Examining the use of ARM PMUs for DVFS and scheduling

CS5199 Individual Masters Project (MSci)

Thomas Ekström Hansen
150015673
Supervisor: Dr. John Thomson

School of Computer Science
University of St Andrews

22nd May 2020

Declaration

Abstract

ARM big.LITTLE systems are everywhere from smart devices, to Systems on Chips, to mobile phones. Common to these is an often limited energy budget, where saving energy without sacrificing too much performance. With asymmetric multicore processors and DVFS comes the potential for vast energy savings by having the scheduler be aware of the energy used by the system tasks and adjusting the DVFS points accordingly, potentially even switching some cores off entirely.

In this project, I explore these points using the gem5 Simulator, a highly accurate simulator with support for both DVFS and power modelling. I present initial findings which suggest that the use of ARM PMUs for energy prediction can be highly effective and accurate for scheduling, motivating further research into the topic.

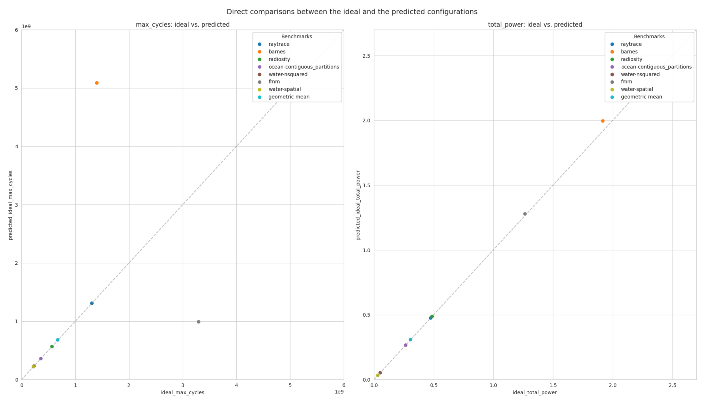
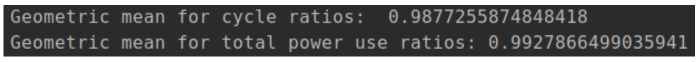


Figure 1: Initial findings encourage looking into the use of ARM PMUs in scheduling, in order to balance performance and energy savings



```
Geometric mean for cycle ratios:  0.9877255874848418
Geometric mean for total power use ratios: 0.9927866499035941
```

Figure 2: Albeit based on a small sample size, the model seems to have narrow optimisation space, which is promising for future research

Contents

1 Introduction **1**

2 Background **3**

2.1 Processes and Threads . . . 3
2.2 Scheduling . . . 4
2.2.1 The Completely Fair Scheduler . . . 5
2.2.2 Multicore Scheduling . . . 6
2.3 ARM big.LITTLE architecture . . . 6
2.3.1 Performance Monitoring Units . . . 7
2.4 Dynamic Voltage and Frequency Scaling . . . 8

3 Context Survey **9**

3.1 The CFS and multicore systems . . . 9
3.2 Adding big.LITTLE awareness to a scheduler . . . 9
3.2.1 The WASH scheduler . . . 9
3.2.2 The COLAB scheduler . . . 10
3.3 Runtime DVFS management via PMUs . . . 10
3.4 ARM's EAS in the Linux kernel . . . 11
3.5 Summary . . . 11

4 The gem5 simulator **12**

4.1 Overview . . . 12
4.2 Configuration and Setup . . . 13
4.2.1 The gem5 binary . . . 13
4.2.2 Python configuration scripts . . . 13
4.2.3 Bootscripts, systems, and m5 instructions . . . 14
4.2.4 CPU types and checkpointing . . . 15
4.3 Interacting with the simulated system . . . 16
4.4 Cross Compilation . . . 17
4.5 Power models and stats . . . 17

4.6 Simulating PMUs 18
4.7 Customising the setups 21
4.7.1 Voltage and Frequency Domains 21
4.7.2 Power Models 24
4.7.3 Output resolution 26
4.8 Stats access from power models 27
4.9 Looking into a different simulator 31

5 Experimental Setup **32**
5.1 Setting up the benchmarks 32
5.2 Setups 33
5.3 Running 33
5.3.1 Technical details 33
5.3.2 Scripts 34

6 Results **35**
6.1 Raw data 35
6.1.1 Problems encountered 35
6.2 Data extraction 36
6.2.1 Mapping gem5 stats to PMU events 36
6.2.2 The `data-aggregate.py` script 39
6.3 Data Processing 39
6.3.1 Pre-processing 39
6.3.2 Predicting and Plotting 40
6.4 Analysis 42
6.4.1 Plots and observations 42
6.4.2 Evaluating the prediction results 43
6.4.3 Using other stock configurations 44

7 Conclusion **46**

A Attempts to find power model stats **54**

B Power plots **55**
B.1 The barnes benchmark 56
B.2 The ocean-contiguous_partitions benchmark 60
B.3 The water-spatial benchmark 64

C Multi-stock prediction results **67**
C.1 Stock 1b1L+2b2L 67
C.2 Stock 2b2L+4b4L 69

D Ethics **71**

Chapter 1

Introduction

As the Internet of Things and mobile devices have become ubiquitous and increasingly powerful, so has demand on their capabilities. Users often want both great performance and great battery life out of numerous devices. The introduction of Asymmetric Multicore Processors (AMPs) like ARM's big.LITTLE technology has helped meet these demands as the big cores can be switched off until needed, thereby saving power. This has led to the concept and management of "dark silicon" – processing elements which have been switched off to save energy.

As with most changes, it takes a bit of time for the software to catch up. The Linux Scheduler had to be redesigned in the 2000s in order to work better with Multicore Processors, and even after the redesign, improvements continued to emerge. Recently, there has been interest in making schedulers energy aware. Since the scheduler manages which tasks run when and on what CPUs, there is an idea that improvements could be made on existing systems if the scheduler was aware of the different types of cores and the energy they use. By having this insight, it could manage the Dynamic Voltage and Frequency Scaling (DVFS) on the cores with much finer control than current approaches, potentially leading to great energy savings at a low performance cost.

The ARM big.LITTLE architecture provides special registers and hardware known as Performance Monitoring Units (PMUs). These allow software to efficiently keep track of how many times certain events happen on the hardware. Although many PMU events (and hence measurements) are supported, typically only 5-7 can be retrieved at a time. This leads to an interesting problem in the form of which ones to prioritise. With the right combination of PMU events and a model built to represent how they affect performance and energy, the scheduler could be able to make intelligent decisions based on the dynamic monitoring of both task and harware performance. As ARM big.LITTLE, energy awareness, and PMUs are only recently beginning to gain attention in the literature, this leaves many interesting and highly applicable research areas open.

In this project, I explore how to use the gem5 simulator to model DVFS and power consumption in big.LITTLE systems. I then use it to explore if predictions based on existing PMU data obtained from a stock configuration can be used to correctly predict the optimal big.LITTLE configuration to run an unseen program on. I find that although gem5 can be challenging to use, the simulations are possible and even limited data can be used to create promising models which come close to the ideal performance ratios when predicting the optimal configuration. I achieve geometric means of $\approx 98.77\%$ and $\approx 99.28\%$ for how close the predictions are to the ideal configurations in terms of number maximum of cycles spent and total power used, respectively. This is highly promising for further research into the area.

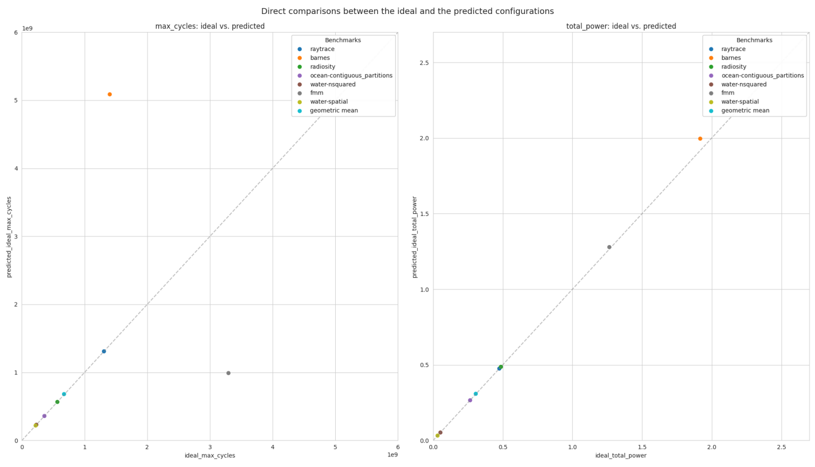
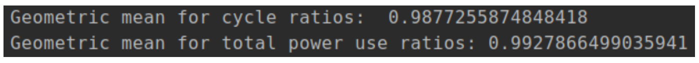


Figure 1.1: Initial results based on data obtained from gem5 seems highly promising in terms of the use of PMUs in scheduling



```
Geometric mean for cycle ratios:  0.9877255874848418
Geometric mean for total power use ratios: 0.9927866499035941
```

Figure 1.2: Although based on a small sample size, the model has very good ratios for ideal / predicted

Chapter 2

Background

2.1 Processes and Threads

The scheduling problem has been around for a while. The earliest papers which discuss scheduling date back to 1962 [11], with hard real-time scheduling (i.e. scheduling for systems where time limits *must* be met) dating back to 1973 [30], and preemptive scheduling being discussed in 1975 [28]. In this section I will give an overview of the scheduling problem, explaining some key concepts and problems, basing a lot on Silberschatz et al.'s book on operating systems [36].

Code running is represented as a *process*. Processes contain the code being run, information such as what process started the current process (its *parent*), what process(es) the current process has created (its *children*), the state of the process (e.g. 'running', 'waiting', 'ready'), and also some information about the current memory and scheduling information about the process's priority, how long it has been running for, etc. A process may contain multiple *threads*. A thread is a simple unit of computation or work being done. A thread has an ID and contains a stack, a register set, and a program counter. The other parts, e.g. the code or memory information, is stored in the process running the thread and is shared between all the process's threads. Hence, creating new threads and inter-thread communication is cheaper than creating new processes and inter-process communication. This is useful for processes which want to do different, often independent things in parallel, e.g. a web browser fetching data from the internet whilst also handling the user's keystrokes. It can also be useful if the problem being solved can be broken up into smaller parts which can then be done in parallel.

In most major operating systems, MacOS, Windows, Solaris, and Linux, there are both user threads and kernel threads. Kernel threads, as the name implies, are managed directly by the operating system kernel. User threads, on the other hand, are created and managed by the user/programmer. Kernel threads are more expensive to create than user threads and as such, most operating systems map user threads to kernel threads during runtime. This means that the user/programmer does not have to worry about how many threads they create, and that the operating system can reuse kernel threads for various applications, thereby improving performance. When it comes to scheduling, the scheduler manages kernel threads.

2.2 Scheduling

When talking about scheduling, the terms "process scheduling" and "thread scheduling" are often used interchangeably. To avoid confusion with the previous section and to be consistent with commonly used Linux terminology, I will use "task" to refer to anything, be it a process or a thread, which needs to be scheduled. Scheduling is the problem determining what tasks need to run and for how long. Most tasks do not require constant computation. Instead, computation happens in bursts and the other time is, for example, spent on waiting for I/O to happen, e.g. reading a value from memory. So in order to maximise CPU usage, a scheduler swaps different tasks on the CPU(s). Another reason scheduling is required, is that if the scheduler did not swap the tasks, a single CPU-intensive task could hog the CPU for a long time. On interactive systems, this could result in loss of interactivity (aka. "freezing") until the heavy task finishes its computations. By swapping tasks, the scheduler tries to keep the CPU-usage as fair as possible. This swap can be done when a task signals that it is waiting, or when it finishes. This is known as *non-preemptive* scheduling. By contrast, *preemptive* scheduling is when the scheduler may pause a task mid-execution in order for a different task to have some time on the CPU as well, thereby ensuring that no one task hogs the CPU leading to the other tasks being unresponsive.

Ideally, the perfect scheduler would let the task that has the least time left run first, thereby minimising the wait time for the other tasks. However, this would require the scheduler to know the future (i.e. when the various tasks would stop) and is therefore unfortunately impossible. Instead, there exist various scheduling algorithms which are used to determine what task to run next. One way to determine this is to assign the tasks a priority and execute the highest-priority task first. This works fine if the high-priority jobs finish. However, there is a very real risk that a low-priority job may never be run as it is kept at the back of the queue by higher-priority jobs. As such, priority scheduling is rarely used in its naïve version, with modern scheduling algorithms changing the priority of a task over time, based on various variables. One example of this is the Linux "Completely Fair Scheduler" (CFS) which was introduced in kernel 2.6.23 [10].

2.2.1 The Completely Fair Scheduler

Typically, the tasks that are waiting to be executed on the CPU are stored in a so-called “ready queue”. However, the CFS stores the tasks in a data structure known as a Red-Black tree (RB-tree). RB-trees were first introduced in 1978 by Guibas and Sedgewick [22]. An RB-tree is a type of self-balancing binary tree. In addition to having the usual binary tree attributes (a key, the left children being less than their parent, and the right children being greater than their parent), each node in the tree is given a colour, red or black, and the tree balances itself by maintaining the following three properties:

1. The root of an RB-tree is black.
2. The children of a red node are black.
3. The paths going from the root to a `null` leaf all contain the same number of black nodes.

A complete overview of how the operations on RB-trees work is beyond the scope of this dissertation. The important attribute is that, like other self-balancing trees, an RB-tree maintains a height that is very close to (or exactly) $logN$, regardless of how many operations have been performed, allowing lookup to stay $O(logN)$. By keeping the tasks in an RB-tree, the CFS can access them quickly while the data structure makes sure to permute the tasks according to their priority. To further increase performance, the CFS caches a pointer to the leftmost node in RB-tree.

The CFS distinguishes between real-time tasks and ‘normal’ tasks. Real-time tasks have a regular priority and are scheduled accordingly. Normal tasks each have a virtual runtime (`vruntime`) and a “nice value”. The `vruntime` is used as the key for the RB-tree storing the tasks and the CFS schedules the leftmost leaf in the RB-tree. The nice value is an integer between -20 and $+19$ which affects the recorded `vruntime` for that task. A task’s `vruntime` is equal to its actual/physical runtime (the time it spent on the CPU) and some modifier based on the task’s nice value. A negative nice value leads to a lower `vruntime` than actual runtime, a nice value of 0 to an equal value, and a positive nice value to a greater `vruntime` than the actual runtime. This means that a task can “be nice” to other tasks by setting its nice value higher, resulting in a bigger `vruntime`, leading it to be scheduled less often than other tasks. And conversely, tasks with a low nice value will be scheduled more often despite them potentially taking up more physical runtime. All tasks eventually get to run, as a task’s `vruntime` can only increase, meaning that even if it does so slowly (due to a low nice value) it will eventually be greater than a task whose `vruntime` has not changed because it was not being scheduled. This task will then be the leftmost node, and so it will get scheduled.

2.2.2 Multicore Scheduling

Most modern CPUs have multiple processing elements, or cores, on the physical chip. These chips are referred to as multicore processors. Each core on the chip has its own set of registers and level 1 (L1) cache, with the other memory components typically being shared across all the processing cores. Having multiple cores can help in terms of *load balancing*, i.e. tasks can be spread across the different cores in order to give them as much runtime as possible without one single core having to run all the tasks. However, tasks cannot easily be resumed on any processor. When swapping a task back in, the scheduler must make sure that the processor and memory state is the same as when the task was swapped out. Since each core has its own set of registers and cache, this means that it is simpler to restore a task to the core it previously ran on, compared to transferring all the task's information to a different core, invalidating the information on the old core, and then restoring the layout on the other core. This is what is known as processor *affinity*; the task has an *affinity* for running on the same core it was swapped out from. Because it is impossible to know how quickly a task will finish, it is possible that some cores will run out of tasks before other ones. In this case, it is necessary to do *load balancing*. This typically involves migrating a task from one core to another. There are two ways a task can be migrated: *push migration* and *pull migration*. Pull migration is when an 'empty' core transfers a ready task from a busy core to itself, 'pulling' the task to it. Push migration, on the other hand, is when an external process notices that it may be better to migrate a task to a different core and 'pushes' that task from its original core to the new core. Since these two migration techniques are not mutually exclusive, the CFS implements both. As previously discussed, migrating a task disturbs its affinity and incurs a lot of overhead. There is no perfect way of prioritising one over the other and schedulers have to strike a balance when managing the tasks of multicore chips, further complicating the already complex scheduling problem.

2.3 ARM big.LITTLE architecture

When running code on CPUs, the code is written, compiled to assembler, and then assembled into an executable binary. How the machine code that the CPU runs behaves is defined by its Instruction Set Architecture (ISA). An ISA defines the registers available; what operations are available, how many arguments they take, and how they behave; how memory is addressed and many other things. When implementing an ISA, for example when designing a CPU, the physical circuitry representing different parts of the ISA may be drastically different between various implementations, but the operations and results are the same (e.g. an `ADD` instruction from an ISA implemented on two different CPUs should behave the same regardless of how the internals of the CPU look).

Having different CPUs implement the same ISA is extremely useful as it allows for binary compatibility as long as the code is compiled for the ISA being used. A classic example of this is the x86 ISA which most modern desktop and laptop processors implement. The same software that runs on a desktop CPU can be run on a laptop CPU without needing to recompile it. The ARM big.LITTLE architecture takes this concept one step further by introducing asymmetric single-ISA multiprocessors (ASMs): A chip which contains multiple, different CPUs implementing the same ISA. The name "big.LITTLE" refers to the two CPU/core types, 'big' and 'LITTLE'. The big cores are more powerful but consume more power, whereas the LITTLE cores are less powerful but also consume less power. This benefits systems where power is limited, e.g. mobile phones, as the LITTLE cores can be used for tasks that are not performance-critical, with the big cores only being used when necessary, thereby saving power. Additionally, since the CPUs are located on the same chip, they have access to the same memory and so tasks can be migrated between cores to save power or (hopefully) increase performance. However, as with the introduction of multicore, the introduction of ASM systems adds a layer of complexity to scheduling: What tasks should run on big cores vs. LITTLE cores? When should tasks be migrated from one to the other? When a new task arrives, how do we know what type of core to start it on (in order to minimise migration overhead)?

2.3.1 Performance Monitoring Units

Some of the more recent ARM ISAs, e.g. ARMv7 and ARMv8, specify Performance Monitoring Units (PMUs). These are special registers and events which allow the hardware to monitor and report certain statistics which can then be examined, either using an external tool or inline assembler. The PMUs can monitor events like the number of CPU cycles passed, the number of level 2 (L2) cache accesses, the number of memory access, how many times the branch predictor speculatively executed a branch, and more. The PMUs have to be enabled by the kernel by setting certain bits in a special register (bit 0 of `PMCR_EL0` in the case of ARMv8) to 1 (`True`). After this, special "event numbers" can be written to other special registers in order to specify what events to count. The cores used in this project (i.e. Cortex A73 cores for big and Cortex A53 cores for LITTLE) support the cycle counter and 6 additional PMU events [1, 2]. Since PMUs can be accessed programmatically using inline assembler, it should theoretically be possible to use these to make informed scheduling decisions. At the cost of yet another thing to consider when scheduling.

2.4 Dynamic Voltage and Frequency Scaling

CPUs used to run at a fixed frequency. As processor design and manufacturing improved, it became possible to change a CPU's frequency dynamically, either through the BIOS or the Operating System (OS). With dynamic frequency scaling, the CPU could be throttled down when idle, thereby saving power, and throttled back up (or sometimes even automatically overclocked) when performance was required. Running a CPU at higher frequencies requires higher voltages in order for the gates in the CPU to stabilise fast enough. If they do not stabilise before the next frequency tick/clock pulse, the hardware output of that CPU gate will be unreliable and/or incorrect. Dynamic Voltage and Frequency Scaling (DVFS) is a technique which saves power by, as the name suggests, varying the frequency and voltage on the fly. It was introduced as an energy savings measure in 1990 [33] and is in every modern computer system. The Linux kernel supports DVFS through its `cpufreq` interface and the "frequency governors" which decide how to change the frequencies [43]. DVFS is also supported on ARM CPUs, including big.LITTLE setups, potentially allowing for even greater power savings than the big.LITTLE setup on its own. However, as with all the previously discussed topics, DVFS adds complexity to the scheduling problem.

To sum up, in order to have both good performance and power efficient, a scheduler could take into account:

- What type of core the task(s) run on (big or LITTLE),
- whether it is worth migrating a task between cores *and* between core types,
- and whether it is worth running the cores at full speed and voltage or if some cores could be throttled to save power at a slight performance hit.

Chapter 3

Context Survey

3.1 The CFS and multicore systems

It is only fairly recently that people have started looking at the problem of Energy Aware Scheduling (EAS) and optimising for big.LITTLE In 2016 the CFS received several fixes based on a paper published in the same year, detailing how some bugs in the CFS decreased performance on multicore systems by 13-24% [32]. This paper addresses multicore in general, and does not focus on big.LITTLE or energy. However, it did drastically improve the performance of the CFS on multicore systems. As mentioned in Jibaja et al. [27], the CFS does well on big.LITTLE for scalable workloads, but it may be possible to accelerate other, non-scalable workloads as well.

3.2 Adding big.LITTLE awareness to a scheduler

3.2.1 The WASH scheduler

Jibaja et al. describe the "WASH" scheduler [27]. The WASH scheduling algorithm is an attempt at optimising big.LITTLE setups on the fly rather than through external hints or guidance from the programmer. By analysing the performance of various tasks depending on the percentage of cycles spent waiting on locks, WASH estimates the criticality, sensitivity, and progress of the tasks currently active on the system. Since the number of instructions executed varies between big and LITTLE cores (due to their different computational power), WASH scales with respect to the instructions per clock (IPC) so as to have a more fair view of a task's performance. Based on all this information, WASH wash then 'accelerates' tasks by either assigning them to a big core from the beginning, or migrating them to a big core if not enough progress is being made according to the parameters monitored. This is done through the POSIX-thread (pthread) API, specifically the `pthread_setaffinity_np` function which sets the affinity of the pthread to the CPU(s) given. Whilst this is a straightforward way to migrate threads, it also means the actual migration is controlled by the CFS rather than WASH itself [44]. WASH's main focus is to make the scheduler big.LITTLE aware and thereby hopefully improve power usage, and as such it does not focus on power or energy along with big.LITTLE, meaning there is likely to be room for improvement. Finally, it was discovered by Yu et al. [44] that WASH's decision-making does not always accelerate the program as a whole, despite running computationally heavier tasks on big cores. Counterintuitively, heavy tasks running on big cores can sometimes be slowed down, due to waiting on results or feedback from lighter tasks running on LITTLE cores, and so it would be better to accelerate these lighter tasks.

3.2.2 The COLAB scheduler

The paper describing the COLAB scheduler, published by Yu et al. earlier this year [44], highlights a number of shortcomings regarding the WASH scheduler and proposes a different, collaborative scheduling algorithm. The COLAB scheduler addresses *thread criticality* (some threads being more critical with respect to program performance than others) and *core sensitivity* (the big and LITTLE cores being designed for different types of workload) whilst maintaining fairness. The collaborative part of COLAB comes from allowing both the core allocator (deciding which core to start on) and the thread selector to label the threads in terms of whether they have a good chance of a high speedup from a big core and whether they are highly blocking or not. The speedup label can then be used to allocate the best type of core for the thread, and a more blocking thread can be selected more often so as to hopefully move it, and by extension threads waiting on it, along faster. This collaboration between core allocation and thread selection allows COLAB to achieve performance gains of 5-15% on average compared to WASH, depending on the hardware configuration. However, similar to WASH, COLAB does not address power or energy, but focuses on improving big.LITTLE. Therefore, this space is still mostly unexplored and it is possible power savings could be made without sacrificing much performance.

3.3 Runtime DVFS management via PMUs

A recent Ph.D. thesis and collection of papers by K. Reddy Basireddy do tackle the DVFS side of things. The thesis seems address power and energy for both asymmetric and symmetric multicore processors, with DVFS being a subset of the methods examined [4]. The DVFS part resulted in the AdaMD paper [3] which details how by using certain PMUs with a “performance monitor”, the DVFS settings can be adjusted to balance power and performance at runtime. AdaMD works by doing an initial prediction of what it thinks the best DVFS and core settings are, and then periodically monitoring the performance of the application and adjusting the DVFS settings and core allocation in terms of “performance constraints”. However, the paper seems somewhat vague in terms of where these performance constraints come from and what they are defined in terms of, e.g. if they are the number of cycles or application runtime or something different. Additionally, as its main focus is DVFS and energy efficiency, and since the paper was published before the COLAB paper, it is possible that the performance of COLAB and the energy efficiency could be combined.

3.4 ARM’s EAS in the Linux kernel

ARM have also looked into EAS. With release 5.0 of the Linux kernel in March 2019 [39] EAS was officially merged into the kernel and released [29]. However, as described in the official documentation [12] this relies on the Energy Model framework being present in the kernel, rather than the PMUs, and also appears to still be a work in progress. The Energy Model framework is likely to be more expensive to use than the hardware-based PMUs, and it is possible that greater performance monitoring accuracy could be achieved by using certain PMU events. Finally, similar to AdaMD, EAS on Linux does not seem to account for the improvements described in the COLAB paper and so it is possible that greater performance and energy savings could be achieved.

3.5 Summary

To summarise, making the scheduler aware of the different core types on Asymmetric Multicore Processors is something only recently started gaining attention in the literature. Making the scheduler energy aware is also only recently beginning to make its way into real systems, e.g. EAS in Linux 5.x. Across the existing literature, nothing seems to combine the two, resulting in a highly interesting and applicable research area, as big.LITTLE systems become increasingly ubiquitous in both mobile devices and the IoT, both of which are could benefit from power savings and/or better scheduling.

Chapter 4

The gem5 simulator

4.1 Overview

The gem5 Simulator [5] is a state-of-the-art hardware simulator used not only for architecture research, but also by the companies which develop the hardware it simulates. ARM, for example, have detailed their use of it at the ARM research summit in 2017. It is the result of merging two simulators in 2011: the m5 simulator, which had great support for multiple hardware architectures and ISAs, and the GEMS simulator, whose main focus was memory simulation and hence allowed for detailed memory models and various cache coherence protocols [5].

Due to the enormous complexity of such a piece of software, the way to start using gem5 is to download and build from source [16]. Building the simulator can be a bit complicated, due to it being sensitive to various system tools and configurations. The gem5 Simulator uses `scons` as its build tool. Specifically, it seems to rely on the Python 2 version of `scons` (and also uses Python 2 for its other Python scripts) despite Python 2 having been officially deprecated on the 1st of January 2020, with the move to do so being announced well in advance [38]. However, this can be easily fixed by using a Python 2 virtual environment. The specific version of the GNU Compiler Collection (GCC) also seems to affect things. I was using GCC version 9.3.0 which must have more warnings than the version used by the gem5 developers, as it failed to build gem5 due to the `-Werror` flag being turned on. This flags turns compiler warnings, which normally highlight suspicious areas of code but still lets the build go ahead, into errors which do not let the build go ahead. The user can tell the compiler that some warnings are exceptions to the `-Werror` flag, but due to the number of warnings I was getting, it was simpler to remove the flag from the `scons` configuration found in the `SConstruct` file. There are also a number of packages that the build script might recommend one installs, most notably the Google

Performance Tools (gperftools) [21] which they suggest improves performance by 12%. Once the necessary software, packages, etc. have been installed and configured, the build process takes around 20-30 minutes when building using 13 threads on an 8th generation Intel Core i7 processor, which also says something about the scale and complexity of gem5.

4.2 Configuration and Setup

4.2.1 The gem5 binary

There are two parts of gem5 that can be configured: the simulator binary and its configuration files. The binary produced by `scons` supports a number of flags which allow the user to specify things like the output directory, whether to redirect gem5's standard output and/or standard error streams, and what to call the various files being output. There are also anumber of debug flags for developing gem5 itself. The most important of the flags is arguably the `--outdir` flag used for specifying the output directory. When running multiple simulations side by side, if no directory is specified, i.e. the default `m5out` directory is used, each instance will try to write to the same files as the others, resulting in non-deterministic race conditions in terms of which process was the last to write to the file, leading to unusable, interweaved output.

4.2.2 Python configuration scripts

After the flags, gem5 takes a Python script containing the setup(s) to run. Since this is a regular Python script, it is possible to add command line arguments here as well, allowing for control over the system configuration without having to change the Python file itself. The gem5 Simulator comes with a number of example scripts for ARM system simulation, located in the `gem5/configs/example/arm` directory. The scripts provide examples for syscall emulation, full system simulation, and full system simulation with power models. Syscall emulation is faster than full system simulation because it only simulates the system calls and their results, and not the disks, memory, kernel, operating system, etc. Because of this, it is also less precise. Full system simulation takes a while and requires a lot more setup. The system has to be specified in its entirety: The full system (FS) section of the "Learning gem5" book [**lowe-power˙full˙nodate**] shows how in order to run a full system simulation, the various caches, disks, memory controllers, and North and South Bridges (for x86 system simulation) have to be specified as Python objects. Fortunately, the `devices.py` file in the example scripts directory already has all of this implemented for ARM full system simulation. The example power model script contains some very simple power models and is likely mainly to show how the power models are specified and added to the objects that support them. By default, the full system scripts support flags allowing to specify the frequency and number of big and LITTLE cores; where the kernel, disk, bootloader, and bootscript are located; whether to instantiate and simulate caches; what CPU type to use; whether to restore from a checkpoint; and much more.

4.2.3 Bootscripts, systems, and m5 instructions

One of the arguments that must be passed to the simulated system (either hard-coded in the Python configuration script or passed via a flag) is the path to the bootscript to run once the OS has booted. The gem5 Simulator comes with a couple of bootscripts, but these are for simulating distributed compute nodes, e.g. a compute cluster with multiple physical systems collaborating and communicating over a network, and therefore contains a lot of networking and other things that are unnecessary for this project. However, it does provide some insight into how to write these. As far as I can tell, even though the bootscripts all end in `.rcS`, the script itself is simply a bash script with most of the usual bash builtins supported. One slight complication is that the `PATH` environment variable seems to always be unset so absolute paths to programs must be specified rather than assuming they are available, e.g. you have to use `/bin/ls` to list directory contents because `ls` is not on the `PATH`.

In order to simulate a full system, a kernel, disk, and bootloader must be provided. The gem5 website hosts a number of Linux disk images at [25]. However, the gem5 website has recently been modernised and in the process the file which displayed an index of [25] seems to have disappeared. Therefore the way to get these system images is to use an old version of the m5 Simulator's webpage [26] which *does* have an index, copy the name of the file, and append it to the URL of [25]. This downloads the most recent version of the system image from the official gem5 website rather than relying on the old website(s) still being maintained. The system downloaded will typically contain `disk` and `binaries` subdirectories. The `disk` directory should contain a disk image (`.img`) file with BusyBox [7] and an `init` process pre-configured. It is possible to mount the disk image through a loopback device on Linux, using the command

```
sudo mount -o loop,offset=32256 \
    /path/to/disk-image.img /path/to/mountpoint
```

The `offset` specifies where the file system starts on the disk image and is derived from 63 sectors $\times$ a 512-byte sector size. Mounting the disk image can be useful in order to find or double-check the absolute paths of certain programs for use in a bootscript, or to place custom programs there in order to be able to run them in the simulator.

One crucial detail which seems to be missing in the gem5 documentation is that in order for gem5 to be able to run the full system, the environment variable `M5_PATH` has to be exported to contain the absolute path to the system files, i.e. the directory containing the '`binaries`' and '`disks`' subdirectories. If `M5_PATH` is not exported to this, gem5 will error saying that the files were not found on the path; there is no indication or mention of what path was searched. However, it is documented in the ARM Research Starter Kit [40].

The `m5` program, which is specific to interfacing with the simulator from within the simulated system, comes "pre-installed" on the gem5-provided disk images. It is typically located at `/sbin/m5` and is used to pass instructions to the simulator from within the simulated system. If the program is run without passing any arguments, it will print a list of the supported commands. The most useful one is `m5 checkpoint` which creates a checkpoint at the current simulation tick.

4.2.4 CPU types and checkpointing

By default, gem5 provides 3 CPU types for ARM system simulation: 'timing', 'atomic', and 'exynos'. The last one likely refers to the Samsung Exynos platform which may have special simulation requirements beyond the default ARM systems. As we were not using this platform, I did not look further into this setting. The 'timing' and 'atomic' CPU types have a key difference: the atomic CPU type is a simplified CPU where every instruction is atomic, i.e. it happens in one go and does not incur latency. This results in much faster simulations (around 20 minutes to boot Linux compared to around $1\frac{1}{2}$-2 hours using the timing CPU) at the cost of simulation realism. In reality, CPU instructions have a latency associated with them, and so some instructions take longer than others. The timing CPU models these latencies. As a result, it is a much more realistic simulation, at the cost of taking a lot longer to simulate.

The atomic CPU type is extremely useful for "fast-forwarding" and "checkpointing". As previously mentioned, running a full system simulation with the timing CPU takes a long time to even get the operating system booted. This is impractical if all we are interested in is the program being run once the operating system has booted, e.g. if we are running a benchmark, as only using the timing CPU would result in long wait times due to a process that the researcher is not interested in. To mitigate this, gem5 allows the user to create and resume from checkpoints: stores of exact system layout at a specific point in time, which can be used to resume the simulation from this state without needing to redo all the simulation preceding it. When resuming from a checkpoint, the CPU type can be changed. This allows us to "fast-forward" parts of the system we are not interested in by using the atomic CPU and calling the `m5 checkpoint` instruction from within the bootscript, before the line(s) where the bootscript starts the benchmark.

Listing 4.1: Example bootscript using checkpointing

```
#!/bin/bash

echo "bootscript.rcS is running"

/sbin/m5 checkpoint
echo "Starting workload"
/path/to/workload

echo "Workload done, exiting simulation."
/sbin/m5 exit
```

Once the bootscript has created the checkpoint, we can then interrupt the simulation and restart it with the timing CPU using the `--cpu-type` flag, and specify the path to the checkpoint to restore from using the `--resume-from` flag. Unfortunately, it seems that the bootscript is loaded into memory and hence stored along with the checkpoint, meaning separate bootscripts have to be created and fast-forwarded for separate workloads. Swapping the bootscript when resuming does not seem to have any effect and the simulator will instead use the bootscript that was initially passed to the simulator when fast-forwarding.

4.3 Interacting with the simulated system

It is possible to connect a terminal emulator to the simulated system and interact with it through this. The system's output is exposed on port 3456 by default, incrementing by 1 as simulations are run in parallel (i.e. 3457 for simulation 2, 3458 for simulation 3, etc.). Once option is to connect via the `telnet` command. However, gem5 also comes with a terminal emulator: `m5term`. The source code for it is found at `gem5/util/term/` and it needs to be compiled. Once `m5term` has been compiled, it can be connected to a system by running the command `./m5term <port>`, replacing `<port>` with the port number. For example, to log in to the simulated system, a bootscript containing the line `/sbin/getty 38400 ttyAMA0` has to be used. This creates a login prompt which can then be used to log in using the username and password 'root'. However, it is somewhat difficult to log in before the timout of 60 seconds occurs: key presses take a long time to be properly registered by the simulated system. Additionally, due to the complexity of running a full system simulation, the system is not the most responsive: tab completion takes around 30 seconds, simple commands like `ls` take 10-20 seconds to execute, and it is possible to follow the line printing with the naked eye. Fast-forwarding and using custom bootscripts is the most straightforward and painless way to use the system.

4.4 Cross Compilation

In order to be able to compile and run custom experiments on the simulated system, a cross compiler is needed. The x86 system(s) running the simulator are based on a different ISA than ARM and so programs compiled for x86 will not run on ARM due to differences in the assembler and binary instructions. A cross compiler is a program which runs on one ISA and compiles programs for a different ISA, e.g. an x86 program which produces ARM binaries. The ARM Developer webpage provides a GNU toolchain which contains the entire GNU Compiler Collection, including cross compilers for x86 based systems [20]. The systems I was examining were ARMv8, i.e. AArch64, systems so the `aarch64-none-linux-gnu` cross compiler was used. I discovered that this cross compiler also produced binaries which were compatible with the Odroid N2 board and vice-versa: programs compiled on the Odroid N2 ran in the simulator when copied over to a disk image. Some programs, e.g. the Linux kernel, provide a `CROSS_COMPILE` setting in the makefile, which can be set to instruct make to use the cross compilation toolchain instead of the host ISA's default toolchain, thereby facilitating cross compilation. The gem5 Simulator comes with two useful programs that may need cross compiling: ARM bootloaders and `m5`.

If setting up a different system from the provided ones, a bootloader is required for full system simulation. The source code for ARM bootloaders is provided in `gem5/system/arm/bootloader/arm[64]/` for 32-bit ARMv7 and 64-bit ARMv8 respectively. If additionally, a custom disk image is being prepared, the `m5` program used to interface with the simulator needs to be cross compiled and put on the image. The source code is found at `gem5/util/m5/` and for ARMv8, the `Makefile.aarch64` should be used.

4.5 Power models and stats

After getting gem5 to build and trying to learn how to set it up and use it, both through the "Learning gem5" book [31] and by looking over the configuration scripts themselves, I decided to try to get the power models working, as power consumption was one of the main interest of the project. Whenever the provided `fs_power.py` would be used as the configuration script, the simulator would crash right before starting the simulation, with the error message:

```
fatal: statistic '' (160) was not properly initialized by a regStats() function
```

I initially tried looking through documentation for both the "Power and Thermal Model" [17] and the "Stats package" [18]. However, neither parts seemed to contain any hints as to what might be causing the crash. I also tried looking through the source code for the CPUs and the stats framework, but due to its scale (around 13000 lines of code) and a lot of the code being C++ which I was not familiar with, this also did not solve or identify the problem.

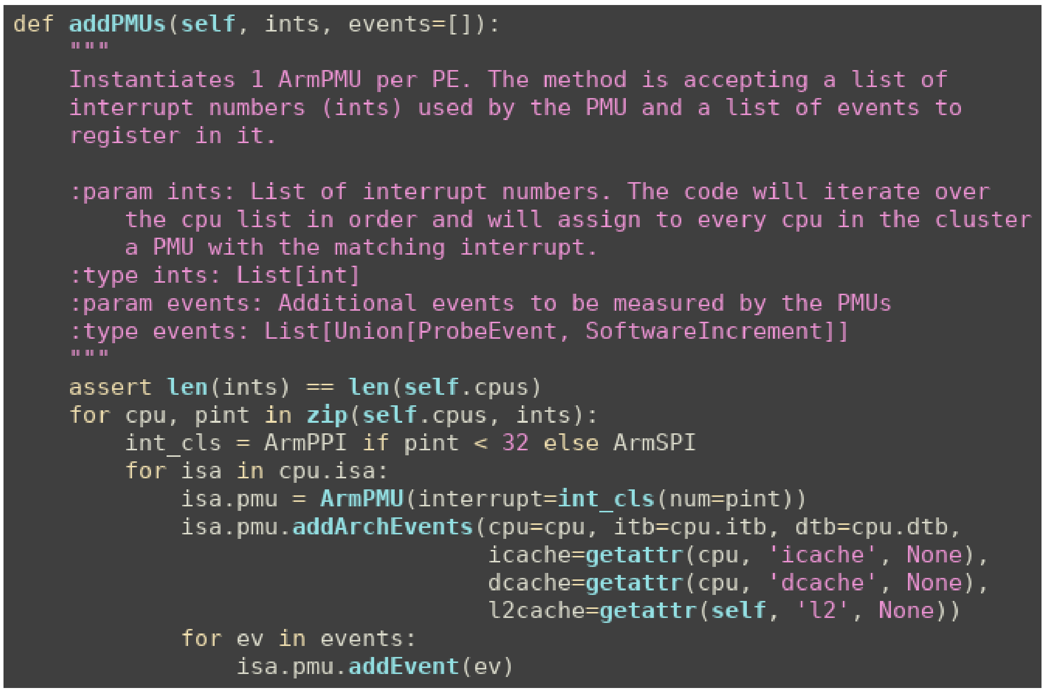
On the 7th of February, I raised an issue on the gem5 Jira pointing out that the script seemed to be broken [24] and continued to look into DVFS, scheduling, and AMPs whilst waiting for a reply. My supervisor also tried to get the script working. Just over a month later, some of the gem5 developers got back. It turned out that they had been completely refactoring the way stats were done and registered, and as a result, certain simulated components were broken when trying to use features that relied on accessing the stats framework whilst a simulation was running, e.g. any subclass of the `MathExprPowerModel`. With one of the developers now being aware of the problem, an initial patch was provided within a day. After applying the patch and rebuilding gem5, the script managed to get past the initialisation step, but now crashed with the error:

```
warn: Failed to find stat 'dcache.overall_misses'
fatal: Failed to evaluate power expressions:
voltage * (2 * ipc + 3 * 0.000000001 * dcache.overall_misses / sim_seconds)
```

Based on the previous issue with registering stats, I assumed it might have been something to do with not registering the stats the power expression was using, however, it turned out to be an additional bug due to the recent shift in the way gem5 was handling stats internally and a further patch was provided within a couple of days of getting back with the new error. With both patches applied, the `fs_power.py` script ran without problems.

4.6 Simulating PMUs

Since PMUs are part of the ARM architecture, it would make sense for gem5 to simulate them. However, as far as I can tell, there is no documentation or guide as to where or how to access them. By looking through the provided `devices.py` script, the `CpuCluster` *does* seem to add some PMU events to the system:

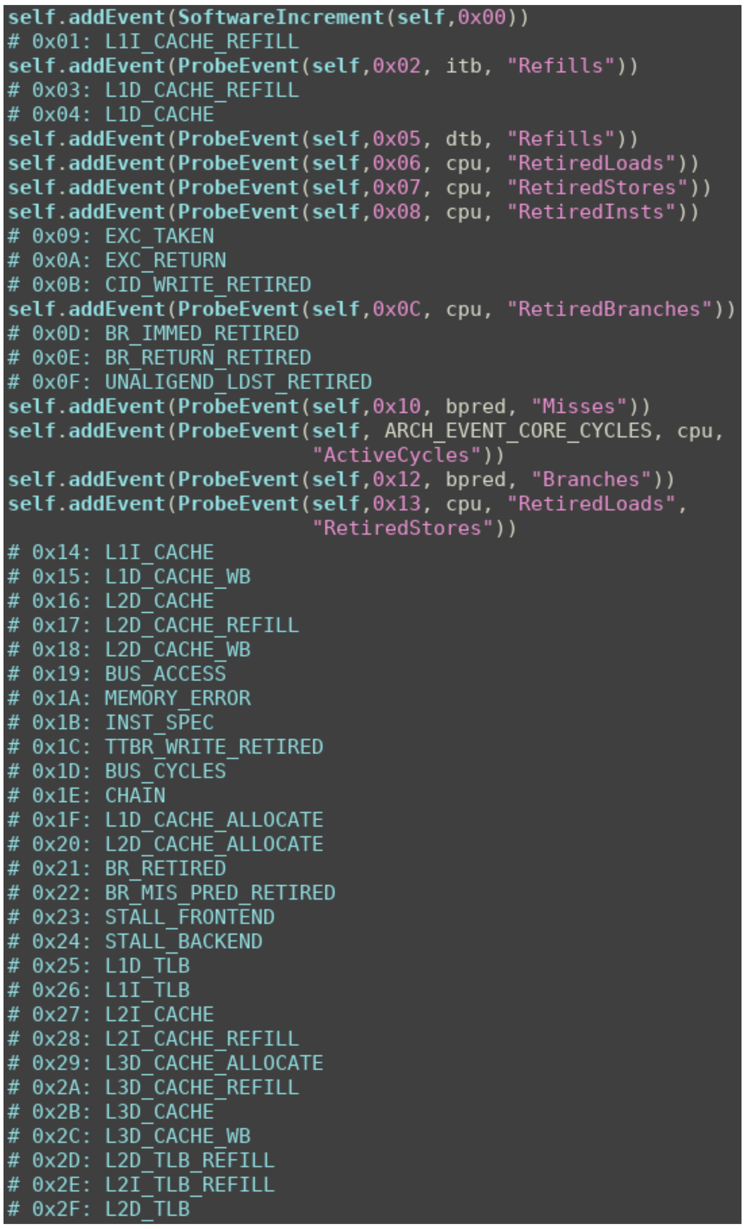


```
def addPMUs(self, ints, events=[]):
    """
    Instantiates 1 ArmPMU per PE. The method is accepting a list of
    interrupt numbers (ints) used by the PMU and a list of events to
    register in it.

    :param ints: List of interrupt numbers. The code will iterate over
        the cpu list in order and will assign to every cpu in the cluster
        a PMU with the matching interrupt.
    :type ints: List[int]
    :param events: Additional events to be measured by the PMUs
    :type events: List[Union[ProbeEvent, SoftwareIncrement]]
    """
    assert len(ints) == len(self.cpus)
    for cpu, pint in zip(self.cpus, ints):
        int_cls = ArmPPI if pint < 32 else ArmSPI
        for isa in cpu.isa:
            isa.pmu = ArmPMU(interrupt=int_cls(num=pint))
            isa.pmu.addArchEvents(cpu=cpu, itb=cpu.itb, dtb=cpu.dtb,
                                  icache=getattr(cpu, 'icache', None),
                                  dcache=getattr(cpu, 'dcache', None),
                                  l2cache=getattr(self, 'l2', None))
            for ev in events:
                isa.pmu.addEvent(ev)
```

Figure 4.1: `addPMUs` function found in `devices.py`

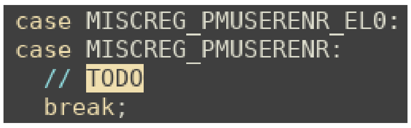
However, setting up a simple program which enables and accesses the PMUs did not make any progress when run on the simulator with the function above being called when initialising the system. The program never seemed to proceed, but when interrupting the simulator, the number of ticks was much greater than the observed average number of ticks to completely boot, indicating that the boot process had completed but the program was not running.

Taking a closer look at the `ArmPMU.py` source code (Figure 4.2; found at `gem5/src/arch/arm/`) I discovered that very few of the PMU events seemed to be implemented. There was unfortunately also no hints in the code comments as to how to access or use the PMUs during the simulation. Looking through the relevant C++ files for `ArmPMU.py`, I discovered a '`//TODO`' at the PMU register which normally enables user access to the PMU registers. This, combined with the complete lack of documentation, seems to suggest the PMUs are unfortunately a work in progress and as such, I was unable to use them for my simulations.



```
self.addEvent(SoftwareIncrement(self,0x00))
# 0x01: L1I_CACHE_REFILL
self.addEvent(ProbeEvent(self,0x02, itb, "Refills"))
# 0x03: L1D_CACHE_REFILL
# 0x04: L1D_CACHE
self.addEvent(ProbeEvent(self,0x05, dtb, "Refills"))
self.addEvent(ProbeEvent(self,0x06, cpu, "RetiredLoads"))
self.addEvent(ProbeEvent(self,0x07, cpu, "RetiredStores"))
self.addEvent(ProbeEvent(self,0x08, cpu, "RetiredInsts"))
# 0x09: EXC_TAKEN
# 0x0A: EXC_RETURN
# 0x0B: CID_WRITE_RETIRED
self.addEvent(ProbeEvent(self,0x0C, cpu, "RetiredBranches"))
# 0x0D: BR_IMMED_RETIRED
# 0x0E: BR_RETURN_RETIRED
# 0x0F: UNALIGEND_LDST_RETIRED
self.addEvent(ProbeEvent(self,0x10, bpred, "Misses"))
self.addEvent(ProbeEvent(self, ARCH_EVENT_CORE_CYCLES, cpu,
                         "ActiveCycles"))
self.addEvent(ProbeEvent(self,0x12, bpred, "Branches"))
self.addEvent(ProbeEvent(self,0x13, cpu, "RetiredLoads",
                         "RetiredStores"))
# 0x14: L1I_CACHE
# 0x15: L1D_CACHE_WB
# 0x16: L2D_CACHE
# 0x17: L2D_CACHE_REFILL
# 0x18: L2D_CACHE_WB
# 0x19: BUS_ACCESS
# 0x1A: MEMORY_ERROR
# 0x1B: INST_SPEC
# 0x1C: TTBR_WRITE_RETIRED
# 0x1D: BUS_CYCLES
# 0x1E: CHAIN
# 0x1F: L1D_CACHE_ALLOCATE
# 0x20: L2D_CACHE_ALLOCATE
# 0x21: BR_RETIRED
# 0x22: BR_MIS_PRED_RETIRED
# 0x23: STALL_FRONTEND
# 0x24: STALL_BACKEND
# 0x25: L1D_TLB
# 0x26: L1I_TLB
# 0x27: L2I_CACHE
# 0x28: L2I_CACHE_REFILL
# 0x29: L3D_CACHE_ALLOCATE
# 0x2A: L3D_CACHE_REFILL
# 0x2B: L3D_CACHE
# 0x2C: L3D_CACHE_WB
# 0x2D: L2D_TLB_REFILL
# 0x2E: L2I_TLB_REFILL
# 0x2F: L2D_TLB
```

(a) `ArmPMU.py`: Very few events seem to be implemented



```
case MISCREG_PMUSERENR_EL0:
case MISCREG_PMUSERENR:
  // TODO
  break;
```

(b) `pmu.cc`: The PMUs might be a work in progress

Figure 4.2: Looking through the PMU-relevant source code

4.7 Customising the setups

While the provided Python scripts help define almost everything, there are certain things that can be modelled which are not present in the scripts, due to their already high complexity. In order to include these things in the simulation, it is therefore necessary to extend and/or modify the provided scripts.

4.7.1 Voltage and Frequency Domains

Part of this project was to look at whether scheduling and DVFS can be combined to achieve more intelligent, power efficient schedules. Therefore, we need the system we are simulating to have DVFS capabilities. In gem5, adding DVFS to a simulation is done by assigning "Frequency Domains" to certain objects. Since DVFS impacts the frequency, i.e. the clock speed, of the object, any `ClockedObject` (which the CPU implementations are a subclass of) can be assigned a Clock Domain. The documentation for adding DVFS to a system is only available on the old website [14] and it does not detail how to add DVFS for the `fs_bigLITTLE.py` configuration. In order to figure this out, it is required to look at the script *and* some of the gem5 source code.

Listing 4.2: Lines 42-52 from `gem5/src/sim/VoltageDomain.py`

```
class VoltageDomain(SimObject):
    type = 'VoltageDomain'
    cxx_header = "sim/voltage_domain.hh"

    # Single or list of voltages for the voltage domain. If only a single
    # voltage is specified, it is used for all different frequencies.
    # Otherwise, the number of specified voltges and frequencies in the clock
    # domain (src/sim/ClockDomain.py) must match.  Voltages must be specified in
    # descending order. We use a default voltage of 1V to avoid forcing users to
    # set it even if they are not interested in using the functionality
    voltage = VectorParam.Voltage('1V', "Operational␣voltage(s)")
```

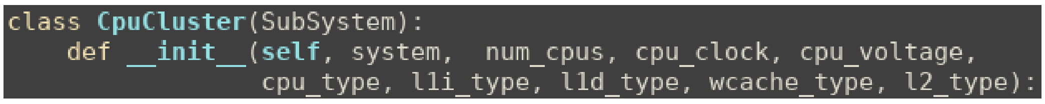
Listing 4.3: Lines 44-69 from `gem5/src/sim/ClockDomain.py`

```
# Abstract clock domain
class ClockDomain(SimObject):
    type = 'ClockDomain'
    cxx_header = "sim/clock_domain.hh"
    abstract = True

# Source clock domain with an actual clock, and a list of voltage and frequency
# op points
class SrcClockDomain(ClockDomain):
    type = 'SrcClockDomain'
    cxx_header = "sim/clock_domain.hh"

    # Single clock frequency value, or list of frequencies for DVFS
    # Frequencies must be ordered in descending order
    # Note: Matching voltages should be defined in the voltage domain
    clock = VectorParam.Clock("Clock period")

    # A source clock must be associated with a voltage domain
    voltage_domain = Param.VoltageDomain("Voltage domain")

    # Domain ID is an identifier for the DVFS domain as understood by the
    # necessary control logic (either software or hardware). For example, in
    # case of software control via cpufreq framework the IDs should correspond
    # to the neccessary identifier in the device tree blob which is interpretted
    # by the device driver to communicate to the domain controller in hardware.
    domain_id = Param.Int32(-1, "domain id")
```

By examining the source code of the Voltage and Clock Domain objects, we discover several things not apparent in the guide/documentation:

1. Voltage and frequency points have to be specified in sorted, descending order.
2. There can either be one voltage value for all frequency steps or there must be exactly the same number of voltage and frequency steps.
3. Clock domains have an ID which must be unique.

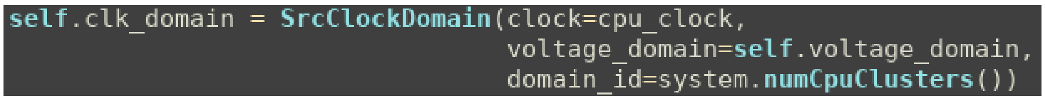
When looking at the provided system devices in `gem5/configs/example/arm/devices.py`, the `CpuCluster` constructor already has constructor support for specifying voltage and clock domains.



```
class CpuCluster(SubSystem):
    def __init__(self, system,  num_cpus, cpu_clock, cpu_voltage,
                 cpu_type, l1i_type, l1d_type, wcache_type, l2_type):
```

Figure 4.3: The `CpuCluster` constructor

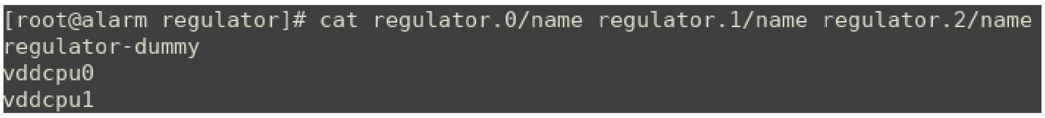
Passing a list of values to the `cpu_clock` and `cpu_voltage` arguments constructs the relevant objects inside the `CpuCluster` object. The only part that needs changing is that the clock domains are not given an ID. This causes the simulator to crash, as the logic detects that the default ID of -1 was given and errors with a message explaining that clock domains must have a unique ID. As part of the project, an Odroid N2 board was acquired. By examining its DVFS capabilities using the `cpupower` tool, it seems the most common setup is to have one DVFS domain per cluster, i.e. per big and LITTLE ‘part’ of the setup. Therefore, I change the line initialising the voltage domain to assign an ID based on the number of clusters in the system. This will return 0 for the first cluster, i.e. cluster 0, and 1 for the second cluster.



```
self.clk_domain = SrcClockDomain(clock=cpu_clock,
                                 voltage_domain=self.voltage_domain,
                                 domain_id=system.numCpuClusters())
```

Figure 4.4: Modified line 126 of `devices.py`

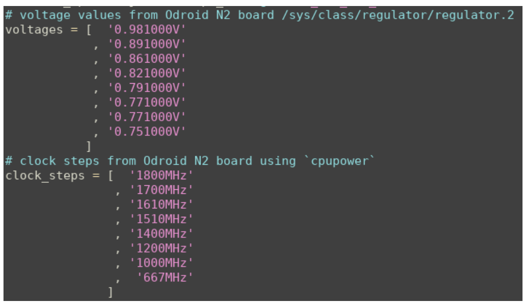
With the devices modified to correctly set up the DVFS objects in the simulator, the only thing that remains to be changed is the `fs_bigLITTLE.py` file. In an attempt to model DVFS as realistically as possible, I used clock and voltage steps from the Odroid N2 board that was provided for this project. The clock steps were collected through the `cpupower` tool. The voltage steps were harder figure out where to find. It turns out that Linux provides some insight into the voltage values through the sysfs, i.e. the `/sys` file system. There are a number of voltage regulators at `/sys/class/regulator/regulator.`{`0,1,2`}. By using the `userspace` frequency governor, which scales the frequency to whatever the user specifies, and the `cpupower` tool to adjust the frequency, I was able to narrow down that regulators 1 and 2 control the voltage of the LITTLE and big clusters respectively. Regulator 0 seems to not do anything, as running `cat` on the `regulator.0/name` outputs the value “`regulator-dummy`”, indicating that the regulator is unlikely to control anything.



```
[root@alarm regulator]# cat regulator.0/name regulator.1/name regulator.2/name
regulator-dummy
vddcpu0
vddcpu1
```

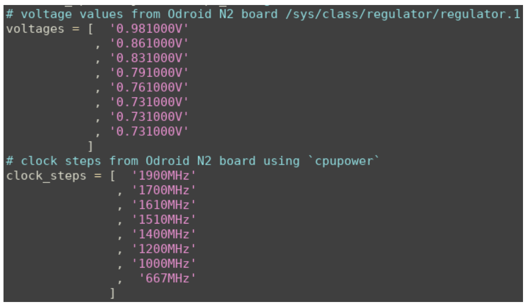
Figure 4.5: The names of the regulators on the Odroid N2 board

In order to model the different DVFS steps for the big and LITTLE clusters, I extended the `CpuCluster` class into 2 clusters: `DVFSBigCluster` and `DVFSLittleCluster`. These subclasses were configured identically to the existing `BigCluster` and `LittleCluster` classes, with the exception of specifying different voltage and clock domains in each one. I also extended the `AtomicCluster` class into a `DVFSBigAtomicCluster` and `DVFSLittleAtomicCluster` class, in order to be able to fast-forward the systems. The changes were made in a copy of the `fs_bigLITTLE.py` file: `fs_bL_extended.py`.



```
# voltage values from Odroid N2 board /sys/class/regulator/regulator.2
voltages = [  '0.981000V'
            , '0.891000V'
            , '0.861000V'
            , '0.821000V'
            , '0.791000V'
            , '0.771000V'
            , '0.771000V'
            , '0.751000V'
           ]
# clock steps from Odroid N2 board using `cpupower`
clock_steps = [  '1800MHz'
               , '1700MHz'
               , '1610MHz'
               , '1510MHz'
               , '1400MHz'
               , '1200MHz'
               , '1000MHz'
               ,  '667MHz'
              ]
```

(a) DVFS values for the big cluster

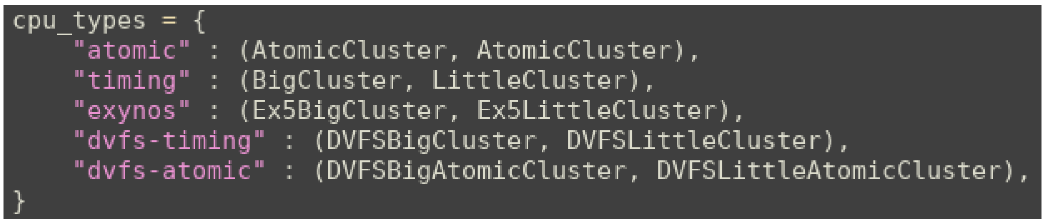


```
# voltage values from Odroid N2 board /sys/class/regulator/regulator.1
voltages = [  '0.981000V'
            , '0.861000V'
            , '0.831000V'
            , '0.791000V'
            , '0.761000V'
            , '0.731000V'
            , '0.731000V'
            , '0.731000V'
           ]
# clock steps from Odroid N2 board using `cpupower`
clock_steps = [  '1900MHz'
               , '1700MHz'
               , '1610MHz'
               , '1510MHz'
               , '1400MHz'
               , '1200MHz'
               , '1000MHz'
               ,  '667MHz'
              ]
```

(b) DVFS values for the LITTLE cluster

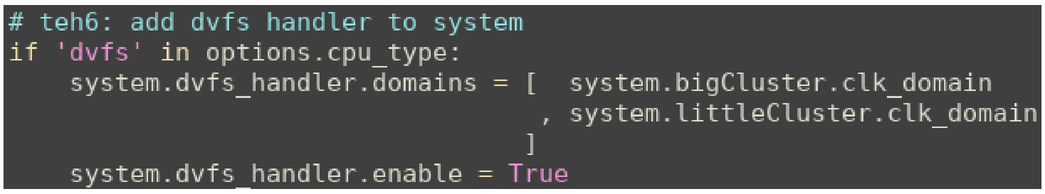
Figure 4.6: DVFS setups based on the Odroid N2 board

Finally, to wire everything up in the script and be able to use the clusters, big.LITTLE pairs of the clusters were added to the `cpu_types` dictionary, allowing them to be selected through the `--cpu-type` flag, and as shown in one of examples in the old documentation [14], the system's DVFS-handler was configured and enabled in cases where the DVFS clusters were being used.



```
cpu_types = {
    "atomic" : (AtomicCluster, AtomicCluster),
    "timing" : (BigCluster, LittleCluster),
    "exynos" : (Ex5BigCluster, Ex5LittleCluster),
    "dvfs-timing" : (DVFSBigCluster, DVFSLittleCluster),
    "dvfs-atomic" : (DVFSBigAtomicCluster, DVFSLittleAtomicCluster),
}
```

(a) The DVFS clusters included in `cpu_types`



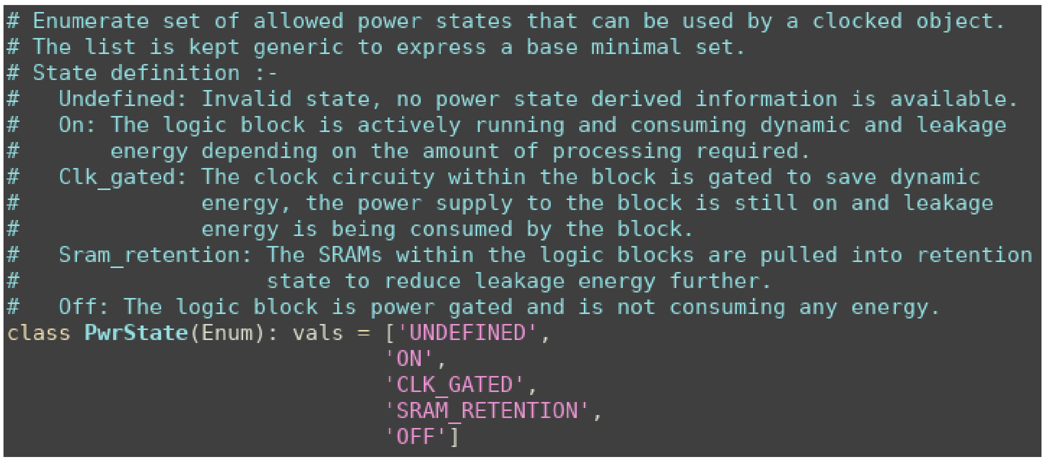
```
# teh6: add dvfs handler to system
if 'dvfs' in options.cpu_type:
    system.dvfs_handler.domains = [  system.bigCluster.clk_domain
                                   , system.littleCluster.clk_domain
                                  ]
    system.dvfs_handler.enable = True
```

(b) If DVFS models are used, configure and enable the DVFS handler

Figure 4.7: Extensions made to be able to use the DVFS extensions

4.7.2 Power Models

The example power models provided with gem5 are explicitly labelled as toy models: when using the `fs_power.py` script, the script will print a warning to the user that "The power numbers generated by this script are examples. They are not representative of any particular implementation or process". However, the script provides a good overview of how to add custom power models. Power models consist of four functions, each describing how to calculate the power consumption in a different power state. The provided example script only assigns a power model to the `ON` power state and assigns a simple function returning 0 to the other states. Although each power state is labelled with a comment, the explanations of what the power states mean can only be found by looking at the `ClockedObject.py` source code:



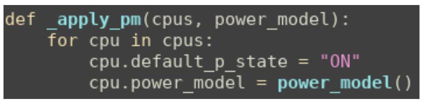
```
# Enumerate set of allowed power states that can be used by a clocked object.
# The list is kept generic to express a base minimal set.
# State definition :-
#   Undefined: Invalid state, no power state derived information is available.
#   On: The logic block is actively running and consuming dynamic and leakage
#       energy depending on the amount of processing required.
#   Clk_gated: The clock circuity within the block is gated to save dynamic
#              energy, the power supply to the block is still on and leakage
#              energy is being consumed by the block.
#   Sram_retention: The SRAMs within the logic blocks are pulled into retention
#                   state to reduce leakage energy further.
#   Off: The logic block is power gated and is not consuming any energy.
class PwrState(Enum): vals = ['UNDEFINED',
                             'ON',
                             'CLK_GATED',
                             'SRAM_RETENTION',
                             'OFF']
```

Figure 4.8: Lines 42-58 of `ClockedObject.py`, explaining the possible power states

Based on the `fs_power.py` script, the functions for modelling power extend the `MathExprPowerModel` class. The `MathExprPowerModel` class allows the equations to use the stats being recorded by gem5, e.g. instructions per cycle, for computing the power consumption [6] and contains two equations: `dyn` and `st`. These represent the dynamic and static power consumptions of the system, respectively. The static power is the power that the component is consuming in the background, i.e. 'statically', and the dynamic power is the power consumption which changes 'dynamically' depending on what the system is doing. Since we are interested in whether DVFS and scheduling can result in power savings, we need to define power models so that gem5 can simulate and output power readings.

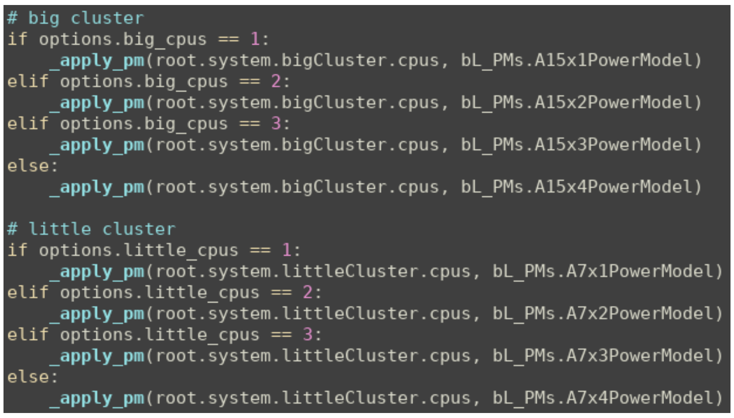
A paper published in 2018 by Walker et al. [42] presents a tool called 'gemstone' which can be used for more accurate power modelling. Specifically, the 'gemstone-applypower' tool [41] can automatically generate power equations which can be used in gem5, based on the data collected in [42]. Using this tool, I created the `fs_bL_power_models.py` file to create and easily import the power models generated by [41]. Since one of the big advantages of using a simulator is that the core configurations can be changed, I customised the power models based on how many cores were available, and created new `MathExprPowerModel` subclasses for each core configuration, e.g.

`BigCpuA15x4PowerOn` or `LittleCpuA7x2PowerOn` for 4 big and 2 LITTLE cpus respectively. The CPUs in the Odroid N2 board are not A15 and A7 cores, but instead A73 and A53 cores. Additionally, modifying equations which were based on hardware readings from a board with a 4 big, 4 LITTLE (4b4L) configuration is unlikely to be accurate. However, the modified power equations should still be better than the toy models provided by the gem5 example script as they take many more factors into account, e.g. the current clock frequency, certain floating point operations, the number of cycles that was simulated, etc. With the various power models in one file, they could be imported and the script configured to apply them based on the number of CPUs specified, using a helper function to make sure the power models were applied to all the CPUs in the cluster:



```python
def _apply_pm(cpus, power_model):
    for cpu in cpus:
        cpu.default_p_state = "ON"
        cpu.power_model = power_model()
```

(a) Helper function to apply power models to all CPUs in the cluster.



```python
# big cluster
if options.big_cpus == 1:
    _apply_pm(root.system.bigCluster.cpus, bL_PMs.A15x1PowerModel)
elif options.big_cpus == 2:
    _apply_pm(root.system.bigCluster.cpus, bL_PMs.A15x2PowerModel)
elif options.big_cpus == 3:
    _apply_pm(root.system.bigCluster.cpus, bL_PMs.A15x3PowerModel)
else:
    _apply_pm(root.system.bigCluster.cpus, bL_PMs.A15x4PowerModel)

# little cluster
if options.little_cpus == 1:
    _apply_pm(root.system.littleCluster.cpus, bL_PMs.A7x1PowerModel)
elif options.little_cpus == 2:
    _apply_pm(root.system.littleCluster.cpus, bL_PMs.A7x2PowerModel)
elif options.little_cpus == 3:
    _apply_pm(root.system.littleCluster.cpus, bL_PMs.A7x3PowerModel)
else:
    _apply_pm(root.system.littleCluster.cpus, bL_PMs.A7x4PowerModel)
```

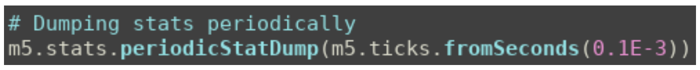
(b) Applying power models based on no. CPUs

Figure 4.9: Extending `fs_bL_extended.py` to apply power models

Finally, in order to keep the `fs_bL_extended.py` script as flexible as possible, I added a `--power-models` flag which would toggle the code block applying the power models instead of always trying to apply them. This was necessary, as the power models only work with the detailed 'timing' CPU type.

4.7.3 Output resolution

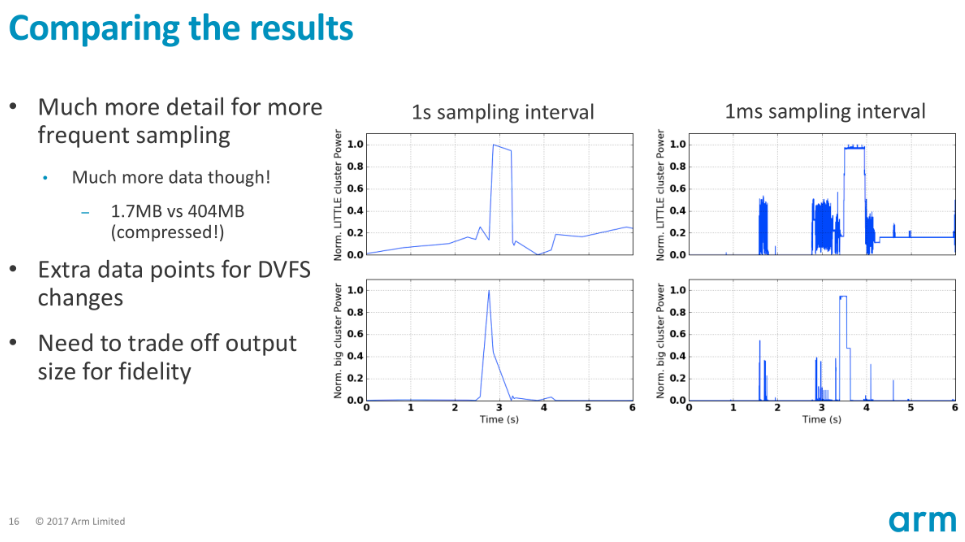
By default, gem5 dumps simulation statistics every simulated second. The statistics, found in the `outdir/stats.txt` file, contain numerous insights and measurements, including memory bus packet sizes, number of floating point units used, number of instructions committed, and much more. It is possible to adjust how often gem5 dumps these stats, by using the `m5.stats.periodicStatDump` function in the Python configuration script. The function takes a period in simulated ticks, but gem5 provides a function `m5.ticks.fromSeconds` to easily be able to specify the frequency in seconds and then convert it to simulated ticks.



```
# Dumping stats periodically
m5.stats.periodicStatDump(m5.ticks.fromSeconds(0.1E-3))
```

Figure 4.10: Lines 129-130 from `fs_power.py`, demonstrating how to control the stats dump frequency

In [6] slide 16 (Figure 4.11), it is shown how drastically different stats dump frequency can affect the resolution of power models. Therefore, I added a `--stat-freq` flag to the `fs_bL_extended.py` script, which allows the user to specify how frequently gem5 should dump stats, including support for scientific notation, e.g. `1e−3` for milliseconds.



Comparing the results

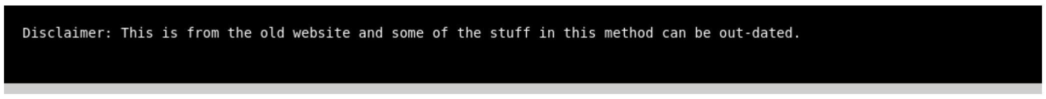
- Much more detail for more frequent sampling
  - Much more data though!
    - 1.7MB vs 404MB (compressed!)
- Extra data points for DVFS changes
- Need to trade off output size for fidelity

16 © 2017 Arm Limited

Figure 4.11: Slide 16 from [6]

4.8 Stats access from power models

After importing the power models generated by [41] and setting up the scripts as detailed in Section 4.7.2, the majority of the stats being used in the more detailed power models were unable to be found by the simulator. This was somewhat perplexing as the exact strings used in the power expressions were present in the `stats.txt` output when a simulation was run without the power models enabled. Trying once more to look through the power [17] and stats [18] documentation did not lead anywhere since the documentation does not detail which stats might exist or how to correctly access them. Unfortunately, I had managed to time the project at the same time that gem5 was migrating their website to a new one and as such, a lot of the documentation was missing or not in the places one would expect to find it. Furthermore, it turned out that the DVFS documentation was officially labelled out of date [15] and trying to look through more of the documentation revealed a similar pattern. Many pages would have a disclaimer at their top saying that they were from the old website and might be outdated as a result.

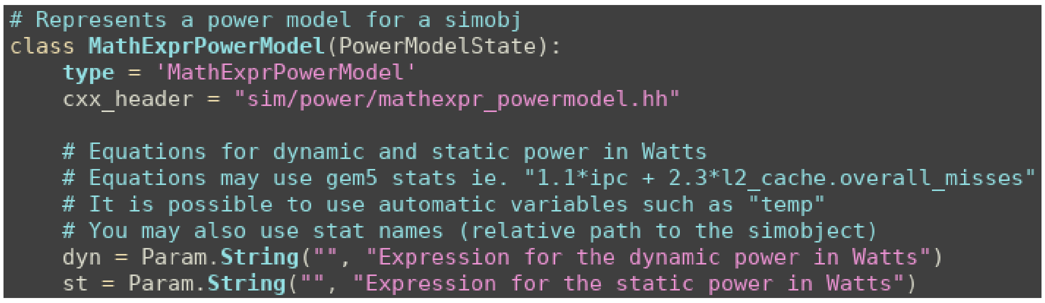


```
Disclaimer: This is from the old website and some of the stuff in this method can be out-dated.
```

Figure 4.12: Disclaimer initially found on many documentation pages due to their recent migration from the old website

Looking at the `MathExprPowerModel` source code (Figure 4.13) suggested that the stat names used possibly had to be relative to the simulated object that was referencing them. This led me to try to print the output of the `getStatGroups` and `getStats` functions which were associated with each `SimObject`. The results of this can be seen in Appendix A. However, this just seemed to confirm that the stats the simulator was crashing on *were* present. Trying to specify ‘relative’ stats, e.g. replacing the use of
`system.bigCluster.clk_domain.clock`
with `clk_domain.clock` or `Parent.clk_domain.clock` (both of which seemed sensible based on the outputs from the `getStats` and `getStatGroups` functions), did not solve the problem and still caused the simulator to crash, now specifying the renamed attempts as the stats not being found. As it was unclear which objects could be assigned a power model, and given that the expressions generated by [41] seemed to specify the full ‘path’, I decided to try assigning the power models to the top-level `system` object and I also tried to assign power models to the clusters. Neither solved the problem: assigning the power model to the system caused the simulator to error in a different way, and assigning them to the clusters caused it to segfault.

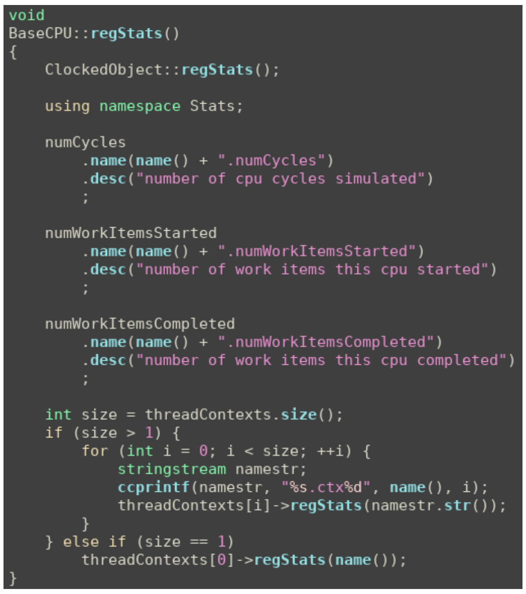
My next idea was that this might have something to do with the new stats framework and the `regStats` function which had been the source of previous problems. However, as far as I was able to tell, there is no documentation of what stats are registered by the various objects. In order to find this out, one has to find the



```
# Represents a power model for a simobj
class MathExprPowerModel(PowerModelState):
    type = 'MathExprPowerModel'
    cxx_header = "sim/power/mathexpr_powermodel.hh"

    # Equations for dynamic and static power in Watts
    # Equations may use gem5 stats ie. "1.1*ipc + 2.3*l2_cache.overall_misses"
    # It is possible to use automatic variables such as "temp"
    # You may also use stat names (relative path to the simobject)
    dyn = Param.String("", "Expression for the dynamic power in Watts")
    st = Param.String("", "Expression for the static power in Watts")
```

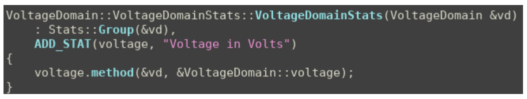
Figure 4.13: A comment in the source code mentions that stats may have to be relative to the `SimObject` they are being used with

`regStats` function in the C++ source code relevant to the `SimObject` in question. Furthermore, some knowledge of the object hierarchy is required as stats registered in a superclass are available to the power models in a subclass. One of the stats that was not mentioned in the gem5 crash messages was the `numCycles` stat. This stat is registered by the `BaseCpu` class and was referable exactly as registered: by the `numCycles` string. Once I found the source code for the voltage and clock domains (located in `gem5/src/sim/power/`) I discovered that they *did* seem to register the stats that the simulator claimed to be unable to find.



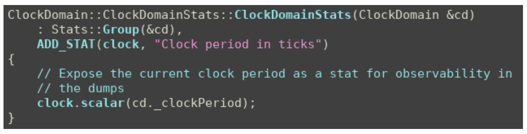
```
void
BaseCPU::regStats()
{
    ClockedObject::regStats();

    using namespace Stats;

    numCycles
        .name(name() + ".numCycles")
        .desc("number of cpu cycles simulated")
        ;

    numWorkItemsStarted
        .name(name() + ".numWorkItemsStarted")
        .desc("number of work items this cpu started")
        ;

    numWorkItemsCompleted
        .name(name() + ".numWorkItemsCompleted")
        .desc("number of work items this cpu completed")
        ;

    int size = threadContexts.size();
    if (size > 1) {
        for (int i = 0; i < size; ++i) {
            stringstream namestr;
            ccprintf(namestr, "%s.ctx%d", name(), i);
            threadContexts[i]->regStats(namestr.str());
        }
    } else if (size == 1)
        threadContexts[0]->regStats(name());
}
```

(a) Stats registered by `BaseCpu`



```
VoltageDomain::VoltageDomainStats::VoltageDomainStats(VoltageDomain &vd)
    : Stats::Group(&vd),
    ADD_STAT(voltage, "Voltage in Volts")
{
    voltage.method(&vd, &VoltageDomain::voltage);
}
```

(b) Stats registered by `VoltageDomain`



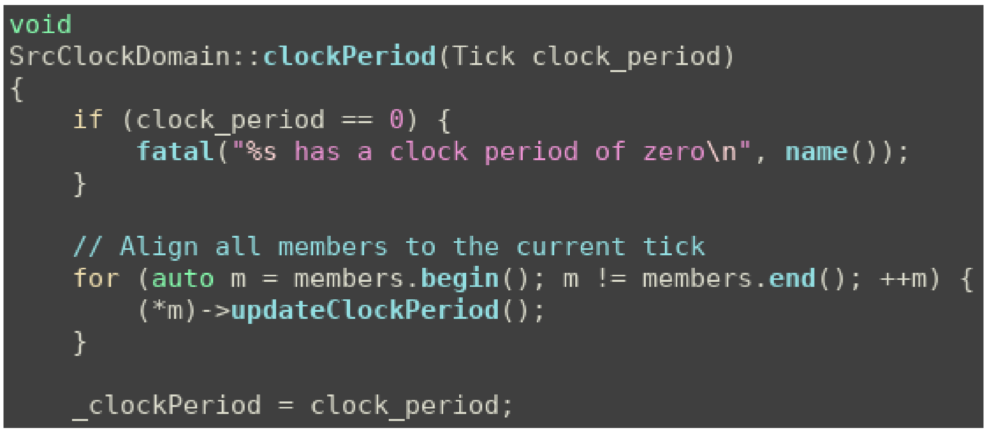
```
ClockDomain::ClockDomainStats::ClockDomainStats(ClockDomain &cd)
    : Stats::Group(&cd),
    ADD_STAT(clock, "Clock period in ticks")
{
    // Expose the current clock period as a stat for observability in
    // the dumps
    clock.scalar(cd._clockPeriod);
}
```

(c) Stats registered by `ClockDomain`

Figure 4.14: Various `regStats` functions searched to find stat names

Having exhausted every attempt at finding and fixing what I presumed was a simple issue, I opened another issue on the gem5 Jira [23]. As this was now very close to the project deadline, I reverted back to using the toy power models because whilst not hardware-validated, they did produce power output which varied depending on the workload and DVFS state of the system, and getting some results was a very high priority. I also looked into potentially using a completely different simulator, in order to get any results. As I did not expect to get an answer immediately or to necessarily have the developers be able to solve it in an instant, I set up the experiments described in Chapter 5 as soon as possible.

I received a response to the issue the next day along with a pointer to some existing patches which could be possible solutions. However, these had merge conflicts with the previous patches that were applied to fix the `regStats` bug and eventually, by discussing the issue with the developer helping me, we found out that it was not a problem with the simulator but with the naming of the stats. The stat that the power expression was trying to reference was `clk_domain.clock`. Based on both the `stats.txt` output and the clock domain's `regStats` function, this seemed to be the name of the stat. Instead, it turns out that the current clock value can be is accessed through the string `clock_period`. This parameter only appears internally in the `clock_domain.cc` file, where it is assigned to the `_clockPeriod` field (see Figure 4.15), which is then used in the `regStats` method (see Figure 4.14c). Other than this, there is no indication anywhere that this might be the stat name. Unfortunately, as I was busy trying to get any results ready and refactoring all the power equations would have taken some time, this solution never made it into the project.



```
void
SrcClockDomain::clockPeriod(Tick clock_period)
{
    if (clock_period == 0) {
        fatal("%s has a clock period of zero\n", name());
    }

    // Align all members to the current tick
    for (auto m = members.begin(); m != members.end(); ++m) {
        (*m)->updateClockPeriod();
    }

    _clockPeriod = clock_period;
```

Figure 4.15: Only uses of the `_clockPeriod` field

4.9 Looking into a different simulator

Having seemingly exhausted most options for getting power output from gem5, my supervisor suggested that we try using the Sniper Simulator [8, 9] due to its claims of having “[f]ull DVFS support” [37]. Sniper is only for x86 simulation and not ARM, but it could still have been useful. Unfortunately, it was impossible to get it to build. Version 7.2, the version available when looking into it, failed to build despite trying different versions of the Intel PIN tool that it depends on [19] and looking at the Google Groups error reporting, it seemed this was a common problem, even with the Docker image [13]. Because of this, and because the toy power models in gem5 seemed to work, no further attempts to get Sniper working were made.

Chapter 5

Experimental Setup

5.1 Setting up the benchmarks

For simulating workloads, the Splash 3 benchmark [35] was chosen. Splash 3 is an iteration on Splash 2 which removes data races from the programs used in the benchmarks [35].

The Splash 3 makefile does not provide a convenient `CROSS_COMPILE` setting for cross compilation. I initially attempted to add this to the makefile but was unsuccessful in getting the the benchmark series to build with the cross compiler. It turned out that some of the benchmark programs compile and run a smaller sub-program as part of their build process, making it impossible to cross compile them due to binary incompatibility between the intermediary programs and the host x86 system. To mitigate this, I copied the benchmark files to a USB-stick and compiled them on the Odroid N2 board.

The disk image provided with the aarch-system-20170616 files fortunately had enough space to fit the entire compiled Splash 3 benchmark. Before copying the files to this, I had attempted to set up my own disk image in order to be certain that there was enough space and that only the things I needed were on there. Creating a file, mounting it using a loopback device, and creating a file system on it was fine. However, in order to have a fully functional system, several tools and libraries had to be installed, e.g. a C standard library. Looking further into it, it seemed very close to Linux From Scratch, or creating a Linux distribution, which was far beyond the scope and time constraints of the project. Once the existing disk image was mounted (method detailed in Section 4.2.3), the files were placed in a new `splash3` directory created at root.

5.2 Setups

The gem5 Simulator supports up to 64 cores when modelling ARM systems [15]. As the number of cores being simulated increases, so does the simulation time. Both due to time constraints and the number of computational resources available, I decided to run 8 different setups: 1b1L, 2b2L, 3b3L, 4b4L, 2b4L, 4b2L, 1b3L, and 3b1L. The 4b2L setup is the same setup as the Odroid N2 board, and the other setups should give a good opportunity to examine how the programs behave when having equal numbers of big and LITTLE cores, and when there is an imbalance between the number of big and LITTLE cores available.

The Splash 3 benchmark comes with a README file containing recommended inputs for running the programs in a simulator. Common across all the programs is that the number of threads can be specified, either through a flag or a specific input file. As such, I decided setup benchmark runs with 1, 2, 4, 8, and 16 threads in order to have a variety of both under- and over-loaded usage across all the hardware configurations.

The latest Long Term Support (LTS) is version 5.4, released in Spring 2020. Since none of the gem5-hosted disk images use this kernel, and in order to have better control over what frequency governor was being used, I downloaded, configured and cross compiled version 5.4.24 (released on the 5th of March 2020) with only the `ondemand` frequency governor available and set as the default governor. This frequency governor should increase the DVFS points under load (on demand) and seems to be the most available on real hardware as both the Odroid boards (from hardkernel) and HiKey boards (from 96 Boards) use custom subversions of kernel 4.9 which is from before EAS and the `schedutil` governor were introduced.

5.3 Running

5.3.1 Technical details

The School of Computer Science's '`sif`' cluster was used to run the simulations. It contains 12 micro servers, with each one having dual quad-core Intel Xeon CPUs at 3.4GHz and 16GB of RAM, running Linux Fedora 28. Since the cluster is shared between all research staff, 9 of the 12 micro servers were used to run simulations. Each node was set up with `gem5.opt` version 2.0, from the git stable branch, with the patches supplied in [24] applied in order to make the stats framework function.

5.3.2 Scripts

Several layers of scripts were created. First, a number of bootscripts were created, one for each benchmark program and the number of threads to use, i.e. 5 bootscripts per benchmark. These scripts can be found in the `gem5-bootscripts/benchmarks` directory, grouped into subdirectories by benchmark program name.

In order to facilitate starting the simulations, a number of bash scripts were created. These took 6 command line arguments: the absolute path to the `gem5` directory, the number of big cores, the number of LITTLE cores, the number of threads, the path to the top-level directory in which to place outputs, and optionally the name of the frequency governor to use. The last argument was in case I managed to have time to run different governors to compare and defaulted to `ondemand`. Each script set up the simulator to redirect standard out and standard error, produce a DVFS config file, and set the output directory in a hierarchy of: program name, big.LITTLE configuration, and input file and number of threads combined with the frequency governor name. For the scripts which were to fast-forward the simulations, the stats output was fixed to `/dev/null` and the CPU-type to `dvfs-atomic`. For the scripts which were resuming from a checkpoint, an extra command line argument was added in order to be able to specify the path to the checkpoint to resume to. Additionally, the resuming scripts kept the stats output, set the CPU-type to the slower but more detailed `dvfs-timing` model, and set the stats dump frequency to once every simulated millisecond. These scripts can be found in the `gem5-commands/benchmarks` directory. The scripts starting with `ff-` are the fast-forwarding scripts.

Finally, in order to facilitate the fast-forwarding and resuming of simulations across the various hardware configurations, and in order easily manage the output directories, the scripts in `gem5-commands/configs` and `gem5-commands/resume-roi` were created. The scripts in the `configs` subdirectory loop over the thread settings. For each thread setting, the script starts 8 instances of the simulator, one per big.LITTLE config, with a timeout of 2 hours. This is enough time for most of the benchmarks to boot the kernel, create a checkpoint, and maybe start computation, apart from the `ocean-non-contiguous` benchmark, which required a timeout of 4 hours. All of the output was put in a `fast-forward` directory in order to easily be able to distinguish the output from fast-forwarded simulations. The scripts in the `resume-roi` subdirectory behave almost exactly the same as the ones in the `configs` subdirectory, apart from finding the checkpoint to resume from, using the non-fast-forwarding scripts, and putting the output in a `roi-out` directory. They also wait for all instances in a thread setting to finish before starting the next one, in order to not overload the 8 cores available on each micro server.

Each node (i.e. micro server) in the cluster was sent the scripts, with each node running the script relevant to a specific benchmark program.

Chapter 6

Results

6.1 Raw data

The simulations took varying amounts of time to complete with some of the longer programs taking 30-40 hours per thread version. Therefore, it took around a week before the majority of the data was available, with only the `volrend` program runs still going; they seemed to take around 70 hours each. Excluding the `volrend` benchmark, there was 320 simulations to retrieve: 8 benchmarks $\times$ 8 big.LITTLE configurations $\times$ 5 different number of threads. A total of 120GB of raw text data was accumulated from the programs excluding `volrend`, with each `stats.txt` file typically being between 2.5-4 million lines long. However, as most of the stats recorded by the simulator are much more detailed than what could be recorded and used by using PMUs, the extracted data size is much smaller.

6.1.1 Problems encountered

Network outage

As mentioned, the `volrend` benchmark took significantly longer than the other programs being run on the simulator. Unfortunately, this meant that it was still running during the University-wide network outage that occurred on the 9th of May. Although the vast majority of the University's network infrastructure was restored quickly, it took a while before access to the cluster was re-established. Additionally, it seems the network outage affected the `tmux` sessions that were being used to run the benchmarks, despite them being local to each micro server. As a result the `volrend` data was incomplete and was inaccessible until shortly before the deadline. Because of these issues, the `volrend` data was not used for the project.

Benchmark crashes

Each bootscript was set to print the line "`Benchmark done, exiting simulation.`" once the benchmark had returned. This allowed me to easily be able to tell if the runs had completed successfully, in case some of them crashed. Unfortunately, once the data was downloaded from the various micro servers and assembled in one place, it became apparent that despite running for a long time, a lot of the benchmarks seemed to have crashed near the end of their execution. By counting the number of times the "`Benchmark done`[...]" line occurred, the following completion ratios were obtained:

benchmark	**n_finished**	**n_crashed**	**completion_ratio**
`barnes`	20	20	50.0%
`fmm`	9	31	22.5%
`ocean-contiguous_partitions`	13	27	32.5%
`ocean-non_contiguous_partitions`	0	40	0.0%
`radiosity`	13	27	32.5%
`raytrace`	8	32	20.0%
`water-nsquared`	9	31	22.5%
`water-spatial`	5	35	12.5%

Table 6.1: Completion ratios for the various benchmarks

As mentioned earlier, most of the crashed benchmarks seem to have crashed towards the end of their execution. The exception is the `ocean-non_contiguous_partitions`, which seems to have crashed within milliseconds of starting. When plotting the power usage of the cores for some of the benchmarks (Appendix B), it becomes clear that the crashes occurred very near the end of the benchmark runs. If one were to look purely at the plots, it would be difficult to say which crashed and which did not based on their power consumption and time taken. Therefore, the data from the crashed benchmarks was kept in the final data set.

6.2 Data extraction

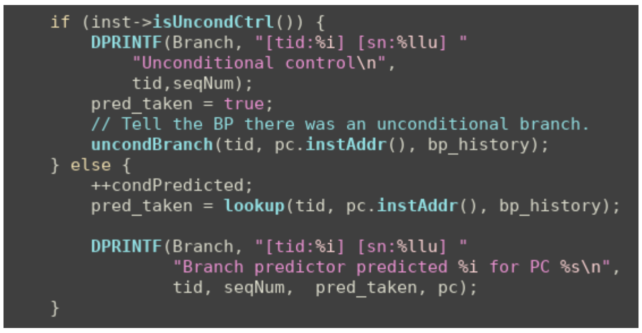
6.2.1 Mapping gem5 stats to PMU events

In order to keep the data as realistic as possible so as to potentially be able to implement the results on real hardware, only data entries which could be obtained through the PMUs were kept. Not all data that the PMUs are capable of recording is available in the gem5 stats. For example, while there is a detailed overview of what amount of data was sent over the memory bus, how many packets were used, and which parts sent them, there does not seem to be an equivalent of the PMU event `0x19`: "Bus access". Some stats also require slightly greater care to extract, e.g. the number of architecturally executed/committed instructions (PMU event `0x08`) is recorded as [...]commit.committedInsts" for CPUs in the big cluster but simply as "[...].committedInsts" for CPUs in the LITTLE cluster. Since the stats blocks in the files are often 1000s of lines, these subtleties are easy to miss when skimming through the stats files. Fortunately, most of the PMU events have a clear, direct equivalent in the stats file. The complete map between stats and PMU events is:

gem5 stat	PMU event	description
branchPred.BTBLookups	0x12	Predictable branch speculatively executed
committedInsts	0x08	Instruction architecturally executed
branchPred.condIncorrect	0x10	Mispredicted or not predicted branch speculatively executed
numCycles	0x11	Cycle [count]
icache.overall_accesses::total	0x14	L1 instruction cache access
dcache.overall_accesses::total	0x04	L1 data cache access
dcache.writebacks::total	0x15	L1 data cache Write-Back
l2.overall_accesses::total	0x16	L2 data cache access
l2.writebacks::total	0x18	L2 data cache Write-Back

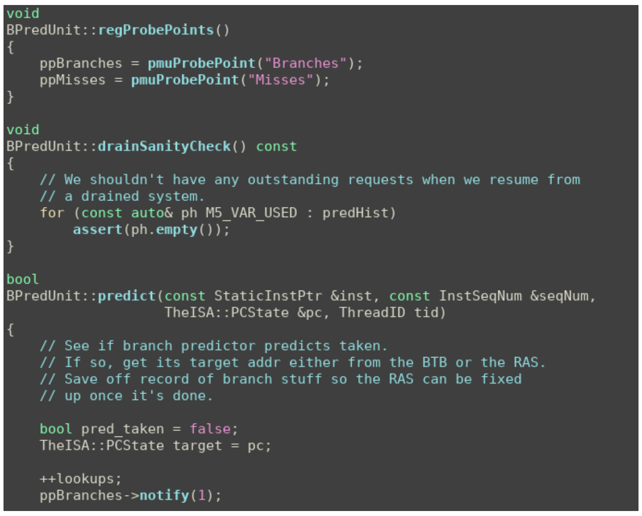
Table 6.2: Mapping between gem5 stats and PMU events (in order of occurrence in the stats files)

The branch prediction stats were somewhat challenging to derive. Initially, I believed that the "branchPred.condPredicted" corresponded to the PMU event for speculatively executing a branch. However, looking through the source code, I discovered that the stat is always incremented *before* predicting if the branch is taken or not (Figure 6.1a). Additionally, I discovered that the PMU registered to the gem5 Simulator seems to get incremented each time the branch predictor is used instead of each time a "[p]redictable branch [is] speculatively executed" (Figure 6.1b). The BTBLookups stat is the only viable option, as it is incremented each time a conditional branch (not a return address) is predicted as taken (Figure 6.1c).



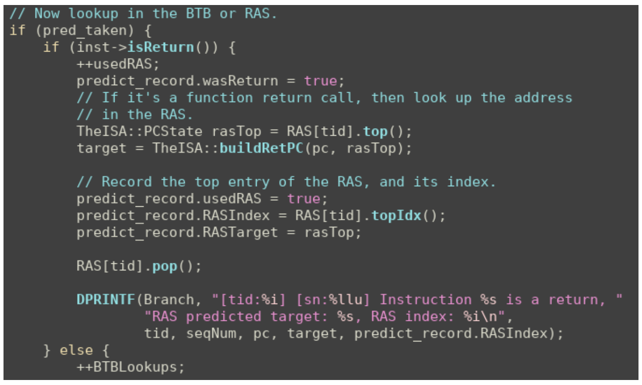
```
    if (inst->isUncondCtrl()) {
        DPRINTF(Branch, "[tid:%i] [sn:%llu] "
            "Unconditional control\n",
            tid,seqNum);
        pred_taken = true;
        // Tell the BP there was an unconditional branch.
        uncondBranch(tid, pc.instAddr(), bp_history);
    } else {
        ++condPredicted;
        pred_taken = lookup(tid, pc.instAddr(), bp_history);

        DPRINTF(Branch, "[tid:%i] [sn:%llu] "
                "Branch predictor predicted %i for PC %s\n",
                tid, seqNum,  pred_taken, pc);
    }
```

(a) `condPredicted` increments before the prediction



```
void
BPredUnit::regProbePoints()
{
    ppBranches = pmuProbePoint("Branches");
    ppMisses = pmuProbePoint("Misses");
}

void
BPredUnit::drainSanityCheck() const
{
    // We shouldn't have any outstanding requests when we resume from
    // a drained system.
    for (const auto& ph M5_VAR_USED : predHist)
        assert(ph.empty());
}

bool
BPredUnit::predict(const StaticInstPtr &inst, const InstSeqNum &seqNum,
                   TheISA::PCState &pc, ThreadID tid)
{
    // See if branch predictor predicts taken.
    // If so, get its target addr either from the BTB or the RAS.
    // Save off record of branch stuff so the RAS can be fixed
    // up once it's done.

    bool pred_taken = false;
    TheISA::PCState target = pc;

    ++lookups;
    ppBranches->notify(1);
```

(b) `lookups`, which is wired up to the gem5 PMU, seems to get incremented on each branch predictor access

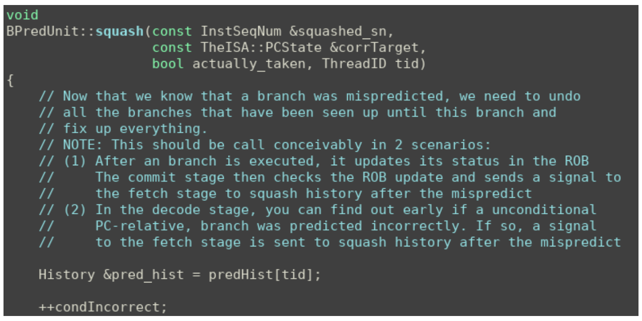


```
// Now lookup in the BTB or RAS.
if (pred_taken) {
    if (inst->isReturn()) {
        ++usedRAS;
        predict_record.wasReturn = true;
        // If it's a function return call, then look up the address
        // in the RAS.
        TheISA::PCState rasTop = RAS[tid].top();
        target = TheISA::buildRetPC(pc, rasTop);

        // Record the top entry of the RAS, and its index.
        predict_record.usedRAS = true;
        predict_record.RASIndex = RAS[tid].topIdx();
        predict_record.RASTarget = rasTop;

        RAS[tid].pop();

        DPRINTF(Branch, "[tid:%i] [sn:%llu] Instruction %s is a return, "
                "RAS predicted target: %s, RAS index: %i\n",
                tid, seqNum, pc, target, predict_record.RASIndex);
    } else {
        ++BTBLookups;
```

(c) `BTBLookups` is incremented if the branch was a conditional branch *and* it was predicted as taken

Figure 6.1: The source C++ code of various branch predictor stats

For branch mispredictions, the condIncorrect fortunately seems to be correct and (according to the code comments) only get incremented when the branch was discovered to be mispredicted.



```
void
BPredUnit::squash(const InstSeqNum &squashed_sn,
                  const TheISA::PCState &corrTarget,
                  bool actually_taken, ThreadID tid)
{
    // Now that we know that a branch was mispredicted, we need to undo
    // all the branches that have been seen up until this branch and
    // fix up everything.
    // NOTE: This should be call conceivably in 2 scenarios:
    // (1) After an branch is executed, it updates its status in the ROB
    //     The commit stage then checks the ROB update and sends a signal to
    //     the fetch stage to squash history after the mispredict
    // (2) In the decode stage, you can find out early if a unconditional
    //     PC-relative, branch was predicted incorrectly. If so, a signal
    //     to the fetch stage is sent to squash history after the mispredict

    History &pred_hist = predHist[tid];

    ++condIncorrect;
```

Figure 6.2: The C++ code confirms that `condIncorrect` stat monitors branch mispredictions

6.2.2 The `data-aggregate.py` script

Having derived which PMU events could be extracted from the gem5 stats, I wrote a script which would scrape each of the many stat files, extract, and aggregate the stats into a single file. The script takes the top-level directory which contains the benchmark output directories as structured through the experimental setup. Since the path to each stats file contains information as to what benchmark, which setup, and how many threads were used, this information is collected when going down the directory structures. For each stat block in the stats file, a line in a CSV-file is then created. Each line contains the information stored in path/directory names, a time increment in the simulation, all the PMU stats, and the power stats. It is possible to specify the name of the output file through the `--output` flag.

Running the script on the 120GB of collected data produced a CSV file of 101MB. This is still a lot of text, but it is also much more manageable than the 120GB of full-detail data.

6.3 Data Processing

6.3.1 Pre-processing

The extracted data needed to be pre-processed before it could be used. For the comparisons between PMU measurements to be fair, the measurements are divided by the number of cycles. Each time slice of 1ms can contain a different number of cycles simulated, due to varying number of stalls or less activity on the core(s), so by dividing the PMU measurements by the number of cycles simulated that time slice, the comparisons become fairer than comparing the time slices directly. The static and dynamic power measurements were added up to be one '`total_power`' measurement.

By creating a multi-index dataframe ordered by benchmark name, big.LITTLE configuration, number of threads, time slice, cluster, and CPU, the data can then be added up over time, resulting in one entry per benchmark, configuration, number of threads, cluster, and CPU which can be easily compared to other instances and/or other benchmarks in order to determine which setup(s) were best in terms of power savings. To be able to compare the total power of the different runs, the power was normalised using Min-Max Feature scaling. This fits the measurements into a range between 0 and 1 inclusive, based on the minimum and maximum values in the dataset, which then makes the power measurements easier to plot and compare since all the records will at most have a power value of 1, whilst also maintaining the relative size differences. The scaling was done using the `minmax_scale` function from Scikit-Learn's [34] `preprocessing` module.

6.3.2 Predicting and Plotting

Since it was too late to try to implement a new scheduler, we instead decided to look at whether the recorded data could show that a more power-efficient schedule existed. The idea was as follows:

1. Select a benchmark that will be arriving as a new program.
2. Get the PMU data by 'running' it on a stock big.LITTLE configuration. This should reveal some details about the program and can simply be looked up since we have all the runs already.
3. Based on the PMU data, predict what benchmark is the most similar by looking at the data which does not contain any information about the benchmark run, for some definition of 'similar'.
4. Find the optimal big.LITTLE configuration for that benchmark, for some definition of 'optimal'.
5. 'Run' the actual benchmark on the configuration and measure/look up the actual results.
6. Based on the data containing all entries, find the true optimal configuration for the benchmark, along with its results.
7. Compare the predicted and actual results, along with a baseline.

To keep the predictions and comparisons simple, only data points from configurations where the number of threads of the benchmark was equal to the number of cores available were kept. The rationale was that this optimised the usage of the configuration and so allowed us to see the fastest the benchmark could perform on that setup, allowing us to see if and how the benchmark scaled with the number of big and LITTLE cores available.

We decided to use Nearest-Neighbours with Euclidean Distance to determine similarity. This was done using the default settings for the `KNearestNeighborsClassifier` in Scikit-Learn's [34] `neighbors` module. The stock configuration was set to be a 2 big 2 LITTLE configuration because it was a configuration which seemed to be somewhere in the middle, hopefully allowing for balanced predictions in either direction. For the same reasons, this configuration was also used as the baseline.

The predictions were done, leaving each benchmark out in turn and retraining the module each time. Since the model returned multiple predictions, one for each core in the stock configuration, these were treated as votes and the benchmark with the most votes was selected for comparison. Based on which benchmark was predicted to be most similar, the optimal configuration for that benchmark was then found by minimising all the number of cycles and amounts of power consumed by the setup for the most most similar benchmark.

To compare across different configurations, which have a different number of cores to compare, the power was accumulated and only the highest number of cycles per cluster was kept. This was done since power consumption is additive, and the cores and clusters were run concurrently, meaning the greatest number of cycles will indicate the last point at which the final thread of the benchmark finished. It was possible to get more than one configuration back if they were equally optimal, however no such situation occurred.

Taking the most similar benchmark and its optimal configuration, the data from the corresponding run with the number of threads equal to the total number of cores was retrieved, as was the data from the actual optimum and the baseline data.

For each benchmark, a plot showing the number of cycles and amount of power for the big and LITTLE clusters was created. These plots allowed me to easily spot and compare the performance of the various benchmarks and configurations. Additionally, in order to get a different look at how the predicted optimum compared to the actual one, these were plotted against each other.

6.4 Analysis

6.4.1 Plots and observations

The behaviour of the Nearest-Neighbours model for the different benchmarks is largely identical. It seems to almost always result in the optimal configuration being 4b4L, apart from when predicting for the `barnes` benchmark, where the 1b1L configuration is predicted to be the most optimal.

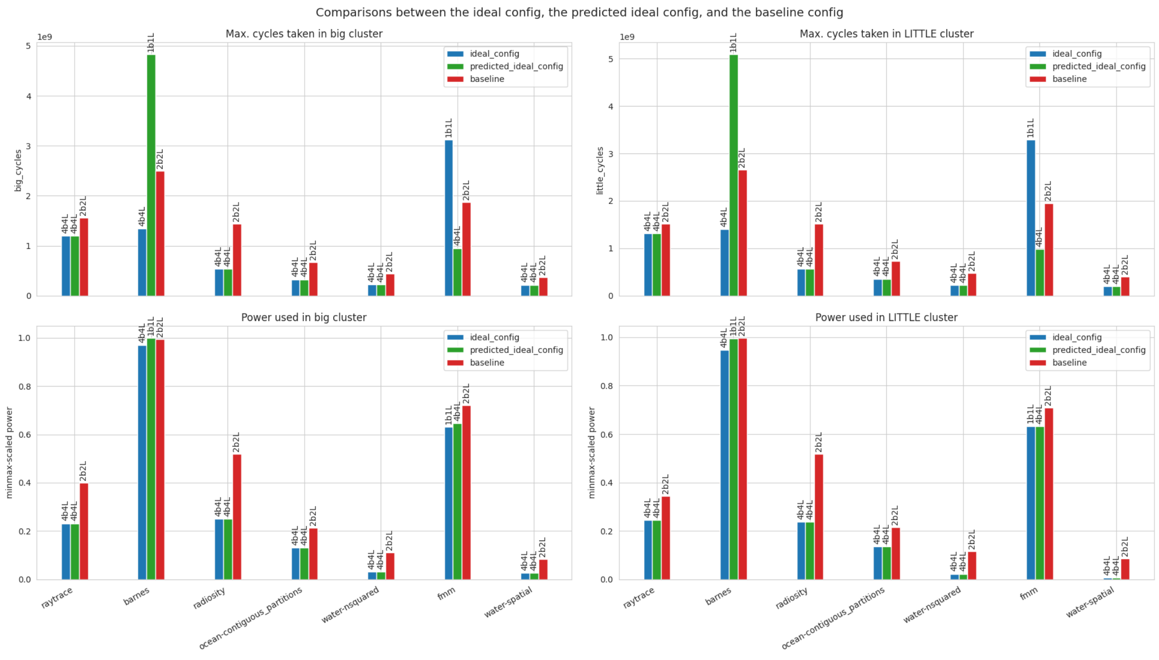


Figure 6.3: Per-cluster comparisons using 2b2L as the baseline

Looking at the plots (note the y-axis for cycles is in Giga, i.e. `1e9`, cycles), it becomes apparent that the trade-offs are there, and that the model has been mostly successful in picking the optimal configuration. For the `barnes` benchmark, the power consumption of the predicted optimal configuration is not far off the true optimum. However, the largest number of cycles is. The `fmm` benchmark reveals an interesting caveat of the optimum finding algorithm: the performance of the true optimum is worse than the predicted, but the power is not. In fact, when looking at the system as a whole, the total power consumption of the 1b1L setup is less than the 4b4L setup. What has likely happened is that when minimising the number of cycles and the power used, the 1b1L configuration was the first to be tried. Subsequently, although the 4b4L configuration has better performance, it does not *also* have better power performance and so it was not selected. It is possible this could be fixed by making the optimum finding algorithm consider the scale of the trade-offs, and not just the values.

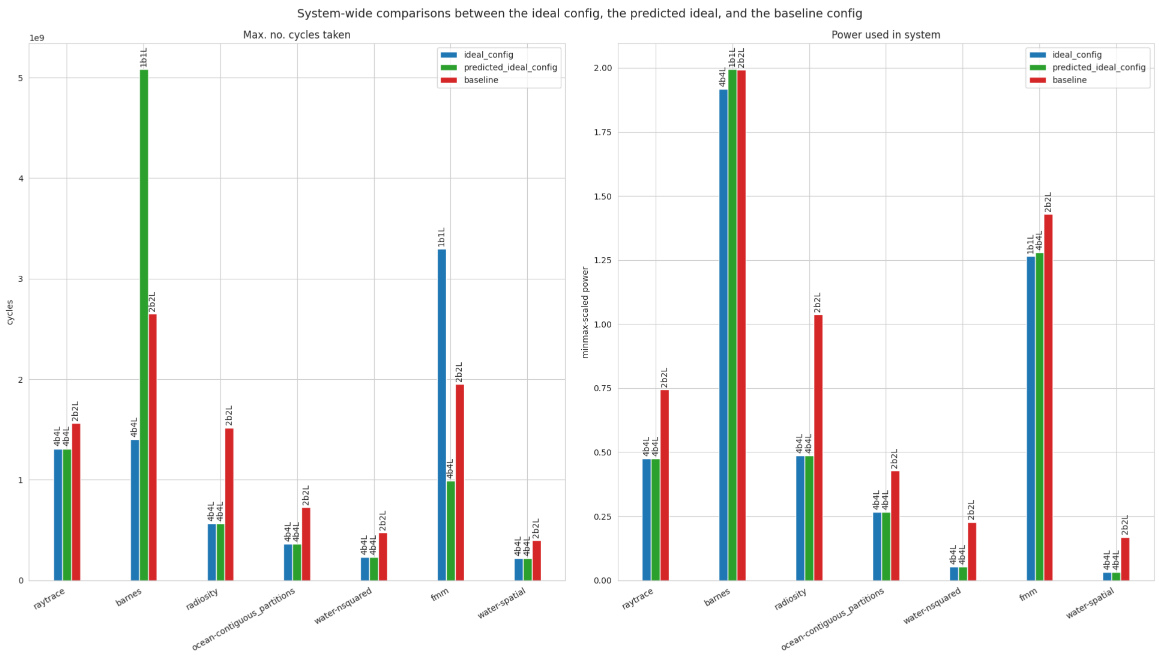


Figure 6.4: System-wide comparisons using 2b2L as the baseline

Interestingly, the tactic of always going with 4b4L does not seem bad when looking at the overall performance. Each time a different configuration manages to do better on one cluster, it seems to come at a cost on the other cluster. This is likely due to 4b4L requiring more power in theory, but finishing the benchmarks so much faster that it does not matter.

6.4.2 Evaluating the prediction results

Plotting the true optima against the predicted optima, helps show how close most of predictions are. The majority of the benchmarks lie exactly on the $y = x$ line, indicating that the prediction was entirely correct. The plots also help explain how the prediction and true optima were off for the `barnes` and `fmm` benchmarks respectively. The power measurements are clearly very close to agreeing, but the performance measurements are much further away from the $y = x$ line. This suggests that, as mentioned with the optimum finding algorithm, some balancing to consider the scale of improvements would be beneficial. That being said, the geometric mean, taken across all the benchmarks and plotted as a cyan dot, lying very close to the $y = x$ line is a good indication that the model was good at predicting the right setup overall.

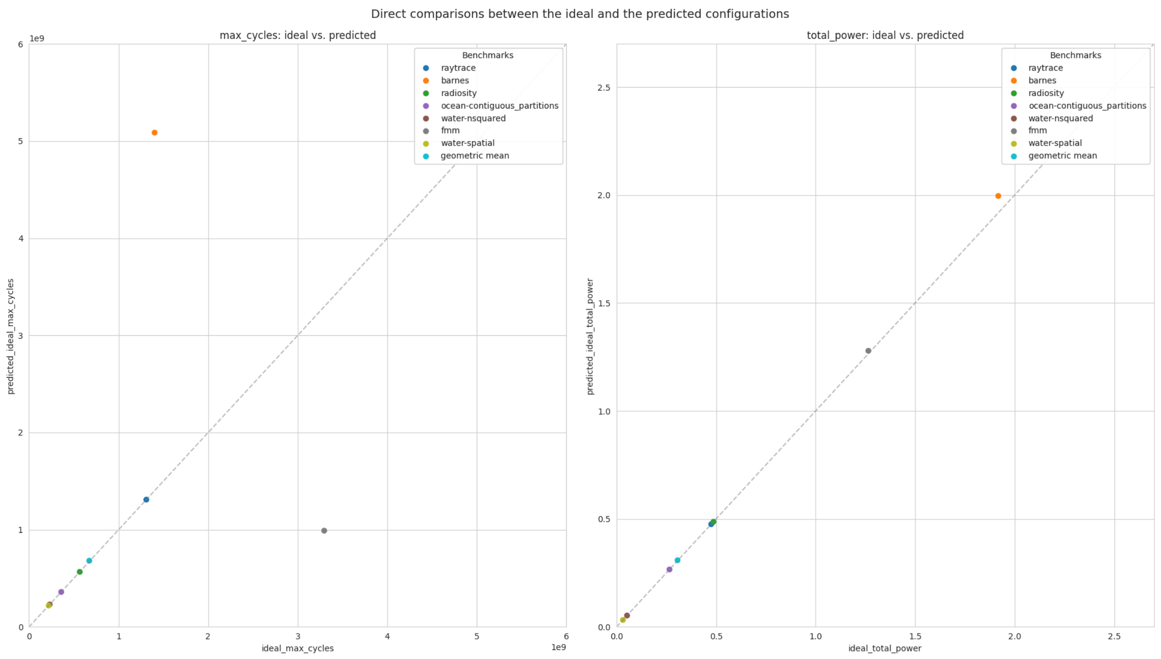
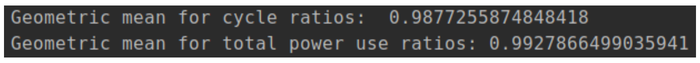


Figure 6.5: Plotting the predicted ideal against the true ideal, for system-wide maximum number of cycles and total power use

Although the sample size is small, computing the improvement ratios of the maximum number of cycles and the total power used by dividing the ideal values by the predicted ones, and then calculating the geometric mean of these ratios seems highly promising. The geometric mean comes out to be $\approx 98.77\%$ for the cycle ratios and $\approx 99.28\%$ for the power ratios. This confirms that there is little to be improved in terms of the power predictions, and also seems to suggest that despite the size of the performance outliers, the space for improvement may be smaller than initially anticipated. However, as mentioned earlier, this is based on a small sample size and more data would be required to say for certain.



```
Geometric mean for cycle ratios:  0.9877255874848418
Geometric mean for total power use ratios: 0.9927866499035941
```

Figure 6.6: The geometric means for improvement ratios seem very promising

6.4.3 Using other stock configurations

In order to improve performance of the model, more data could be supplied. One way of doing this would be to ‘run’ the arriving benchmark on more than one stock configuration and using the results from all the stock configurations as input when predicting the most similar. I tried using pairs of configurations: 2b2L+4b4L, 1b1L+2b2L. The idea was that the change in balance between the configurations would lead to a change in predictions as there were ‘examples’ of which cores affected what. However, when plotted, the results were identical. The resulting system-wide plots can be seen in Appendix C. The complete set of plots can be found in the `results-plots` directory.

Chapter 7

Conclusion

Accurate full system simulators, with support for the latest hardware and its features are hugely complex, both for the developers and the users. The gem5 simulator is definitely a powerful tool, but it unfortunately seems to be in a somewhat fragile and undocumented state concerning DVFS and power models at the moment.

Whilst the predictions done were not based on much data, it was demonstrated that predictions can be done and that even with little data, the predictions can be very good. Changing the configurations was shown to affect the performance and energy consumption of the setups, sometimes drastically. This demonstrates the importance of managing the DVFS of cores and clusters, and that there is potentially a lot to be gained in terms of power savings. Based on the results from the predictions, the true optima, and the baselines, it seems schedulers should be able to use the PMUs to make informed decisions as to what silicon to keep dark in order to balance energy consumption and performance, although some attention will likely have to be paid to ensure the scale of performance and energy savings is balanced along with the values. Additionally, it seems that even a simple model can have a relatively high success rate. This could mean that PMU-aware schedulers could be implemented at a low performance cost, which is highly desirable for schedulers.

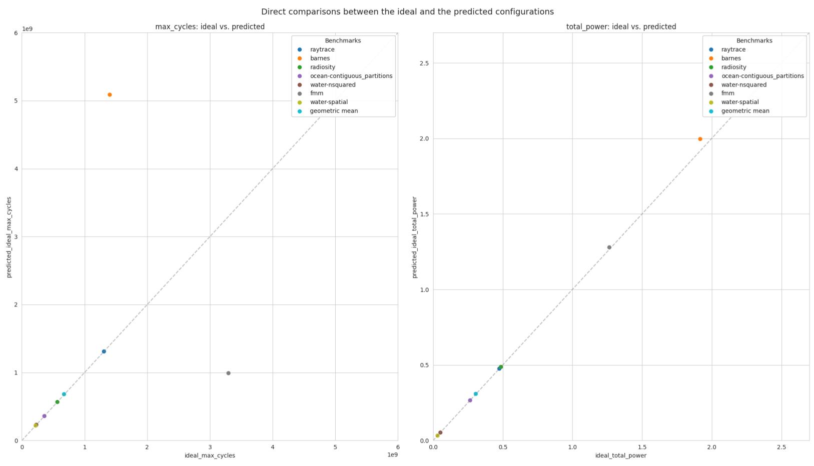


Figure 7.1: Even a simple model based on little data has a relatively good accuracy at predicting the best configurations

Given more time, I would have liked to get more data by also running configurations with only big or LITTLE cores and try to get the predictions to work for varying numbers of threads as well. Other future work could include the implementation on real hardware with multiple programs running simultaneously; improving the predictions by determining the feature importance of the different PMU events; playing with the balance of the optimum finder, to see if better optima could be found if power is prioritised over number of cycles, or vice versa. Although the initial results obtained in this project are based on a small sample size, they suggest that there is narrow space for improvement in terms of achieving the best performance, but also motivates the future work by demonstrating that even a relatively simple model trained on little data seems to produce this narrow optimisation space.

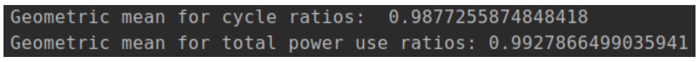

```
Geometric mean for cycle ratios:  0.9877255874848418
Geometric mean for total power use ratios: 0.9927866499035941
```

Figure 7.2: Although based on a small sample size, the model seems to have a narrow optimisation space

Acknowledgements

I would like to thank my supervisor for helping me when I thought this project had crashed and burned, the gem5 developers for being friendly and providing fixes so that I could get the statistics and some power output working, and Teng Yu and Runxin Zhong for their help with understanding the ARM PMUs and providing source code from a recently submitted paper on a similar topic to this project.

Bibliography

[1] "Arm Cortex-A53 MPCore Processor Technical Reference Manual". In: (2016), pp. 12–1. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500j/DDI0500J_cortex_a53_trm.pdf.

[2] "Arm® Cortex®-A73 MPCore Processor Technical Reference Manual". In: (2015), pp. 11–442. URL: http://infocenter.arm.com/help/topic/com.arm.doc.100048_0100_06_en/cortex_a73_trm_100048_0100_06_en.pdf.

[3] Basireddy, K. R. et al. "AdaMD: Adaptive Mapping and DVFS for Energy-efficient Heterogeneous Multi-cores". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019), pp. 1–1. ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2019.2935065. URL: https://ieeexplore.ieee.org/document/8796426/.

[4] Basireddy, K. R. "Runtime Energy Management of Concurrent Applications for Multi-core Platforms". Doctoral. University of Southampton, Apr. 2019. 202 pp. URL: https://eprints.soton.ac.uk/433546/.

[5] Binkert, N. et al. "The gem5 simulator". In: *ACM SIGARCH Computer Architecture News* 39.2 (31st May 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://dl.acm.org/doi/10.1145/2024716.2024718.

[6] Bischoff, S. "gem5: empowering the masses". ARM Research Summit. 2017. URL: http://old.gem5.org/wiki/images/b/bf/Summit2017_powerframework.pdf.

[7] *BusyBox*. URL: https://busybox.net/.

[8] Carlson, T. E., Heirman, W. and Eeckhout, L. "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. Seattle, Washington: ACM Press, 2011, p. 1. ISBN: 978-1-4503-0771-0. DOI: 10.1145/

2063384.2063454. URL: http://dl.acm.org/citation.cfm?doid=2063384.2063454.

[9] Carlson, T. E. et al. “An Evaluation of High-Level Mechanistic Core Models”. In: *ACM Transactions on Architecture and Code Optimization* 11.3 (27th Oct. 2014), pp. 1–25. ISSN: 1544-3566, 1544-3973. DOI: 10.1145/2629677. URL: https://dl.acm.org/doi/10.1145/2629677.

[10] *CFS Scheduler — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[11] Corbató, F. J., Merwin-Daggett, M. and Daley, R. C. “An Experimental Time-Sharing System”. In: *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*. AIEE-IRE ’62 (Spring) (May 1962), p. 10. DOI: 10.1145/1460833.1460871.

[12] *Energy Aware Scheduling — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html#energy-aware-task-placement.

[13] *Error compiling sniper 7.2 - Google Groups*. 14th Jan. 2020. URL: https://groups.google.com/forum/#!topic/snipersim/De4-E9wkeU4.

[14] *Experimenting with DVFS*. 2019. URL: http://old.gem5.org/Running_gem5.html#Experimenting_with_DVFS.

[15] *gem5: About*. URL: http://www.gem5.org/about/.

[16] *gem5: Getting Started with gem5*. URL: http://www.gem5.org/getting_started/.

[17] *gem5: Power and Thermal Model*. URL: https://www.gem5.org/documentation/general_docs/thermal_model.

[18] *gem5: Statistics*. URL: http://www.gem5.org/documentation/general_docs/statistics/.

[19] *Getting Started - Sniper*. 17th Feb. 2019. URL: http://snipersim.org/w/Getting_Started.

[20] *GNU Toolchain — GNU-A Downloads*. ARM Developer. 19th Dec. 2019. URL: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads.

[21] *Google Performance Tools*. Version 2.7. original-date: 2014-01-28T11:15:07Z. URL: https://github.com/gperftools/gperftools.

[22] Guibas, L. J. and Sedgewick, R. "A dichromatic framework for balanced trees". In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978).* 19th Annual Symposium on Foundations of Computer Science (sfcs 1978). Ann Arbor, MI, USA: IEEE, Oct. 1978, pp. 8–21. DOI: 10.1109/SFCS.1978.3. URL: http://ieeexplore.ieee.org/document/4567957/.

[23] Hansen, T. E. and Nikoleris, N. *[GEM5-463] The "clock" stat is inaccessible in power models - Jira.* 22nd Apr. 2020. URL: https://gem5.atlassian.net/browse/GEM5-463 (visited on 15/05/2020).

[24] Hansen, T. E. and Travaglini, G. *[GEM5-319] gem5 ARM example* {{*fs_power.py*}} *script does not work. - Jira.* 7th Feb. 2020. URL: https://gem5.atlassian.net/browse/GEM5-319.

[25] *Index of /dist/current/arm (new).* 2020. URL: http://dist.gem5.org/dist/current/arm/.

[26] *Index of /dist/current/arm (old).* 2017. URL: http://m5sim.org/dist/current/arm/.

[27] Jibaja, I. et al. "Portable performance on asymmetric multicore processors". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization - CGO 2016.* the 2016 International Symposium. Barcelona, Spain: ACM Press, 2016, pp. 24–35. ISBN: 978-1-4503-3778-6. DOI: 10.1145/2854038.2854047. URL: http://dl.acm.org/citation.cfm?doid=2854038.2854047.

[28] Kleinrock, L., Gail, R. and Kleinrock, L. *Computer applications.* Queueing Systems Leonard Kleinrock; Richard Gail ; Vol. 2. OCLC: 299649623. New York, NY: Wiley, 1976. 549 pp. ISBN: 978-0-471-49111-8.

[29] *Linux Kernel — Energy Aware Scheduling (EAS).* ARM Developer. 2019. URL: https://developer.arm.com/tools-and-software/open-source-software/linux-kernel/energy-aware-scheduling.

[30] Liu, C. L. "Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment". In: *J. ACM* 20.1 (Jan. 1973), p. 16. DOI: 10.1145/321738.321743.

[31] Lowe-Power, J. *Learning gem5.* gem5 Tutorial — gem5 Tutorial 0.1 documentation. 2019. URL: http://learning.gem5.org/book/index.html.

[32] Lozi, J.-P. et al. "The Linux scheduler: a decade of wasted cores". In: *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys '16.* the Eleventh European Conference. London, United Kingdom: ACM Press, 2016, pp. 1–16. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901326. URL: http://dl.acm.org/citation.cfm?doid=2901318.2901326.

[33] Macken, P. et al. “A voltage reduction technique for digital systems”. In: *1990 37th IEEE International Conference on Solid-State Circuits*. 1990 37th IEEE International Conference on Solid-State Circuits. San Francisco, CA, USA: IEEE, 1990, pp. 238–239. DOI: 10.1109/ISSCC.1990.110213. URL: http://ieeexplore.ieee.org/document/110213/ (visited on 07/05/2020).

[34] Pedregosa, F. et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* (2011), p. 6. URL: http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html.

[35] Sakalis, C. et al. “Splash-3: A properly synchronized benchmark suite for contemporary research”. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Uppsala, Sweden: IEEE, Apr. 2016, pp. 101–111. ISBN: 978-1-5090-1953-3. DOI: 10.1109/ISPASS.2016.7482078. URL: http://ieeexplore.ieee.org/document/7482078/.

[36] Silberschatz, A., Galvin, P. B. and Gagne, G. *Operating system concepts*. 9. ed., internat. student version. OCLC: 854717328. Hoboken, NJ: Wiley, 2014. 829 pp. ISBN: 978-1-118-09375-7.

[37] *Sniper*. 14th Apr. 2020. URL: http://snipersim.org/w/The_Sniper_Multi-Core_Simulator.

[38] *Sunsetting Python 2*. Python.org. URL: https://www.python.org/doc/sunset-python-2/.

[39] Torvalds, L. *Linux 5.0*. E-mail. 4th Mar. 2019. URL: https://lore.kernel.org/lkml/CAHk-=wjuG6HiGbD7DCGfvDvhr_1WZUR-eYF2qWGbYyn9k6unvg@mail.gmail.com/T/.

[40] Tousi, A. and Zhu, C. “Arm Research Starter Kit: System Modeling using gem5”. In: (2017), p. 32. URL: https://github.com/arm-university/arm-gem5-rsk.

[41] Walker, M. *mattw200/gemstone-applypower*. GitHub. 2018. URL: https://github.com/mattw200/gemstone-applypower.

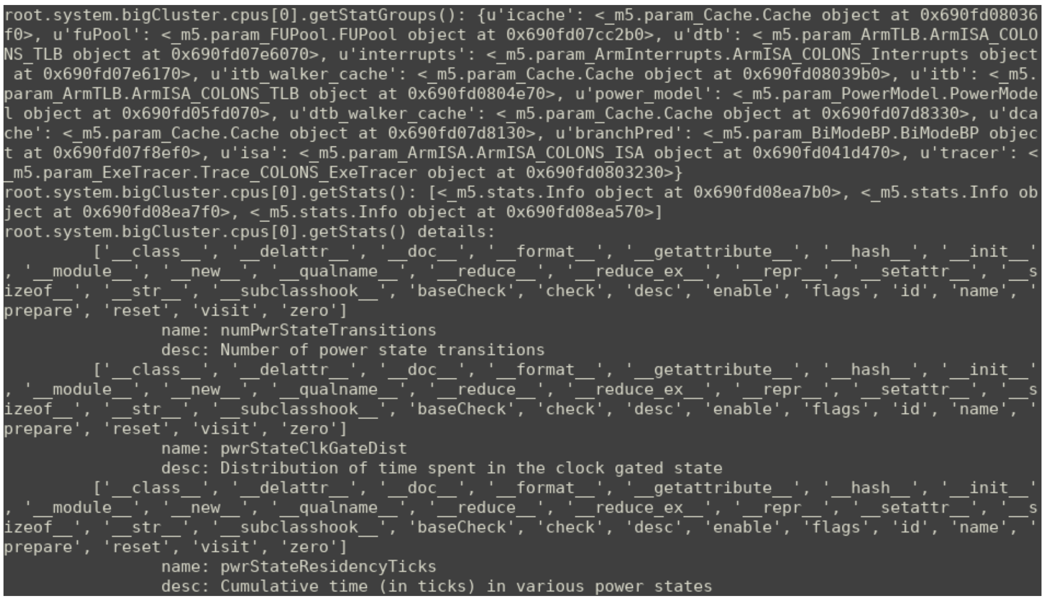
[42] Walker, M. et al. “Hardware-Validated CPU Performance and Energy Modelling”. In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Belfast: IEEE, Apr. 2018, pp. 44–53. ISBN: 978-1-5386-5010-3. DOI: 10.1109/ISPASS.2018.00013. URL: https://ieeexplore.ieee.org/document/8366934/.

[43] Wysocki, R. J. *CPU Performance Scaling — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html.

[44] Yu, T. et al. “COLAB: a collaborative multi-factor scheduler for asymmetric multicore processors”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. CGO ’20: 18th ACM/IEEE International Symposium on Code Generation and Optimization. San Diego CA USA: ACM, 22nd Feb. 2020, pp. 268–279. ISBN: 978-1-4503-7047-9. DOI: 10.1145/3368826.3377915. URL: https://dl.acm.org/doi/10.1145/3368826.3377915.

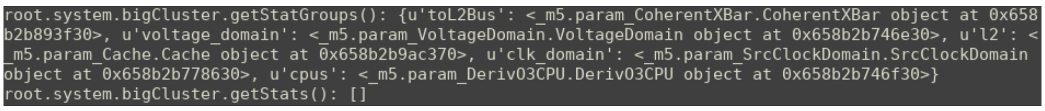
Appendix A

Attempts to find power model stats



```
root.system.bigCluster.cpus[0].getStatGroups(): {u'icache': <_m5.param_Cache.Cache object at 0x690fd08036
f0>, u'fuPool': <_m5.param_FUPool.FUPool object at 0x690fd07cc2b0>, u'dtb': <_m5.param_ArmTLB.ArmISA_COLO
NS_TLB object at 0x690fd07e6070>, u'interrupts': <_m5.param_ArmInterrupts.ArmISA_COLONS_Interrupts object
 at 0x690fd07e6170>, u'itb_walker_cache': <_m5.param_Cache.Cache object at 0x690fd08039b0>, u'itb': <_m5.
param_ArmTLB.ArmISA_COLONS_TLB object at 0x690fd0804e70>, u'power_model': <_m5.param_PowerModel.PowerMode
l object at 0x690fd05fd070>, u'dtb_walker_cache': <_m5.param_Cache.Cache object at 0x690fd07d8330>, u'dca
che': <_m5.param_Cache.Cache object at 0x690fd07d8130>, u'branchPred': <_m5.param_BiModeBP.BiModeBP objec
t at 0x690fd07f8ef0>, u'isa': <_m5.param_ArmISA.ArmISA_COLONS_ISA object at 0x690fd041d470>, u'tracer': <
_m5.param_ExeTracer.Trace_COLONS_ExeTracer object at 0x690fd0803230>}
root.system.bigCluster.cpus[0].getStats(): [<_m5.stats.Info object at 0x690fd08ea7b0>, <_m5.stats.Info ob
ject at 0x690fd08ea7f0>, <_m5.stats.Info object at 0x690fd08ea570>]
root.system.bigCluster.cpus[0].getStats() details:
         ['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__'
, '__module__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__s
izeof__', '__str__', '__subclasshook__', 'baseCheck', 'check', 'desc', 'enable', 'flags', 'id', 'name', '
prepare', 'reset', 'visit', 'zero']
                name: numPwrStateTransitions
                desc: Number of power state transitions
         ['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__'
, '__module__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__s
izeof__', '__str__', '__subclasshook__', 'baseCheck', 'check', 'desc', 'enable', 'flags', 'id', 'name', '
prepare', 'reset', 'visit', 'zero']
                name: pwrStateClkGateDist
                desc: Distribution of time spent in the clock gated state
         ['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__'
, '__module__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__s
izeof__', '__str__', '__subclasshook__', 'baseCheck', 'check', 'desc', 'enable', 'flags', 'id', 'name', '
prepare', 'reset', 'visit', 'zero']
                name: pwrStateResidencyTicks
                desc: Cumulative time (in ticks) in various power states
```

Figure A.1: Stat groups present for a CPU



```
root.system.bigCluster.getStatGroups(): {u'toL2Bus': <_m5.param_CoherentXBar.CoherentXBar object at 0x658
b2b893f30>, u'voltage_domain': <_m5.param_VoltageDomain.VoltageDomain object at 0x658b2b746e30>, u'l2': <
_m5.param_Cache.Cache object at 0x658b2b9ac370>, u'clk_domain': <_m5.param_SrcClockDomain.SrcClockDomain
object at 0x658b2b778630>, u'cpus': <_m5.param_DerivO3CPU.DerivO3CPU object at 0x658b2b746f30>}
root.system.bigCluster.getStats(): []
```

Figure A.2: Stat groups present for the big cluster

Appendix B

Power plots

B.1 The barnes benchmark

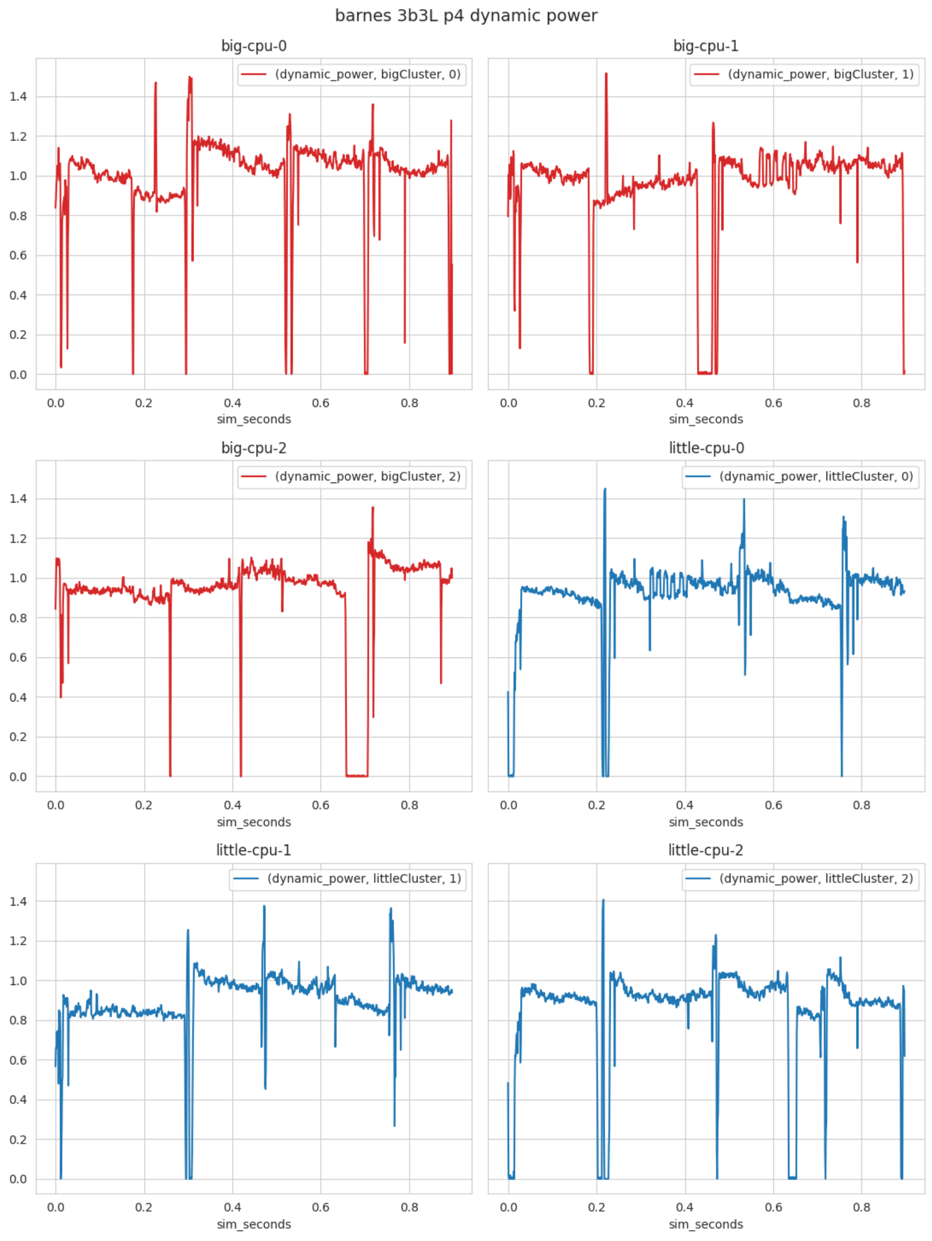


Figure B.1: Dynamic power for 4-threaded `barnes` on 3 big and 3 LITTLE cores (completed)

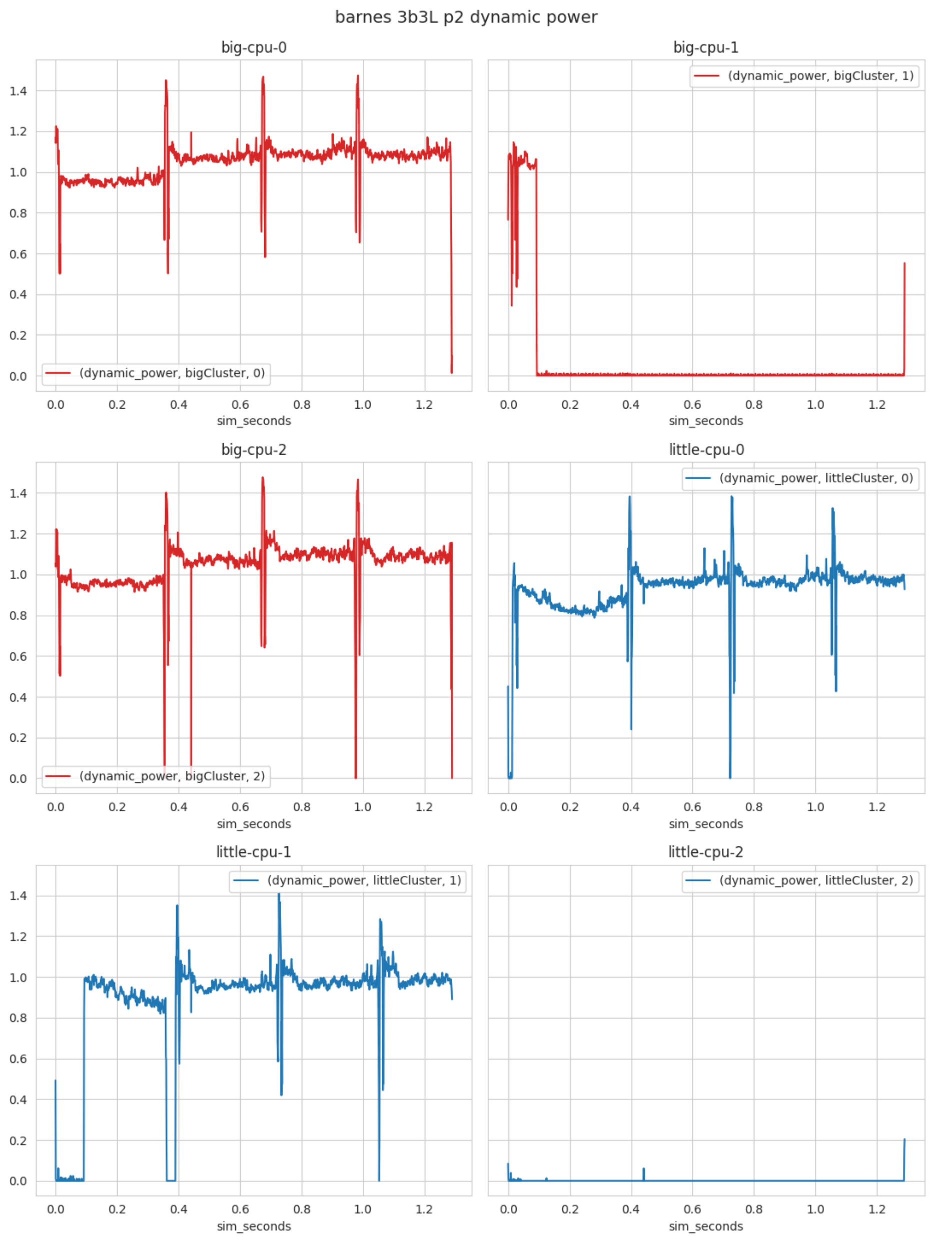


Figure B.2: Dynamic power for 2-threaded `barnes` on 3 big and 3 LITTLE cores (crashed)

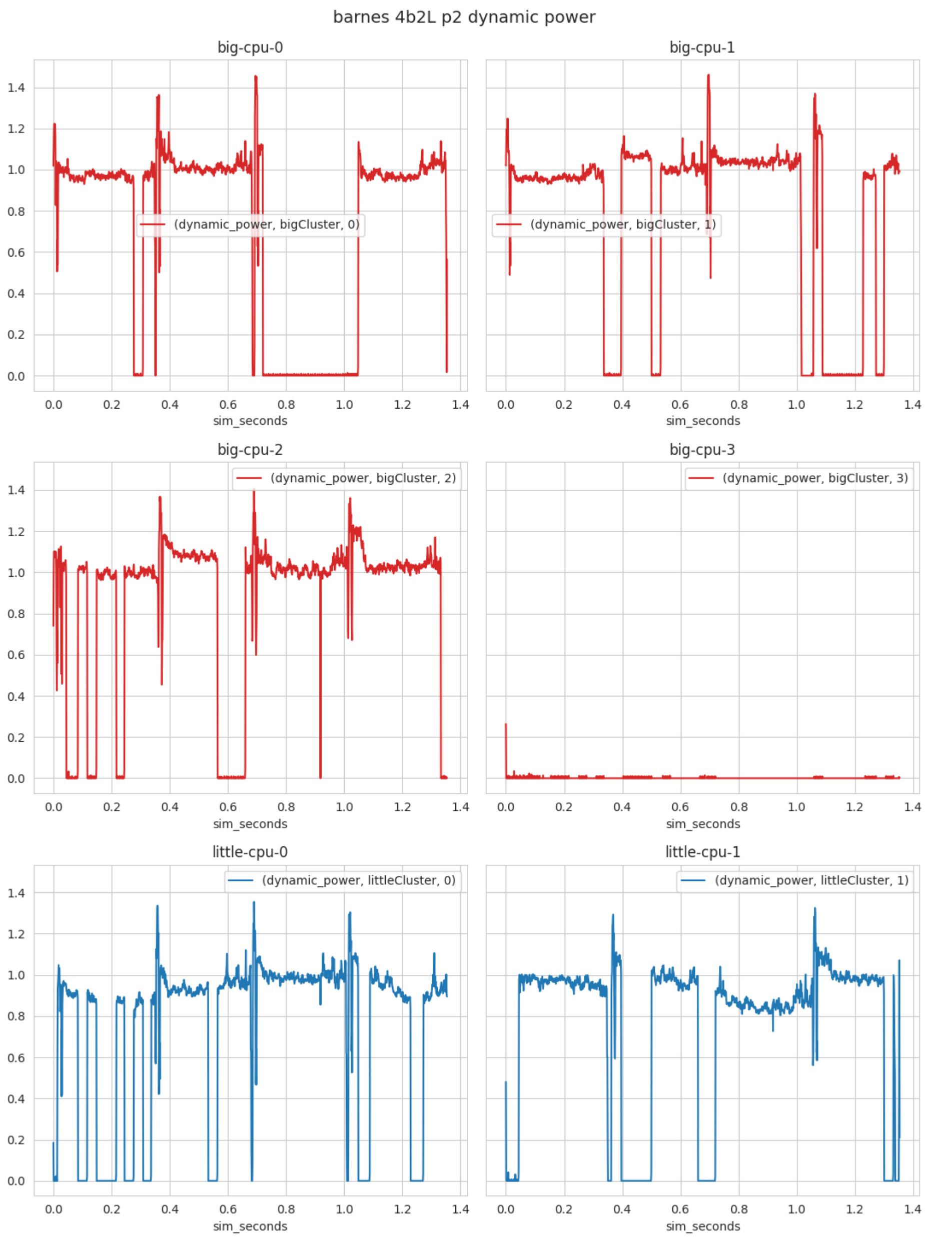


Figure B.3: Dynamic power for 2-threaded `barnes` on 4 big 2 LITTLE cores (completed)

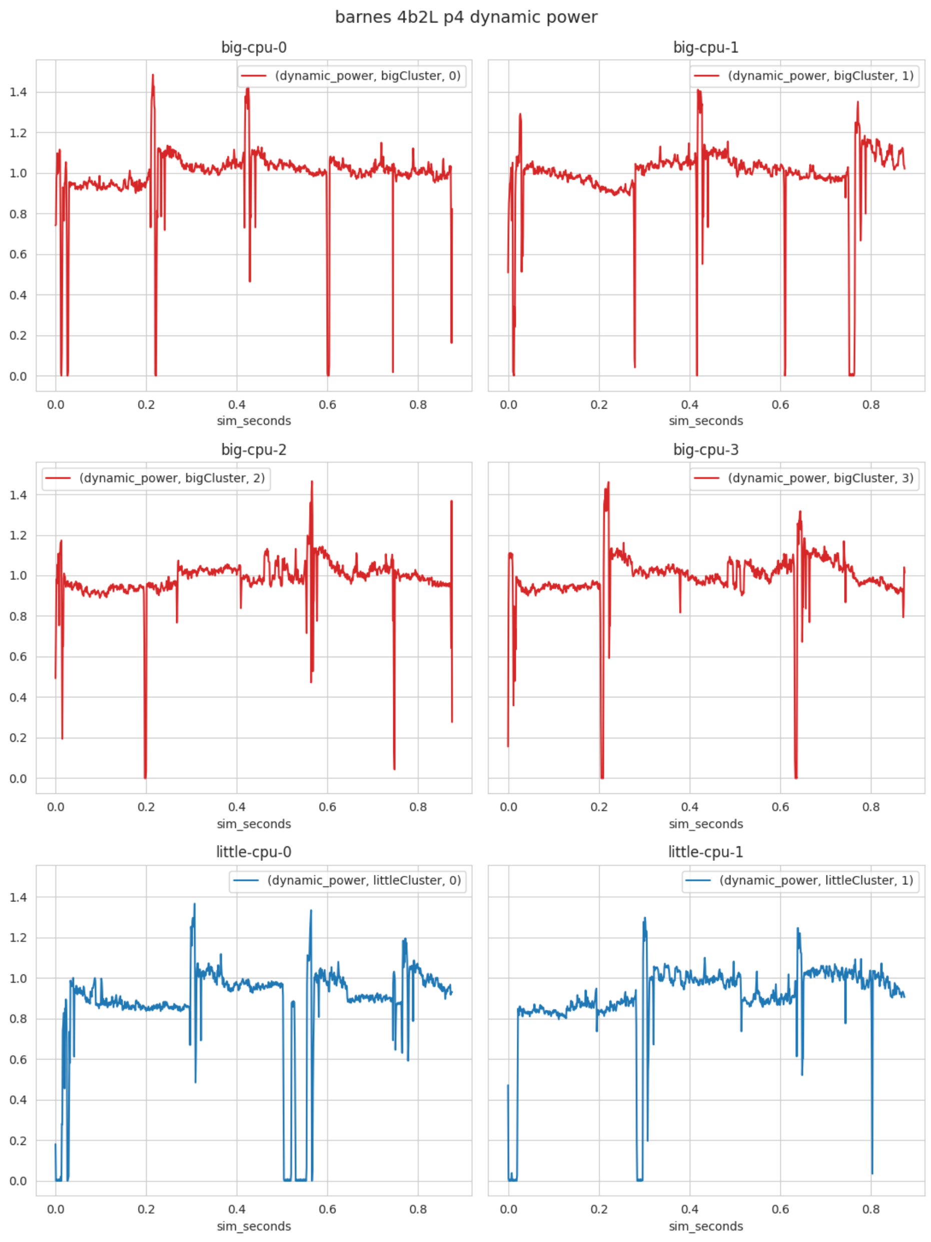


Figure B.4: Dynamic power for 4-threaded `barnes` on 4 big 2 LITTLE cores (crashed)

B.2 The ocean-contiguous_partitions benchmark

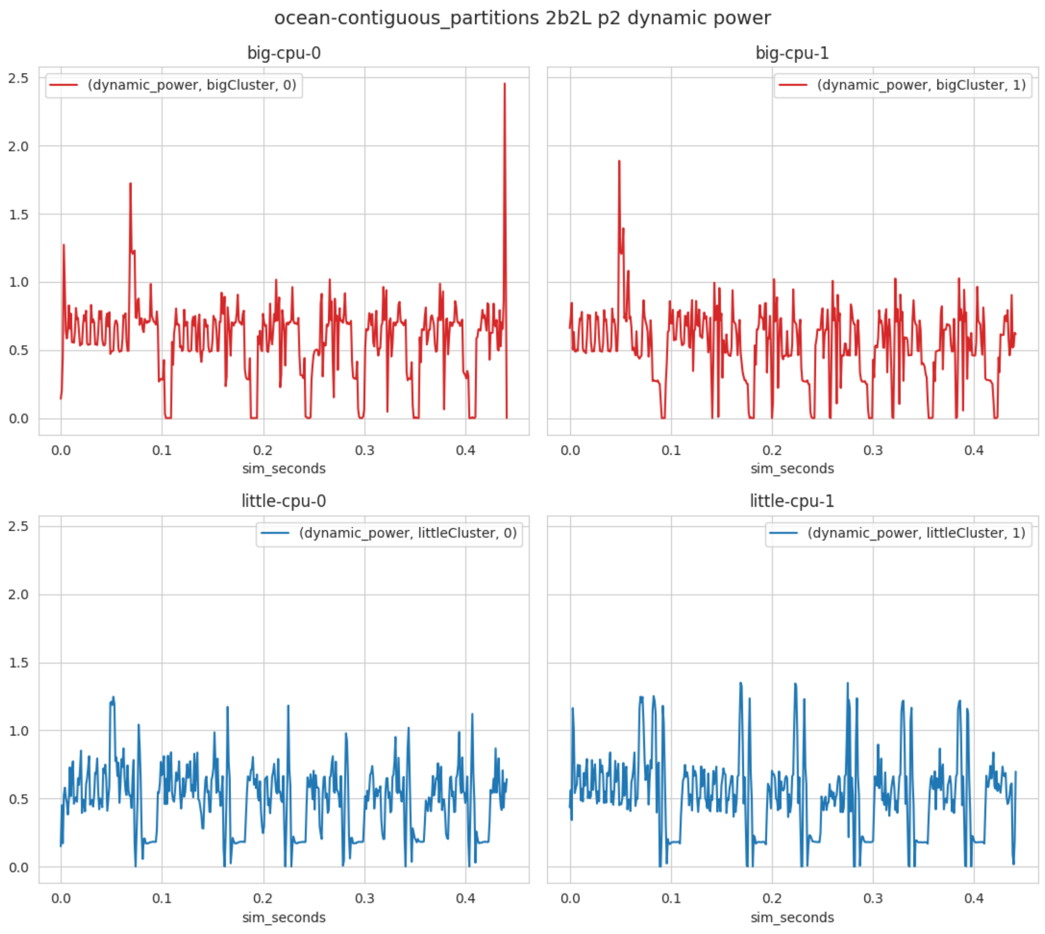


Figure B.5: Dynamic power for 2-threaded `ocean-contiguous_partitions` on 2 big 2 LITTLE cores (completed)

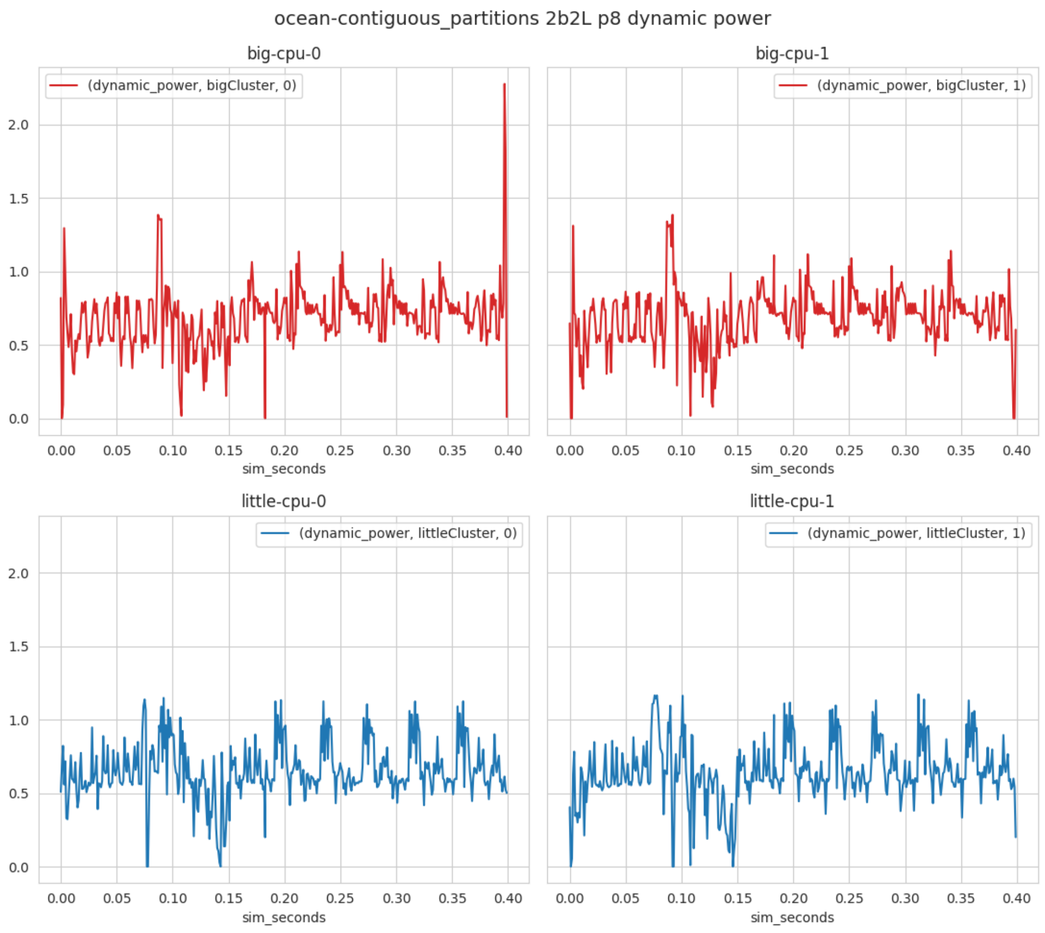


Figure B.6: Dynamic power for 8-threaded `ocean-contiguous_partitions` on 2 big 2 LITTLE cores (crashed)

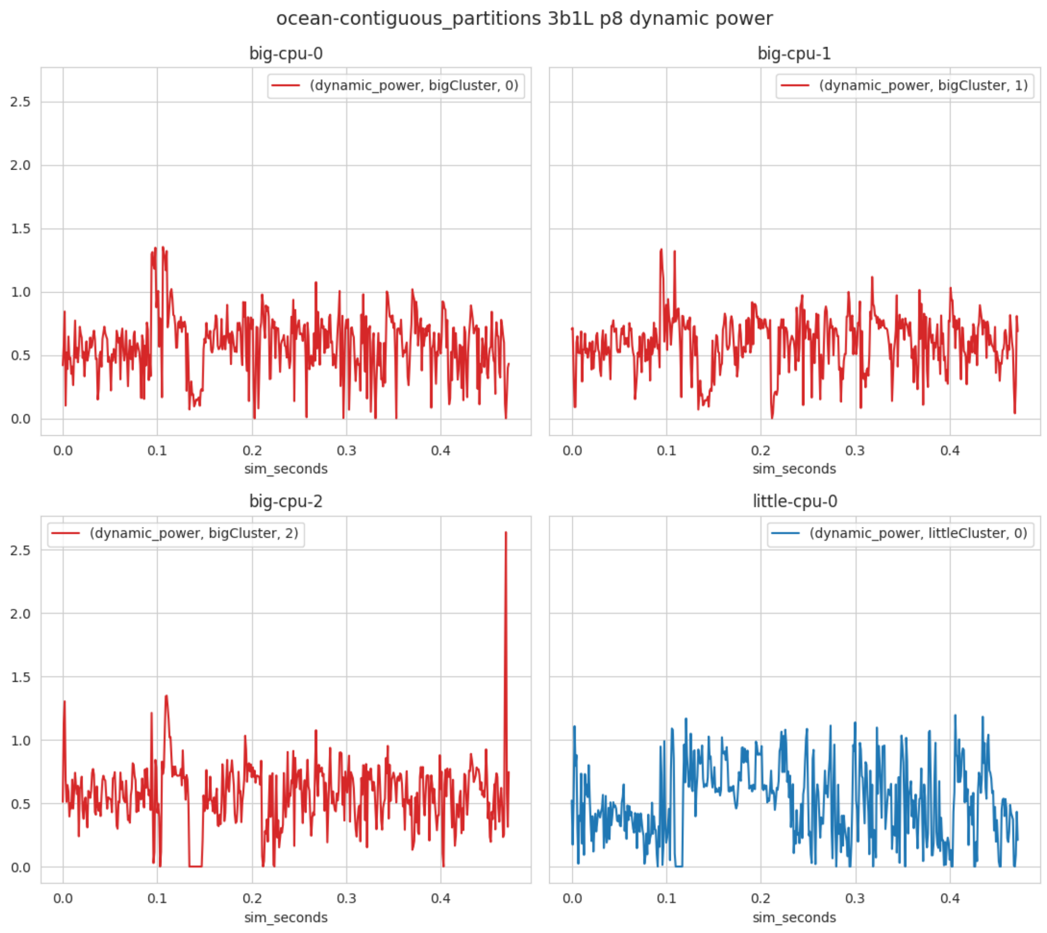


Figure B.7: Dynamic power for 8-threaded `ocean-contiguous_partitions` on 3 big 1 LITTLE cores (completed)

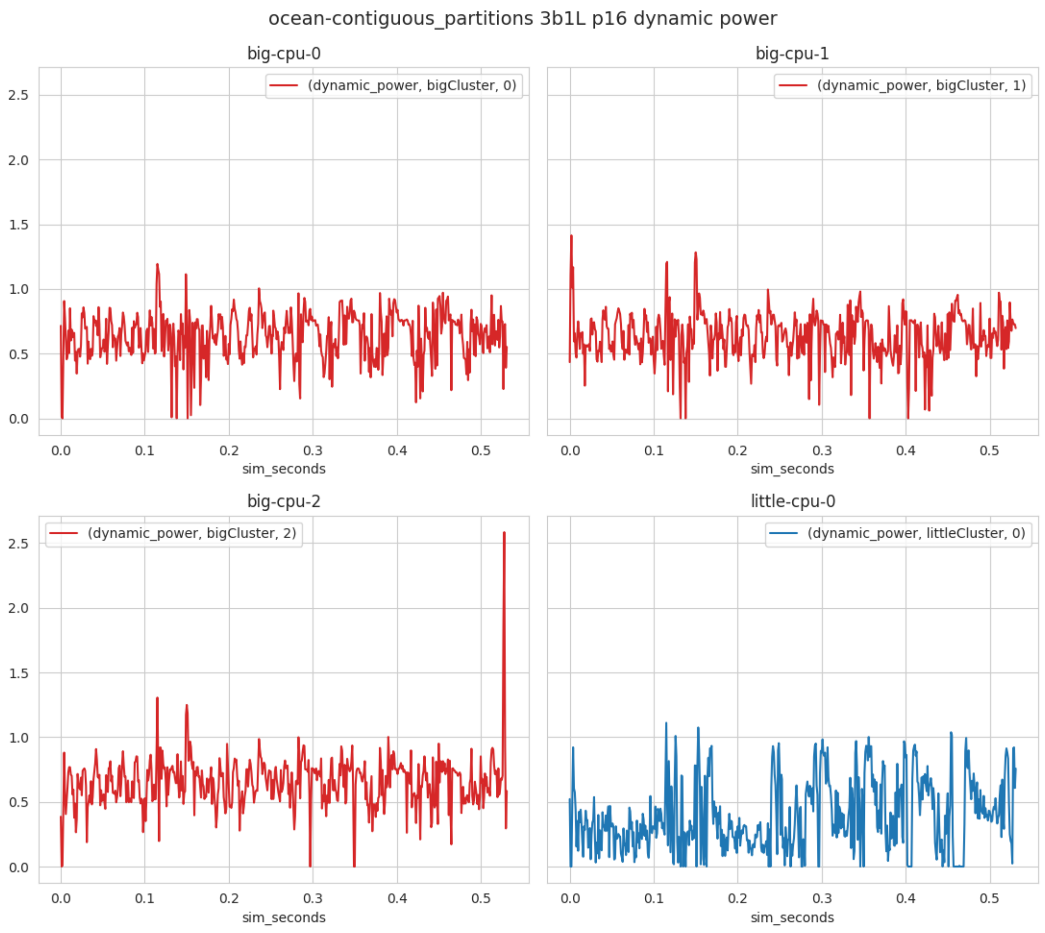


Figure B.8: Dynamic power for 16-threaded `ocean-contiguous_partitions` on 3 big 1 LITTLE cores (crashed)

B.3 The water-spatial benchmark

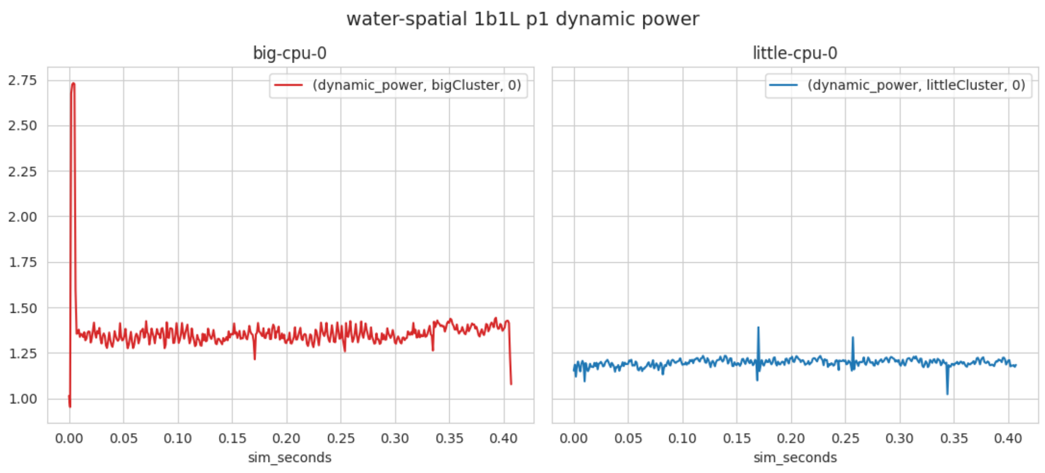


Figure B.9: Dynamic power for single-threaded `water-spatial` on 1 big 1 LITTLE core (completed)

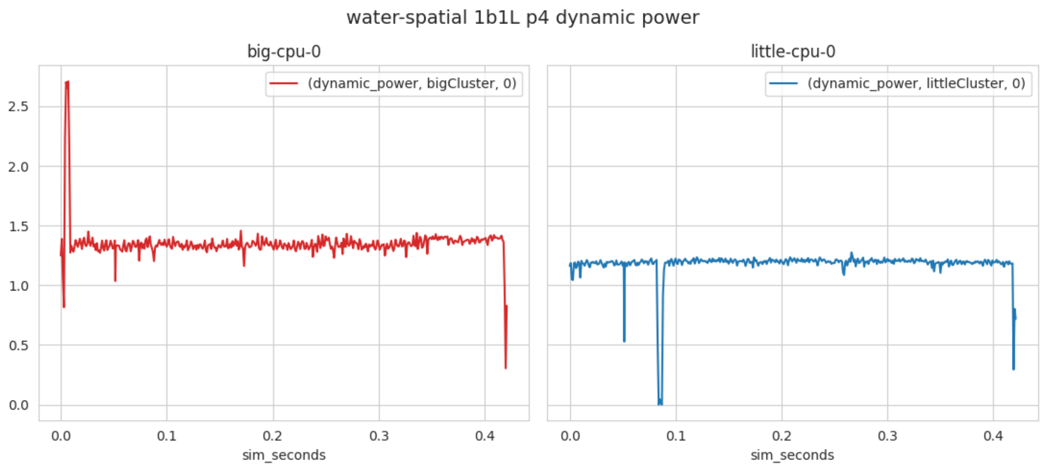


Figure B.10: Dynamic power for 4-threaded `water-spatial` on 1 big 1 1 LITTLE core (crashed)

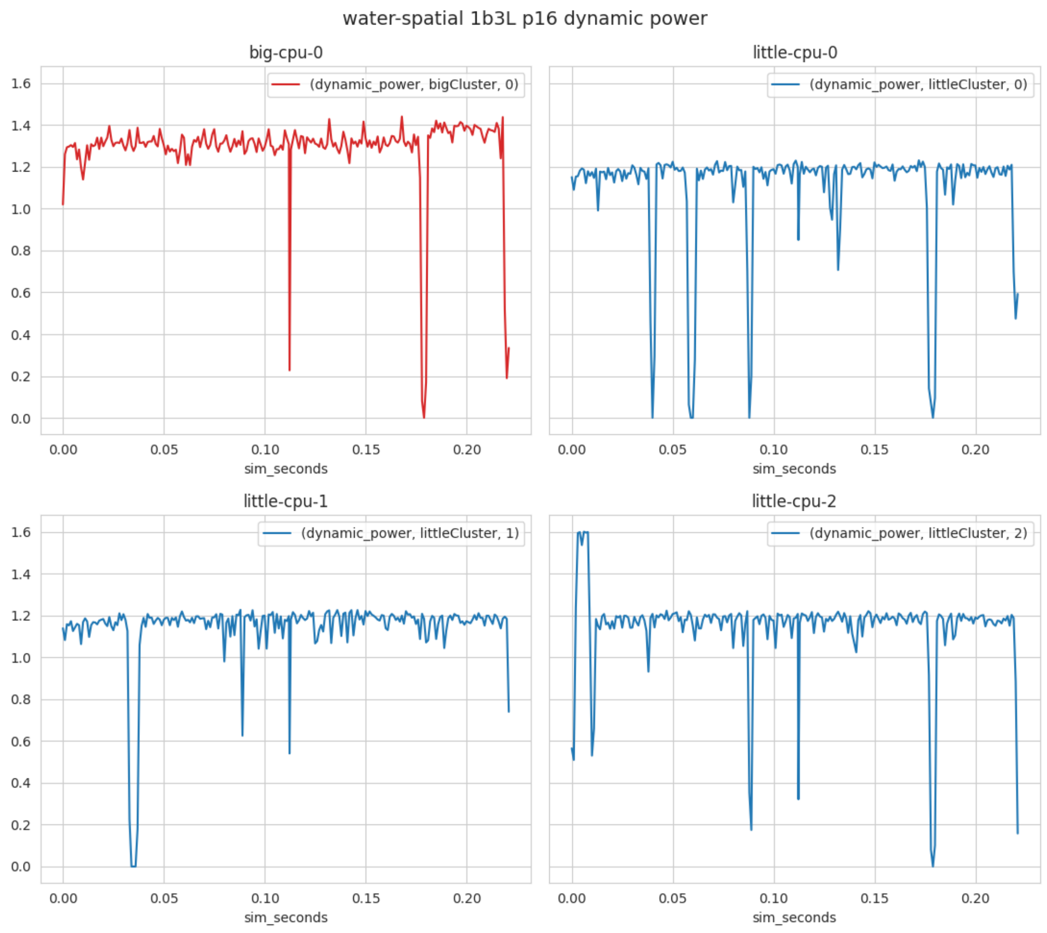


Figure B.11: Dynamic power for 16-threaded `water-spatial` on 1 big 3 LITTLE cores (completed)

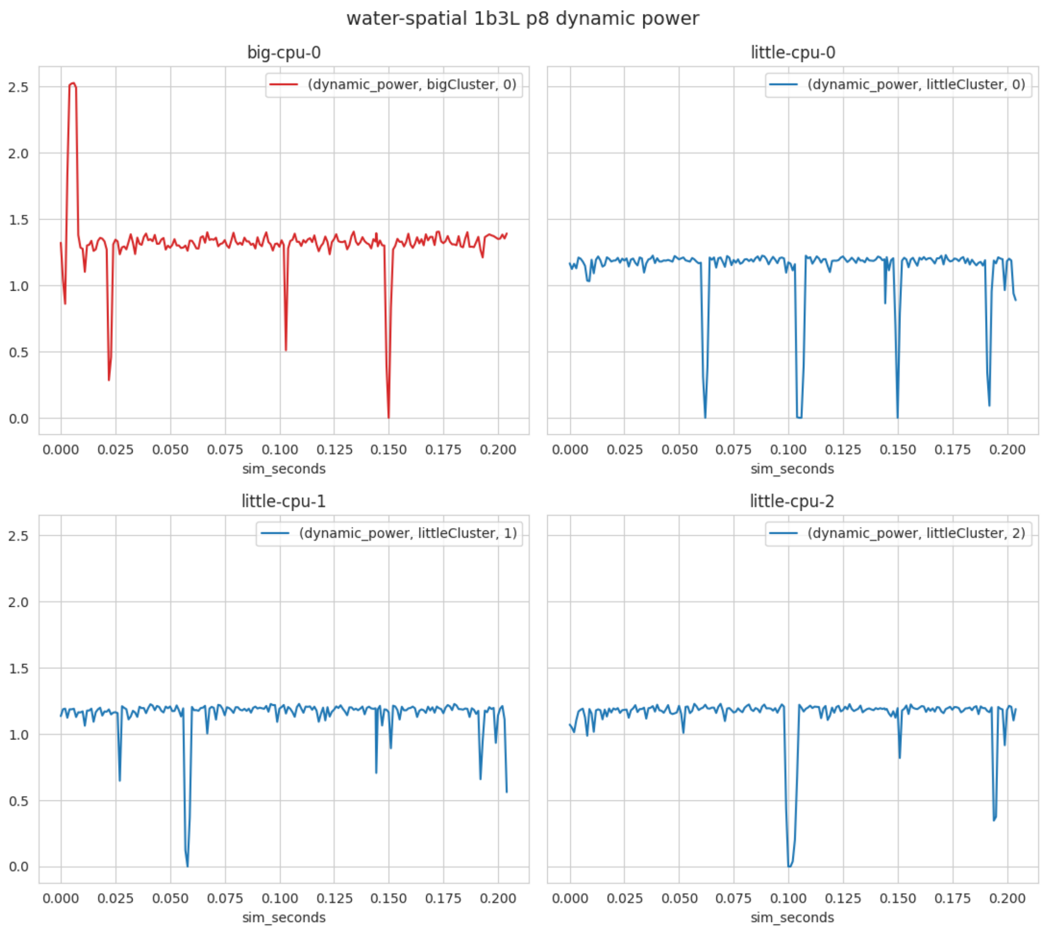


Figure B.12: Dynamic power for 8-threaded `water-spatial` on 1 big 3 LITTLE cores (crashed)

Appendix C

Multi-stock prediction results

C.1 Stock 1b1L+2b2L

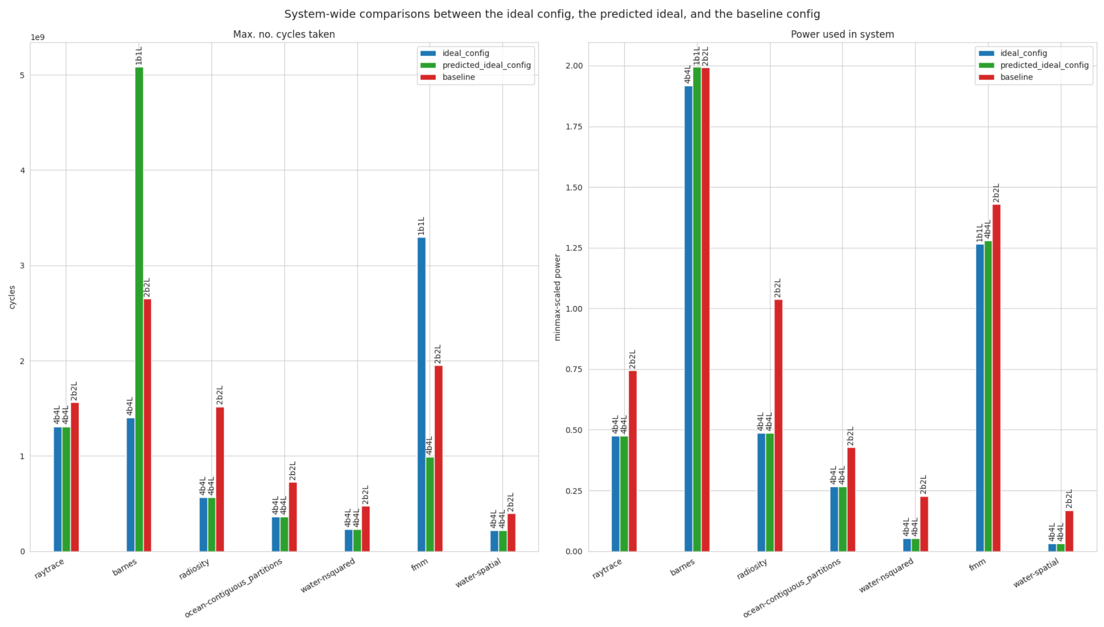


Figure C.1: Performance comparisons using 1b1L+2b2L as stock, with 2b2L as the baseline

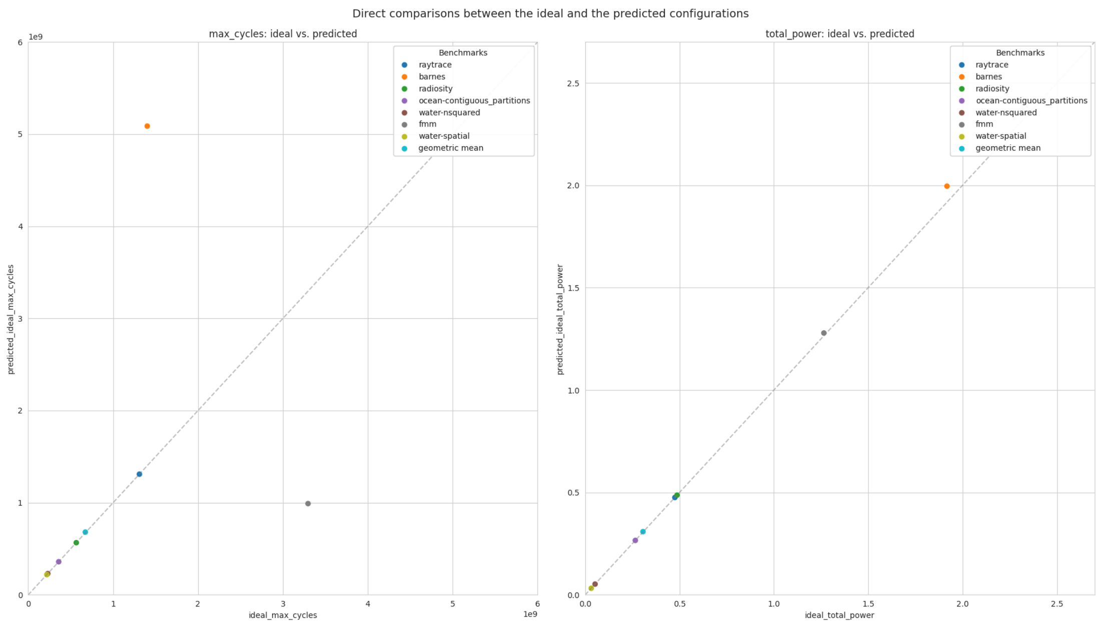


Figure C.2: Direct comparisons using 1b1L+2b2L as stock

C.2 Stock 2b2L+4b4L

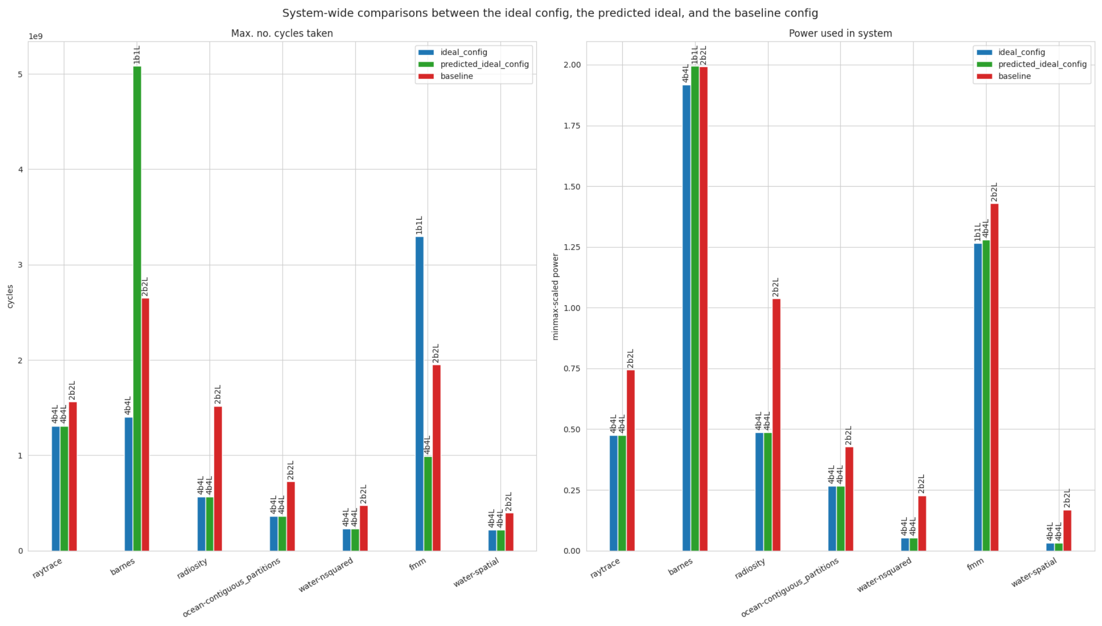


Figure C.3: Performance comparisons using 2b2L+4b4L as stock, with 2b2L as the baseline

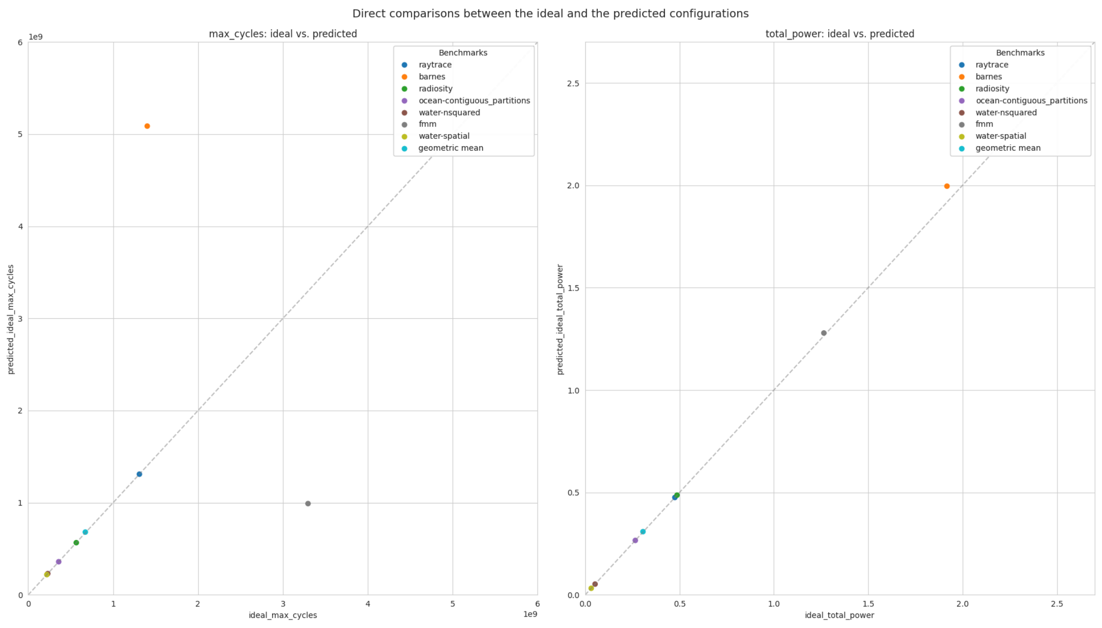


Figure C.4: Direct comparisons using 2b2L+4b4L as stock

Appendix D

Ethics

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

☐ **Staff Project**
☐ **Postgraduate Project**
[✓] **Undergraduate Project**

Title of project

ADDING DVFS TO SCHEDULING FOR ARM big.LITTLE

Name of researcher(s)

MR. THOMAS EKSTRÖM HANSEN

Name of supervisor (for student research)

DR. JOHN THOMSON

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted [✓ **YES**] **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

TEHans

Print Name

THOMAS EKSTRÖM HANSEN

Date

14/02/2020

Signature Lead Researcher or Supervisor

[illegible]

Print Name

JOHN THOMSON

Date

14/2/20

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.