CMU 18-741: Advanced Computer Architecture

Handout 5: Cache Prefetching Competition

**** Due 10/11/2005 ****

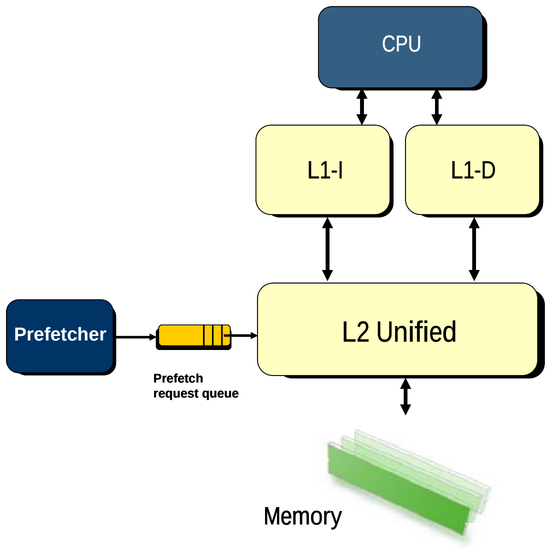
1. Introduction

Architectural innovations along with accelerating processor speeds have led to a tremendous disparity between processor and memory performance. Techniques have been proposed to allow prefetching of cache blocks in anticipation of future cache accesses. In this project, you will 1) design a basic strided prefetcher and 2) participate in a cache prefetching competition. The objective of the competition is to design a prefetcher that maximally reduces the L2 cache miss rate without undue cost in terms of extra bandwidth and storage.

This project must be completed in groups of 2 or 3 students. After forming a group, one member should email the TAs with the names and email addresses of all members by September 29. Use the subject heading "Project 2 Group".

2. Prefetcher Design

In this project, you will be designing prefetchers for a unified L2 cache. Your prefetchers will monitor memory traces (same as those in project 1) as well as cache events within the cache hierarchy (e.g., L2-U miss or hit) in order to decide when and what block addresses to prefetch. The figure below illustrates the high-level organization of the cache hierarchy. The prefetcher monitors the address and the outcome of each memory references from the CPU and issue prefetch requests via a finite-capacity request queue.



Upon receiving a request for a prefetch, a fill from memory for the prefetch address block can be brought directly into the L2 cache (and possibly evicting another block) or, optionally, a prefetch buffer that acts as an auxiliary cache to the L2. Specifically, this means that misses in the L1 caches will search the L2 and the prefetch buffer in parallel. If a hit occurs in the prefetch buffer, the line is moved over into the L2 cache. You can request a prefetch buffer of any size and one of the following three placement/replacement policies: **direct-mapped**, **fully-associative with FIFO replacement**, and **fully-associative with random replacement**. Of course you will be charge a cost according to the size and complexity of your prefetch buffer. See section 4 for more detail on these structures.

Your prefetch requests are queued in a 32-entry request queue. You are given the opportunity to inject one request following each memory reference unless the request queue is full. (**If the request queue is full, any prefetch you**

issue will be dropped.) The requests are drained at a maximum rate of one-request per **20** instructions (to simulate the effect of limited memory bandwidth). Furthermore, a request is delayed in the buffer for a minimum of 40 instructions before it is processed (to simulate the effect of memory delay---that is a prefetch must be issued 40 instructions early to eliminate a L2 miss.). If a prefetch request address hits in the L2 cache or the prefetch buffer (at the time of processing) then it has no effect.

The following are the parameters that have been fixed:

L1 Data cache	32kB, 2-way set associative, 32B blocks
L1 Instruction cache	32kB, 2-way set associative, 32B blocks
L2 Unified	512kB, 4-way set associative, 32B blocks
Request queue size	32 entries
Request queue enqueue bandwidth	Following each instruction or memory reference
Request queue dequeue bandwidth	at most one request every 20 instruction references
Request queue delay	40 instructions
# Instructions to skip	1,000,000

To ensure a consistent cache model for everyone, we provide you with our verified cache models (in the form of object files and header files) to be compiled with your prefetcher model. You may use any language as long as you can provide a wrapper that will interface to our object code developed for C. Your prefetcher takes the form of a function that is called after each memory reference with the outcome as arguments. Your function can optionally return one prefetch request per call.

All together, there are three functions your prefetcher code must implement:

int initPrefetcher (void)
long long Prefetch (int referenceType, long address, int hitL1, int hitL2, int hitPrefetch, int numPrefetchQueueEntryRemaining)
void evictionL2(long addr, int evictType)
void printPrefetcherStats ()
int dumpInterval ()

initPrefetcher( ) is called once by our code before trace processing begins. The 32-bit return value should encode whether you would like to instantiate one of the three types of prefetch buffer available to you. (See Section 4 for more detail).

prefetch( ) is called following every memory reference in the trace is processed by our cache model. The arguments provide your prefetcher with information about the cache events associated with the memory reference (e.g., hitL1, hitL2, etc.). **The 64-bit return value should encode whether a prefetch should be made, where to prefetch to (directly into L2 or into a prefetch buffer), and what is the prefetch address.**

evictionL2( ) notifies your predictor whenever a L2 eviction takes place. The arguments indicate whether the eviction was caused by a regular miss to L2 or due to a prefetch. Therefore, one expects this function to be called at most twice in between two consecutive calls to prefetch( ).

printPrefetcherStats( ) allows you to print any internal prefetcher statistics you desire. It is called at the end of the simulation.

dumpInterval( ) allows you to choose how often you wish to see cache statistics.

Additional details will be included in the source package for the skeleton code. The following code fragment gives you an idea of the sequence of events in simulation loop:

```
initPrefetcher( );
```

```
... do some initializing

long long prefetch;

// Our cache model’s main loop
while(Address = getAddress()) {
  ... process Address in our cache model
  // Notify your prefetcher if an eviction occurred due to a normal fill to
L2
  if(eviction out of L2)
      evictionL2(Address, L2evictionNormal);

  ... process eligible prefetch requests in the request queue
  // Notify your prefetcher if any evictions occurred due to a prefetch
  if(eviction out of L2 due to prefetch)
      evictionL2(Address, L2evictionByPrefetch);

  prefetch = Prefetch(instType, address, hitL1, hitL2,
                                          hitPrefetch, numPrefetchQueueFree);
  ... put your prefetch into the request queue (will be processed
      no sooner than 40 instructions from now and no sooner than 20
      instructions after the previous request is processed.)
}
```

3. Basic Prefetcher

Your first requirement is to implement a prefetcher that reduces the average global L2 miss rate. One prefetcher that will do this is a strided prefetcher with lookahead capability for memory accesses. A strided prefetcher makes its predictions by tracking memory access patterns that stride across cache blocks. For example, if a memory access pattern for the load instruction at PC=A is 0, 32, 64 … etc., the strided prefetcher should attempt to prefetch ahead of the memory reference to avoid future misses. For a more detailed description of this prefetcher, read the paper “Stride Directed Prefetching in Scalar Processors” by John W. C. Fu et al. under `/afs/ece/class/ece741/project2/prefetchPapers/`. To earn credit for this portion of the project, your prefetcher only needs to reduce the average global L2 miss rate.

4. Competition

After satisfying the basic strided prefetcher requirements, you can attempt a competition prefetcher. **Your prefetcher will be judged based on a compound metric that measures extra storage (prefetcher and prefetch buffer), extra memory bandwidth, and reduction of L2 misses.**

Cost
For any basic SRAM used (e.g., strided prefetcher history table), cost will be calculated as the total number of bits. Additional memory ports and content-addressable capability are further penalized by a multiplicative factor. For example, in the strided prefetcher, the total history storage overhead is “# of entries x (PC tag bits + Valid bit + Memory block address bits + Stride bits + …)”. An n-port SRAM is n times more expensive than a 1-port SRAM. Each content addressable memory port cost is two times more than a simple port. The number of entries in each SRAM structure must be a power of two. (You may want to divide your table into separate SRAMs if only some columns need multi-porting or content addressability.)

Along with any SRAMs you use, we also provide three different styles of prefetch buffers. You may only use one and it is placed next to the L2. If you use this, its cost is computed as follows:

Direct-mapped prefetch buffer	Number of entries x (27 – $\log_2$(number of entries) + 1)
Fully-associative prefetch buffer (FIFO replacement)	Number of entries x (27 + 1) x 2 = 56 x Number of entries
Fully-associative prefetch buffer (Random replacement)	Number of entries x (27 + 1) x 2 = 56 x Number of entries

Let

C' = total # bits for L2 cache + storage cost in prefetcher + storage cost of prefetch buffer.
C = total # bits for L2 cache (baseline cache)

Coverage
Coverage metric is related to the improvement to miss rate averaged over all benchmarks.

Let

M' = average global miss rate with a prefetcher
M = average global miss rate without a prefetcher (baseline cache)

Bandwidth
Bandwidth metric is related to the increase in bandwidth across all benchmarks. The total bandwidth accounts for the number of fill requests issued to main memory (normal L2 fills and prefetches) and the total number of writeback requests from the L2 to main memory.

Let

B' = total number of reads and writes to memory with a prefetcher
B = total number of reads and writes to memory without a prefetcher (baseline cache)

Score function
Your overall prefetcher score will be determined by following:

$$S = \frac{C}{C'}\frac{M}{M'}\frac{B}{B'}$$

The goal is to maximize *S* by minimizing the cost C', average global L2 miss rate M', and average bandwidth B' when your prefetcher is added.

Design constraints
Think carefully about the operations that your prefetcher performs. Your prefetcher should not be performing unrealistic operations such as updating more than 1 entry at a time when using a single-ported history table. Your prefetcher should not be able to "walk" a table and update 100 entries (unless the work per entry is parallelizable as a content-addressable operation.) **In general, avoid "magic" operations in your code. When in doubt, consult with the TAs.**

`/afs/ece/class/ece741/project2/prefetchPapers/` contains a collection of various prefetching designs and evaluations from real research papers. If you adapt any of these designs, please mention so in your report.

5. Requirements

The following is a list of what we expect for a completed project:

- All source files
- Makefile
- Readme file
- Design review answers (see section 7)
- Report

You may use any operating system for your project, but your project will be tested on the Greek cluster (e.g. alpha.ece.cmu.edu), so make sure your final submission works on these machines.

When building your predictors, we will be executing:

```
make basic

make competition
```

The executables generated should be "**pf_basic**", and "**pf_competition**", respectively. Like in project 1, each of these executables should run stand-alone and process the configuration file **config.default**, which will be provided in the skeleton code.

Please include a README file with your code, which should contain the names of the group members, and e-mail addresses at which you can be contacted in case there are any problems. If there are any other comments about the project (e.g. project does not work), please include it in the README file. A sample README file can be found at /afs/ece/class/ece741/project2/.

6. Grading

Your project will be graded according to the following:

40%	Prefetcher builds and runs without crashing
20%	A prefetecher that reduces L2 global cache miss rate (by any amount)
20%	Design review answers + Report
20%	Competition Score

To achieve the first 40%, the make commands (see section 5) should generate the correct executables and execute benchmarks without crashing. 20% of your grade will depend on a demonstration that your prefetcher really does reduce L2 global cache miss rate. 20% of your grade will depend on the quality of your design review answers and report.

The final 20% is based on the competition score. If you are in the top quartile, you will receive the full 20%. If you are in the second quartile, you will receive 15%. If your score is at least as good as the TAs' prefetcher score, you will receive at least 10%. If your score is below the TAs' prefetcher score, you will receive 0%. (The TAs have promised to not work too hard. The target score will be posted one week from now.) If you are the top-scoring team, you will be rewarded with a token prize and bragging rights.

7. Design Review

1. If the L1-I and L1-D caches are doubled in size, would you expect a change in your competition prefetcher's performance? Why or why not?

2. List 2 problems with the score metric we are using. Why might the score not accurately represent reality? For the two problems listed, propose a better method for computing the score.

8. Report

Please submit a report (maximum of two pages) briefly summarizing the results from your basic prefetcher and competition prefetcher. Please report L2 misses for your basic prefetcher and show your calculations for computing the score function for the competition prefetcher. **The cost of your competition prefetcher must be accurately reported (we will check!).**

Also, describe the implementation of your competition prefetcher and what the results were. Explain the hardware structures that you use and justify the operations you perform on it.

9. Submission

The final submission will be done electronically. Copy all relevant files to the directory in /afs/ece/class/ece741/project2/submission/your_email which matches the email of the student who e-mailed the group names. The directory is unlocked until the deadline so you can make changes until then if you submit early.