timer (timebase)
- Timebase synchronisation of the testing loop iterations is selected by litmus7 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

User synchronization : (-barrier user)

- Threads synchronize with each other using software mechanisms like polling, signaling, or barriers
- This method is less precise because thread execution can be affected by factors like OS scheduling delays, thread startup times, or system load, leading to potential desynchronization between threads.

-barrier user execution

```
$ mkdir barrier_user

$ litmus7 -barrier user -vb true -o
barrier_user test.litmus

$ cd barrier_user

$ make

$ ./test.exe
```

Output of user synchronization

```
63483:*  -418
59547:*   282
58843:*  1010
35922:*   330
96356:*  -260
Test SB Allowed
Histogram (4 states)
5     *>0:EAX=0; 1:EAX=0; x=1; y=1;
499999:>0:EAX=1; 1:EAX=0; x=1; y=1;
499993:>0:EAX=0; 1:EAX=1; x=1; y=1;
3     :>0:EAX=1; 1:EAX=1; x=1; y=1;
Ok

Witnesses
Positive: 5, Negative: 999995
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=2d53e83cd627ba17ab11c875525e078b
Observation SB Sometimes 5 999995
Time SB 0.15
```

- In context of test output **63483** represents the test number i.e., 63483rd execution of test
- * on every line indicates that this specific run resulted in the **targeted outcome** being observed
- **-vb true :** this option prints the starting delays of each thread relative to first thread P0 in terms of timebase ticks

63483:* -418

- P1 thread started 418 ticks before P0(indicated by negative value)

59547:* 282

- P1 thread started 282 ticks after P0

-barrier timebase execution

```
$ mkdir barrier_timebase

$ litmus7 -barrier timebase -vb true -o barrier_timebase test.litmus

$ cd barrier_timebase

$ make

$ ./test.exe
```

Output of timebase synchronization

```
0008:* 2048[59]  2050[56]
0006:* 2048[62]  2050[59]
0004:* 2068[64]  2062[67]
0002:* 2068[69]  2058[65]
0000:* 2052[58]  2050[61]
Test SB Allowed
Histogram (4 states)
315533*>0:EAX=0; 1:EAX=0; x=1; y=1;
231467:>0:EAX=1; 1:EAX=0; x=1; y=1;
250834:>0:EAX=0; 1:EAX=1; x=1; y=1;
202166:>0:EAX=1; 1:EAX=1; x=1; y=1;
Ok

Witnesses
Positive: 315533, Negative: 684467
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=2d53e83cd627ba17ab11c875525e078b
Observation SB Sometimes 315533 684467
Time SB 6.01
```

- In context of test output **0008** represents the test number i.e., 8th execution of test
- * on every line indicates that this specific run resulted in the **targeted outcome** being observed
- **-vb true :** this option prints the starting delays of each thread relative to first thread P0 in terms of timebase ticks

0008:* 2048[59] 2050[56]

- Threads read timebase values like **2048** and **2050**.
- Thread 1 looped **59 times**, and thread 2 looped **56 times** while waiting for the timebase value to reach the expected target timebase value

Prefetch Control

- Prefetching refers to the process of loading data into the cache before it is actually needed by the CPU.
- The custom prefetch mode allows users to control how cache management instructions are applied during the execution of memory consistency tests.
- This mode is enabled using the **-preload custom** option with the **litmus7** tool.

```
$ mkdir preload
$ litmus7 -mem indirect -preload custom -o preload test.litmus
$ cd preload/
$ make
$ ./test.exe --help
.
.
.
-pra (I|F|T|W) set all prefetch
-prf <list> set prefetch, default ''
List syntax is comma separated proc:name=(I|F|T|W)
```

-pra option

- The executable test.exe accepts two new command line options: **-prf and -pra.**
- Those options takes arguments that describe cache management instructions
- The **-pra** option is used to specify the cache management instructions. It takes one letter that stands for a cache management instruction as we here describe:
    - **I: Do nothing** (no cache management operation).
    - **F: Cache flush** (invalidate the cache line, forcing the data to be written back to memory).
    - **T: Cache touch** (preload the cache line into the cache, typically for reading).
    - **W: Cache touch** for a write (preload the cache line into the cache, anticipating a write operation).
- Example of usage of -pra option: **$ ./test.exe -pra T**
- Not all architectures support every cache management instruction.
- If an instruction is not supported:
    - **F (Flush)** will do nothing.
    - **T (Touch)** will do nothing.
    - **W (Touch for a write)** will behave as **T (Touch)**

-prf option

The **-prf** option provides a more selective approach to cache management. It allows you to specify different cache management instructions for specific threads and specific memory locations.

Syntax :

-prf n:loc=X

- n: Thread number (e.g., 0, 1, 2, etc.).
- loc: Program variable or memory location (e.g., x, y, z).
- X: Cache management control letter (I, F, T, or W).

$ ./test.exe -prf 0:x=T,1:y=F

- Instructs **thread 0** to **"touch"** (preload into cache) memory location **x** before executing the test code.
- Instructs **thread 1** to **"flush"** (load data to main memory) memory location **y** before executing the test code.

Prefetch metadata

- The source code of tests may include prefetch directives as metadata prefixed with **“Prefetch=”**. In particular, the generators of the diy7 suite produce such metadata

  Prefetch=0:x=F,0:y=T,1:y=F,1:z=T

- **-preload custom** lets you use prefetch metadata specified in the litmus test file.

  ```
  $ litmus7 -mem indirect -preload custom -o R 6.SB+Prefetch.litmus
  ```

- **-hints** option allows you to override or specify additional prefetch directives using a mapping file

  ```
  $ cat map.txt
  SB Prefetch=0:x=W,0:y=F
  $ litmus7 -mem indirect -preload custom -hints map.txt -o hints/ test.litmus
  $ cd hints/
  $ make
  $ ./test.exe
  ```

Static prefetch control

Static Preload Modes : These modes offer a way to execute cache management instructions consistently and predictably without relying on the runtime -prf options.

1. **-preload static :** Executes cache management instructions as commanded by the Prefetch metadata.
2. **-preload static1 :** Similar to -preload static, but without the -prs option to control the execution probability at runtime.
3. **-preload static2 :** Cache management instructions are executed with probability ½ (-prs = 2).

Runtime option “-prs”

It allows you to control the probability of executing cache management instructions.

- **-prs 0**: Disables cache management instructions.
- **-prs 1**: Always executes cache management instructions.
- **-prs 2**: Executes cache management instructions with probability 1/2.
- **-prs n**: Executes cache management instructions with probability 1/n.

Static prefetch execution

-prefetch static execution

```
$ mkdir static
$ litmus7 -mem indirect -preload static -o static/
test.litmus
$ cd static/
$ make
$ ./test.exe --help
usage: ./test.exe (options)*
  -v      be verbose
  -q      be quiet
.
```

-prs <n> prefetch probability is 1/n, -prs 0 disables feature, default 1

-prefetch static1 execution

```
$ mkdir static1
$ litmus7 -mem indirect -preload static1 -o static1/
test.litmus
$ cd static1/
$ make
$ ./test.exe --help
usage: ./test.exe (options)*
  -v      be verbose
  -q      be quiet
.
.
.
```

THANK YOU

Presented By :
S.Lochani Vilehya
Emp ID : 43317