

EN292-10: Advanced Computer Architecture
Homework 3 — Data Prefetching
Due Tuesday November 14, 2006

This homework problem consists of adding next line (i.e., sequential) data prefetching to SIMPLESCALAR. The project is to be done in groups of 2 (or 3 as appropriate). If there are more than 2 people in your group, you will be required to add more functionality to SIMPLESCALAR proportional to the extra manpower (personpower??) available. For instance, a group of three may be required to add either stream buffers or stride prefetching capability to the code.

This is meant to be a three week project. There are three distinct parts to this project:

- Modifying SIMPLESCALAR to accommodate data prefetching;
- Running simulations using SPEC2000 benchmarks using various processor parameters to evaluate the performance benefit of prefetching.
- Reporting results.

This is not an easy assignment. Start early for best results. As a first step, you should make sure you read the paper *When Caches Aren't Enough: Data Prefetching Techniques*, available through the course webpage (found under the listing for Homework #3).

Part I: Modifying the Source Code

In this part, I will describe (in limited detail) what you need to do to add data prefetching to your simulator.

Issues to Consider

1. Prefetch instructions should proceed through the pipeline in a manner similar to load instructions, with the following exceptions:
 - The effective address should be computed based on the address of the last load miss instruction. This means that you do not need to go through the usual process of computing the effective address using an integer add functional unit.
 - Prefetch instructions to not produce any results. Therefore, these instructions should not proceed through the Writeback or Commit stages. More specifically, you do not need to queue event onto the event queue after they are issued.
 - Prefetch instructions should not cause exceptions.
2. You may have noticed through your previous SIMPLESCALAR assignment, instructions are scheduled for issue using the `ready_queue` structure. This structure is actually a linked list of reservation stations (i.e., `static struct RS_link *ready_queue;`, where the `RS_link` structure contains a link to a particular reservation station `rs`) within the RUU or LSQ.

In order to reduce confusion of what type of instruction was inserted into the `ready_queue`, I suggest handling prefetch instructions through a separate structures, say `prefetch_queue` and `PFQ`, that are defined in a similar manner as `ready_queue` and `LSQ` respectively. You will need to be careful on how you initialize and update all the fields within the `RS_link` and `PFQ` structures so they make sense within the context of prefetching. For instance, each `rs` structure defined in the `PFQ` (and

pointed to by `RS_link`) should be marked as a load and the `in_LSQ` entry in `rs` should be marked as true). In addition, the `addr` field should be valid and computed appropriately (e.g., according to the next sequential address after the load miss). See also the function `readyq_enqueue` for an example of how to insert instructions only your `prefetch_queue`.

3. When selecting instructions for issue, priority should be given to non-prefetching instructions. That is, within the `ruu_issue` subroutine, you should first process all possible instructions from the `ready_queue`. If issue slots remain (and there are still load/store functional units available), then scan the `prefetch_queue` for any ready instructions and issue them. You can follow a similar approach for scanning through the `prefetch_queue` as is done for the `ready_queue` in `ruu_issue`.
4. Remember that after prefetched instructions are issued, they do not require any further action in the `ruu_writeback` or `ruu_commit` stages. This means that you need to free the PFQ entry as well as the `prefetch_queue` entry immediately after issuing. See the `ruu_commit` subroutine for an example of how instructions are removed from the RUU. Basically, this is done by incrementing the entry's tag field and updating the `RUU_head` and `RUU_num` values.
5. Scheduled prefetch instructions should be cancelled whenever a branch misprediction is detected. Note that you can do this by incrementing the tag field in the appropriate PFQ entry (as if the instruction already issued). See, for instance, the subroutine `ruu_recover`, to get an idea of how this is handled for the RUU and LSQ when a branch misprediction is detected. Although squashing prefetch instructions after a mispredict makes sense, it is not essential for correct operation. Because of this, it may make sense to implement "prefetch squashing" after you get everything else running.
6. You should consider making some variables parameterizable so you can avoid having to recompile your code every time you need to redefine a variable. For instance, the size of your PFQ should be set by a parameter from the command line. Another command line parameter should be defined for turning prefetching on or off.
7. You should include in your code some means of measuring the effectiveness of prefetching. For instance, you should have a counter keep track of the number of times a prefetch instruction was issued. For extra credit, you can also keep track of the number of times prefetched data was actually used by the program. One way of doing this is to add an extra field in the data cache entry that is set when prefetched data is placed there and cleared when it is accessed the first time. You could also manage a sorted list in software just for gathering statistics (i.e., it doesn't have to reflect actual hardware implementation. Note that this extra credit part is actually mandatory for a group of 3 team.

There are bound to be other important issues that I haven't mentioned here. The important thing is to think through your design before hacking through the code. It is probably impossible to anticipate all issues before you begin coding, but you can save time and effort by working through as many issues as possible up front first.

Part II: Validating your design and evaluating performance improvement

Simulations should be run first just to verify that data prefetching is functioning correctly within SIMPLESCALAR. In this case you may find it particularly useful to run `test_fmATH` with the simulator. Once you make it through `test_fmATH`, then you can run experiments on the SPEC2000 benchmarks.

Some things you need to consider:

- The file `baseline.cfg` may need to be updated to reflect your new design.

- Data prefetching may be particularly effective for some applications or microarchitectural configurations, but not for others. It is up to your group to determine how the performance benefit may be dependent on the microarchitectural configuration. This is called a sensitivity analysis. For instance, prefetching is not going to help you much if your program never misses in the L1 data cache. A smaller (or less associative) cache will increase the likelihood of a miss, but how much will data prefetching be able to compensate for a small cache? For instance, will an 8K data cache with prefetching give me the same performance as a 16K data cache without prefetching? Also, does the size of the L1 data cache have any impact on the usefulness of prefetch data (i.e., the probability that it will be accessed by an instruction)?
- Your simulations should be run on at least 6 benchmarks (3 integer and 3 floating point). Keep in mind that your simulations may take longer to complete now with your added code. Make sure you fastforward and set a limit for the maximum number of instructions committed as you did for homework #2.

Part III: Project Writeup

- Your writeup for this project should be more elaborate than for homework #2. In addition, you should also include an explanation of the modifications you made to SIMPLESCALAR to implement data prefetching. As with homework #2, when reporting your simulation results, I'm not just interested in the numbers you get in your output files (though you should report these numbers in your writeup). More specifically, I would like a detailed analysis of your results, explaining why your results came out the way they did, how they compare with experimental results reported in previous research papers and why there may be discrepancies.

Include in your report an explanation of the kinds of implications your results may have on the hardware implementation of the processor. What kind of recommendations can you make for a superscalar processor design based on these results?