arm

# Arm Research Starter Kit: System Modeling using gem5

Ashkan Tousi and Chuan Zhu
July 2017 (updated January 2022)

arm

# License

arm

# Abstract

This document is part of the Arm Research Starter Kit on System Modeling. It is intended to guide you through Arm-based system modeling using the gem5 simulator. The gem5 simulator is a modular platform for system architecture research.

We first introduce the gem5 simulator and its basics. gem5 provides two main simulation modes: the System call Emulation (SE) mode, where system services are provided by the simulator, and the Full System (FS) mode, where a complete system with devices and an operating system can be simulated.

We then introduce a High-Performance In-order (HPI) CPU model in gem5, which is tuned to be representative of a modern in-order Armv8-A implementation.

We also use some benchmarks to test the performance of our HPI system. The Stanford SingleSource workloads from the LLVM test-suite are used for benchmarking in the gem5 SE mode. In the FS mode, we use the PARSEC Benchmark Suite to examine our model. We run the PARSEC benchmarks on both single-core and multi-core HPI systems and present the results.

This document aims to help early-stage computer system researchers to get started with Arm-based system modeling using gem5. It can also be used by professional gem5 users who would like to use a modern Arm-based CPU model in their research.

arm

# Contents

arm

# Starting with Arm System Modeling in gem5

## Introduction

System-on-chip (SoC) computing systems are complex and prototyping such systems is extremely expensive. Therefore, simulation is a cost-effective way to evaluate new ideas. Modern simulators are capable of modeling various system components, such as different CPU models, interconnection topologies, memory subsystems, etc. Simulators also model the interactions among system components.

The gem5 simulator [1] is a well-known sophisticated simulator used for computer system research at both architecture and micro-architecture levels. gem5 is capable of modeling several ISAs, including Arm and x86, and supporting both 32 and 64-bit kernels and applications. It does so with enough detail such that booting unmodified Linux distributions is possible. In the case of Arm, gem5 is able to boot latest versions of the Android operating system. This Research Starter Kit will guide you through Arm-based system modeling using the gem5 simulator and a 64-bit processor model based on Armv8-A.

### Arm Cortex-A Processors

Arm Cortex-A processors [2] are powerful and efficient application processors, which are widely deployed in mobile devices, networking infrastructure, IoT devices, and embedded designs. Along with their high performance and power efficiency, Cortex-A processors provide compatibility with many popular operating systems, such as various Linux distributions, Android, IOS and Windows Phone. A full memory management unit is employed in Cortex-A models, and multiple CPU cores can work together seamlessly.

Besides the 32-bit architecture used in Armv7-A, which is referred to as AArch32 state, the later Armv8-A represents a fundamental change to the Arm architecture by introducing an optional 64-bit architecture, named the AArch64 state.

In order to enable researchers to evaluate their ideas on state-of-art Arm systems and carry out further studies and research, we provide guidance on how to simulate an Arm system in gem5.

## Using this Research Starter Kit

We show you how to get the materials required for this Research Starter Kit (RSK), and also how to build gem5 for Arm.

### Hardware and OS Requirements

In order to compile gem5 and get it working, we suggest some simple guidelines on choosing a suitable host platform.

The simulation of a 64-bit Arm platform is memory-thirsty. A modern 64-bit host platform can utilize more memory and prevent further slowdowns. Also, gem5 can require up to 1GB memory per core to compile. We also suggest using the same endianness between the host platform and the simulated Arm ISA, as cross-endian simulation is not extensively tested on gem5. Finally, gem5 runs on the Unix-like OSs.

Our experiments are carried out on an `8-core x86_64, i7-4790 CPU @ 3.60GHz` host machine running an `Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-112-generic x86_64)`. The examples have also been tested more recently on an `Ubuntu 18.04 LTS` host.

### gem5 Dependencies

For the purpose of this work, the following dependencies are needed to build gem5 on the host platform:

| Software | Version Requirement |
|---|---|
| g++ | 4.8+ |
| Python and Python-Dev | 2.7 |
| SCons | 0.98.1+ |
| zlib | recent |
| m4 | recent |
| protobuf | 2.1+ |
| pydot | recent |

protobuf [1] and pydot are not mandatory but are highly recommended for trace capture, playback support and graphical representations of the simulated system topology. Most of the software packages can be installed from the repositories of the popular Linux distributions.

## Getting the Research Starter Kit

The Arm Research Starter Kit (RSK) on System Modeling is comprised of two main parts:

- gem5: the gem5 repository is located at: `https://gem5.googlesource.com/public/gem5`
- arm-gem5-rsk: the scripts, patches and files required to run the examples and benchmarks listed in this document can be found at the Arm gem5 RSK repository: `https://github.com/arm-university/arm-gem5-rsk.git`
  - Document: this document is also part of the `arm-gem5-rsk` repository
  - Cheat Sheet: all code and examples provided in this document are listed in the Wiki section of the `arm-gem5-rsk` repository: `https://github.com/arm-university/arm-gem5-rsk/wiki`

You can clone the above repositories separately, or just download our `clone.sh` bash script and run it. It will clone the repositories into their corresponding `gem5` and `arm-gem5-rsk` directories. The script itself can also be found in the `arm-gem5-rsk` repository.

```
$ wget https://raw.githubusercontent.com/arm-university/arm-gem5-rsk/master/clone.sh
$ bash clone.sh
```

You can find more information about the structure of both repositories in their `README` files.

## Building gem5 binaries for Arm

In order to build the gem5 binaries for Arm, simply use `scons` with parameters in the `<build_dir>/ARM/<target>` format from the gem5 root directory, where `ARM` in the middle selects the simulator functionalities associated with the Arm architecture. You can also enable parallel builds by specifying the `-jN` option, where `N` is the number of concurrent build processes. Also, `gem5.opt` is the name of the gem5 binary to build, which balances between the simulation speed and the debugging functionality.

```
$ scons build/ARM/gem5.opt -jN
```

In total, five binary versions are supported, namely `.debug`, `.opt`, `.fast`, `.prof` and `.perf`, which specify the set of compiler flags used. Their differences can be found below:

| Build Name | Explanation |
|---|---|
| gem5.debug | supports run time debugging with no optimization. |
| gem5.opt | provides a fair balance between debugging support and optimization. |
| gem5.fast | provides best performance with optimization turned on, but debugging is not supported. |
| gem5.prof | is similar to "gem5.fast" but also includes instrumentation to be used by profiling tools. |
| gem5.perf | is complementary to "gem5.prof", includes instrumentation, but uses Google perftools. |

[1] `sudo apt install build-essential git m4 scons zlib1g zlib1g-dev libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev python-dev python`

## Simulating Arm in gem5

We are now ready to simulate an Arm system in gem5. First, we introduce the corresponding command line, before looking at some examples. The gem5 command line has the following format. This command calls the gem5 binary and its related options, as well as a simulation Python script and its corresponding options.

```
$ <gem5_ARM_binary> [gem5_options] <simulation_script> [script_options]
```

In order to check the available options for the gem5 binary, you may use the command `$ <gem5_ARM_binary> -h`. Also, to list the script options, the following command may be used.

```
$ <gem5_ARM_binary> [gem5_options] <simulation_script> -h
```

The aforementioned Python script in the gem5 command line sets up and executes the simulation, by setting the SimObjects in the gem5 model [3], including CPUs, caches, memory controllers, buses, etc. A selection of example simulation scripts for Arm can be found in `configs/example/arm/` within the `gem5` directory.

There are two simulation modes in gem5: the System call Emulation (SE) and the Full System (FS) simulation modes.

### System Call Emulation (SE) Mode

The SE mode simulation focuses on the CPU and memory system, and does not emulate the entire system. We just need to specify a binary file to be executed. So, we can run a simple simulation using the following command, where `starter_se.py` is the simulation script and `hello` is the binary file passed as a positional argument.

```
$ ./build/ARM/gem5.opt configs/example/arm/starter_se.py --cpu="minor" \
 "tests/test-progs/hello/bin/arm/linux/hello"
```

We can then see the gem5 output in the terminal, including some basic information about the simulation and standard output of the test program, which is "Hello world!" in this case. If the specified memory in the configuration is smaller than the memory available, you might get the following warning: `Warn: DRAM device capacity (x) does not match the address range assigned (y)`, which can be safely ignored.

You can also run a multi-core example in the SE mode, where you have to specify one program per core:

```
$ ./build/ARM/gem5.opt configs/example/arm/starter_se.py --cpu="minor" \
 --num-cores=2 "tests/test-progs/hello/bin/arm/linux/hello" "tests/test-progs/hello/bin/arm/linux/hello"
```

### Full System (FS) Mode

In the FS mode, the complete system can be modeled in an OS-based simulation environment. In order to simulate using the FS mode, we have to take some extra steps by downloading the Arm Full-System Files from the gem5 Download page. After extracting, we let gem5 know the location of our `disks` and `binaries` directories using the `M5_PATH` environment variable, i.e. the path to the `m5_binaries` directory.

Please note that the URLs specified in the example below may change in the future as new binaries are released by the gem5 project. Please consult the downloads page for URLs for the latest releases.

```
$ export M5_PATH="${PWD}/../m5_binaries"
$ mkdir -p "${M5_PATH}"
$ wget -P "${M5_PATH}" http://dist.gem5.org/dist/v21-2/arm/aarch-system-20210904.tar.bz2
$ tar -xjf "${M5_PATH}/aarch-system-20210904.tar.bz2" -C "${M5_PATH}"
$ wget -P "${M5_PATH}/disks" http://dist.gem5.org/dist/v21-2/arm/disks/ubuntu-18.04-arm64-docker.img.bz2
$ bunzip2 "${M5_PATH}/disks/ubuntu-18.04-arm64-docker.img.bz2"
$ echo "export M5_PATH=${M5_PATH}" >> ~/.bashrc
```

Before using the simulation script, it is good to check its options using the command below:

```
$ ./build/ARM/gem5.opt configs/example/arm/starter_fs.py --help
```

So, we can run the FS simulation using the following command, where `starter_fs.py` is the simulation script and `--disk-image=$M5_PATH/disks/ubuntu-18.04-arm64-docker.img` specifies the image file.

```
$ ./build/ARM/gem5.opt configs/example/arm/starter_fs.py --cpu="minor" --num-cores=1 \
          --disk-image=$M5_PATH/disks/ubuntu-18.04-arm64-docker.img --root-device=/dev/vda1
```

This should boot up Linux and start a shell on the system console associated with a TCP port. In order to reach the console, we can connect to the default port, `3456`, by using the telnet command on another terminal:

```
$ telnet localhost 3456
```

By doing so, we can interact with a simulated Arm system.

FS simulation usually takes a long time. One way to reduce the simulation time is to create a checkpoint, which is basically a snapshot of the simulation at a specific time, e.g. after the Linux has booted up. We can resume from the checkpoint at a later time, without having to wait for the booting process.

The easiest way to create a checkpoint is to run the following command on the telnet console after booting. This will generate a `cpt.TICKNUMBER` directory under the `gem5/m5out/` directory, where the `TICKNUMBER` refers to the tick value at which the checkpoint was created.

```
$ m5 checkpoint
```

In order to restore from a checkpoint, one can specify the checkpoint by passing the `--restore=m5out/cpt.TICKNUMBER/` option to the `starter_fs.py` script:

```
$ ./build/ARM/gem5.opt configs/example/arm/starter_fs.py --restore=m5out/cpt.TICKNUMBER/ --cpu="minor" \
          --num-cores=1 --disk-image=$M5_PATH/disks/ubuntu-18.04-arm64-docker.img --root-device=/dev/vda1
```

By doing so, you can quickly access the booted system and run commands on it.

When restoring the checkpoint, the `--num-cores` and `--disk-image` arguments should be the same, but the `--cpu` type may be changed.

## gem5 Statistics

After running gem5, three files will be generated in the default output directory called `m5out`. This output directory can be changed by using the option `-d`.

- Simulation parameters: the files named `config.ini` and `config.json` contain a list of each system component called SimObject as well as their parameters.
- Statistics: the file named `stats.txt` contains the statistics of each SimObject, which is detailed in the form of name, value, and description.

A typical example of the content of `stats.txt` is shown below:

```
---------- Begin Simulation Statistics ----------
sim_seconds                     0.000039               # Number of seconds simulated
sim_ticks                       38574750               # Number of ticks simulated
sim_freq                        1000000000000          # Frequency of simulated ticks
host_inst_rate                  77421                  # Simulator instruction rate (inst/s)
host_op_rate                    89376                  # Simulator op (including micro ops) rate (op/s)
host_tick_rate                  583563834              # Simulator tick rate (ticks/s)
host_mem_usage                  2245976                # Number of bytes of host memory used
host_seconds                    0.07                   # Real time elapsed on the host
sim_insts                       5116                   # Number of instructions simulated
sim_ops                         5907                   # Number of ops (including micro ops) simulated
system.voltage_domain.voltage 3.300000                 # Voltage in Volts
system.clk_domain.clock         1000                   # Clock period in ticks
```

The first column is the name of the parameter, with the corresponding value in the second column and a brief description in the third column. For example, `sim_seconds` is the total simulated seconds, `host_inst_rate` is the instruction rate of running gem5 on the host, and `sim_insts` is the number of instructions simulated. After the basic simulation parameters, the statistics related to each module of the simulated system are printed. For example, `system.clk_domain.clock` is the system clock period in ticks.

## Summary

In this chapter, we introduced the gem5 simulator as a powerful tool for modeling Arm-based systems. We also introduced our Research Starter Kit (RSK), and showed how to get all the required materials, build gem5 and run simple examples.

gem5 supports two simulation modes: I) System call Emulation (SE), where it only simulates the program, traps system calls made to the host and emulates them, and II) Full System (FS), where gem5 simulates a complete system, which provides an operating system based simulation environment.

So far, we have run basic SE and FS test examples. In the following chapters, we will cover more advanced topics, such as CPU types, memory accesses, and core configuration scripts. Moreover, we will introduce an Arm-based high performance in-order CPU, and build a system around it. We will then demonstrate how to run benchmarks on top of the simulated system and collect the results.

# A High-Performance In-order (HPI) Model

## Introduction

gem5 is such a powerful tool that it can be configured to model the latest CPU models of various architectures [4], namely Alpha, Arm [5], x86, etc. In this chapter, we focus on the modeling of a High-Performance In-order (HPI) CPU, which follows the Arm architecture using gem5.

Before prototyping a certain CPU model, let us get a better understanding of gem5. The gem5 simulator was briefly introduced in its original publication [4] with the focus on its overall goals, design features and simulation capabilities. The authors of the aforementioned paper also created tutorial slides [6] for an older release of gem5. The official documentation of gem5 [7] can be used for further details. Together with the University of Wisconsin-Madison, the gem5 community provide several courses on gem5 and computer architectures, as well as a tutorial for gem5 beginners [3, 8]. Finally, an introduction to a distributed version of gem5 can be found in [9], covering more advanced topics.

gem5 is gaining popularity for prototyping novel ideas in computer architecture research. For instance, the authors of [5] elaborated on the micro-architectural simulation of in-order and out-of-order Arm microprocessors with gem5. The authors of [10] proposed a fast, accurate and modular DRAM controller model for gem5 full-system simulation, while some other works [11, 12, 13] focused on GPU modeling using gem5.

## In-order CPU Models in gem5

gem5 provides different CPU models which suit different requirements, namely simple CPUs, detailed in-order CPUs, detailed out-of-order CPUs, and KVM-based CPUs. In this section, we focus on the in-order CPU models provided in gem5. We start with the simplified CPU models and the MinorCPU, which is a more realistic in-order CPU model.

The simplified CPU models are functional in-order models for fast simulation, while some of the details are ignored. These models can be used for simple tests, or for fast-forwarding to the regions of interest during a simulation. The base class for the simplified CPU models is named BaseSimpleCPU, which defines basic functions for handling interrupts, fetch requests, pre-execute setup and post-execute actions. Since it implements a simplified CPU model, it simply advances to the next instruction after executing the current one, with no instruction pipelining. In contrast, the detailed CPU models are more realistic and certainly more time-consuming to simulate.

Here we introduce two models derived from BaseSimpleCPU, namely AtomicSimpleCPU and TimingSimpleCPU, with the inheritance structure shown in Fig 1. On top of that, we will focus on the more comprehensive MinorCPU model, and then introduce our approach to build a High-Performance In-order (HPI) CPU based on it.

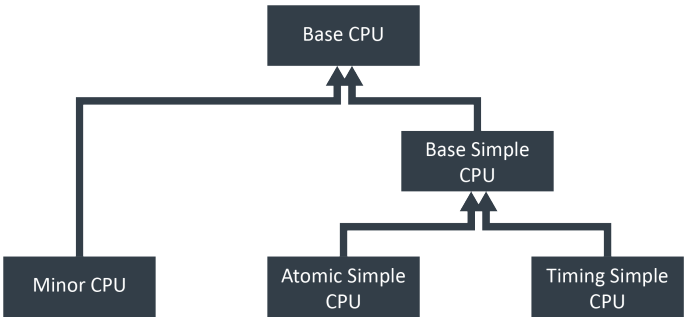



*Figure 1: Simple / Minor CPU classes defined in gem5*

### Memory Access

CPU models depend on memory access and there are three types of access supported in gem5, namely timing, atomic and functional methods.

- Atomic access is the fastest of the three, and completes a transaction in a single function call. The function `sendAtomic()` is used for the atomic requests, and the requests complete before `sendAtomic()` returns. It models state changes (cache fills, coherence) and calculates the approximate latency without contention or queuing delay, making it suitable for loosely-timed simulation (fast-forwarding) or warming caches.
- Functional access is similar to Atomic access in that it completes a transaction in a single function call and the access happens instantaneously. Additionally, functional accesses can coexist in the memory system with Atomic or Timing accesses [14]. Therefore, functional accesses are suitable for loading binaries and debugging.
- Timing access is the most realistic access method and is used for approximately-timed simulation, which considers the realistic timing, and models the queuing delay and resource contention [14]. Timing and Atomic accesses cannot coexist in the system.

## Implementation of the gem5 In-order CPU Models

Based on the above memory access models, let us introduce a few in-order CPU models in gem5.

### AtomicSimpleCPU

As shown in Fig 2, the AtomicSimpleCPU uses Atomic memory accesses. In gem5, the AtomicSimpleCPU performs all operations for an instruction on every CPU `tick()` and it can get a rough estimation of overall cache access time using the latency estimates from the atomic accesses [1]. Naturally, AtomicSimpleCPU provides the fastest functional simulation, and is used for fast-forwarding to get to a Region Of Interest (ROI) in gem5.

### TimingSimpleCPU

The TimingSimpleCPU adopted Timing memory access instead of the simple Atomic one. This means that it waits until memory access returns before proceeding, therefore it provides some level of timing. TimingSimpleCPU is also a fast-to-run model, since it simplifies some aspects including pipelining, which means that only a single instruction is being processed at any time. Each arithmetic instruction is executed by TimingSimpleCPU in a single cycle, while memory accesses require multiple cycles [8, 3]. For instance, as shown in Fig 2, the TimingSimpleCPU calls `sendTiming()` and will only complete fetch after getting a successful return from `recvTiming()`.

### MinorCPU

We need a more comprehensive and detailed CPU model in order to emulate realistic systems, therefore we should utilize the detailed in-order CPU models available in gem5. In older versions of gem5, a model named InOrder CPU was capable of doing the job for us, but now there is a new model called MinorCPU.

The MinorCPU is a flexible in-order processor model which was originally developed to support the Arm ISA, and is applicable to other ISAs as well. As shown in Fig 3, MinorCPU has a fixed four-stage in-order execution pipeline, while having configurable data structures and execute behavior; therefore it can be configured at the micro-architecture level to model a specific processor.

The four-stage pipeline implemented by MinorCPU includes fetching lines, decomposition into macro-ops, decomposition of macro-ops into micro-ops and execute. These stages are named Fetch1, Fetch2, Decode and Execute, respectively. The pipeline class controls the cyclic tick event and the idling (cycle skipping) [15]. We briefly introduce these pipeline stages.

- Fetch1: two pipeline stages are used for instruction fetch (and two cycles for access to the data cache) in order to give us time to convert a virtual address to a physical one, check for a hit in the cache, read the data back and do operations like branch prediction. If these all had to be done in a single cycle, rather than two, the maximum frequency of the processor would be much slower. Fetch1 fetches cache lines or partial cache lines from the ICache, and handles the stages of translating the address of a line fetch (via the TLB). Fetch1 then passes the fetched lines

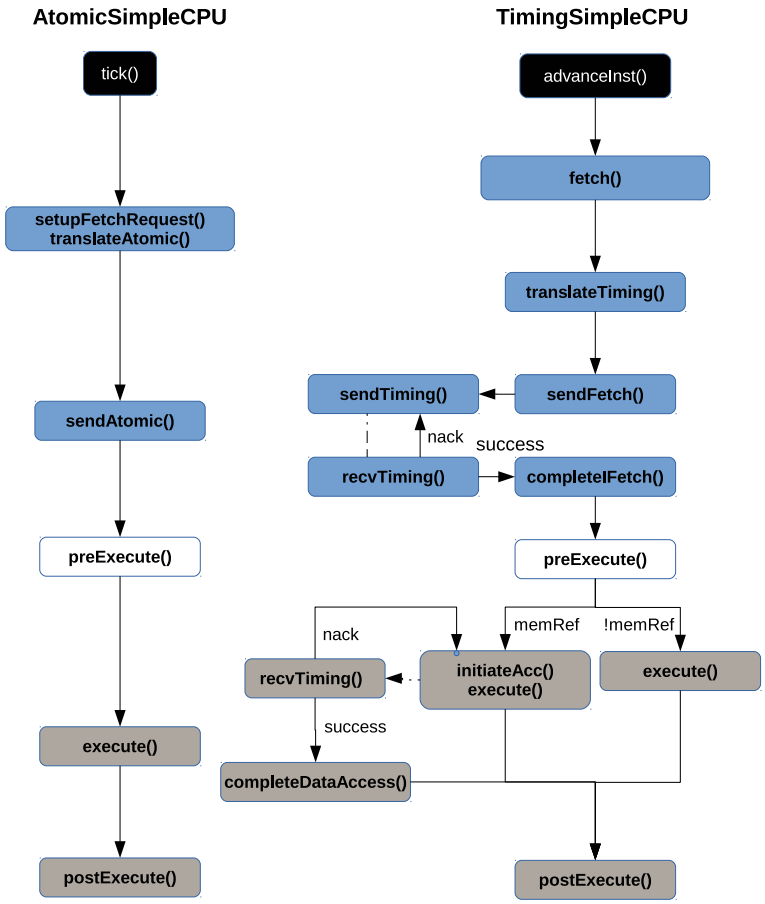



*Figure 2: Comparison of the CPU execution flows for the simplified CPU models in gem5*

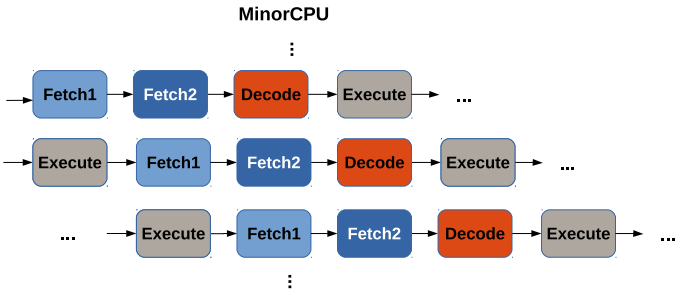



*Figure 3: Pipeline stages of the MinorCPU model in gem5*

to Fetch2, and the latter will decompose them to instructions. Some configurations related to the Fetch1 stage are listed in Table 1 for the HPI core, which uses a similar pipeline. For example, the parameter `fetch1LineSnapWidth` sets the data snap size, which controls the size to be fetched from the ICache. `fetch1LineWidth` controls the fetch size of subsequent lines. Some other parameters sets the delays. For instance, `fetch1ToFetch2BackwardDelay` models the delay caused by branch predictions from Fetch2 to Fetch1.

- Fetch2: firstly, puts the line from Fetch1 into the input buffer and divides the data in the buffer into individual instructions. These instructions are then packed into a vector of instructions and passed on to Decode stage. If any fault is found in the input line or a decomposed instruction, packing instructions can be aborted early. As shown in Table 1, the parameter `decodeInputWidth` sets the number of instructions, which can be packed into the output of Fetch2 stage per cycle. The parameter `fetch2CycleInput` controls whether Fetch2 can take more than one entry

from the input buffer per cycle.

The branch predictor is included in the Fetch2 stage, which predicts branches for all the control instructions. The predicted branch instruction is then packed into the Fetch2's output vector, the prediction sequence number is incremented, and the branch is communicated to Fetch1 [15]. Branches (and corresponding instructions) processed by Execute will generate branch outcome data, which is sent forward to Fetch1 and Fetch2. Fetch1 and Fetch2 will uses it for corresponding updates. In gem5, a two-level branch predictor can be implemented by inheriting the `tournamentBP` base class, with the parameters listed in Table 2. These parameters characterize the local and global predictor, including the number of counters, number of bits of those counters, the history table size, etc. The local and global predictors use their own history tables to index into their table of counters. A choice predictor chooses between the two. The global history register is speculatively updated, the rest are updated upon branches committing or misspeculating.

- Decode: decomposes the instructions from Fetch2 into micro-ops and outputs them as an instruction vector for the execute stage. By setting the corresponding parameters `executeInputWidth` and `decodeCycleInput`, as shown in Table 1, we can set the number of instructions packed into the output per cycle, and whether to take more then one entry in the input buffer per cycle [15].

- Execute: takes a vector of instructions from the Decode stage and provides the instruction execution and memory access functionalities. The Execute stage for an instruction can take multiple cycles and its precise timing can be modeled by a functional unit pipeline FIFO. Similarly, by setting the corresponding parameters `executeInputWidth` and `executeCycleInput`, as shown in Table 1, we can set the number of instructions in the input vector, and whether or not to examine more than one instruction vector. Additionally, `executeInputBufferSize` decides the depth of the input buffer. The Execute stage includes the Functional Units (FU), which model the computational core of the CPU. By configuring the `executeFuncUnits`, as shown in Table 3, we can define functional units associated with our core. Each functional unit models a number of instruction classes, the delay between instruction issues, and the delay from instruction issue to commit [15]. Since Execute utilizes a Load Store Queue (LSQ) for memory reference instructions, there are some parameters that configure LSQ, as shown in Table 1.

## High-Performance In-order (HPI) CPU

By introducing the basic CPU models in gem5, especially MinorCPU, we have paved our way to build a High-Performance In-order CPU based on the Arm architecture, and we name it HPI. The HPI CPU timing model is tuned to be representative of a modern in-order Armv8-A implementation. In Fig 4, we portrayed the major components included in our model. We introduce each of these components more in detail.

### Processor Pipeline

The pipeline of our HPI CPU uses the same four-stage model as the MinorCPU described in Section "MinorCPU". Specifically, the parameters of our HPI CPU can be found in Table 1, and the parameters for the HPI CPU branch predictor are listed in Table 2.

### Interrupt Controller

In the Arm architecture, a Generic Interrupt Controller (GIC) supports routing of software generated, private and shared peripheral interrupts between cores in a multi-core system. The GIC enables software to mask, enable and disable interrupts from individual sources, to prioritize (in hardware) individual sources and to generate software interrupts.

Interrupts are identified in the software by a number, called an interrupt ID. An interrupt ID uniquely corresponds to an interrupt source. Software can use the interrupt ID to identify the source of interrupt and to invoke the corresponding handler to service the interrupt.

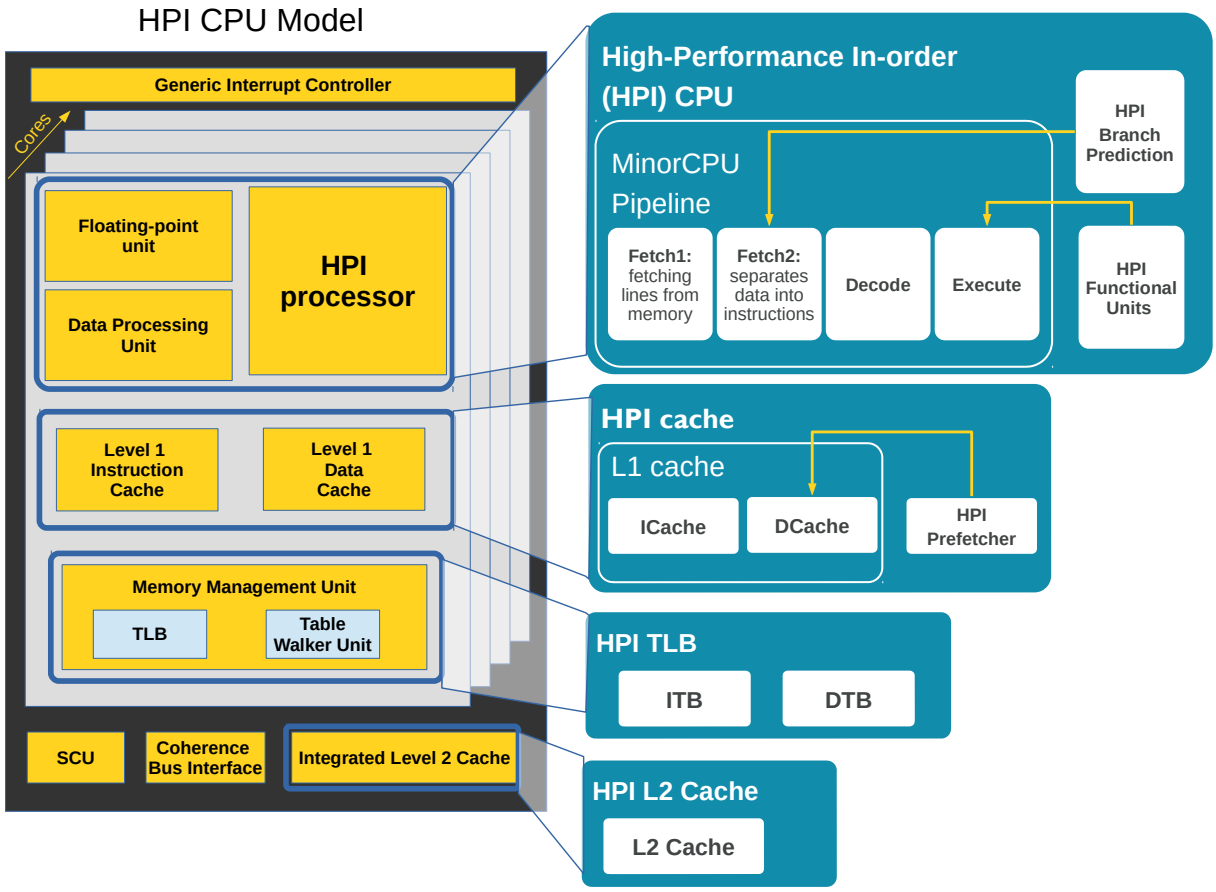



*Figure 4: Building blocks of the Arm High-performance In-order CPU in gem5*

The interrupt controller is implemented in gem5 under the path `src/arch/[ISA]/interrupts.hh`. To include the interrupt controller in the system, we may use the derived CPU models of BaseCPU. The interrupt controller is initialized by `BaseCPU::createInterruptController()` in the Python script `src/cpu/BaseCPU.py`.

### Floating-Point Unit and Data Processing Unit

The Data Processing Unit (DPU) shown in Fig 4 holds most of the program-visible state of the processor, such as general-purpose registers and system registers. It provides configuration and control of the memory system and its associated functionality, and decodes and executes instructions.

The Floating-Point Unit (FPU) includes the floating-point register file and status registers. It performs floating-point operations on the data held in the floating-point register file.

The DPU and FPU can be modeled by the Functional Units (FU) in the Execute stage of the MinorCPU pipeline (Section "MinorCPU"). Specifically, we can model them by configuring the `executeFuncUnits`, as shown in Table 3.

### Level 1 Caches

In the architecture used by our HPI core, we have separate instruction and data buses, hence an instruction cache (ICache) and a data cache (DCache). So, there are distinct instruction and data L1 caches backed by a unified L2 cache.

Typically, the instruction and data caches are configured independently during implementation, to sizes of 8KB, 16KB, 32KB, or 64KB. In our case, the L1 instruction and data memory system has the key features listed in Table 4. For instance, we characterize the delay by hit latency and response latency. We also specify the number of Miss Status Holding Registers (MSHR). Additionally, we override the associativity and size of the L1 caches.

*Table 1: Parameter specifications for the pipeline of the HPI CPU*

| Module Name | Element Name | Element Value |
|---|---|---|
| Fetch1 stage | fetch1LineSnapWidth | 0 |
| | fetch1LineWidth | 0 |
| | fetch1FetchLimit | 1 |
| | fetch1ToFetch2ForwardDelay | 1 |
| | fetch1ToFetch2BackwardDelay | 1 |
| Fetch2 stage | fetch2InputBufferSize | 2 |
| | fetch2ToDecodeForwardDelay | 1 |
| | fetch2CycleInput | True |
| Decode stage | decodeInputBufferSize | 3 |
| | decodeToExecuteForwardDelay | 1 |
| | decodeInputWidth | 2 |
| | decodeCycleInput | True |
| Execute stage | executeInputWidth | 2 |
| | executeCycleInput | True |
| | executeIssueLimit | 2 |
| | executeMemoryIssueLimit | 1 |
| | executeCommitLimit | 2 |
| | executeMemoryCommitLimit | 1 |
| | executeInputBufferSize | 7 |
| | executeMaxAccessesInMemory | 2 |
| | executeLSQMaxStoreBufferStoresPerCycle | 2 |
| | executeLSQRequestsQueueSize | 1 |
| | executeLSQTransfersQueueSize | 2 |
| | executeLSQStoreBufferSize | 5 |
| | executeBranchDelay | 1 |
| | exeuteSetTraceTimeOnCommit | True |
| | executeSetTraceTimeOnIssue | False |
| | executeAllowEarlyMemoryIssue | True |
| | enableIdling | True |

*Table 2: Branch Prediction parameter specifications for HPI*

| Module Name | Element Implementation | Base Class | Element Value |
|---|---|---|---|
| Branch Prediction | branchPred = HPI_BP(numThreads = Parent.numThreads) | TournamentBP | localPredictorSize = 64 |
| | | | localCtrBits = 2 |
| | | | localHistoryTableSize = 64 |
| | | | globalPredictorSize = 1024 |
| | | | globalCtrBits = 2 |
| | | | choicePredictorSize = 1024 |
| | | | choiceCtrBits = 2 |
| | | | BTBEntries = 128 |
| | | | BTBTagSize = 18 |
| | | | RASSize = 8 |
| | | | instShiftAmt = 2 |

Our data cache implements an automatic prefetcher that monitors cache misses in the core. When a data access pattern is detected, the automatic prefetcher starts linefills in the background. The prefetcher recognizes a sequence of data cache misses at a fixed stride pattern that lies in four cache lines, plus or minus. Any intervening stores or loads that hit in the data cache do not interfere with the recognition of the cache miss pattern. Our configurations for the prefetcher is listed in Table 5.

### Memory Management Unit

An important function of the Memory Management Unit (MMU) in Fig 4 is to enable the system to run multiple tasks as independent programs running in their own private virtual memory space. They do not need any knowledge of the physical

*Table 3: Functional Units parameter specifications for HPI*

| Module Name | Element Implementation | Base Class | Element Value |
|---|---|---|---|
| Functional units | executeFuncUnits =HPI_FUPool() | MinorFUPool | HPI_IntFU() |
| | | | HPI_Int2FU() |
| | | | HPI_IntMulFU() |
| | | | HPI_IntDivFU() |
| | | | HPI_FloatSimdFU() |
| | | | HPI_MemFU() |
| | | | HPI_MiscFU() |

*Table 4: L1 Cache parameter specifications for HPI*

| Module Name | Element Implementation | Base Class | Element Value |
|---|---|---|---|
| Level 1 Cache | HPI_ICache, HPI_DCache | Cache | tag_latency = 1 |
| | | | data_latency = 1 |
| | | | response_latency = 1 |
| | | | HPI_ICache.mshrs = 2 |
| | | | HPI_DCache.mshrs = 4 |
| | | | tgts_per_mshr = 8 |
| | | | size = '32kB' |
| | | | HPI_ICache.assoc = 2 |
| | | | HPI_DCache.assoc = 4 |

*Table 5: L1 Data Cache Prefetcher parameter specifications for HPI*

| Module Name | Element Implementation | Element Value |
|---|---|---|
| Level 1 DCache Prefetcher | prefetcher = StridePrefetcher() | queue_size=4 |
| | | degree=4 |

memory map of the system, that is, the addresses that are actually used by the hardware, or about other programs that might execute at the same time.

The Translation Lookaside Buffer (TLB) is a cache of recently accessed page translations in the MMU. For each memory access performed by the processor, the MMU checks whether the translation is cached in the TLB. If the requested address translation causes a hit within the TLB, the translation of the address will be immediately available. Each TLB entry typically contains not just physical and virtual addresses, but also attributes such as memory type, cache policies, access permissions, the Address Space ID (ASID), and the Virtual Machine ID (VMID). If the TLB does not contain a valid translation for the virtual address issued by the processor, known as a TLB miss, an external translation Table Walk or lookup is performed. Dedicated hardware within the MMU enables it to read the translation tables in memory. The newly-loaded translation can then be cached in the TLB for possible reuse if the translation table walk does not result in a page fault. The exact structure of the TLB differs between implementations of the Arm processors. In our HPI core model, we used the `ArmTLB` base model in gem5, and used the parameters listed in Table 6 for our data and instruction TLBs.

*Table 6: TLB in MMU parameter specifications for HPI*

| Module Name | Element Implementation | Base Class | Element Value |
|---|---|---|---|
| TLB | itb = HPI_ITB() | ArmTLB | size = 256 |
| | dtb = HPI_DTB() | | size = 256 |

### Level 2 memory system

The level 2 memory system consists of the Snoop Control Unit (SCU), the Coherence Bus Interface and the level 2 cache.

- Snoop Control Unit: the SCU maintains coherency between the individual data caches in the processor, and it contains buffers that can handle direct cache-to-cache transfers between cores without having to read or write any data to the external memory system. Cache line migration enables dirty cache lines to be moved between cores, and there is no requirement to write back transferred cache line data to the external memory system. Each core has tag and dirty RAMs that contain the state of the cache line. Rather than sending a snoop request to each core, the SCU contains a set of duplicate tags that allow it to check the contents of each L1 data cache.
- Level 2 Cache: data is allocated to the L2 cache only when evicted from the L1 memory system. The only exceptions to this rule are for memory marked with the inner transient hint, or for non-temporal loads, that are only ever allocated to the L2 cache. The L2 cache is 16-way set associative. The L2 cache tags are looked up in parallel with the SCU duplicate tags. If both the L2 tag and SCU duplicate tag hit, a read accesses the L2 cache in preference to snooping one of the other cores. In our Python code for the HPI core model, the parameter specifications for the L2 cache are listed in Table 7. The delays are characterized by hit latency and response latency. We also specify the number of MSHRs and override the associativity and size of the L2 caches.

*Table 7: L2 Cache parameter specifications for HPI*

| Module Name | Element Implementation | Base Class | Element Value |
|---|---|---|---|
| L2 Cache | HPI_L2 | Cache | data_latency = 13 |
| | | | tag_latency = 13 |
| | | | response_latency = 5 |
| | | | mshrs = 4 |
| | | | tgts_per_mshr = 8 |
| | | | size = '1024kB' |
| | | | assoc = 16 |
| | | | write_buffers = 16 |

### Python Configuration Script for the HPI Core

In order to create our HPI model, we need to add all the configurations (including the parameter specifications mentioned above) into a Python configuration script. The source code for the HPI core model, which is named `HPI.py` is located at `gem5/configs/common/cores/arm/`.

## System Modeling in the SE Mode

We introduced the configuration parameters for the HPI core. The MinorCPU models uses the parameters specified in the `gem5/configs/example/arm/devices.py` script, such as `devices.L1I`, `devices.L1D`, and `devices.L2`. In order to build a system in the gem5 SE mode, we need to set up the cache, memory and other modules which are associated with the CPU models, and properly connect them.

In our `starter_se.py` simulation script, located in the `gem5/configs/example/arm` directory, we provide three different CPU types, namely AtomicSimpleCPU, MinorCPU and the HPI CPU:

```
cpu_types = {
    "atomic" : ( AtomicSimpleCPU, None, None, None, None),
    "minor" : (MinorCPU,
               devices.L1I, devices.L1D,
               devices.L2),
    "hpi" : ( HPI.HPI,
              HPI.HPI_ICache, HPI.HPI_DCache,
              HPI.HPI_L2)
}
```

Since gem5 does not simulate caches for the Atomic models, the cache-related options for the AtomicSimpleCPU are set to `None`.

We also add several arguments to the parser. When using the simulation script, you can use the command line arguments below to change the default values:

```
def addOptions(parser):
    parser.add_argument("commands_to_run", metavar="command(s)", nargs='*',
                        help="Command(s) to run")
    parser.add_argument("--cpu", type=str, choices=cpu_types.keys(),
                        default="atomic",
                        help="CPU model to use")
    parser.add_argument("--cpu-freq", type=str, default="4GHz")
    parser.add_argument("--num-cores", type=int, default=1,
                        help="Number of CPU cores")
    parser.add_argument("--mem-type", default="DDR3_1600_8x8",
                        choices=MemConfig.mem_names(),
                        help = "type of memory to use")
    parser.add_argument("--mem-channels", type=int, default=2,
                        help = "number of memory channels")
    parser.add_argument("--mem-ranks", type=int, default=None,
                        help = "number of memory ranks per channel")
    parser.add_argument("--mem-size", action="store", type=str,
                        default="2GB",
                        help="Specify the physical memory size")
    return parser
```

## Simulating the HPI Model in the SE mode

Considering that you have built the gem5 binaries for Arm using the command introduced in Section "Building gem5 binaries for Arm", we can test-run our HPI model in the SE mode by running the command below, like before. We configure the CPU model using `--cpu` (`hpi` in this case) and the number of cores by setting the `--num-cores` argument.

```
$ ./build/ARM/gem5.opt configs/example/arm/starter_se.py --cpu="hpi" --num-cores=1 \
 "tests/test-progs/hello/bin/arm/linux/hello"
```

Like before, we will see the "Hello world!" output in the console.

## System Structure for Different CPU Models

After running an SE simulation, the system structure diagram can be found in `run_scripts/m5out/config.dot.svg` (unless one specifies a different output directory). Using such diagrams, we show the differences between the three CPU types used for the SE simulation.

- AtomicSimpleCPU: the system structure for the Atomic model in the gem5 SE simulation mode is depicted in Fig. 5, from which we can see that the simplified atomic memory accesses is used, and caches are ignored.
- MinorCPU: the system structure for the MinorCPU model in the SE mode is depicted in Fig. 6. In addition to the modules in the Atomic model, this system uses walk caches as well as L1 and L2 caches.
- HPI: the system structure for the HPI model in Fig. 7 is quite similar to the MinorCPU in the SE mode, although the parameter values are different.

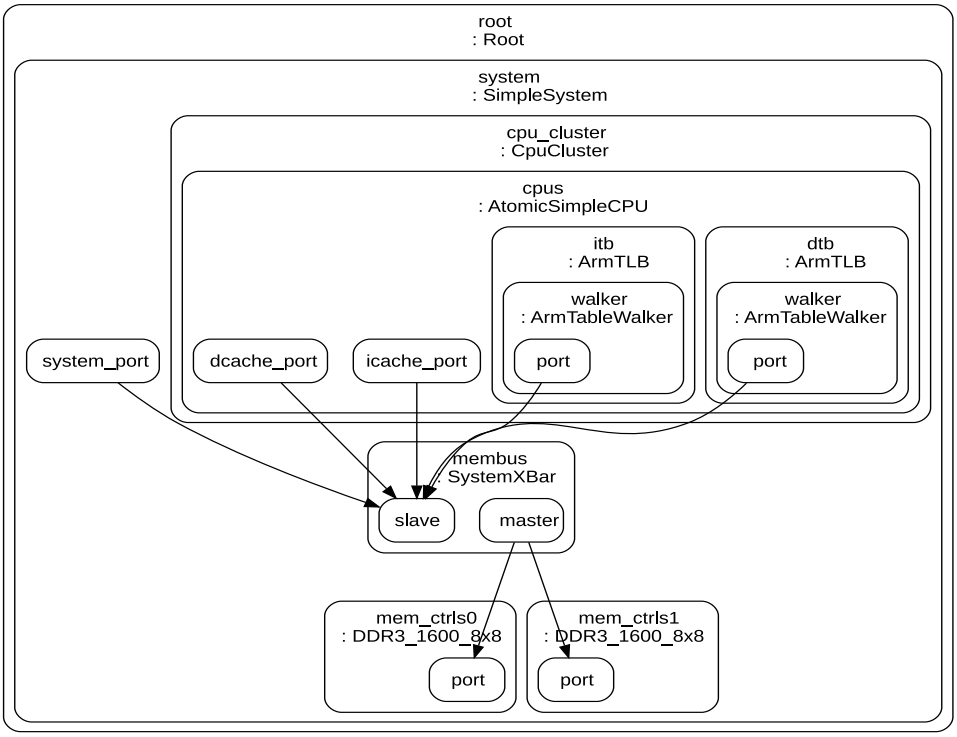



*Figure 5: System structure for the Atomic model in the SE mode*

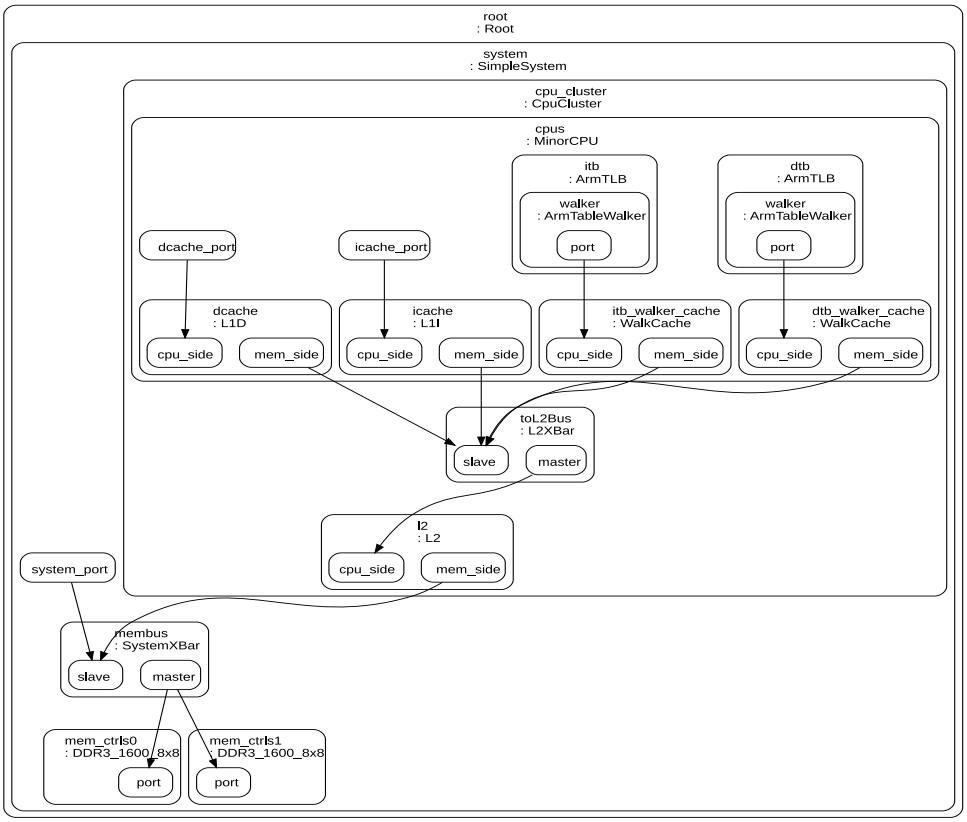



*Figure 6: System structure for the Minor model in the SE mode*

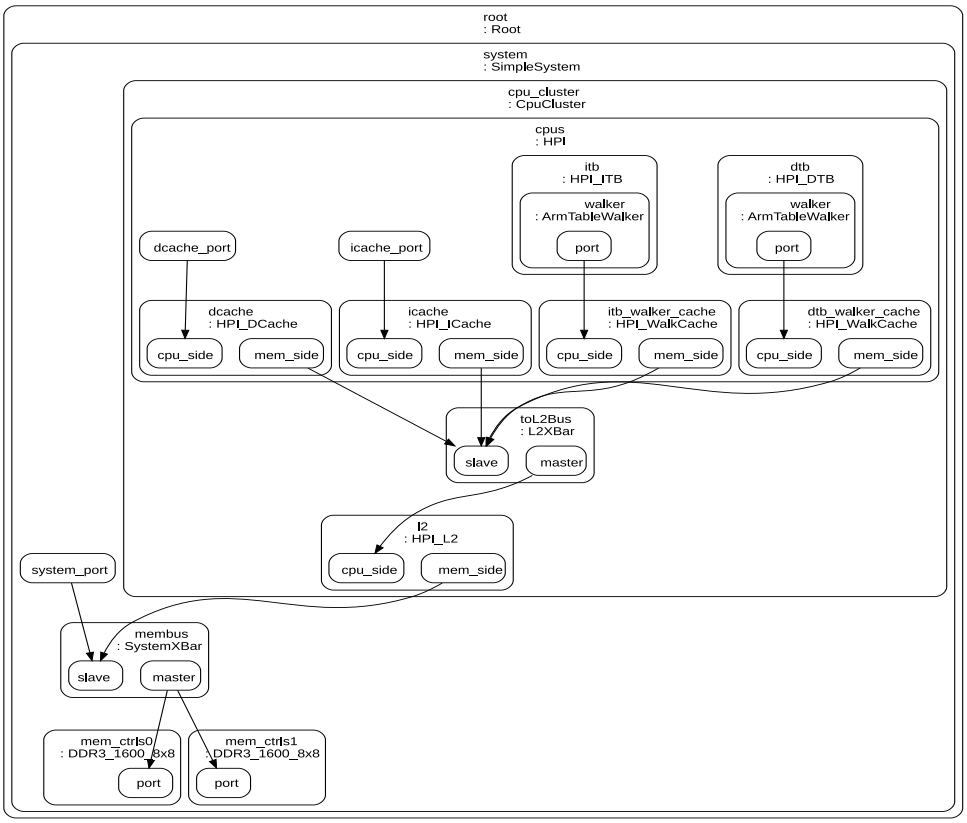



*Figure 7: System structure for the HPI model in the SE mode*

## System Modeling in the FS Mode

In the FS simulation, we can run unmodified OS binaries, simulate all devices and get more realistic results. We need a disk image, containing all the bytes and structure of a storage device, just as you would find on a real hardware device. Also, since gem5 skips the bootloader part of the boot and loads the kernel directly into the simulated memory, we can provide the kernel separately, without having to modify the disk image.

In our `starter_fs.py` simulation script located in the `gem5/configs/example/arm` directory, we provide the following information. As described in Section "Full System (FS) Mode", we first need to let gem5 know the location of our disk image and kernel binary (taken relative to `${M5_PATH}/disks` and `${M5_PATH}/binaries` respectively):

We provide additional options to specify the disk image and kernel binary. Also, a runscript can be specified by using the `--script` option. Runscripts are bash scripts that are automatically executed after Linux boots.

```
def addOptions(parser):
    parser.add_argument("--dtb", type=str, default=None,
                        help="DTB file to load")
    parser.add_argument("--kernel", type=str, default=default_kernel,
                        help="Linux kernel")
    parser.add_argument("--disk-image", type=str,
                        default=default_disk,
                        help="Disk to instantiate")
    parser.add_argument("--script", type=str, default="",
                        help = "Linux bootscript")
```

### Simulating the HPI Model in the FS mode

If you have already built the gem5 binaries for Arm, downloaded the disk image, and set the `M5_PATH` variable, as described in "Full System (FS) Mode", then you are ready to test-run the HPI model in the FS mode (you do not have to specify the disk image, as we are using the default one):

```
$ ./build/ARM/gem5.opt configs/example/arm/starter_fs.py --cpu="hpi" --num-cores=1 \
          --disk-image=$M5_PATH/disks/ubuntu-18.04-arm64-docker.img --root-device=/dev/vda1
```

We can then interact with the simulated system using `telnet localhost 3456`. You can also check the `m5out/system.terminal` to see the details of the booting process.

### System Structure for the HPI Model

As shown in Fig 8, the system structure for the HPI model in the FS mode contains additional details and components required to model the entire system.

## Summary

In this chapter, we briefly introduced the in-order CPU models and different memory access types in gem5. We described the pipeline stages of the MinorCPU model, and then introduced our HPI CPU model and its major components. The HPI CPU timing model is tuned to be representative of a modern in-order Armv8-A implementation.

Finally, we used our SE and FS simulation scripts, namely `starter_se.py` and `starter_fs.py` to run test examples and compare the system structures of different CPUs. In the next chapter, we will run some benchmarks on our HPI model in both simulation modes.

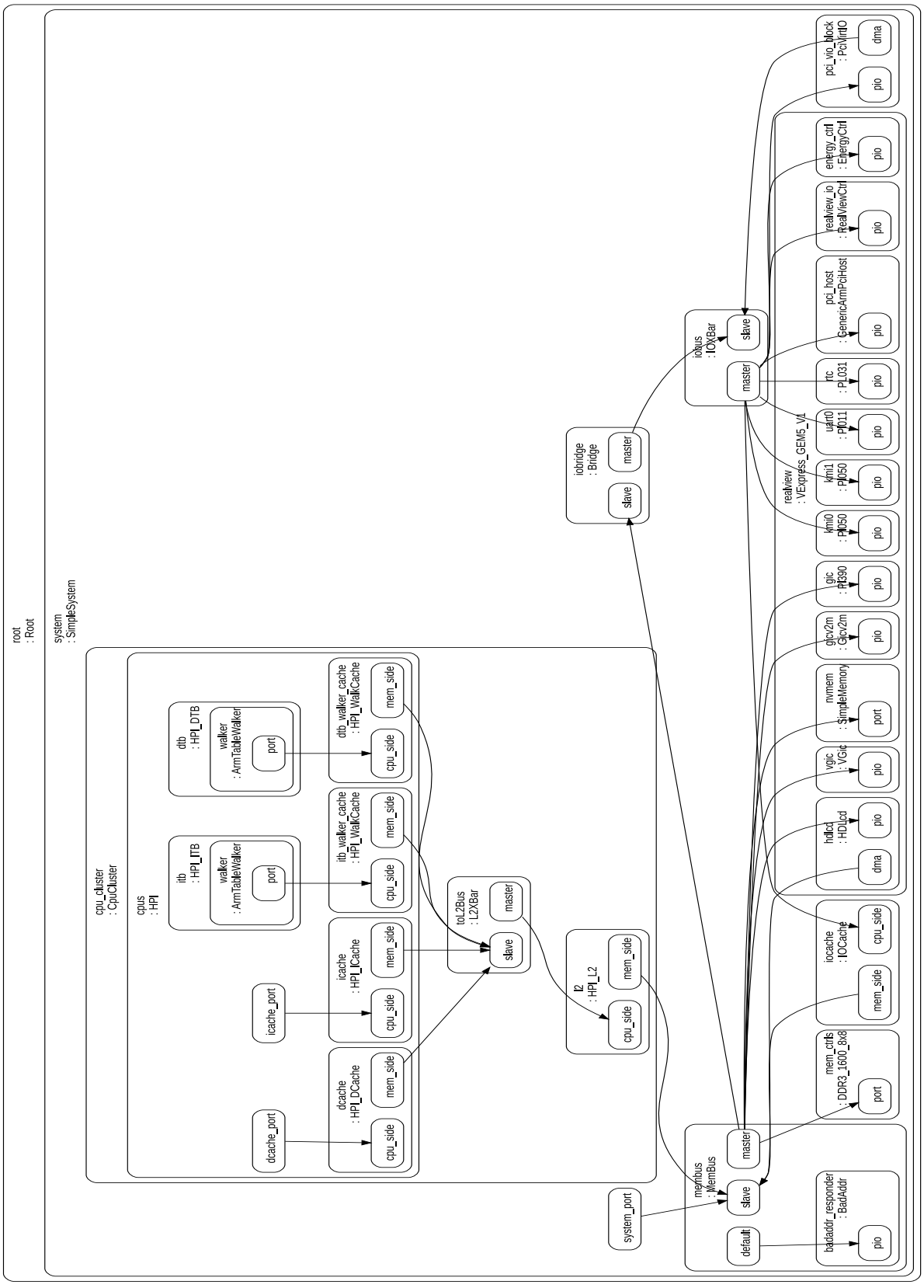



*Figure 8: System structure for the HPI model in the FS mode*

arm

# Running Benchmarks on the HPI Model

## Introduction

Given the system model equipped with our HPI CPU, let us test out the system performance using some well-known benchmarks.

System evaluation using gem5 has been the focus of many other related works. For example, the accuracy evaluation of gem5 simulator system is carried out in [16], the sources of error in full-system simulation using gem5 is discussed in [17] and the authors of [18] proposed a structured approach to the simulation, analysis and characterization of smartphone applications.

In this section, we give examples of benchmarking a single-core HPI CPU model in the gem5 SE mode. We also use the PARSEC benchmark suite on both single-core and multi-core HPI models in the gem5 FS mode, and present the results.

Note: it is important to mention that we have run our experiments with an older Arm disk image (`aarch-system-2014-10`) using the gem5 tree `109cc2caa6`. You might get slightly different results with the new settings.

## Benchmarking in the SE Mode

We use the Stanford SingleSource workloads from the LLVM test-suite for benchmarking in the SE mode. The svn command below downloads the benchmarks to your local machine:

```
$ svn export https://github.com/llvm/llvm-test-suite/tags/llvmorg-3.9.0/SingleSource/Benchmarks/Stanford \
          se-benchmarks
```

The source code files (.c files) can now be found in the `se-benchmarks` directory. Next, you need to install the Arm cross compiler toolchain:

```
$ sudo apt-get install gcc-aarch64-linux-gnu
```

The next step is to replace the `se-benchmarks/Makefile`'s content with the code below, and then run `make` to compile the benchmarks using the Arm cross compiler.

```
SRCS = $(wildcard *.c)
PROGS = $(patsubst %.c,%,$(SRCS))
all: $(PROGS)
%: %.c
      aarch64-linux-gnu-gcc --static $< -o $@
clean:
      rm -f $(PROGS)
```

### Running the SE Benchmarks

Having compiled all the workloads, we can run them in the gem5 SE mode, using the command below. You need to set `<benchmark>` and `/path_to_benchmark`.

```
$ ./build/ARM/gem5.opt -d se_results/<benchmark> configs/example/arm/starter_se.py \
 --cpu="hpi" /path_to_benchmark
```

For example, to run the `Bubblesort` benchmark, run the following command:

```
$ ./build/ARM/gem5.opt -d se_results/Bubblesort configs/example/arm/starter_se.py \
 --cpu="hpi" /path_to/se-benchmarks/Bubblesort
```

By doing so, the statistics will be recorded in `stats.txt` files in the `gem5/se_results/<benchmark>` directories. To compare the benchmarks, we first need to create a simple configuration file and specify a list of benchmarks to be compared, comparison parameters from their `stats.txt` statistics and an output file. Then, we just pass this configuration file to the `read_results.sh` bash script provided as part of the `arm-gem5-rsk` Research Starter Kit.

An example configuration file `exe_time.ini` looks like this:

```
[Benchmarks]
Bubblesort
IntMM

[Parameters]
simSeconds

[Output]
res_exe_time.txt
```

As the second step, we change the directory to `se_results` and run the `read_results.sh` bash script by passing the `exe_time.ini` file as a parameter:

```
$ cd se_results # where the results of SE runs are stored
$ bash ../../arm-gem5-rsk/read_results.sh exe_time.ini
```

The final result file `res_exe_time.txt` will look like this:

```
Benchmarks          simSeconds
Bubblesort          0.092932
IntMM               0.006827
```

We compare the workloads running in the SE simulation mode in terms of the instructions/operations count, execution time and overall average cache-miss latency.

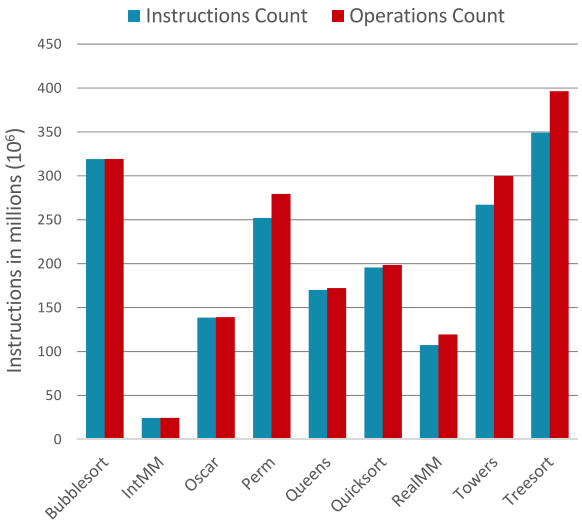



*Figure 9: Instructions/operations count obtained in the SE mode for the HPI model*

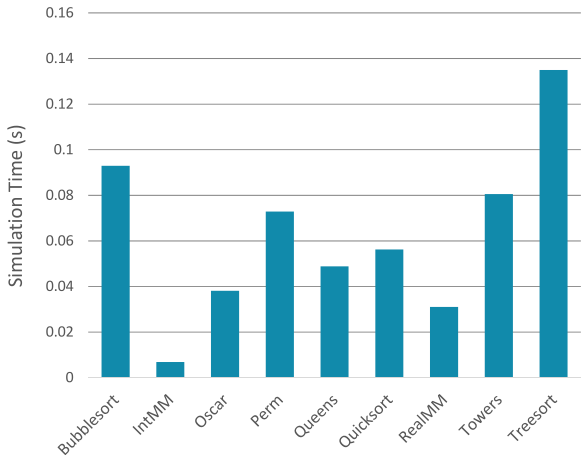



*Figure 10: Execution time in seconds obtained in the SE mode for the HPI model*

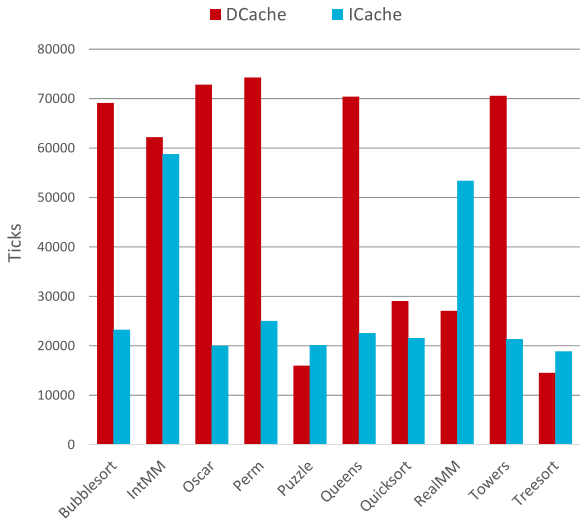



*Figure 11: Overall average miss latency obtained in the SE mode for the HPI model*

## Benchmarking in the FS Mode

In order to examine different aspects of our HPI system in the FS mode, the Princeton Application Repository for Shared-Memory Computers (PARSEC) version 3.0 [19] is used. PARSEC contains multi-threaded applications covering a diverse range of domains including financial analysis, search, computer vision, animation, data mining and so on. Make sure to check the system requirements in [20].

### Compiling the PARSEC Benchmarks

First, we need to download PARSEC 3.0:

```
$ wget http://parsec.cs.princeton.edu/download/3.0/parsec-3.0.tar.gz
$ tar -xvzf parsec-3.0.tar.gz
```

Considering that you may not have access to a local Arm machine, we describe two ways to compile PARSEC benchmarks for Arm:

- Cross-Compiling: using a cross-compiler on an x86 machine

- Compiling on QEMU: using QEMU to emulate an Arm machine on x86

We introduce a couple of common steps for both approaches. Then we show you how to compile your PARSEC benchmarks using either of these approaches.

### Common Steps

The distributed PARSEC package only supports the 32bit Armv7-A version of the Arm architecture, however, our HPI system focuses on the 64bit Armv8-A. Therefore, in our common steps, we have to make the following changes to PARSEC (note that all the patches are included in the `parsec_patches` directory of the `arm-gem5-rsk`):

- Step1: from the `parsec-3.0` directory, apply the `static-patch.diff` patch. This patch is used for generating static binaries. It also modifies the "canneal" benchmark, which does not support the AArch64 architecture by default. So, the patch adds the header file `atomic.h` from the FreeBSD project [21] to your PARSEC source tree.:

```
$ patch -p1 < ../arm-gem5-rsk/parsec_patches/static-patch.diff
```

- Step2: to recognize the AArch64 architecture, replace the `config.guess` and `config.sub` files. Do not forget to set path to `/parsec_dir/` and `/absolute_path_to_tmp/`

```
$ mkdir tmp; cd tmp # make a tmp dir outside the parsec dir
$ wget -O config.guess 'http://git.savannah.gnu.org/gitweb/?p=config.git; \
a=blob_plain;f=config.guess;hb=HEAD'
$ wget -O config.sub 'http://git.savannah.gnu.org/gitweb/?p=config.git; \
a=blob_plain;f=config.sub;hb=HEAD'
$ cd /parsec_dir/ # cd to the parsec dir
$ find . -name "config.guess" -type f -print -execdir cp {} config.guess_old \;
$ find . -name "config.guess" -type f -print -execdir cp /absolute_path_to_tmp/config.guess {} \;
$ find . -name "config.sub" -type f -print -execdir cp {} config.sub_old \;
$ find . -name "config.sub" -type f -print -execdir cp /absolute_path_to_tmp/config.sub {} \;
```

PARSEC provides some hook functions, which are called at specific locations (beginning and end of a phase) by the benchmarks. A set of hook functions called `__parsec_roi_begin()` and `__parsec_roi_end()` are used to remove the impact of the initialization and cleanup phases, and define a Region of Interest (ROI) for each benchmark. ROI is a part of the benchmark that contains "interesting" computations, e.g. the parallel phase.

We can instruct gem5 to measure statistics only for the ROI, by using the m5-related functions, namely `m5_checkpoint()`, `m5_reset_stats()` and `m5_dump_stats()`. We have added these functions to both cross-compile and QEMU patches. So, when using either of these techniques, the PARSEC benchmarks will be annotated by the necessary gem5 functions to measure the statics only for the Region of Interest.

### Cross-Compiling on x86

In order to cross-compile the PARSEC benchmarks for AArch64, we have to configure the PARSEC package to use the correct Arm compiling toolchain. Furthermore, we have to specify the target build-platform for the PARSEC benchmarks. We need to download the `aarch64-linux-gnu` toolchain from Linaro:

```
$ wget https://releases.linaro.org/components/toolchain/binaries/latest-5/aarch64-linux-gnu/gcc-linaro
    -5.5.0-2017.10-x86_64_aarch64-linux-gnu.tar.xz
$ tar xvfJ gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu.tar.xz
```

Before applying the cross-compile patch, we need to change the `CC_HOME` and the `BINUTIL_HOME` in the `xcompile-patch.diff` to point to the downloaded `<gcc-linaro directory>` and `<gcc-linaro directory>/aarch64-linux-gnu` directories.

We can then apply the patch from the `parsec-3.0` directory using the command below:

```
$ patch -p1 < ../arm-gem5-rsk/parsec_patches/xcompile-patch.diff
```

Finally, the `parsecmgmt` tool can be used to cross-compile the benchmarks. You need to set the `<pkgname>`:

```
$ export PARSECPLAT="aarch64-linux" # set the platform
$ source ./env.sh
$ parsecmgmt -a build -c gcc-hooks -p <pkgname>
```

### Compiling on QEMU

A more comprehensive solution to compile the PARSEC benchmarks is to use an emulated AArch64 Linux system. We use the free and open-source QEMU (Quick Emulator) [22], which can be hosted on an x86-64 machine and provide an emulated Arm machine (System Emulation).

Firstly, we need to apply the `qemu-patch.diff` patch from the `parsec-3.0` directory using the command below:

```
$ patch -p1 < ../arm-gem5-rsk/parsec_patches/qemu-patch.diff
```

This patch will add the m5-related hooks, but will not modify the build options.

Before downloading QEMU, we need to resolve the dependencies:

```
$ sudo apt-get install libglib2.0-dev libpixman-1-dev libfdt-dev libcap-dev libattr1-dev libcap-ng-dev ninja-
    build
```

Then, we download and compile QEMU for the targeted AArch64 platform.

```
$ git clone git://git.qemu.org/qemu.git qemu
$ cd qemu
$ ./configure --target-list=aarch64-softmmu --enable-virtfs
$ make
```

Having the QEMU binaries ready, we can go back to our main working directory and download the kernel and disk image for AArch64 from Linaro, using:

```
$ wget http://releases.linaro.org/archive/15.06/openembedded/aarch64/Image
$ wget http://releases.linaro.org/archive/15.06/openembedded/aarch64/vexpress64-openembedded_lamp-armv8-gcc
    -4.9_20150620-722.img.gz
$ gzip -dc vexpress64-openembedded_lamp-armv8-gcc-4.9_20150620-722.img.gz > vexpress_arm64.img
```

Let us boot up QEMU using the command below. In order to let the emulated system have access to the PARSEC source files, we need to share our PARSEC directory with QEMU, by setting the path to the `/shared_directory/`:

```
$ ./qemu/aarch64-softmmu/qemu-system-aarch64 -m 1024 -cpu cortex-a53 -nographic -machine virt \
 -kernel Image -append 'root=/dev/vda2 rw rootwait mem=1024M console=ttyAMA0,38400n8' \
 -drive if=none,id=image,file=vexpress_arm64.img -netdev user,id=user0 -device \
 virtio-net-device,netdev=user0 -device virtio-blk-device,drive=image -fsdev \
 local,id=r,path=/shared_directory/,security_model=none -device virtio-9p-device,fsdev=r,mount_tag=r
```

After booting, you will be logged in to a fully working Linux system on AArch64 as `root@genericarmv8`, which allows you to run commands like on a local AArch64 machine. Use the following command to mount the shared directory.

```
$ mount -t 9p -o trans=virtio r /mnt
```

Now that we have our working directory mounted under `/mnt`, we can access the PARSEC directory and compile the benchmarks (make sure that you are using the bash shell):

```
$ cd /mnt # cd to the mounted PARSEC directory
$ source ./env.sh
$ parsecmgmt -a build -c gcc-hooks -p <packagename>
$ poweroff # quit QEMU
```

After compilation, we can quit the emulation environment by issuing `poweroff`.

## Running the PARSEC Benchmarks

The gem5 FS mode does not support shared directories with the host. Thus we need to copy the directory of compiled benchmarks `parsec-3.0` to our FS disk image. However, the distributed FS images are usually quite small in size and do not have enough empty space for our binaries. Let us create a copy of the Ubuntu disk image (`disks/ubuntu-18.04-arm64-docker.img`), rename it to `expanded-ubuntu-18.04-arm64-docker.img` and expand it using the following commands:

```
$ cp ubuntu-18.04-arm64-docker.img expanded-ubuntu-18.04-arm64-docker.img
$ dd if=/dev/zero bs=1G count=20 >> ./expanded-ubuntu-18.04-arm64-docker.img # add 20G zeros
$ sudo parted expanded-ubuntu-18.04-arm64-docker.img resizepart 1 100% # grow partition 1
```

Now we have a disk image with enough space on it. Let us find its start sector and unit size, and then mount it to a new directory, e.g. `disk_mnt`.

To find the start sector and unit size, run `fdisk -l` on the expanded disk image, and note the `Units` (sector size) in bytes (in this case `512 bytes`) and the `Start` sector of the partition (in this case `128 sectors`). These values can then be multiplied together to find the start offset in bytes (in this case `65536 bytes`).

```
$ fdisk -l expanded-ubuntu-18.04-arm64-docker.img
Disk expanded-ubuntu-18.04-arm64-docker.img: 21.9 GiB, 23474836480 bytes, 45849290 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x93d4895b

Device                                  Boot Start      End  Sectors  Size Id Type
expanded-ubuntu-18.04-arm64-docker.img1        128 45849289 45849162 21.9G 83 Linux
```

Using the start offset in bytes calculated above, mount the disk image:

```
$ mkdir disk_mnt
$ sudo mount -o loop,offset=65536 expanded-ubuntu-18.04-arm64-docker.img disk_mnt
```

By using `df`, we can find the corresponding loop device for the `disk_mnt` directory (`/dev/loopX`). We then use `resize2fs` to resize the file system to be able to use the available space on the disk image (if you have a graphical interface, you can also use `GParted` to resize your disk image), and finally we copy the compiled PARSEC benchmarks to the FS disk image.

```
$ df # find /dev/loopX for disk_mnt
$ sudo resize2fs /dev/loopX # resize filesystem
$ df # check that the Available space for disk_mnt is increased
$ sudo cp -r /path_to_compiled_parsec-3.0_dir/ disk_mnt/root # copy the compiled parsec-3.0 to the image
$ sudo umount disk_mnt
```

At this point, we have an FS disk image containing the compiled PARSEC benchmarks. We then need to generate benchmark runscripts and pass them via the `--script` option to the simulation script `starter_fs.py`. The runscripts can be generated using the `gen_rcs.sh` bash script included in this Research Starter Kit, where `-p <pkgname>` sets the PARSEC package to use, `-i <simsmall/simmedium/simlarge>` sets the input size, and `-n <nth>` sets the minimum number of threads to use.

```
$ cd arm-gem5-rsk/parsec_rcs
$ bash gen_rcs.sh -p <pkgname> -i <simsmall/simmedium/simlarge> -n <nth>
```

After taking the above steps, we can run the benchmarks using the expanded image and the corresponding runscripts:

```
$ ./build/ARM/gem5.opt -d fs_results/<benchmark> configs/example/arm/starter_fs.py --cpu="hpi" \
 --num-cores=1 --disk-image=$M5_PATH/disks/expanded-ubuntu-18.04-arm64-docker.img --root-device=/dev/vda1 \
 --script=../arm-gem5-rsk/parsec_rcs/<benchmark>.rcS
# Example: canneal on 2 cores
$ ./build/ARM/gem5.opt -d fs_results/canneal_simsmall_2 configs/example/arm/starter_fs.py --cpu="hpi" \
 --num-cores=2 --disk-image=$M5_PATH/disks/expanded-ubuntu-18.04-arm64-docker.img --root-device=/dev/vda1 \
 --script=../arm-gem5-rsk/parsec_rcs/canneal_simsmall_2.rcS
```

Similar to the SE mode, we compare the benchmarks running in the FS simulation mode in terms of the instructions/-operations count, execution time and overall average cache-miss latency. Additionally, we illustrate multi-core speedups in Fig. 15. You can use the `read_results.sh` bash script to collect the results from the gem5 statistics, as in Section .

Note: we have used the `simsmall` input sets for all the benchmarks except `blackscholes`. For `blackscholes`, the `simmedium` input set is used. The first three graphs below show the results of running the benchmarks on a dual-core HPI system. Also, as mentioned before, only the statistics inside the Regions of Interest are measured.

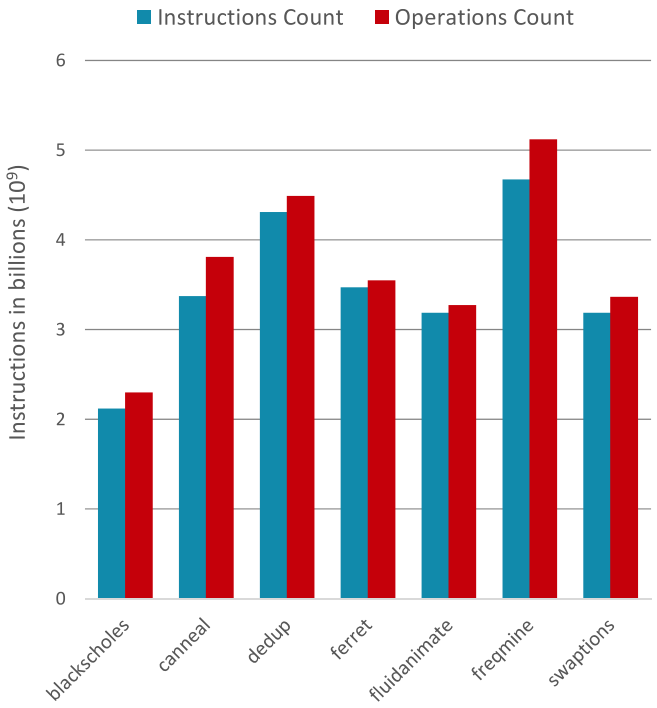



*Figure 12: Instructions/operations count obtained in the FS mode for the dual-core HPI model*

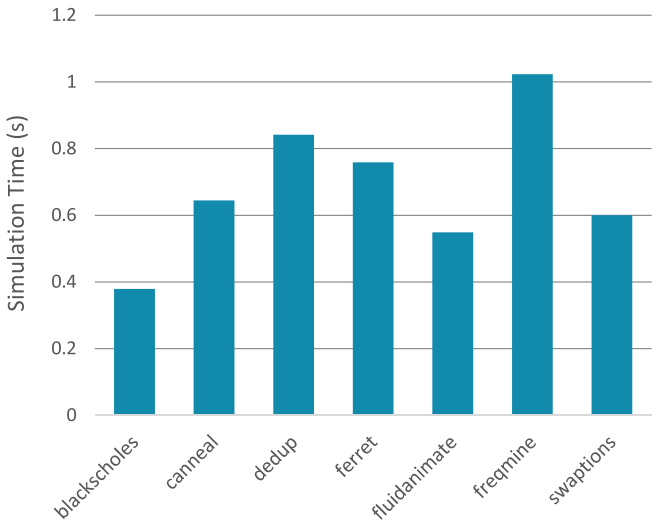



*Figure 13: Execution time in seconds obtained in the FS mode for the dual-core HPI model*

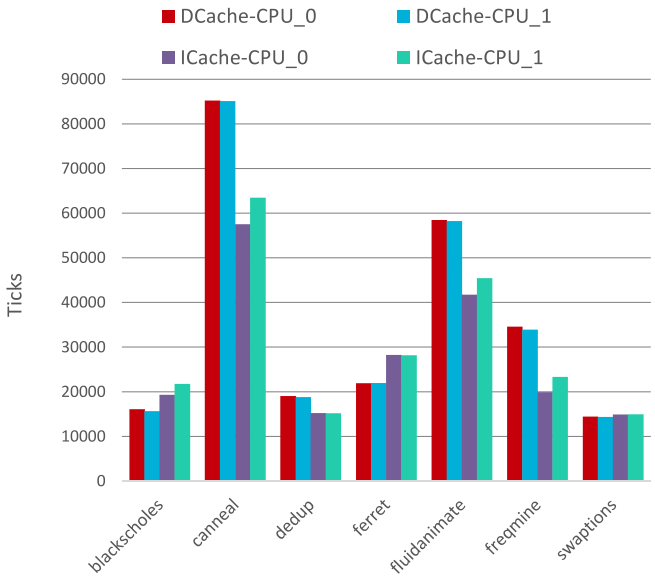
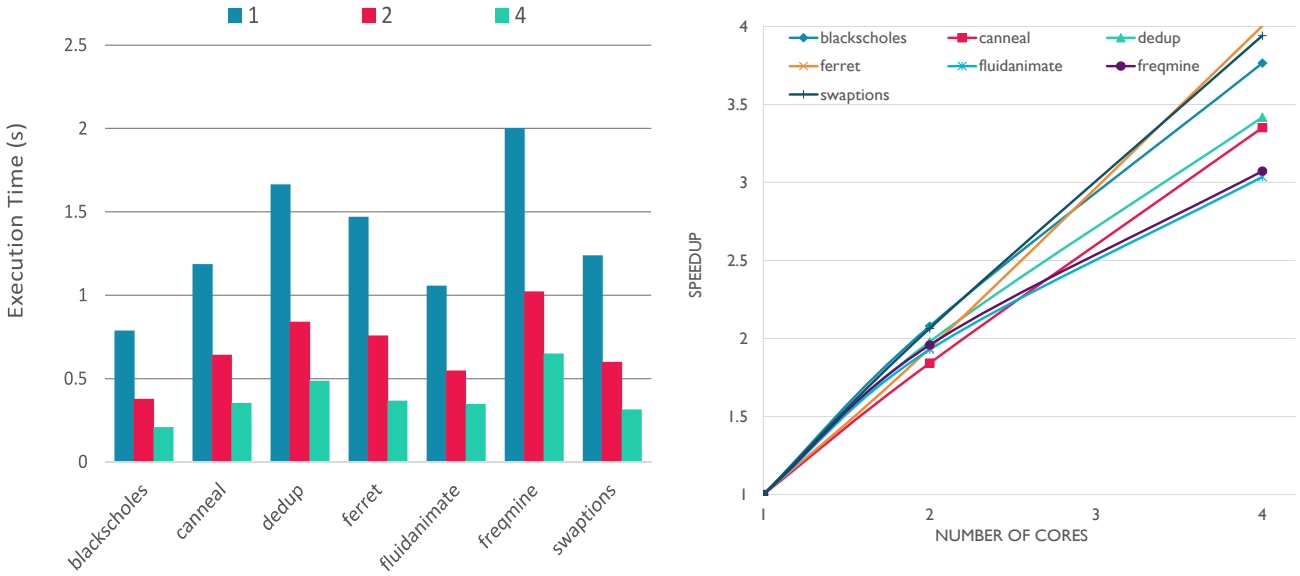



*Figure 14: Overall average miss latency obtained in the FS mode for the dual-core HPI model*




*(a) Multi-core execution time*

*(b) Multi-core speedup*

*Figure 15: Scalability of the PARSEC benchmarks on the HPI model*

## Summary

In this chapter, we focused on running benchmarks on the HPI model in both SE and FS simulation modes. For benchmarking in the SE mode, Stanford SingleSource workloads from the LLVM test-suite were used. We showed how to compile these workloads for the Arm HPI model. We also showed how to collect the results in order to compare the benchmarks.

For benchmarking in the FS mode, we used the PARSEC Benchmark Suite version 3.0. The FS simulation is more complicated and requires additional steps, such as downloading and modifying an FS disk image, creating runscripts, etc. We described two different ways to compile the PARSEC benchmarks for the Armv8 architecture: cross-compiling and compiling on QEMU. We then explained how to modify a disk image and copy the compiled benchmarks to it. Finally, we ran the PARSEC benchmarks on single-core and multi-core HPI models, and provided comparisons between them using the gem5 statistics.

# Bibliography


[1] "The gem5 simulator." https://www.gem5.org/.

[2] "Cortex-A series - Arm." https://www.arm.com/products/processors/cortex-a/index.php.

[3] "gem5 tutorial - gem5 tutorial 0.1 documentation." http://pages.cs.wisc.edu/~david/courses/cs752/Fall2015/gem5-tutorial/index.html.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[5] F. A. Endo, D. Courousse, and H. P. Charles, "Micro-architectural simulation of in-order and out-of-order Arm microprocessors with gem5," in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pp. 266–273, July 2014.

[6] B. Beckmann, N. Binkert, A. Saidi, J. Hestness, G. Black, K. Sewell, and D. Hower, "The gem5 simulator." http://dist.gem5.org/tutorials/isca_pres_2011.pdf.

[7] "Documentation - gem5." https://www.gem5.org/documentation.

[8] "Cs 752 course wiki: Fall 2015 : Course calendar browse." http://pages.cs.wisc.edu/~david/courses/cs752/Fall2015/wiki/index.php?n=Main.CourseCalendar.

[9] M. Alian, D. Kim, N. S. Kim, G. Dozsa, and S. Diestelhorst, "Dist-gem5 architecture." http://publish.illinois.edu/icsl-pdgem5/files/2015/12/dist-gem5-arch-v4.pdf.

[10] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi, "Simulating dram controllers for future system architecture exploration," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 201–210, March 2014.

[11] R. de Jong and A. Sandberg, "Nomali: Simulating a realistic graphics driver stack using a stub gpu," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 255–262, April 2016.

[12] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, pp. 34–36, Jan 2015.

[13] Y. Ko, T. Kim, Y. Yi, M. Kim, and S. Ha, "Hardware-in-the-loop simulation for cpu/gpu heterogeneous platforms," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2014.

[14] "General memory system - gem5." http://old.gem5.org/General_Memory_System.html.

[15] "Inside the minor cpu model." https://www.gem5.org/documentation/general_docs/cpu_models/minor_cpu.

[16] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pp. 1–7, July 2012.

[17] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 13–22, March 2014.

[18] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver, "A structured approach to the simulation, analysis and characterization of smartphone applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pp. 113–122, Sept 2013.

[19] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[20] "The PARSEC Benchmark Suite." http://parsec.cs.princeton.edu/download.htm.

[21] A. Turner, "The freebsd github: arm64." https://github.com/freebsd/freebsd/blob/master/sys/arm64/include/atomic.h.

[22] "QEMU." http://www.qemu.org.