ECE 463/521: Fall 2002
Project #1: Data-Cache System Design
Due: Monday, Sept. 30, 11:00 PM

Project rules

1. All students are encouraged to work in teams of two, using pair programming. Pair programming means programming where two people sit at the same workstation, writing the code collaboratively.

2. ECE 521 students will have additional parts to do; therefore both members of a pair should be from the same class (463 or 521).

3. You may not work with the same partner on more than one project this semester. You must register your partnership by posting on the Pair-Programming Partners message board, under the topic Project 1.”

4. Sharing of code between teams will be considered cheating, and will be penalized in accordance with the Academic Integrity policy.

5. It is acceptable for you to compare your results with other groups to help debug your program. It is not acceptable to collaborate on the final experiments.

6. You must do all your work in the C/C++ or Java languages. C++ (or Java) is encouraged because it enables straightforward code-reuse and division of labor.

7. Homework will be submitted over the Wolfware Submit system and run in the Eos/Unity environment.

Project description

This project will give you a chance to build three general data-cache models: a one-level (L1) cache, and two hybrid caches (L1+victim cache, L1+MRU cache). You will write a trace-driven simulator, which inputs a “trace” (a sequence of references) from a dynamic instruction stream to simulate hardware operations.
**Input: Trace file**

The simulator reads a trace file that contains one reference per line in the following format (an address consists of 32 bit):

```
r | w <hex address>
r | w <hex address>
... 
```

Example:

```
r ffe04540
r ffe04544
w 0eff2340
r ffe04548
... 
```

**Simulator: Your task**

**Specification of simulator**

**Cache simulation capabilities**

a. The simulator models a memory hierarchy with (1) an L1 data cache and (2) an optional victim cache (the system can be configured with or without a victim cache). **Tip:** If you are using C++ or Java, note that the victim cache code is quite different from the L1 cache code, so it is best to implement different classes for each.

b. L1 cache description:

- **SIZE:** total bytes of data storage
- **ASSOC:** the associativity of the cache \((ASSOC = 1\) is a direct mapped cache)
- **BLOCKSIZE:** the number of bytes in a block
- LRU replacement policy

There are a few constraints on the above parameters: (i) **BLOCKSIZE** is a power of two and (ii) the number of **sets** is a power of two. *Note that ASSOC (and, therefore SIZE) need not be a power of two.* The number of sets is determined by the following equation:

\[
\# \text{sets} = \frac{\text{SIZE}}{\text{BLOCKSIZE} \times \text{ASSOC}}
\]

You may assume that a miss to the L1 cache that has to be satisfied from memory takes as long to process as 20 hits.

c. Victim cache description:

- The victim cache is fully-associative and uses LRU replacement policy.
- The number of victim cache entries, \(V\), is adjustable.

\(\textbf{L1 cache miss / victim cache hit:}\) If there is a miss in the L1 cache and a hit in the victim cache (say for block \(X\)), block \(X\) is placed in the L1 cache. Then, the block that \(X\) replaced in the L1 cache (say block \(Y\), which we call the victim block) is
placed in the victim cache. Block \( Y \) goes into the victim cache entry where block \( X \) was found, replacing block \( X \) in the victim cache. That is, the two caches swap blocks \( X \) and \( Y \). This also means that a given block will never reside in both caches at the same time.

A special case is when there is no victim block from the L1 cache (which occurs when there are invalid entries in the L1 cache). In this case, block \( X \) is simply invalidated in the victim cache, instead of being replaced by a victim block.

**L1 cache miss / victim cache miss:** If there is a miss in the L1 cache and a miss in the victim cache (say for block \( X \)), block \( X \) is placed in the L1 cache. Then, the block that \( X \) replaced in the L1 cache (say block \( Y \), which we call the victim block) is placed in the victim cache. We cannot perform a swap, as we did above, because block \( X \) was not found in the victim cache. Instead, block \( Y \) replaces the LRU block in the victim cache. In the special case that there is no victim block from the L1 cache, the victim cache does nothing.

**d.** An “MRU cache” consists of the most recently referenced line in each set of a set-associative cache. For the purposes of this simulator, you may assume that when an MRU cache is being modeled, it takes 1 cycle to access the MRU line of a cache, and two cycles to access the non-MRU lines of each set. However, because the MRU cache is direct mapped, its cycle time is shorter than the cycle time of a set-associative cache. Specifically, assume that the cycle time with an MRU cache in use is \( 4/5 \) (80%) of the cycle time when only a set-associative L1 cache is in use.

When there is a hit to a non-MRU line in the set-associative cache, the block that was just hit needs to be moved to the MRU cache. Assume that this causes a stall of two cycles.

**e.** Prefetching should be used in your simulation. This means that when a reference to a block, say block \( i \), causes a miss, block \( i \) is fetched, but immediately after block \( i \) is in the cache, block \( i+1 \) is also fetched. This means that another line needs to be replaced from the cache. Note that this line will always come from the next set after the set into which line \( i \) was fetched. The prefetched line then becomes the MRU line of the set.

Since it takes as long to process a miss as to process 20 hits, when a block is prefetched, you should “schedule” the prefetch to begin as soon as the cache miss that triggered the prefetch has been processed. That is, if the current time is \( t \) when a miss is encountered, the processor will be stalled until time \( t+20 \) waiting for the miss to be serviced. Then a prefetch will begin at time \( t+20 \) and finish at time \( t+40 \).

Once a prefetch has begun, the block that is being prefetched is neither valid nor invalid, but “in transition.” That is, if it is referenced, it does not cause another miss, but the processor can’t continue right away either. Rather, the processor is stalled until the prefetch has finished.

Here are some thoughts on how to implement this. Instead of a valid/invalid bit for each cache line, we now need a variable that can take on the values VALID, INVALID, and IN TRANSITION. When a prefetch begins, the line into which the block is being prefetched changes state to IN TRANSITION.

What happens if a block is referenced while it is IN TRANSITION? The processor must stall till the prefetch finishes. So, the simulator can have a variable called, e.g., \( \text{time\_prefetch\_done} \). Whenever a prefetch occurs, this variable is set to the time when it will be finished (e.g., \( t+40 \) from
the example above). Now, if a block that is IN TRANSITION is referenced, a stall must occur until that time. So in the simulator, we can simply set current_time = time_prefetch_done. This takes care of handling the stall. (Note that we can get by with a single time_prefetch_done variable, since only one block can be being prefetched at a time.)

Of course, before we handle a prefetch-induced stall, we need to be careful that the block is still IN TRANSITION. If the current time is after the time that the prefetch finishes, we don’t need to stall (indeed, we better not stall!). We just need to set the block’s status to VALID. An easy way to implement references to blocks IN TRANSITION would be to set current_time = min(current_time+1, time_prefetch_done) and set the block’s status to VALID.

There’s another special case we need to consider. That is, what happens when a miss occurs while a prefetch is in progress? We can’t start to process the miss until the prefetch is finished. (Actually, high-performance processors use split-transaction buses, which allow transfers to memory to be overlapped, but in our first project, we will opt for the simpler approach of assuming that only one transfer is active at a time.) So this means that if the current time is t, the miss will not finish being processed at time t+20, but rather at time_prefetch_done+20. Thus, prefetching makes it possible to have stalls that are almost twice as long as the stalls without prefetching. We hope that this doesn’t happen very often, and that the cost is outweighed by having prefetched blocks made available without any processor stalls.

f. The write policy is either WTNA or WBWA, described in detail below.

g. WTNA:
   - Write hits in L1 cache: The write also goes to the victim cache (write-through). It should not hit in the victim cache, since a block cannot be in both caches at once (see victim cache description). You should access the victim cache anyway, just to assert that the block isn’t found.
   - Write misses in L1 cache: Since we do not allocate on a write miss (WTNA), there cannot be a victim (no block is replaced in L1 cache). So, victim cache activity is minimal. The write is also issued to the victim cache (write-through) to keep its data up-to-date. In the simulator, you only need to update LRU if it hits; nothing is done if it misses.

h. WBWA: Both the L1 cache and victim cache have dirty bits. Blocks are managed exactly as described above (see victim cache description), only make sure to carry dirty bits with blocks wherever they go. Definition of “writeback”: In a system without a victim cache, replacing a dirty block in the L1 cache causes a writeback to memory, and it is counted towards the “writeback” statistic. However, if there is a victim cache, the replaced block and its dirty bit are sent to the victim cache, and we do not call this a “writeback” because it isn’t sent to memory. When a LRU dirty block is replaced in the victim cache (not swapped!), it is written back to memory and counts toward the “writeback” statistic. In summary, in a system with a victim cache, writebacks can only occur from the victim cache; in a system without a victim cache, writebacks occur from the L1 cache.

i. Five parameters completely specify the system: SIZE, ASSOC, BLOCKSIZE, WRITE_POLICY, V, and M. The variable V is the size of the victim cache. If V=0, the victim cache is disabled. M is a boolean; iff M is false, there is no MRU cache.

j. The size of an address is 32 bits.
**Output: Statistical information**

In overview, this is the kind of information you will be collecting for each of the cache systems. For more complete information, refer to the specific lists in the descriptions of each type of cache system, below.

1. Number of cache reads
2. Number of read misses
3. Number of cache writes
4. Number of write misses
5. Number of writebacks (if WBWA policy was used)
6. Total memory traffic (number of blocks transferred to/from main memory, including misses to memory and writebacks)
7. Cache miss rate = \( \frac{\text{cache read misses} + \text{cache write misses}}{\text{cache reads} + \text{cache writes}} \)

**Project Specification**

**Model 1: L1 cache**
(Both 463 and 521 students should complete this part)

**Requirements**
1. The parameters, such as cache size, block size, and associativity \((n \geq 1)\), and write mode (WBWA or WTNA) are adjustable in your simulator.
2. LRU is the only replacement policy.

**Output of your simulator**
1. Number of L1 reads
2. Number of L1 read misses
3. Number of L1 writes
4. Number of L1 write misses
5. Number of writebacks
6. Total memory traffic (number of blocks transferred to/from main memory, including misses to memory and writebacks)
7. L1 miss rate = \( \frac{\text{L1 read misses} + \text{L1 write misses}}{\text{L1 reads} + \text{L1 writes}} \)

**Data analysis:**
Use your data collected from your experiments to analyze the relationship of the miss rate, AAT (average access time—refer to AAT calculation and CACTI tool), cache size, block size, and other parameters of cache (better with table and graphic). Find two data cache systems with the best AAT for each trace file.
Model 2: L1 cache+Victim cache
(for both of 463/521 students)

Requirements
1. The cache size, block size, cache associativity, write mode and prefetching policy of L1 and Victim cache (fully associative) are adjustable in your simulator.
2. LRU is the only replacement policy.

Output of your simulator:
1. Number of L1 reads
2. Number of L1 read misses
3. Number of L1 writes
4. Number of L1 write misses
5. Number of VC reads
6. Number of VC read misses
7. Number of VC writes
8. Number of VC write misses
9. Number of writebacks
10. Total memory traffic
11. L1 miss rate = \( \frac{L1 \text{ read misses} + L1 \text{ write misses}}{L1 \text{ reads} + L1 \text{ writes}} \)
12. VC miss rate = \( \frac{VC \text{ read misses} + VC \text{ write misses}}{L1 \text{ read misses} + L1 \text{ write misses}} \)

Data analysis
Take the two best combinations of parameters from Model 1, and vary the size of the victim cache. Find the two data-cache systems with the best AAT for each trace file. Describe why you think the best-performing system outperforms the others.

Model 3: MRU cache
(for 521 students only)

Requirements
1. The cache size, block size, cache associativity and write mode of the MRU cache are adjustable in your simulator.
2. LRU is the only replacement policy.
3. When a line is prefetched, it could come into either the MRU cache, or the non-MRU portion of the L1 cache. Assume it comes into the non-MRU portion of the cache, as the most recently used line in that portion (i.e., as the second most recently used line in the set).
Output of your simulator

1. Number of cache cache reads
2. Number of cache read misses
3. Number of cache writes
4. Number of cache write misses
5. Number of MRU read misses
6. Number of MRU write misses
9. Number of writebacks
10. Total memory traffic
11. MRU miss rate = \( \frac{\text{MRU read misses} + \text{MRU write misses}}{\text{MRU reads} + \text{MRU writes}} \)
12. Cache miss rate = \( \frac{\text{cache read misses} + \text{cache write misses}}{\text{cache reads} + \text{cache writes}} \)

Data analysis

Use the data collected from your experiments to analyze the relationship of miss rate, AAT (average access time), cache size, block size, and other parameters of the cache (ideally, with tables and graphics). Find two data-cache systems with the best AAT for each trace file.

Overall analysis

From an architectural perspective, analyze and compare the characters of those three models with the results from your experiments.

CACTI tool

Installing

Log into your Unity home directory in UNIX.

// link to the tool on the server
eos% ln -s /afs.eos.ncsu.edu/courses/ece/ece521/lec/001/www/projects/CACTI.FUDGE.FIXWARNING
// the directory name CACTI.FUDGE.FIXWARNING will show up at your home directory
eos% cd CACTI.FUDGE.FIXWARNING
// then cacti will work.
eos% cacti

Using

CACTI Syntax
cacti <csize> <bsize> <assoc> <tech>
OR

cacti <csize> <bsize> <assoc> <tech> <RWP> <RP> <WP>

csize - size of cache in bytes (i.e. 16384)
bsize - block size of cache in bytes (i.e. 32)
assoc - associativity of cache (i.e. 2 or FA)
direct mapped caches - DM
set associative caches - number n (where cache is n-way associative)
fully associative caches - FA
technology size in microns (0.35um we use)
RWP - number of read/write ports (we use defaults)
RP - number of read ports (we use defaults)
WP - number of write ports (we use defaults)

Example:

```
eos% cacti 16384 32 2 0.35um
```

will return the following timing and power analysis for a 16K 2-way set associative cache
with a 32-byte blocksize at a 0.35um feature (technology) size:

```
 Technology Size: 0.35um
 Vdd: 2.6V
 Access Time (ns): 2.36865
 Cycle Time (wave pipelined) (ns): 0.78955
 Power (nJ): 3.76872
 Wire scale from mux drivers to data output: 0.10
```

**AAT calculation**

The access times of the L1 cache, MRU cache and victim cache can be determined using
the CACTI tool. Below, $T_{L1}$, $T_{MRU}$ and $T_{V}$ refer to the L1 cache, MRU cache and victim
cache access times (in ns), respectively.

However, for the case of MRU+L1 cache, we will assume that the access time to the L1
cache is two cycles. It needs to be some multiple of the MRU cache’s cycle time, and 2
cycles seems reasonable in almost all cases.

For L1 cache

$$\text{AAT (s) = } T_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

It takes 20 cycles to fetch a block from memory, and the remaining bytes are transferred
at a rate of 8 bytes/cycle. Assuming the clock period of the processor is set by the L1
access time (i.e., the clock period in ns is equal to $T_{L1}$)
Memory-to-L1 transfer time = 20 cycles
Miss Penalty$_{L1}$ = 20 cycles
For L1+Victim cache

AAT (s) = T_{L1} + Miss Rate_{L1} \times Miss Penalty_{L1}

Transferring a block from the victim cache to the L1 cache occurs at a rate of 8 bytes/cycle. The block must be completely transferred before the processor resumes. The V-to-L1 transfer time in ns is equal to:

V-to-L1 transfer time = (BLOCKSIZE / 8) \times (clock period) = (BLOCKSIZE / 8) \times T_{L1}

Miss Penalty_{L1} = T_v + (1 – Miss Rate_v) \times (BLOCKSIZE / 8) \times T_{L1} + Miss Rate_v \times Miss Penalty_v

Miss Penalty_v = 20 cycles

For MRU+ L1 cache

The time for an MRU hit will be 1 cycle.
The time for a non-MRU (L1) hit will be 2 cycles.
AAT (s) = T_{MRU} + Miss Rate_{MRU} \times Miss Penalty_{MRU}

Miss Penalty_{MRU} = T_{L1} + (1– Miss Rate_{L1}) \times (BLOCKSIZE / 8) \times T_{L1} + Miss Rate_{L1} \times Miss Penalty_{L1}

Miss Penalty_{L1} = 40 cycles (this is 20 \times the cycle time of L1 cache, as in the other configurations).

Program Interface Requirements

To assure that your code can be run and graded easily, …

1. Your code must be able to run in the Unix environment on the Eos/Unity system.
2. A makefile must be provided.
3. A “make” will create three executable files, corresponding to three models. The names of three executable files are “basic”, “victim” and “MRU”.
4. The program must be executable from the command line, e.g.
   eos% basic <tracefile>
5. Every simulator must have a good interface for cache parameter setting.

Example for L1 cache:

eos% basic trace1.txt 8 1024 1 B <return>

   The first parameter is the block size, in bytes (8).
   The second parameter is the cache size, in bytes (1024),
   The third parameter is the associativity (1), and
   The fourth parameter is the write model (here, B for WBWA; use T for WTNA).

For a victim cache, there is a fifth parameter V, the size of the victim cache in number of lines.

What to hand in

1. Makefile
2. Source code
3. Project report (Microsoft Word .doc file recommended)

* No executable file is needed; just submit your *source code and makefile*.

**Grading**

0% You do not hand in (submit electronically) anything by the due date.
+10% Your Makefile works, and creates three simulators (executable files).
+10% Your simulator can read trace files from the command line and has the proper interface for parameter settings.
+50% Your simulator produces the correct output.
+30% You have done a good analysis of the experiment.