The Library Scheme Competition

Brandon Cho (mhcho@mit.edu)

Table of contents

Introduction

'Library' schemes work for the timeout mechanism which allows data blocks retrieved from home core in S-NUCA architecture to be cached in the cores that requested data for a limited period time; when a remote core requests a cache block for read, the home core decides until when (as an absolute time based on a global time) the remote core may retain the data and read from the block. The expiration time (although it is probably not a great name, we call it timestamp for now) is sent to the requester along with the data, and also written in the cache of the home core. All remote cores must throw away remote copies of data when the global time reaches the time given by the timestamp of each block. Therefore, the home core knows that no other cores may have a remote copy of a given cache block if the timestamp corresponding to the block has expired. Consequently, data consistency is hold only if the home does not allow writing to a cache block before its timestamp gets expired.

There are a number of design choices in implementing this mechanism. Anyone who comes up with the best scheme will win the library competition, and get a pair of Bose noise-canceling headphones (sponsored by Srini) as the prize!

The competition ends at Monday, February 14 at 5PM EST. DO NOT check in your codes in the git repository. Send a brief description on your scheme, test results (if available), and archived source codes to mhcho@mit.edu.

Build & Run

We will use DARSIM with shared memory support to evaluate library schemes.

1. Build DARSIM for the competition

Get a fresh copy of DARSIM code by:

```
git clone /afs/csail/group/csg/gitroots/dar.git
```

Configure building scripts as the following:

```
./bootstrap
./configure --prefix=$HOME/SOMEWHERE
```

Executables will be copies in `$HOME/SOMEWHERE`. The simplest way to use configurations for the competition is to use `-DLIBRARY_COMPETITION` option when you compile. You may also give an optimization-level option too.

```
make install CXXFLAGS='-DLIBRARY_COMPETITION -O3'
```

2. Run DARSIM with various traces

Under the retrieved DARSIM source directory, go into `examples/library-competition`. First, you have to make an configuration image file from the input script provided in the directory by:

```
$HOME/SOMEWHERE/bin/darimg
   shmem_raonly_p64_1vc_il1l2.cfg
```

This will generate `output.img` file. The simulation will use memory trace files, and you will find five traces (FFT, LU_CONTIGUOUS, OCEAN_CONTIGUOUS, RADIX, WATER-N-SQUARED) in `/scratch/mhcho/library-competition` at `beast1.csail.mit.edu` (under the root directory. email to mieszko@mit.edu if you have no access to beast1 machine). Then you can run the simulation by:

```
$HOME/SOMEWHERE/bin/darsim
   output.img
   --memory-traces=TRACE_FILE_TO_EXECUTE
   --cycles=100000000
```

Make it sure to set a very large number for `--cycles`, because the simulation will end as soon as finishing the traces. In order to observe deterministic results for each run, you may give a fixed random seed:

```
   --random-seed=7
```

Also, you may find it useful to give a log level to monitor how the simulation is running. You may specify the level of log messages to print:

```
   --verbosity=1
```

WARNING: Although those traces are generated with a very small size of problems, file sizes are still large, and it take very long to finish (expect tens of hours, or a couple days). You may want to use synthetic traces for quick evaluations of your schemes (see Section 4. Using synthetic traces).

3. Statistics report and competition criteria

At the end of the standard statistics report of DARSIM, the following information is provided:

```
Parallel Completion Time : 93020.00000 cycles
  Fastest thread completion time : 68634.00000 cycles
  Average thread completion time : 83863.56250 cycles

Average memory latency : 35.93178 cycles +/- 124.84397
  average memory latency - READ : 17.97659 cycles +/- 21.37446
  average memory latency - WRITE : 89.45338 cycles +/-
   238.04922

Remote Access Rate : 48.16719%
  ( f0d6 / 1f400 )
```

Although the parallel completion time shows the overall performance of your scheme, the other statistics may affect the competition results; the best to way to evaluate a library scheme should be discussed more. Also, the implementation cost will be used as a tie-breaker if the performance of two schemes are within 3% of each other.

4. Using synthetic traces

If you feel the traces from SPLASH-2 benchmarks take too much time, you may want to use synthetic traces. You can generate synthetic traces using `examples/library-competition/generate_synthetic_threads.py` file. It takes the following options:

```
Options (defaults)
 -n <arg> number of threads (64)
 -l <arg> number of instructions per thread (10000)
 -m <arg> % of memory instructions (20)
 -w <arg> % of writes in memory instructions (25)
 -p <arg> % of private data accesses in memory instructions
   (70)
 -t <arg> temporal locality index [0 to 99] (40)
 -o <arg> output filename (synthetic.memtrace)
```

Use 64 threads, and adjust the number of instructions for your purpose on each simulation (10000 instructions take < 5 minutes to finish). Generated traces will make memory instructions at a fixed interval, and whether it will access private data of a thread or remote data will be decided on two-state Markov modulated Bernoulli process. The temporal locality index tells how likely the next memory access will be on the private data of running thread, if the previous was on the private data. Depending on the combination of option `-p` and `-t`, the actual percentages of private data accesses and the actual degree of temporal locality may change from the specified values. You can check the adjust value from the standard output of the script:

```
./generate_synthetic_threads.py -p 15 -t 80
 - memory instructions in every 5 non-memory instructions.
 - actual private access ratio : 13.0434782609 %
     - the probability of accessing remote data after
   accessing a local data : 20 %
     - the probability of accessing local data after accessing
   a local data : 20 %
```

You can see although the portion of private data access is set to 15%, the actual percentage is adjusted to 13% (This is an expected value, and the percentage on each output trace file may differ too).

Where to make important design choices

In order to implement your own schemes, you will mostly work on source files in `src/exec` directory. This chapter explains where to look for each main optimization point. For more information on the shared memory framework for DARSIM, read **Shard Memory Framework for Hornet** (`src/exec/shared_memory_framework.pdf`).

Be careful in evaluating the hardware costs of your scheme because subtle details may result in huge differences. For example, the granularity of your timestamp logic (how finely you decide the expiration time) will decide the size of caches and network traffic.

1. Timestamp logic

Probably the most important design problem is how to decide the timestamp value for each remote access request. The decision is made on the line 233 of `src/exec/core.cpp` (inside `exec_mem_server()` function. search for `LIBRARY_COMPETITION`).

A limited number of information is provided, such as the current timestamp of the cache block, when the cache block is accessed in last, how long the data has been cached (since the last time it gets in), and for how

many cycles writes have been pending on the cache block. Keep in mind that each information has a different implementation costs.

Of course, you may use different types of information. Read `src/exec/homeCache.hpp` and modify if needed, to get the information you decided to use.

2. Pass additional information with memory requests

If you want to include additional information in remote access requests for the home core to consider in deciding timestamp values, read line 163 of `src/exec/memtraceCore.cpp` (inside `exec_core()` function. search for `LIBRARY_COMPETITION`). This is where a thread makes a remote access, and you can add more information in `req` (an instance of `memoryRequest`). You will probably find that you want to modify `src/exec/memoryRequest.hpp` and `src/exec/memoryRequest.cpp` (in order to include more information in requests) and also `src/exec/memtraceThread.hpp` and `src/exec/memtraceThread.cpp` as well (in order to take information from running threads). Obviously, you will also modify the previous file `src/exec/core.cpp` to use those information.

3. Replacement policies of home cache & away cache

The replacement policies of home cache (where local data is cached) and away cache (where remotely accessed data is cached) is defined in line 192 of `src/exec/homeCache.cpp` and in line 221 of `src/exec/awayCache.cpp`, respectively.

4. Other optimizations

Your implementation is not limited to the described above. For example, you may try different schemes in how to deal with remote write accesses. In most cases, you will work on the files that are already introduced in this chapter (files of class `core`, `memtraceCore`, `homeCache`, `awayCache`, `memoryRequest`). However, if you need more controls on the framework, you may also want to look at `src/exec/remoteMemory.hpp`, `src/exec/remoteMemory.cpp` and `src/exec/message.hpp`.

Test configurations

This section summarizes the system setup used for the competition. Although it is not subject to optimization, please note that any assumptions here may change if we find there are better configurations.

Network	
# Cores	64
# VC sets	4
# VCs	4 per link (1 per set)
Flit width	128 bits
Per-hop latency	1 cycle
Core	
Architecture	RA only
I-cache	Magic
L1 Data Cache (both for home cache and away cache)	
# L1 caches	64+64 (each core has 1 L1 home and 1 L2 home)
Configuration	2-way associative
Block size	16 bytes
Total size	1Kbytes
Process time	1 cycle

L2 Data Cache	
# L2 caches	64 (each core has 1 L2)
Configuration	2-way associative
Block size	16 bytes
Total size	4Kbytes
Process time	1 cycle
DRAM Controller	
# DRAM Controllers	16 (Along the first and the last columns)
DRAM Controller <-> DRAM latency	60 cycles
DRAM Controller process time	1 cycle
DRAM process time	10 cycles
Throughput	16 GB/s for 1GHz
Off-chip memory request header size	8bytes

Revision History

Date	Description
2011/01/28	The initial release