M5 simulator system
TDT4260 Computer Architecture
User documentation

Last modified: January 25, 2022

Contents

Chapter 1

Introduction

You are now going to write your own hardware prefetcher, using a modified version of M5, an open-source hardware simulator system. This modified version presents a simplified interface to M5's cache, enabling you to concentrate on a specific part of the memory hierarchy: a prefetcher for the second level (L2) cache.

1.1 Overview

This documentation covers the following:

- Installing and running the simulator
- Machine model and memory hierarchy
- Prefetcher interface specification
- Using the interface
- Testing and debugging the prefetcher on your local machine
- Statistics

1.2 Chapter outlines

The **first chapter** gives a short introduction, and contains an outline of the documentation.

The **second chapter** starts with the basics: how to install the M5 simulator. There are two possible ways to install and use it. The first way is to use one of the virtual machines provided, which has everything pre-installed. This is the best option for those who use Windows or Mac as their operating system of choice. For Linux enthusiasts, there is also the option of downloading a tarball, and installing a few required software packages.

The chapter then continues to walk you through the necessary steps to get M5 up and running: building from source, running with command-line options that enables prefetching, running local benchmarks and compiling and running custom test-programs.

The **third chapter** gives an overview of the simulated system, and describes its memory model. There is also a detailed specification of the prefetcher interface, and tips on how to use it when writing your own prefetcher. It includes a very simple example prefetcher with extensive comments.

The **fourth chapter** contains definitions of the statistics used to quantitatively measure prefetchers.

The **fifth chapter** gives details on how to debug prefetchers using advanced tools such as GDB and Valgrind, and how to use trace-flags to get detailed debug printouts.

Chapter 2

Installing and running M5

There are two options for obtaining and running the simulator:

1. Using one of the provided virtual machines (easiest, subsection 2.1.1)
2. Installing M5 on your own Linux machine (subsection 2.1.2)

2.1 Installation

2.1.1 Virtual machine

As part of the course, we have set up virtual machines (VMs) running Ubuntu Linux for all the students. The VMs come with the M5 simulator pre-installed. Simply log in with your regular NTNU username and password via SSH:

```
ssh <username>@tdt4260-<username>.idi.ntnu.no
```

Both Linux and Mac come with an SSH client installed by default. Windows users should download an SSH client such as PuTTY.

Once logged in, the simulator can be setup in a new 'prefetcher' directory in your home using:

```
setup_prefetcher ./prefetcher.
```

You can copy files to/from the virtual machine using the `scp` command (Linux and Mac) or PSCP (Windows, available from the PuTTY website).

To copy a file from the VM to your local machine, use:

```
scp <username>@tdt4260-<username>.idi.ntnu.no:~/prefetcher/src/prefetcher.cc
./prefetcher.cc
```

To copy a file from your local machine to the VM, use:

```
scp prefetcher.cc
<username>@tdt4260-<username>.idi.ntnu.no:~/prefetcher/src/prefetcher.cc
```

Or you can just use your NTNU home folder to comfortably edit the source files, which is also the home folder on the VM.

2.1.2 Installing M5 on Linux

First, download the modified M5 simulator and SPEC CPU2000 benchmarks suite from `https://goo.gl/fWcygV` (NB! 564MiB large file).

Software requirements (specific Debian/Ubuntu packages mentioned in parentheses):

- 3.4.6 <= g++ <= 4.8 (g++-4.8)
- Python and libpython >= 2.4 (`python` and `python-dev`)
- Scons > 0.98.1 (`scons`)
- SWIG >= 1.3.31 (`swig`)
- zlib (`zlib1g-dev`)
- m4 (`m4`)

To install all required packages in one go, issue instructions to `apt-get`:

```
sudo apt-get install g++-4.8 python-dev scons swig zlib1g-dev m4
```

The simulator framework comes packaged as a gzipped tarball. Start the adventure by unpacking it to `/opt/` using the following commands:

`sudo tar xvzf prefetcher.tgz -C /opt/`.

Now setup the prefetcher using the just extracted framework:

```
/opt/prefetcher/bin/setup_prefetcher ./prefetcher
```

The now created prefechter directory contains the sourcecode and necessary build scripts for this assignment.

2.2 Build

To build M5 you need to execute `make compile`, which executes `compile.sh` under `scripts` and includes the prefetcher from the `src` directory.

To see how M5 is builded or to make changes look up the `compile.sh` script. Inside there you can see that M5 uses the `scons` build system:

`scons -j2 ~/prefetcher/build/ALPHA_SE/m5.opt` builds the optimized version of the M5 binaries. To execute `scons` you need to be in the m5 directory which is located under `/opt/prefetcher/m5`, or on your local setup right in the framework under `prefetcher/m5`.

`-j2` specifies that the build process should build two targets in parallel. This is a useful option to cut down on compile time if your machine has several processors or cores.

For this assignment you only need to build M5 using the provided Makefile. If you want to debug M5, you should adjust `compile.sh` to build the debugging version as described in Chapter 5.

2.3 Run

Before running M5, it is necessary to specify the architecture and parameters for the simulated system. This is a nontrivial task in itself. Fortunately there is an easy way: use the included example python script for running M5 in syscall emulation mode, `/opt/prefetcher/m5/configs/example/se.py`. When using a prefetcher with M5, this script needs some extra options, described in Table 2.1.

Option	Description
`--detailed`	Detailed timing simulation
`--caches`	Use caches
`--l2cache`	Use level two cache
`--l2size=1MB`	Level two cache size
`--prefetcher=policy=proxy`	Use the C-style prefetcher interface
`--prefetcher=on_access=True`	Have the cache notify the prefetcher on *all* accesses, both hits and misses
`--cmd`	The program (an Alpha binary) to run

Table 2.1: Basic `se.py` command line options.

For an overview of all possible options to `se.py`, in the `m5` directory do

```
./build/ALPHA_SE/m5.opt
/opt/prefetcher/m5/configs/example/se.py --help
```

When combining all these options, the command line will look something like this:

```
./build/ALPHA_SE/m5.opt
/opt/prefetcher/m5/configs/example/se.py --detailed
--caches --l2cache --l2size=1MB --prefetcher=policy=proxy
--prefetcher=on_access=True
```

This command will run `se.py` with a default program, which prints out "Hello, world!" and exits. To run something more complicated, use the `--cmd` option to specify another program. See subsection 2.3.2 about cross-compiling binaries for the Alpha architecture. Another possibility is to run a benchmark program, as described in the next section.

2.3.1 CPU2000 benchmark tests

To evaluate your prefetcher against the SPEC CPU2000 benchmarks, run `make test`. This executes the `test_prefetcher.py` script from the `scripts` directory. It runs a selected suite of CPU2000 tests with your prefetcher, and compares the results to some reference prefetchers.

The per-test statistics that M5 generates are written to `output/<testname-prefetcher>/stats.txt`. The statistics most relevant for hardware prefetching are then filtered and aggregated to a `stats.txt` file in the base directory.

See chapter 4 for an explanation of the reported statistics.

Since programs often do some initialization and setup on startup, a sample from the start of a program run is unlikely to be representative for the whole program. It is, therefore, desirable to begin the performance tests after the program has been running for some time. To save simulation time, M5 can resume a program state from a previously stored checkpoint. The prefetcher framework comes with checkpoints for the CPU2000 benchmarks taken after one billion (10^9) instructions.

It is often useful to run a specific test to reproduce a bug. To run the CPU2000 tests outside of `test_prefetcher.py`, you will need to set the `M5_CPU2000` environment variable. If this is set incorrectly, M5 will give the error message "Unable to find workload". To export this as a shell variable, do

```
export M5_CPU2000=$(pwd)/data/cpu2000
```

Near the top of `test_prefetcher.py` there is a commented-out call to `dry_run()`. If this is uncommented, `test_prefetcher.py` will print the command line it would use to run each test. This will typically look like this:

```
build/ALPHA_SE/m5.opt --remote-gdb-port=0 -re
--outdir=output/ammp-user
/opt/prefetcher/m5/configs/example/se.py
```

```
--checkpoint-dir=/opt/prefetcher/lib/cp
--checkpoint-restore=1000000000
--at-instruction --caches --l2cache --standard-switch
--warmup-insts=10000000 --max-inst=10000000 --l2size=1MB
--bench=ammp --prefetcher=on_access=true:policy=proxy
```

This uses some additional command line options, these are explained in Table 2.2.

Option	Description
`--bench=ammp`	Run one of the SPEC CPU2000 benchmarks.
`--checkpoint-dir=...`	The directory where program checkpoints are stored.
`--at-instruction`	Restore at an instruction count.
`--checkpoint-restore=`n	The instruction count to restore at.
`--standard-switch`	Warm up caches with a simple CPU model, then switch to an advanced model to gather statistics.
`--warmup-insts=`n	Number of instructions to run warmup for.
`--max-inst=`n	Exit after running this number of instructions.

Table 2.2: Advanced `se.py` command line options.

2.3.2 Running M5 with custom test programs

If you wish to run your self-written test programs with M5, it is necessary to cross-compile them for the Alpha architecture. The easiest way to achieve this is to download the precompiled compiler-binaries provided by crosstool from the M5 website. Install the one that fits your host machine best (32 or 64 bit version). When cross-compiling your test program, you must use the `-static` option to enforce static linkage.

To run the cross-compiled Alpha binary with M5, pass it to the script with the `--cmd` option. Example:

```
./build/ALPHA_SE/m5.opt
/opt/prefetcher/m5/configs/example/se.py --detailed
--caches --l2cache --l2size=512kB --prefetcher=policy=proxy
--prefetcher=on_access=True --cmd /path/to/testprogram
```

Chapter 3

The prefetcher interface

3.1 Memory model

The simulated architecture is loosely based on the DEC Alpha Tsunami system, specifically the Alpha 21264 microprocessor. This is a superscalar, out-of-order (OoO) CPU that can reorder a large number of instructions, and do speculative execution.

The L1 caches are organized as a 32KiB instruction cache and a 64KiB data cache. Each cache block is 64B. The L2 cache size is 1MiB, also with a cache block size of 64B. The L2 prefetcher is notified on every access to the L2 cache, both hits and misses. There is no prefetching for the L1 cache.

The memory bus runs at 400MHz, is 64 bits wide, and has a latency of 30ns.

3.2 Interface specification

The interface the prefetcher will use is defined in a header file located at `src/interface.hh`. To use the prefetcher interface, you should include `interface.hh` by putting the line `#include "interface.hh"` at the top of your source file.

`#define`	Value	Description
`BLOCK_SIZE`	64	Size of cache blocks (cache lines) in bytes
`MAX_QUEUE_SIZE`	100	Maximum number of pending prefetch requests
`MAX_PHYS_MEM_SIZE`	$2^{28} - 1$	The largest possible physical memory address

Table 3.1: Interface `#defines`.

NOTE: All interface functions that take an address as a parameter block-align the address before issuing requests to the cache.

Function	Description
`void prefetch_init(void)`	Called before any memory access to let the prefetcher initialize its data structures
`void prefetch_access(AccessStat stat)`	Notifies the prefetcher about a cache access
`void prefetch_complete(Addr addr)`	Notifies the prefetcher about a prefetch load that has just completed

Table 3.2: Functions called by the simulator.

Function	Description
`void issue_prefetch(Addr addr)`	Called by the prefetcher to initiate a prefetch
`int get_prefetch_bit(Addr addr)`	Is the prefetch bit set for `addr`?
`int set_prefetch_bit(Addr addr)`	Set the prefetch bit for `addr`
`int clear_prefetch_bit(Addr addr)`	Clear the prefetch bit for `addr`
`int in_cache(Addr addr)`	Is `addr` currently in the L2 cache?
`int in_mshr_queue(Addr addr)`	Is there a prefetch request for `addr` in the MSHR (miss status holding register) queue?
`int current_queue_size(void)`	Returns the number of queued prefetch requests
`void DPRINTF(trace, format, ...)`	Macro to print debug information. `trace` is a trace flag (`HWPrefetch`), and `format` is a `printf` format string.

Table 3.3: Functions callable from the user-defined prefetcher.

`AccessStat` member	Description
`Addr pc`	The address of the instruction that caused the access (**P**rogram **C**ounter)
`Addr mem_addr`	The memory address that was requested
`Tick time`	The simulator time cycle when the request was sent
`int miss`	Whether this demand access was a cache hit or miss

Table 3.4: `AccessStat` members.

The prefetcher must implement the three functions `prefetch_init`, `prefetch_access` and `prefetch_complete`. The implementation may be empty.

The function `prefetch_init(void)` is called at the start of the simulation to allow the prefetcher to initialize any data structures it will need.

When the L2 cache is accessed by the CPU (through the L1 cache), the function `void prefetch_access(AccessStat stat)` is called with an argument (`AccessStat stat`) that gives various information about the access.

When the prefetcher decides to issue a prefetch request, it should call `issue_prefetch(Addr addr)`, which queues up a prefetch request for the block containing `addr`.

When a cache block that was requested by `issue_prefetch` arrives from memory, `prefetch_complete` is called with the address of the completed request as parameter.

Prefetches issued by `issue_prefetch(Addr addr)` go into a prefetch request queue. The cache will issue requests from the queue when it is not fetching data for the CPU. This queue has a fixed size (available as `MAX_QUEUE_SIZE`), and when it gets full, the oldest entry is evicted. If you want to check the current size of this queue, use the function `current_queue_size(void)`.

3.3 Using the interface

Start by studying `interface.hh`. This is the only M5-specific header file you need to include in your header file. You might want to include standard header files for things like printing debug information and memory allocation. Have a look at what the supplied example prefetcher (a very simple sequential prefetcher) to see what it does.

If your prefetcher needs to initialize something, `prefetch_init` is the place to do so. If not, just leave the implementation empty.

You will need to implement the `prefetch_access` function, which the cache calls when accessed by the CPU. This function takes an argument, `AccessStat stat`, which supplies information from the cache: the address of the executing instruction that accessed cache, what memory address was access, the cycle tick number, and whether the access was a cache miss. The block size is available as `BLOCK_SIZE`. Note that you probably will not need all of this information for a specific prefetching algorithm.

If your algorithm decides to issue a prefetch request, it must call the `issue_prefetch` function with the address to prefetch from as argument. The cache block containing this address is then added to the prefetch request queue. This queue has a fixed limit of `MAX_QUEUE_SIZE` pending prefetch requests. Unless your prefetcher is using a high degree of prefetching, the number of outstanding prefetches will stay well below this limit.

Every time the cache has loaded a block requested by the prefetcher, `prefetch_complete` is called with the address of the loaded block.

Other functionality available through the interface are the functions for getting, setting, and clearing the prefetch bit. Each cache block has one such tag bit. You are free to use this bit as you see fit in your algorithms. Note that this bit is *not* automatically set if the block has been prefetched, it has to be set manually by calling `set_prefetch_bit`. `set_prefetch_bit` on an address that is not in cache has no effect, and `get_prefetch_bit` on an address that is not in cache will always return false.

When you are ready to write code for your prefetching algorithm of choice, put it in `src/prefetcher.cc`. When you have several prefetchers, you may want to to make prefetcher.cc a symlink.

The prefetcher is statically compiled into M5. After `prefetcher.cc` has been changed, recompile with `make compile`.

3.3.1 Example prefetcher

```
/*
 * A sample prefetcher that does sequential one-block lookahead.
 * This means that the prefetcher fetches the next block _after_
 * the one that was just accessed.
 * It also ignores requests to blocks already in the cache.
 */

#include "interface.hh"

void prefetch_init(void)
{
    /*
     * Called before any calls to prefetch_access.
     * This is the place to initialize data structures.
     */

    DPRINTF(HWPrefetch, "Initialized sequential-on-access prefetcher\n");
}

void prefetch_access(AccessStat stat)
{
    /* pf_addr is now an address within the _next_ cache block */
    Addr pf_addr = stat.mem_addr + BLOCK_SIZE;

    /*
     * Issue a prefetch request if a demand miss occured,
     * and the block is not already in cache.
     */
    if (stat.miss && !in_cache(pf_addr)) {
        issue_prefetch(pf_addr);
    }
}

void prefetch_complete(Addr addr) {
    /*
     * Called when a block requested by the prefetcher has been loaded.
     */
}
```

Chapter 4

Statistics

This chapter gives an overview of the statistics by which your prefetcher is measured and ranked.

IPC instructions per cycle. Since we are using a superscalar architecture, IPC rates > 1 is possible.

Speedup Speedup is a commonly used proxy for overall performance when running benchmark tests suites.

$$\text{speedup} = \frac{\text{execution time}_{\text{no prefetcher}}}{\text{execution time}_{\text{with prefetcher}}} = \frac{IPC_{\text{with prefetcher}}}{IPC_{\text{no prefetcher}}}$$

Good prefetch The prefetched block is referenced by the application before it is replaced.

Bad prefetch The prefetched block is replaced without being referenced.

Accuracy Accuracy measures the number of useful prefetches issued by the prefetcher.

$$\text{acc} = \frac{\text{good prefetches}}{\text{total prefetches}}$$

Coverage How many of the potential candidates for prefetches were actually identified by the prefetcher?

$$\text{cov} = \frac{\text{good prefetches}}{\text{cache misses without prefetching}}$$

Identified Number of prefetches generated and queued by the prefetcher.

Issued Number of prefetches issued by the cache controller. This can be significantly less than the number of *identified* prefetches, due to duplicate prefetches already found in the prefetch queue, duplicate prefetches found in the MSHR queue, and prefetches dropped due to a full prefetch queue.

Misses Total number of L2 cache misses.

Degree of prefetching Number of blocks fetched from memory in a single prefetch request.

Harmonic mean A kind of average used to aggregate each benchmark speedup score into a final average speedup.

$$H_{avg} = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + ... + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

Chapter 5

Debugging the prefetcher

5.1 m5.debug and trace flags

When debugging M5 it is best to use binaries built with debugging support (m5.debug), instead of the standard build (m5.opt). So let us start by recompiling M5 to be better suited to debugging. Therefore, you need to edit `compile.sh` under `scripts`:

```
scons [...]  "${SCRIPT_DIR}/../build/ALPHA_SE/m5.debug"
```

Change only the target to m5.debug an leave the rest of the scons call in place, as indicated by the 3 dots.

To see in detail what's going on inside M5, one can specify enable trace flags, which selectively enables output from specific parts of M5. The most useful flag when debugging a prefetcher is `HWPrefetch`. Pass the option `--trace-flags=HWPrefetch` to M5:

```
./build/ALPHA_SE/m5.debug --trace-flags=HWPrefetch [...]
```

Warning: this can produce a lot of output! It might be better to redirect `stdout` to file when running with `--trace-flags` enabled.

5.2 GDB

The GNU Project Debugger `gdb` can be used to inspect the state of the simulator while running, and to investigate the cause of a crash. Pass GDB the executable you want to debug when starting it.

```
gdb --args build/ALPHA_SE/m5.debug --remote-gdb-port=0
-re --outdir=output/ammp-user
/opt/prefetcher/m5/configs/example/se.py
--checkpoint-dir=lib/cp --checkpoint-restore=1000000000
```

```
--at-instruction --caches --l2cache --standard-switch
--warmup-insts=10000000 --max-inst=10000000 --l2size=1MB
--bench=ammp --prefetcher=on_access=true:policy=proxy
```

You can then use the `run` command to start the executable.

Some useful GDB commands:

`run <args>`	Restart the executable with the given command line arguments.
`run`	Restart the executable with the same arguments as last time.
`where`	Show stack trace.
`up`	Move up stack trace.
`down`	Move down stack frame.
`print <expr>`	Print the value of an expression.
`help`	Get help for commands.
`quit`	Exit GDB.

GDB has many other useful features, for more information you can consult the GDB User Manual at http://sourceware.org/gdb/current/onlinedocs/gdb/.

5.3 Valgrind

Valgrind is a very useful tool for memory debugging and memory leak detection. If your prefetcher causes M5 to crash or behave strangely, it is useful to run it under Valgrind and see if it reports any potential problems.

By default, M5 uses a custom memory allocator instead of `malloc`. This will not work with Valgrind, since it replaces `malloc` with its own custom memory allocator. Fortunately, M5 can be recompiled with `NO_FAST_ALLOC=True` to use normal `malloc`:

```
scons -j2 NO_FAST_ALLOC=True [...]
```

To avoid spurious warnings by Valgrind, it can be fed a file with warning suppressions. To run M5 under Valgrind, use

```
valgrind --suppressions=lib/valgrind.suppressions
./build/ALPHA_SE/m5.debug [...]
```

Note that everything runs *much* slower under Valgrind.