Power Profiling: From Measurements to Simulation Models

Stian Hvatum
Terje Runde

Master of Science in Computer Science

Submission date:	June 2014
Supervisor:	Gunnar Tufte, IDI
Co-supervisor:	Asbjørn Djupdal, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Energy efficiency is currently one of the biggest challenges in modern computer design. High power density limits further performance growth, and energy efficiency affects both the power bill for supercomputers and battery lifetime for embedded devices. A better understanding of energy efficiency during the design stage eases development of better architectures. In this thesis, we investigate energy consumption and architectural properties of an ARM Cortex-A9 processor. Further, this information is used to create a tool for estimating its power consumption through simulation.

Instruction level energy consumption is determined through measurements and experiments on real hardware, which are further mapped to certain architectural events found in the gem5 simulator. The tool utilizes these events together with a simulator trace log and outputs a representation of energy consumption over time.

This method can be applied during the development process at the simulator level, while traditional methods typically involves hardware synthesis. The results show that this tool can estimate energy consumption with margin of error of 5 % on general workloads, and is able to identify power consumption trends throughout a program.

Sammendrag

Energieffektivitet er en av de største utfordringene i moderne datamaskindesign. Videre ytelsesøkning begrenses av høy strømtetthet, i tillegg har energieffektivitet stor betydning i alt fra strømregningen på superdatamaskiner til batterlevetid for små innebygde enheter. Bedre forståelse for energieffektivitet vil gjøre det lettere å utvikle bedre arkitekturer. I denne masteroppgaven ser vi nærmere på arkitekturen og energiforbruket til en ARM Cortex-A9. Vi lager deretter et verktøy for å forutsi dens strømforbruk gjennom simulering.

Gjennom målinger og eksperimenter gjort på ekte maskinvare bestemmes strømforbruk på instruksjonsnivå. Videre blir dette koblet til bestemte hendelser i den samme arkitekturen modelert i gem5-simulatoren. Verktøyet vårt benytter så disse hendelsene, sammen med loggfiler fra simulatoren, til å lage en representasjon av prosessorens strømforbruk over tid.

Vår metode kan benyttes i prosessorutvikling allerede i simulatorfasen, mens tradisjonelle metoder ikke virker før maskinvaren er ferdig syntetisert. Resultatene viser at verktøyet vårt kan estimere strømforbruk innenfor 5 % feilmargin på normale arbeidslaster. Det kan også identifisere positive og negative utviklinger i strømforbruket gjennom kjøringen av et program.

Preface

This report is submitted to the Norwegian University of Science and Technology in fulfillment of the requirements for master thesis. This work has been performed at the Department of Computer and Information Science, NTNU, with Associate Professor Gunnar Tufte as supervisor, and Asbjørn Djupdal as co-supervisor.

Thanks to Gunnar Tufte and Asbjørn Djupdal for all help with the technical work and the report. Also thanks to Kenneth Sivertsvik, Håkon Øye Amundsen and Joakim Andersson for help with proofreading the thesis.

Title:	Power Profiling: From Measurements to Simulation Models
Students:	Terje Runde & Stian Hvatum

Problem description: The SHMAC prototype is an ongoing research project within the Energy Efficient Computing Systems (EECS) strategic research area. SHMAC is planned to run in an FPGA and be an evaluation platform for research on heterogeneous multi-core systems. Due to the Dark silicon effect, future computing systems are expected to be power limited. The goal of the SHMAC project is to propose software and hardware solutions for future power-limited heterogeneous systems.

The micro architecture level is an implementation of the Instruction Set Architecture (ISA). Energy efficiency of an ISA is as such given by the chosen micro architecture. To be able to take the "right" design choice to optimize for energy efficiency, knowledge of energy and power for instruction types, e.g., instructions of type float, nop, copy, are needed.

The goal of this sub-project within the SHMAC platform is to gain knowledge of energy/power consumption of different instruction types to be able to extract information that can be used to improve the micro architecture design of SHMAC-cores. This project will take a twofold approach; 1) Investigate the power/energy consumption of simple benchmark programs on real hardware, i.e. create benchmark programs and evaluate performance by measurements. 2) Investigate the same benchmark programs in simulations as to ensure a good understanding of the relation between measurements and simulated results.

The project will include:

- Devising small benchmark programs, e.g., C or assembly, that isolate specific functions at the micro architecture level.
- Run test on real hardware to collect data.
- Run tests in simulation to relate measurements to simulation results.

An ARM processor is going to be the target ISA for measurements and simulations.

Supervisor:	Gunnar Tufte, IDI
Co-supervisor:	Asbjørn Djupdal, IDI

Contents

Chapter 1

Introduction

The steep increase in single-threaded performance during the last few decades seems to have come to an end. Figure 1.1 shows how Moore's law is continuing [1, 2]; the transistor count on-die is still increasing exponentially. As we have seen the end of Dennard scaling [3, 4], power density increases as more transistors are crammed together. Too much power on a tiny area leads to more heat than conventional cooling solutions can dissipate. Today, computer designers are striving to achieve higher performance without further increase in power density.

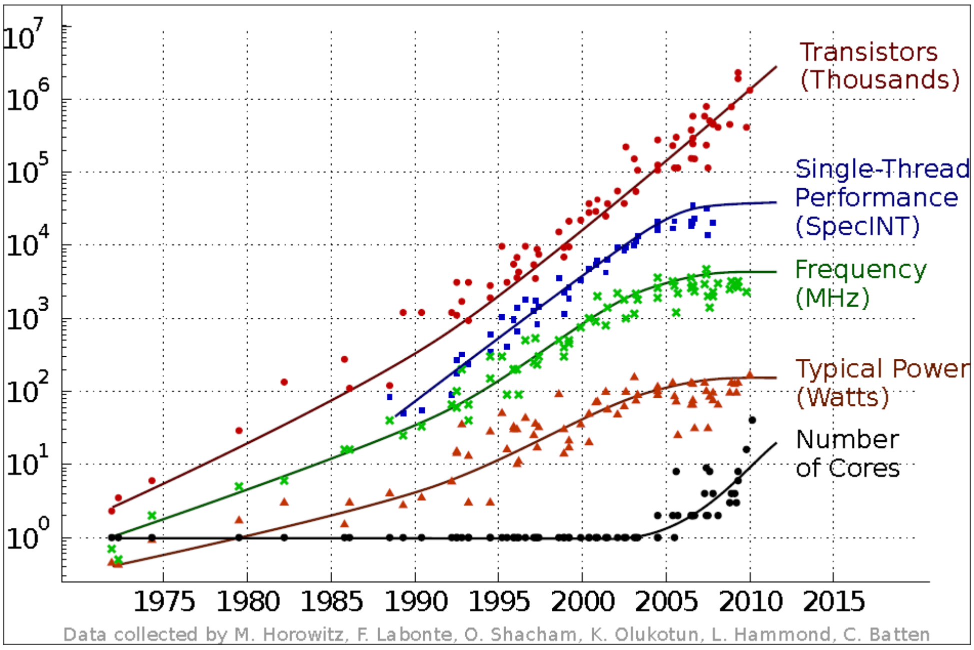


Figure 1.1: Historical trends in CPU performance, from [5].

1.1 Historical Perspective

Computers have emerged in many roles in our society, and the demand for greater computer resources is ever increasing. Throughout the '80s and '90s, the increasing demand for performance was met by increasing the clock frequency. Shortening the critical path and exploiting instruction level parallelism allowed the CPU to run at higher clock speeds to improve throughput [2]. Consequently, processor manufacturers were able to double single-threaded performance approximately every 18th month [1]. The tradeoff, however, was an increased amount of complex logic added to the processor core. Techniques such as pipelining, superscalarity and out-of-order execution all improved performance by leveraging the increased number of transistors [6]. For a long time, new process technologies allowed for smaller and less energy consuming transistors, but as we approached the end of Dennard scaling [3, 4], the amount of gates required to accommodate speedups could not fit on the die due to thermal constraints. Heat generation on-chip became overwhelming; one could no longer add more logic and increase the frequency to gain additional performance.

1.2 Demand for Energy Efficiency

We are now at the beginning of an era where energy efficiency and performance are tightly coupled. When improving performance, one must take care not to exceed the physical limitation of power dissipation. Thus, energy efficiency is key to additional performance gain; performance per Watt must be emphasized.

Heat is not the only motivating factor to keep energy consumption down. Processors targeting laptops, cellphones and other mobile devices have always been energy-constrained due to their use of batteries. Lower energy consumption would allow for longer battery life and/or heavier applications. More recently, mobile processors have become increasingly popular in alternative domains, such as supercomputing. Their low cost and high performance per Watt ratio makes them attractive for massively parallel problems, which is currently done on large and expensive supercomputers. These machines have huge energy budgets and are taken out of service after just a couple of years, being replaced by new machines that offer better performance for less power. Building data centers from low-cost embedded processors is believed to have a huge potential and could change the landscape of supercomputing in the future [7].

Not only data centers benefit from the use of mobile processors. The SHMAC research project at NTNU aims to build a single-ISA heterogeneous computing platform with processing cores specialized for energy efficiency. Using the most efficient processor or hardware accelerator – in terms of both energy and performance – is the key to success for such platforms.

There are several reasons to minimize a processors energy consumption. Batteries would last longer, applications become richer and it will enable processor performance growth to continue. Energy efficiency has become crucial; performance alone is no longer the single most important attribute of processors.

1.3 Optimizing for Energy Efficiency

Given the availability of advanced hardware design tools, it is possible to model and simulate performance of an unimplemented architecture with decent accuracy. However, modeling power consumption is a more elaborate process: current techniques works on a low level and uses circuit-level models to obtain energy metrics. This method makes them accurate, but also heavy and time consuming. Being able to rapidly prototype and visualize how changes in the microarchitecture affects energy performance is an advantage when designing energy efficient hardware. Some solutions already exists [8, 9], but most of them inspect energy consumption at a fine granularity and requires ASIC synthesis of HDL code. During the design process, there is a great need for tools that help developers predict the changes in power consumption when new features are implemented.

The immediate lack of a system that is easy to use and set up motivates the creation of a new high-level tool. We introduce PET; a tool that is able to estimate power usage over time for a given workload on a given architecture. It will use an energy metric profile together with a simulator trace log to calculate energy usage. Using this approach, PET will be able to detect if hardware modifications done to the simulation level model will be beneficial in the realized hardware. PET will also tell if a processor implementation is more energy efficient than another given a specific workload. Thus, it can help building workload optimized tiles for the SHMAC project [10]. Using PET, one can also adjust the energy metric profile and simulate power usage with one component cheaper or more expensive to use in terms of energy. This will enable hardware designers to understand which optimizations are most beneficial and identify possible routes of exploration in their journey of processor energy optimization.

1.4 Assignment Interpretation

Based on the assignment description text, the following main tasks were identified.

Task 1: Quantify the exact cost of executing specific instructions on a modern out-of-order CPU core.

Task 2: Create a software suite that accentuate energy consumption during software execution on various hardware.

Task 1 involves performing energy measurements on hardware components to obtain numbers of a processors energy characteristics. *Task 2* depends on the results from the former and can only be solved after the completion of *Task 1*.

Task 1 was solved as a part of the specialization project (TDT4501) during the fall of 2013, and the final report in its entirety is attached in Appendix A. The review of *Task 2* is the main emphasis of this master's thesis. In order to solve it, a simulation environment must be created and combined with a method for power estimation.

1.5 Report Organization

Chapter 1: Introduction provides a historical perspective to the trends in processor design and motivates the need for energy efficient hardware.

Chapter 2: Background contains supportive material on subjects used throughout this thesis, as well as explanations that justify decisions made later in the report.

Chapter 3: Building a Power Estimation Tool is the main piece of our problem solution. It contains an in-detail review of what PET does and how it is built.

Chapter 4: PET Performance Tuning describes considerations needed when porting the use of PET to support new hardware configurations.

Chapter 5: Experiments and Results presents the tests used to evaluate PET, along with accuracy and performance data.

Chapter 6: Conclusion and Further Work provides the concluding remarks on the work described in this thesis and suggests possible areas of interest for further research.

Chapter 2 Background

This thesis touches subjects such as artificial intelligence, computer architecture and electronics. Some background information on the most important subjects is provided in this chapter.

2.1 Energy Consumption in CPUs

Two sources of energy consumption in a CPU can be distinguished: static and dynamic. Static energy consumption is caused by a small current continuously leaking through the transistors, while dynamic is due to charges being moved towards ground when transistors are toggling [11]. Figure 2.1 shows how current flows through a NOT gate at the transistor level. The green arrow indicates where static leakage occurs and the two red arrows shows where charges escapes when switching. As feature size decreases, a significant part of overall energy consumption is due to static leakage [12]. This means that simply powering the chip without any toggling generates a significant amount of heat [13, 14]. Static power consumption origins from transistor size and layout, while dynamic power consumption depends on the amount of transistor switching. From an architectural point of view, the dynamic power consumption is easier to influence.

2.2 Instruction Level Energy Measurements

High precision instruction level energy models can be derived for pipelined processors by monitoring the instantaneous current drawn by the processor at each clock cycle, as explained in [15]. Modern processors commonly operate at a few GHz, and the Nyquist-Shannon sampling theorem [16] states that the sampling frequency must be at least twice the frequency of the signal being measured. The signal sampled from the processor might change at least once per clock cycle, so obtaining accurate measurements would require use of very expensive instruments.

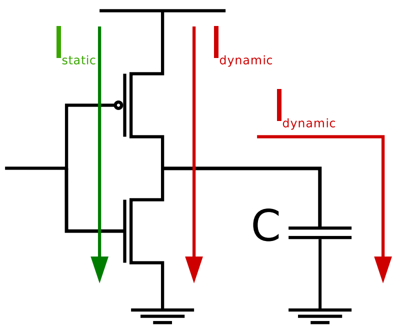


Figure 2.1: Static and dynamic power through a NOT gate.

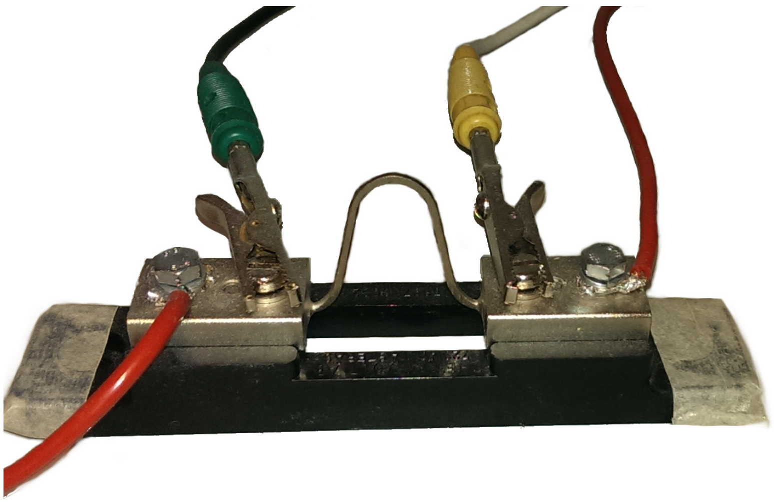


Figure 2.2: The shunt resistor used in our experiments.

In [17], single instructions were measured by exploiting fast-loop-mode [18] and looping over a group of equal instructions. A bench multimeter was used to measure the voltage drop over a shunt resistor as seen in Figure 2.2, set up as illustrated in Figure 2.3. From this, the current flowing from the V_{core} power rail and through the processor core was inferred. The peripherals were isolated and excluded from the measurements. The shunt resistor was chosen such that the voltage drop over it resided in the range $0 - 100$ mV. According to the datasheet [19], the voltage readings would then have 0.003 % margin of error.

The use of a shunt resistor to infer current is equivalent to how ammeters work internally. However, the internal shunt resistor is not scaled for the dynamic range of a specific target. Also, a too large resistor would drop the voltage relative to the impedance in the load, and in the case of a processor this can give unpredictable

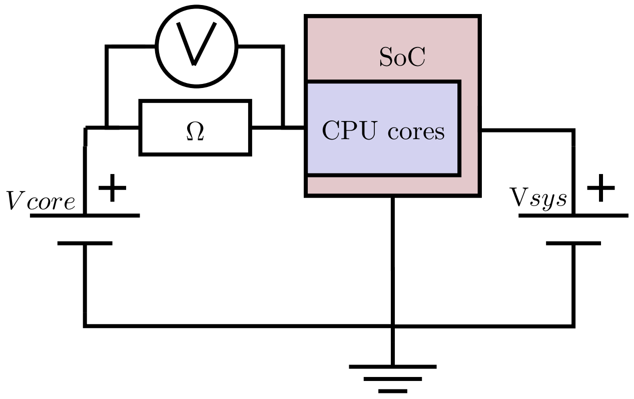


Figure 2.3: Experiment setup for measuring single instruction current drain.

results. Figure 2.4 illustrates two scenarios where voltage drops over a 0.1 Ω shunt resistor and a variable load (e.g., the CPU core). There is a trade-off between accuracy in measurements and voltage variations across the circuit. If the shunt resistor is too small, the voltage drop diminishes and is difficult to measure.

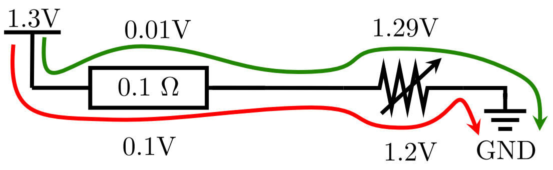
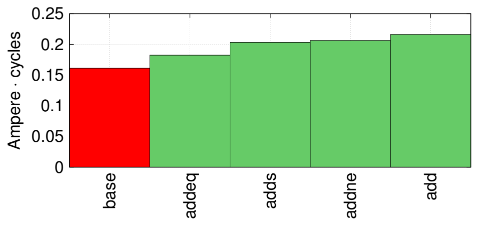


Figure 2.4: The red and green lines represents two snapshots in time with different variable loads. A higher current drain through the circuit changes the ratio of voltage drop between the two loads.

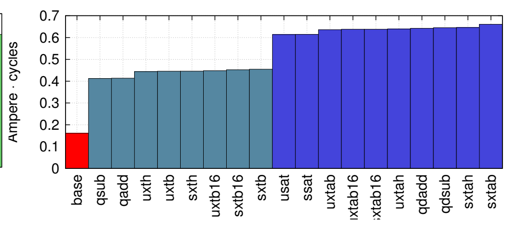
Figure 2.5 presents results from [17], and shows how different instructions use different amounts of energy. This indicates that the architecture impacts how efficient each instruction is. *base* refers to the cheapest instruction in the ISA and roughly corresponds to the static power consumption.

2.3 Hardware Platform

The ODROID-X2 developer platform [20] was used as the reference hardware for all experiments in this thesis. An image of the platform is shown in Figure 2.6. Its core voltage is accessible on an attached daughter board beneath the heat sink, making it easy to conduct power measurements. The board is equipped with a Samsung Exynos 4412 “Prime” [21], a modern SoC (“System-on-Chip”) featuring four ARM Cortex-A9 [18] CPU cores, a Mali-400 GPU and 2 GB of on-chip DRAM. The Cortex-A9 is



(a) Conditional execution (`eq` is false).



(b) Non-multiply multi-cycle instructions.

Figure 2.5: Figures from [17] showing the results of measuring the current drain through the CPU core while running isolated instructions in a loop. The values are measured current drain multiplied by the average number of cycles used.

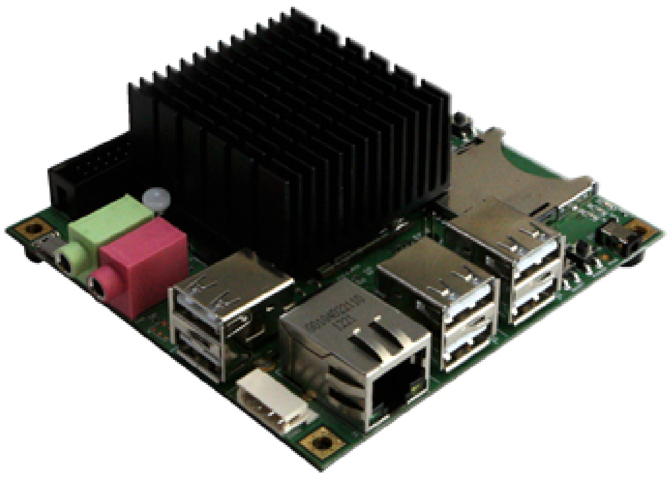


Figure 2.6: Image of the Hardkernel ODROID-X2 from the Hardkernel ODROID-X2 product page [20].

one of ARM's mid-to-high-range application processors. It features an out-of-order dual-issue speculative RISC core, and it seems to be designed with emphasis on energy efficiency. This processor is primarily found in low-powered embedded devices with a modest demand for performance, such as smartphones and tablets.

Figure 2.7 shows an overview of the ARM Cortex-A9 architectural structure. It features an out-of-order multi-issue module after the decode stage. This module can do speculative issue and schedules two arithmetic operations per cycle. It also features a multiplexed lane for address operations and floating point operations (the *NEON* FPU). In this experiment, a 4-core variant of the processor was used, but 3 of the cores were disabled to ease both the measurement and the simulation process. Table 2.1 enumerates the most important properties of the SoC.

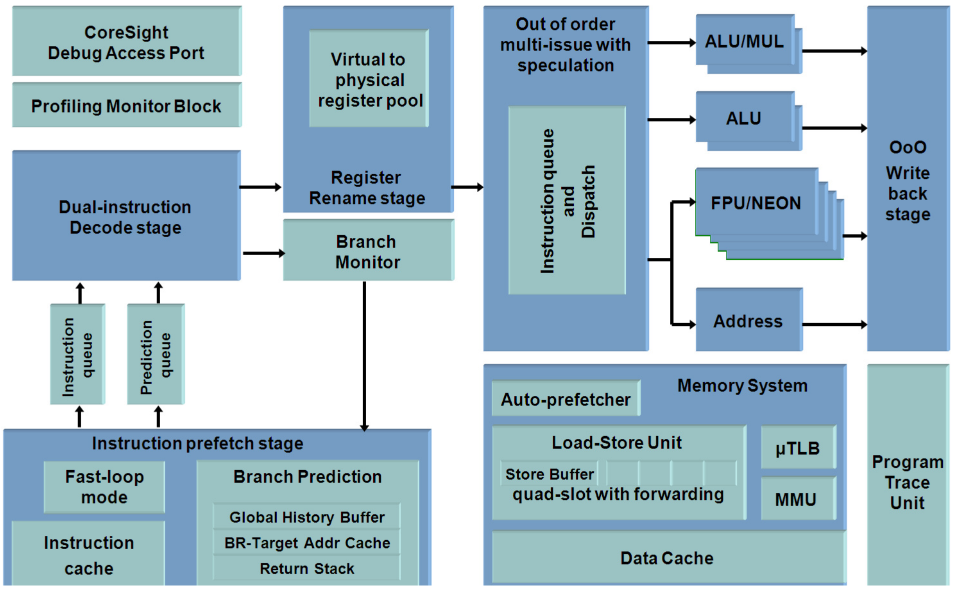


Figure 2.7: An overview of the Cortex-A9 Pipeline, figure taken from the ARM Cortex-A9 White paper [18].

Manufacturer	Hardkernel
Platform	ODROID-X2
SoC	Samsung Exynos 4412 “Prime”
CPU Core	ARM Cortex-A9 (r3p0)
Number of Cores	4
Clock Frequency	1.7 GHz
Core Voltage	1.3 V
L1 Cache	Dual 32 KB
L2 Cache	1 MB
Main Memory	2 GB LP-DDR2 880 MHz

Table 2.1: Hardware specifications ODROID-X2.

2.4 Hardware Simulators

As computer architecture development meets more challenging demands, a versatile set of software tools have been developed to help the designers. In this collection of tools lies a set of computer architecture simulators meant to evaluate processors at the architectural level. They provide the ability to model hardware at a higher abstraction layer than what is expressed by the underlying circuit.

2.4.1 A Brief Comparison of Hardware Simulators

To support our power estimation tool, a simulator front-end must provide a good picture of events that occur in a hardware implementation of the architecture. The out-of-order property significantly increases the level of complexity which the simulator must handle. The following simulators were considered:

Sniper

Sniper is a high-speed, multicore, multi-threaded and cycle-accurate computer architecture simulator [22, 23, 24]. It already integrates with McPAT [25, 26] and it is open source. Sniper only works with x86 targets, and is therefore not applicable for simulation of ARM-based architectures.

SimpleScalar

SimpleScalar is a popular commercial architectural simulator that comes with a free academic license providing full source code [27, 28, 29]. SimpleScalar supports the ARM instruction set among many others, and looks like a decent simulator for advanced out-of-order core simulation. SimpleScalar is also the simulator used by the Wattch-project [30]. However, the SimpleScalar project seems to be in a state of abandonment. The source code for SimpleScalar v3 is still available and received patches in 2003.

QEMU

QEMU is a generic open source machine emulator which enables near real-time performance on architectures like ARM, even on x86 host machines [31, 32, 33]. However, QEMU is a machine emulator rather than an architectural simulator. Despite its great performance of running ARM-binaries, it will not produce CPU and memory event trace logs, and is not suitable for this project.

gem5

gem5 is a merger between the M5 simulator [34] and the GEMS simulator project [35]. gem5 includes ARM-support with out-of-order execution and provides cycle-accurate trace logs which are appropriate for this project [36]. Its core is written in C++ and has a highly modular interface that allows users to specify simulator targets through Python scripts. Many of the maintainers are employees of ARM Corp., and the activity on the mailing lists suggests high project activity [37].

Provided this comparison of simulators, and given that NTNU has previous experience with gem5, gem5 was the natural winner and our choice of an architectural simulator.

2.4.2 The gem5 Simulator

The gem5 project [38] merges the best features of M5 [34] and GEMS [35] and includes a wide range of CPU and memory models [39].

The gem5 simulator comes bundled with different CPU models ranging from in-order models without timing constraints, such as `AtomicSimpleCPU`, to detailed out-of-order cores such as `O3CPU`. During the merge between M5 and GEMS, two memory systems emerged: M5's simple memory system and the more advanced Ruby Memory System from the GEMS project. M5's memory system is simple, and works by settings delays to each memory request, depending on how they hit in the memory hierarchy. Ruby is a more complete memory system simulation tool, and can be used to model new types of memory systems. The Ruby memory system is currently unsupported for ARM architectures.

The simulator has two main execution modes: *Syscall Emulation* (SE) or *Full System* (FS). In SE mode, the simulator runs without any real operating system. gem5 traps system calls from the executable and emulates them, often by passing them to the host operating system. In FS mode, the simulator can load an entire operating system, e.g., a GNU/Linux distribution, and run applications within the OS. gem5 supports many architectures; it can run binaries compiled for ALPHA, SPARC, MIPS, ARM, x86 and POWER architectures.

During simulation, gem5 keeps track of hundreds of different events related to the CPU core and memory system. In-detail statistics, similar to performance counters on real hardware, are then dumped for subsequent inspection. gem5 is also able to output a trace log while it runs, originally intended for debugging of gem5. A trace log contains user-selected events that happens within the simulated hardware. These trace logs grow quickly in size, easily tens of gigabytes, but provides useful insights of the simulated execution. In particular, they describes CPU activity down to the microarchitecture level and outputs simulated processor activity.

2.5 Global Optimization

Optimization is a field of applied mathematics that deals with finding the best set of parameters to optimize an objective function. A problem with N variables $\{n_0, \ldots, n_{N-1}\}$ in range $n_i \in \{0, \ldots, k-1\}$ will have a search space with k^N possible solutions. Each solution can be evaluated by applying the objective function to it, to obtain the solutions *fitness*. The fitness is a measure of how good the solution is and it is used to assist the selection of candidate solutions for evaluation.

The solution space can be thought of as a (N+1)-dimensional space that directly relates to the number of variables in the solution, plus one axis for the fitness value.

E.g., when a problem contains a single variable, its solution space might look like Figure 2.8 where the X-axis is the value of the single variable and the Y-axis is the fitness of this solution. Problems with more variables will have more axes. The optimization algorithms can be thought of as methods for exploring and finding the highest or lowest point in of the fitness-axis.

As k and N grow large, it becomes infeasible to search through and evaluate the vast amount of permutations and other techniques must be employed. Finding an arbitrary local optimum is often straight forward using classical *local* optimization methods such as a simple hill climbing algorithm. However, these methods cannot always be used to find a *global* optimum. A wide range of algorithms to search through a subset of the solution space exists, each with different approaches and properties. Many of these are described in detail in [40].

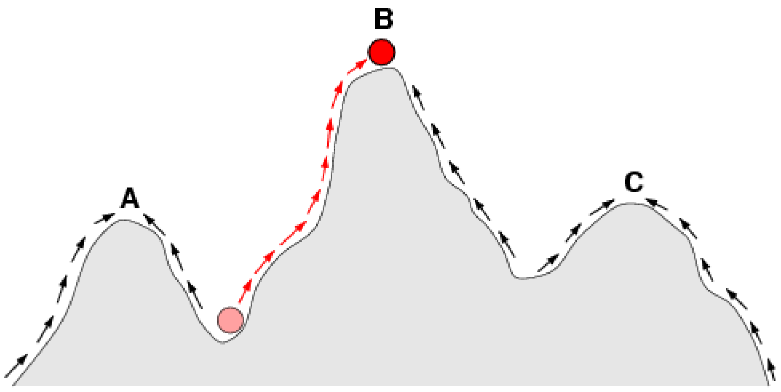


Figure 2.8: A one-dimensional fitness landscape. The arrows indicate the preferred flow of a population on the landscape, and the points A and C are local optima. The red ball indicates a population that moves from a very low fitness value to the top of a peak. Borrowed from [41].

In general terms, optimization algorithms can be divided into deterministic and non-deterministic approaches. The deterministic approaches can be thought of as a single path in the solution space that starts at a defined but most likely suboptimal solution and ends at the best solution. The non-deterministic approaches usually selects one or more paths at random such that each run might yield a different outcome. Deterministic algorithms will always find the same solution, but if the search space is large this might be extremely time consuming. Non-deterministic algorithms tend to have an explorative behavior [42]; each computation of the next state includes some form of randomness. E.g., simulated annealing will do the same as the hill climbing algorithm, but will have some chance of moving downhill, thus it will be less prone to be stuck in a local optimum. A brief overview of algorithms considered for this thesis is provided below.

Regression is described as a study of dependence between properties [43]. When data set contains values from at least two properties, regression can be used to find one value as a function of the other. An example of a regression technique is the least square method. This method will try to find a linear function, $y = a_1x_1 + a_2x_2 + ... + a_nx_n + b$, that minimizes the least square error. Implementations of linear regression solvers often formulate the problem as a system of linear equations [44], which can be solved by Gaussian elimination. More advanced forms for regressions can be used when the problem cannot be well mapped with a linear function, such as the Gauss-Newton algorithm [45].

Simulated Annealing is a technique that belongs to the field of stochastic optimization and metaheuristics, inspired by the process of annealing in metallurgy [46]. It starts in a random state s, and for each iteration it probabilistically decides between moving the system to a neighboring state s' or staying in state s. To avoid ending up in a local optimum, the probability starts high, but decreases over time. Hence, simulated annealing can quickly consider the most important parts of the state if configured adequately.

Evolutionary Algorithms is a term that refers to computational methods inspired by the process and mechanisms of biological evolution [47]. They differ from conventional algorithms by selecting the best-fit individuals in a population for reproduction and applying crossover and mutation to produce offspring. Only the best fit individuals go on to the next generation [48].

Global optimization problems are a well studied research area and there are countless ways to solve them. It can be difficult to know in advance which methods will provide the most rewarding results. Also, multiple approaches can be combined in efforts to extract the best properties from several branches, or to speed up convergence.

In this thesis, we have chosen to use an evolutionary approach called $1 + \lambda$, with $\lambda = 4$. It includes the following steps:

1. Generate λ random individuals.
2. Evaluate population.
3. Select the best individual.
4. Mutate λ new individuals from the best-fit individual.
5. Repeat step 2 through 4 for each generation.

In addition, we have borrowed ideas from simulated annealing by letting the probability of mutation decrease as fitness improves.

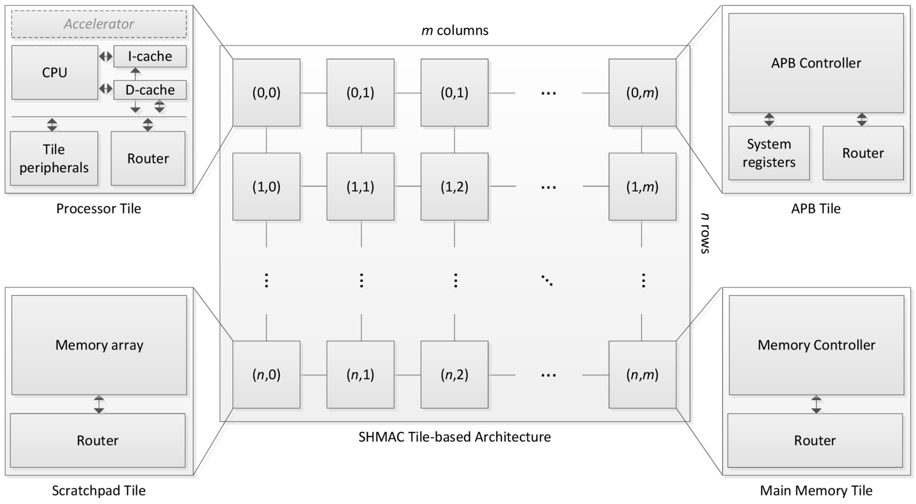


Figure 2.9: The SHMAC architecture. The figure shows how different kind of tiles are combined together and form a complete architecture. From [10].

2.6 SHMAC

SHMAC is a hardware prototype of a Single-ISA Heterogeneous MAny-core Computer [49, 10, 50]. It is an ongoing research project within the Energy Efficient Computing Systems (EECS) research area at the Department of Computer and Information Science, and Department of Electronics and Telecommunications at NTNU. The SHMAC project is driven by the *dark silicon effect*: as transistors become smaller, only parts of a chip can be powered simultaneously [4]. SHMAC implements two main strategies to mitigate the dark silicon effect, heterogeneity and specialization.

The SHMAC architecture is tile-based. Processing elements are laid out in a rectangular grid with connections to their nearest neighbor, as depicted in Figure 2.9. A router device present in all SHMAC tiles handles communication and data flow between tiles. In SHMAC, different kinds of specialized tiles/accelerators can be composed as desired, to form a computer tailored to the application. With the ability to evaluate different tiles with respect to energy and performance, the most advantageous core composition can be chosen.

Chapter 3 Building a Power Estimation Tool

The ultimate goal for this project is to model and estimate energy consumption for not yet implemented computer architectures. This allows new ideas to be prototyped and evaluated with respect to energy efficiency already at the design stage, easing the process of building energy efficient hardware. These evaluations can only serve as estimates and will doubtedly be truly accurate. Nevertheless, it can be used to test specific workloads and applications on specific processor configurations and evaluate ideas rapidly during the design phase.

As we are already supplied with a computer architecture simulator capable of tracing all sorts of hardware events, the next step is to extract power information from these event logs. In this chapter we introduce PET, a Power Estimation Tool. PET provides guided information about power usage for computer architectures and represents a major piece of our problem solution.

PET is implemented in C++ using the Boost Library [51]. The source code for PET and the rest of this project can be found Github:

```
git clone https://github.com/terjr/thesis.git
```

3.1 What is PET?

PET is a tool for power estimation of new as well as old architectures. It is built by measuring existing hardware with great detail, capturing discrete events and assigning each event a certain energy cost. When selected events have been weighted, one can run test programs through a simulator which is configured to act as the target hardware, as depicted in Figure 3.1. The simulator will generate a trace log containing the weighted events, which is then processed by PET. From this workflow, PET can produce a data set containing power consumption distributed over the simulation lifetime – a power profile of the program execution.

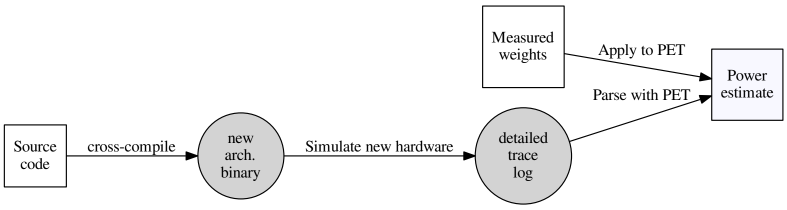


Figure 3.1: Example usage of the PET program.

When estimating power for an unimplemented hardware platform, the new hardware will be weighted similar to an existing implementation. As a consequence, this method requires a certain similarity between old and new hardware. We claim that in general, all modern computer architectures are built from comparable principles, and thus mappable to each other.

The accuracy will indeed suffer the more the model deviates from the reference hardware. PET's primary use is to identify *variations* between two implementations of the same instruction set architecture. For instance, one can experiment with a larger L2 cache to see how it affects energy usage and performance. The additional energy used by a larger L2 cache can be derived from existing implementations.

3.2 Approach

There are many considerations to take before creating a tool that should pretend to understand the implementation of hardware and the implications of features regarding energy efficiency. Through the next sections, we a model to be used in PET is developed and its inputs are defined.

3.2.1 Energy Modeling

Song et. al [52] identifies three major approaches to processor power modeling used in the past, and introduces an instruction-based energy estimation model that can be used for energy simulation at high speed. Their proposed method is expressed through the following equation, and includes the desired features of previous energy models.

$$P_{core}(t) = \frac{E_{unit} \cdot A_{datapath} \cdot w(t) + E_{static}}{T_{sampling}} \tag{3.1}$$

This method has two dependencies. First, one must have sufficient details of the processor in order to identify datapath components to form the $A_{datapath}$ matrix. The entries in $A_{datapath}$ are the invocation counts of physical components in the datapath with respect to the workload metric $w(t)$. $w(t)$ is typically comprised of instruction types or key operational parameters such as cache miss, ratio, pipeline stall cycles and number of executed instructions. Secondly, the energy unit vector E_{unit}, a vector enumerating the per-access energy cost, requires circuit-level knowledge of the target processor to calculate. $A_{datapath}$ can often be found by reverse engineering and benchmarking. The E_{unit}, however, is rarely available for commercial processors.

When building the model for PET, the model from [52] is simplified by combining $A_{datapath}$ and E_{unit} to form a vector of weights that directly corresponds to the cost of an event. We call this vector C. Power for each core over time, $P_{core}(t)$, is then modeled by the following formula.

$$P_{core}(t) = \frac{C \cdot w(t) + E_{static}}{T_{sampling}} \tag{3.2}$$

Here, C represents the global cost vector – a matrix enumerating the cost for all event types. Note that it is global and do not depend on time. $T_{sampling}$ represents the sampling period and E_{static} the static energy consumption.

3.2.2 Power Consuming Events

Choosing which events should be tracked and which workloads that would give good metrics is an important part of our method. We account for two main groups of events; CPU instruction events and memory activity events. The events in these groups are listed in Table 3.1. It is desirable to estimate energy consumption on literally all types of computing systems, ranging from large-size clusters to embedded systems. To provide this flexibility it was decided that PET should parse log files from the simulators rather than being built-in on a specific simulator. Most active and working architectural simulators supports this sort of trace logs. Even if they are formatted differently, the effort of adjusting to a new format is a lot less than the effort of building this tool within a simulator.

The trace logs contains information about everything that goes on within the fictional computer. Such a piece of information is defined in PET as a *simulator event*. A simulator event can be thought of as a unit of work that uses a specified amount of energy. When PET finds such an event, it increases the modeled energy consumption at the correct point in time where the event took place.

IntAlu	Basic integer ALU operation
IntMult	Integer multiply ALU operation
MemRead	Memory Read issued, triggers LS unit
MemWrite	Memory Write issued, triggers LS unit
SimdFloatMisc	NEON unit activated

(a) CPU core events.

L1IR	L1 instruction cache, read
L1IW	L1 instruction cache, write
L1DR	L1 data cache, read
L1DW	L1 data cache, write
L2R	L2 cache, read
L2W	L2 cache, write
PhysR	Main memory, read
PhysW	Main memory, write

(b) Memory events.

Table 3.1: Power consuming events.

The events described in Table 3.1 are the ones currently recognized by PET, but adding more (or removing) events is trivial. These events are selected mainly based on the information which is easily extracted from a gem5-formatted trace log, but also adjusted according to what could be checked with performance counters on the target hardware. Most of this information is available from [17], where different instruction loops were measured with both ammeter and performance counters. This is then correlated with the properties of the pipeline (as seen in Figure 2.7).

3.3 Input

PET needs two types of data in order to model power; a simulator trace log and event-type weights. Optionally, PET can also read an annotation file and display function calls in the output.

3.3.1 gem5 Trace Logs

In order to create the simulator trace log with the information required by PET, gem5 must be run with a specific set of parameters. By executing gem5 with `--debug-flags=Bus,Cache,MemoryAccess,Exec`, gem5 will output trace files that look like Listing 3.1.

```
1  3021: system.physmem: Write of size 8 on address 0x82fe0 data 0xe1a0f00eee1d0f70
2  3021: system.cpu.icache: access for ReadReq address 9c0 size 64
3  3021: system.cpu.icache: ReadReq (ifetch) 9c0 miss-
4  ...
5  3432: system.cpu.dcache: Block addr 81f0 moving from state 0 to state:7 valid: 1
6  3432: system.cpu.dcache: Leaving recvTimingResp with ReadResp for address 81f00
7  3432: system.tol2bus.respLayer1: The bus is now busy from tick 234320 to 236376
8  1642: system.cpu T0 : 0x89d4.0 : ldr   r1, [sp] #4      : MemRead :  D=0x00000000
9  1642: system.cpu T0 : 0x89d4.1 : addi_uop   sp, sp, #4 : IntAlu :  D=0x00000000b
10 1701: system.cpu T0 : 0x89d8   : mov   r2, sp          : IntAlu :  D=0x00000000b
11 1701: system.cpu T0 : 0x89dc.0 : str   r2, [sp, #-4]!  : MemWrite :  D=0x0000000
12 1760: system.cpu T0 : 0x89dc.1 : subi_uop   sp, sp, #4 : IntAlu :  D=0x00000000b
13 1760: system.cpu T0 : 0x89e0.0 : str   r0, [sp, #-4]!  : MemWrite :  D=0x0000000
14 4000: system.membus: recvTimingResp: src system.membus.master[0] ReadResp 0x1640
15 4000: system.l2: Handling response to ReadResp for address 1640
16 4000: system.l2: Block for addr 1640 being updated in Cache
```

Listing 3.1: gem5 trace log.

Each line in Listing 3.1 represents an event that happens in the simulated hardware. Line 1 tells that a write access to physical memory has happened. Line 2 is the event of instruction cache access, while Line 3 shows that this request failed. During this simulation, there is also events like Line 5 which represents that the data cache updates some content. The discrete instructions running through the CPU is also logged, e.g., Line 8 shows a load instruction and Line 9 shows an add instruction.

The trace log input can be opened as a file or read from a `Unix` pipe. Discrete events are extracted from the trace log and power consumption is accumulated in equally sized timeslots in PET. Internally, these time steps are called buckets and its size is parameter controlled. Often, it is more practical to specify the number of buckets in the output rather then specifying the number of simulator ticks in each bucket. PET is able to estimate the bucket size by peeking at the last tick of a trace file. This is not possible when reading from a pipe, i.e., `stdin`. The trace file is not necessarily in tick order, but close enough to set a reasonable bucket size. The bucket size estimation algorithm is shown in Algorithm 3.1.

Listing 3.1 shows that the events in the trace log is not necessarily in their correct order. This means that PET must accumulate power consumption to the entire timeline at all times. Consequently, it is not possible to produce a continuous output flow. The results are stored in memory and written out when the entire input is parsed.

3.3.2 PET Weight Files

Equally important as finding the correct events is assigning each event the correct amount of power consumption. As each event will count differently depending on the architecture, PET will read a weight file along with the gem5 trace log. A sample

Algorithm 3.1 Bucket size detection algorithm.

```
function numTicks( traceFile ):
    # Find file size
    eof_pos = traceFile.getSize()

    # Seek almost to end, avoid last newline
    traceFile.seek( eof_pos - 3 )

    # Trace from back of file to second last newline
    while not traceFile.currentChar is '\n':
        traceFile.seek_backwards

    # File stream position is now at beginning of last line
    # Parse this line
    simulatorEvent = parseLine( traceFile.getLine() )

    # Return the tick of the retreived event
    return simulatorEvent.getTick()
```

weight file is shown in Listing 3.2. As the timeslots are specified in simulator ticks instead of CPU cycles, the values have been chosen to match a 2 GHz processor, i.e., one CPU cycle per 500 simulator ticks [1]. If this method was to be applied to a processor with a different clock speed than 2 GHz, the weights would have to be scaled proportionally. This is not the case for the static power drain, as it is added to each timeslot, and not scaled in accordance with bucket size. It is also important to understand that the weight is applied once for each event, so events that naturally takes a number of cycles will have a high weight, which is in reality distributed over many ticks. This will not be accurate if an expensive event is applied at the border of a bucket. It is assumed that accuracy at this level is not important enough to increase the complexity of PET.

The weights displayed in Listing 3.2 are accumulated each time PET discovers a recognizable event in the log file. A simplified version of this algorithm can be found in Algorithm 3.2

3.3.3 Program annotation files

PET has the ability to annotate its output using a map from PC to function name (or rather, symbol name). The simulated binary itself is not an input to PET, instead PET comes bundled with an annotation tool: `scripts/annotate.sh`. This tool extracts symbols from the binary file, compiled with debugging symbols, to a text file in the format seen in Listing 3.3. The left column represents the address where the function is found, and the right column is the function name. PET will tag

[1]This is the default gem5 simulation granularity.

```
# This file contains weights for the ARM Cortex-A9 embedded within
# Samsung Exynos 4412 Prime meassured on an Odroid X2. Details in
# Runde & Hvatum 2013 "Exploring Instruction Level Energy Efficiency"

# CPU Core Activity
IntAlu 170
IntMult 300
MemRead 80
MemWrite 50
SimdFloatMisc 400

# Memory related activity
L1IR   230
L1IW   340
L1DR   230
L1DW   340
L2R   1100
L2W   1300
PhysR 2600
PhysW 2800

# Static power
Static 70
```

Listing 3.2: Weight file example.

Algorithm 3.2 Power accumulation algorithm.

```
# map of accumulated power for each time step
map<time,power> output

# input is all trace log lines, elements in weight file and
# the determined bucket size (number of simulator ticks in
# each bucket)

function assignWeights( traceLogLines, weightMap, bucketSize )
    # run through each line
    for each line in traceLogLines:
        # extract event parameters from line
        simulatorEvent = parseLine( line )

        # get the assigned weight from weight file
        eventWeight = weightMap[simulatorEvent.getEventType()]

        # add this weight to the output map
        output[simulatorEvent.getTick()/bucketSize] += eventWeight
    return output
```

```
00008120 read_int
00008194 group_number
00008680 strip
00008734 read_int
000087a4 group_number
000089bc fini
00008a08 call_weak_fn
00008a2c deregister_tm_clones
00008a64 register_tm_clones
00008aec frame_dummy
00008b44 main
00008f40 check_one_fd
00009420 abort
000097ac exit
000099ec rand
00009ed0 flush_cleanup
00009ff0 save_for_backup
0000bd44 malloc_init_state
0000bda4 ptmalloc_unlock_all2
0000be3c mem2mem_check
0000beb8 mem2chunk_check
0000c128 ptmalloc_lock_all
0000c318 new_heap
```

Listing 3.3: Annotation File Example.

each bucket with the last seen symbol within that bucket. `scripts/annotate.sh` is constructed using `objdump` and is shown in Listing 3.4.

```
#!/bin/sh
BINARY=$1
OBJDUMP=$(which arm-linux-gnueabi-objdump)
if [ -z "$OBJDUMP" ]
then
    OBJDUMP=$(find / -name 'arm-*-objdump' -print -quit 2>/dev/null )
fi

$OBJDUMP -S $BINARY | grep '<.*>:' | grep -v '<_' | tr -d '<>:'
```

Listing 3.4: `scripts/annotate.sh`: Script to extract symbols from a binary.

It should be mentioned that a program compiled with debugging symbols contains hundreds, if not thousands of symbols. Often, many of these symbols are called in clusters, while long periods of the program are spent in loops that are not using any symbols at all. This often renders a graphical representation of the power log with complete annotation as a complete mess. We recommend creating the annotation file using the mentioned tools, but manually filtering out only the symbols of interest.

3.4 Output

When a log file is consumed by PET, the output should be usable for many applications. In early stages of the design phase, or when great differences are expected, a sparse annotated graphical output might be the best way of visualizing power consumption. As the project evolves and more subtle changes are evaluated, a textual output will be easier to compare. PET supports three different output options:

graph
: This format is the default, and provides an overview of the entire program in an easily digestible format. An example of such a graph is printed in Figure 3.2.

plain
: The example in Listing 3.5 shows the *plain* format, which is intended to be used for further machine processing.

table
: The table format, with an example shown in Listing 3.6, shows a terminal-printable output which is easier to read. It might come in handy as the default format might be hard to read when you are looking for specific information.

3.4.1 Units

The output format is understood as timeslots in which the architecture has a certain current drain, which should be multiplied with applied voltage to get consumed energy. The values are given as milliamperes, equal to milliwatt if voltage is 1 V. Milliamperes are used as it is easier to find current drain rather than wattage with the setup used in this project, as described in section 2.2. When power is estimated for a new architecture, the resistance of the circuit is difficult to deduce, and voltage might also be an unknown factor. Given Ohms law in Equation 3.3 and the definition of electric power in Equation 3.4

$$I = \frac{U}{R} \tag{3.3}$$

$$P = U \cdot I \tag{3.4}$$

it can be found that power equals current squared times resistance

$$U = R \cdot I$$

$$P = (R \cdot I) \cdot I$$

$$P = I^2 \cdot R \tag{3.5}$$

and that power equals voltage squared divided by resistance

$$P = U \cdot \frac{U}{R}$$

$$P = \frac{U^2}{R} \tag{3.6}$$

Thus, estimating only the current drain means that the power at each point will be unknown without knowing resistance or voltage. Further, energy consumption cannot be estimated unless the new architecture is similar in terms of voltage and resistance to a chip where these numbers are available. Even the current drain might not be representable at all; if resistance or voltage is unequal to the levels found in the reference chip, the final numbers will be far off.

Equation 3.5 and Equation 3.6 states how voltage and current is important for energy consumption. The current is, from Equation 3.3, dependent on resistance as well as voltage. With this in mind, and knowing that power in a complex environment is a delicate matter, the most important application for PET is to point in the right direction. PET will never give accurate power estimations for new chips, but will provide useful information for seeing if a new feature or architectural fix will render the final architecture more energy efficient or not.

3.4.2 Examples of Output Data

Visualization is often a good thing when inspecting old or trying to understand new problems. Figure 3.2 shows an example of PET `graph` output format, with annotations.

Example of the `plain` output format can be seen in Listing 3.5. The left column is the bucket number, while the right column is instant current draw from the modeled architecture.

When reading the output directly from console, a more descriptive output format is the `table` format. An example using this option is rendered in Listing 3.6.

3.5 Architecture

The trace logs read by PET can easily grow to 10s of gigabytes. Due to memory constraints on commonly available computers, it is not feasible to read the entire log

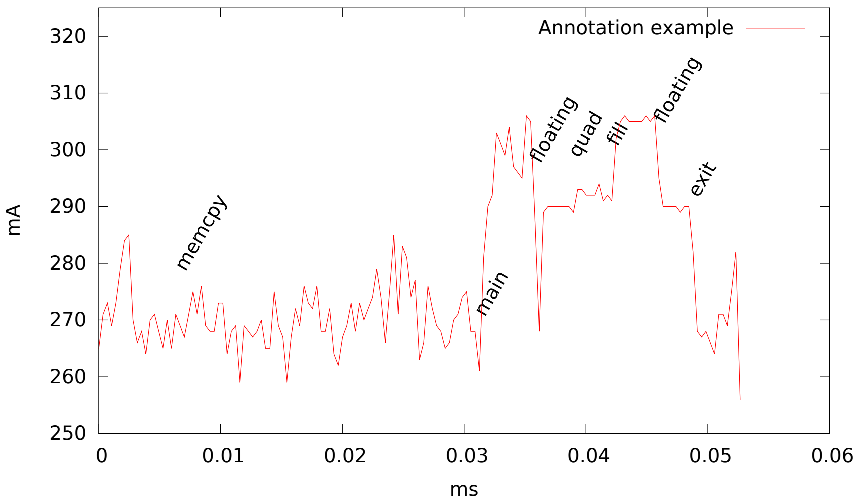


Figure 3.2: PET graphical output. This example contains annotations, each label represents the entrance of a function.

```
0 120 memcpy
1 113 start
2 150 main
3 123 main
4 133 fun1
5 117 main
```

Listing 3.5: PET plain output with function annotations.

file into memory and then start parsing. One of PET's major design goals is to be user friendly and convenient to use, and as a consequence it must be reasonably fast. To gain speed, the PET core is built around a parallel pattern similar to MapReduce [53].

3.5.1 Overview

In order to obtain acceptable performance, we have looked at different ways of digesting large data sets. The final implementation of PET follows a scheme borrowing ideas from the producer-consumer pattern as explained by Gamma et. al. in [54] and the MapReduce algorithm. As depicted in Figure 3.3, this scheme makes it rather easy to let a producer (sequentially) read the lines from the log file into ring buffers (produce) and let multiple consumers pick from their ring buffer (consume). Each consumer parses the log lines they pick, and apply the weight of each read event to

```
/-----------------------------------------\
|    Bucket    |   milliAmps |     Symbol     |
|------------|------------|---------------|
|            0 | 120.000000 |     memcpy     |
|            1 | 113.000000 |     start      |
|            2 | 150.000000 |     main       |
|            3 | 123.000000 |     main       |
|            4 | 133.000000 |     fun1       |
|            5 | 117.000000 |     main       |
\-----------------------------------------/
```

Listing 3.6: PET table output with function annotations.

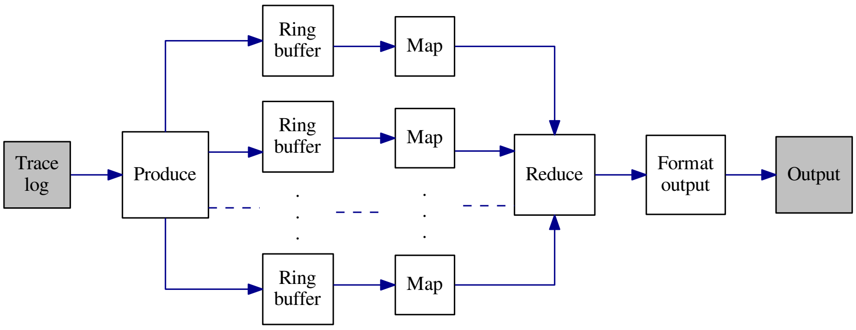


Figure 3.3: How PET works.

their result vector (map). When all lines are read and parsed, the result vectors are merged (reduce) and idle-task power and static power consumption is added. This combination of algorithms allows PET to take advantage of all available cores.

The next subsections will describe in detail the most important parts of the workflow, in sequential order. For further understanding of the program flow, a call graph is seen in Figure 3.4.

3.5.2 Argument Parsing and Program Options

As any other non-trivial programs, PET has to adapt to input options given from the command line or from a settings file. PET makes extensive use of the `Boost` library and utilize `Boost::Program_options` for parsing the command line. This allows easy extraction of program options, both with long (`--option=`*`value`*) and short (`-o` *`value`*) option style.

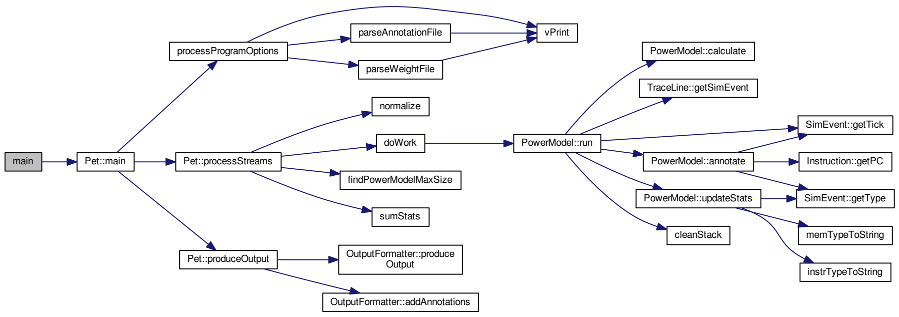


Figure 3.4: Call graph.

3.5.3 Reading Trace Logs

When arguments are parsed and a trace log has been specified, either by path or as `stdin`, a single thread is kicked off reading each line of the log file into a C++ string container. This happens in the `Pet::processStreams` class member function seen in Figure 3.4. The string container is then inserted into one of many circular buffers. The circular buffers are implemented with `boost::lockfree::spsc_queue`, a lock-free single producer, single consumer queue. The property of being lock-free is explained by Tim Blechmann in [55]:

> Data structures are *lock-free*, if some concurrent operations are guaranteed to be finished in a finite number of steps. While it is in theory possible that some operations never make any progress, it is very unlikely to happen in practical applications.

In PET, this queue has a fixed size of 8192 elements, but dynamic size is also available in the library implementation. Which buffer the string is inserted into is determined by a simple circular algorithm; the next ring buffer is selected when the current one is full. When the buffers are small, each are filled fast enough to keep all workers occupied. We observed that this method avoids locking better than using a single ring buffer shared by all worker threads. The number of threads and the size of the ring buffers are tightly coupled with how fast the host computer is able to feed PET with the log files.

3.5.4 String to Event Mapping and Power Accumulation

String parsing and mapping are the most compute-intensive parts of PET. PET spawns multiple worker threads as specified by the user. As the producer fills the ring buffers for each of the workers, the workers pick strings from their pool. The strings are popped from the ring buffer, thus making space for new elements right away. Each string is parsed by the `TraceLine` class, which looks for patterns in the strings containing known event types. When connecting PET with gem5, the trace logs as previously seen Listing 3.1 contains an event type designation in the second colon-separated column. The `TraceLine` class extracts this part by string trimming.

The event types are instantiated as objects of their parent type (Instruction- or Memory-event). The right parameters are found from progressive string parsing. If the event is unrecognized, a dummy object of type `UnknownEvent` is returned. This type has zero cost later in the reduce phase. Each event object is able to figure out its own weight as written in the *weights*-file. After the event has been parsed, the weight is added to the power model at the corresponding time step.

In order to reduce the time used for disposal of the string objects after they are parsed, they are placed in a static-size array. When this array is full, or the ring buffer is empty, the worker frees all the string data. This helps keeping the memory footprint low while avoiding unnecessary calls to `free()`. This optimization does not have a massive impact on performance, but as can be seen from [56], the `free()` implementation contains enough pointer arithmetic to make a difference in a tight loop.

3.5.5 Data Reduction

When all lines from the trace log have been consumed by the workers, the threads are joined and their data is returned as standard C++ vectors. These vectors are further wrapped in yet another standard C++ vector. The inner vectors are then combined; the value from the corresponding buckets in each vector is added together and put in a result vector. This reduction happens as the last part of `Pet::processStreams`, as seen in Figure 3.4.

After this reduction, the number of idle cycles is estimated by subtracting recorded events from the maximum number of events in a bucket. This is done more simplistic than accurate using Equation 3.8. `eventsInWorkerBucket` is the number of events recorded in each vector at each bucket, and each event is pinned to the cycle where it originated. Note that N is the number of worker threads, not the number of buckets.

$$eventsInBucket = \sum_{n=1}^{N} eventsInWorkerBucket_n \tag{3.7}$$

$$idleEvents = \frac{ticksInBucket}{ticksInCycle} - eventsInBucket \tag{3.8}$$

It should also be noted that even though this method might work well in a single-cycle in-order CPU, the out-of-order nature of the Cortex-A9 makes it hard to tell how many idle cycles actually occurred. E.g., a single cycle may fill the pipeline with four events, then idle the three next cycles; this would be calculated as no idle time. When the approximate numbers of `idleEvents` have been estimated, that number is multiplied by the *Idle*-weight and added to the sum in the result vector. Finally, the entire vector is normalized according to bucket size and then static current drain is added.

3.5.6 Output Production and Annotations

After the result vector has been completely accumulated, annotation is added to a new vector in the same manner as the data reduction. With the new, merged annotation map containing all last matches between a symbol and the program counter within each measure point, this map is fed to the `OutputProducer` object as seen in Figure 3.4. The `OutputProducer` is responsible for generating output as defined by the input arguments. Its options have already been described in section 3.4, and its implementation is a simple nested if-else-clause that calls internal functions for each output type. The graphical output is produced using a wrapper around `gnuplot`, while the textual outputs are created by `printf`-statements.

3.5.7 Unit Tests

All internal string parsing is verified by unit tests. The unit tests are written with help from the Boost Test Library [57].

The test library generates a new binary with the same program content, except the main function, thus the program flow is different. The test binary will run through the listed functions with a certain input, and if the output is unexpected, the test binary will print to the console an error message containing a description of what went wrong.

Chapter 4
PET Performance Tuning

Several factors affect the PET model accuracy. We now consider the steps needed to adapt the use of PET on a new architecture. First, one must create a CPU model using gem5's Python interface, resembling the modeled hardware. Then, finding proper weights is formulated as a global optimization problem and matched up against existing hardware.

4.1 Measuring a Real World Processor

The results contributed in this work relies on the existence of a method to isolate and measure core voltage on a hardware implemented reference CPU. This is possible due to the V_{core} separation on the development kit, as mentioned in section 2.3.

In [17], we conducted experiments to quantify the energy cost of an instruction executing on a modern out-of-order mobile processor. We were able to do this by completely bypassing the memory hierarchy utilizing special hardware (*fast-loop mode*) and sampling a running average. Voltage drop over a shunt resistor set in series with the ODROID-X2 development board was measured. This voltage drop was used to calculate energy used in the processor core. The results obtained are further used to tune PET towards this architecture.

4.2 Simulator Environment

PET relies heavily on a front-end that can execute a binary on a simulator and output $(time, event)$ tuples as a trace of execution. In this thesis, we are using the gem5 simulator, but technically PET could be modified to support any simulator front-end. The power profile generated by PET are derived from the weight configuration given as input and the simulation trace, so it is important that the simulator trace is similar to a hardware execution.

Butko et al. [58] reports gem5 *runtime* accuracy to vary from 1.39 % to 17.94 %, with the most accurate results coming from programs with low memory usage. The benchmarks used represents a wide variety of scientific workloads, as well as media applications and memory intensive synthetic benchmarks. The benchmarks used has a high degree of instruction diversity, so they get away with having a simple CPU model. The real program flow gets diluted and accurate timings are obtained simply by setting the correct CPI value. In fact, they used an in-order model to simulate an out-of-order core.

We claim that the key to obtain high precision power estimates is an architectural simulator with great accuracy. The simulator must exhibit similar timing characteristics as the target hardware under all workloads considered, such that the simulated execution resembles the hardware execution as close as possible. This means that accurate power estimation needs these discrete events to happen, and they must happen with a realistic timing related to their triggering cause. Without a simulator capable of providing decent accuracy over system events, power estimation using methods as suggested in this report will fail. We will now explain how gem5 can be configured to improve simulator accuracy for a particular CPU core, as well as pointing out difficulties with this approach.

4.2.1 gem5 CPU configuration

The gem5 simulator is bundled with an out-of-order core implementing the ARMv7 ISA. It serves as a basis for the evolution of our custom core meant to model a Cortex-A9 on the Exynos 4412 SoC. ARM cores can be configured with caches of varying sizes decided by the implementing vendor and will have varying performance accordingly. Thus, it is important to tune all CPU parameters so that it matches the modeled SoC. It is not publicly known which SoC the default configuration attempts to model, but according to the gem5 mailing list it is neither Cortex-A9 nor Cortex-A15 [59]. However, the fact that it is made for the ARMv7 instruction set and is out-of-order leads us to think that minor modifications will make it a decent Cortex-A9 model.

The model gem5 uses in its simulations can easily be configured using a Python interface. We started with gem5 changeset `aaf017eaad7d` and added a new CPU configuration file for the Exynos 4412; `gem5/configs/common/Exynos_4412P.py`. A few other files were edited to accommodate the new processor definition. Please refer to Appendix B to see all patches applied in these experiments.

We found many sources of information claiming to know implementation details of the Cortex-A9, including [58, 60, 61, 62, 63, 64, 65, 66].

Combining this information helps us build a gem5 model for the Exynos SoC, but it is still not trivial. We found the simulator to perform different from the physical hardware, even though the system parameters were set equal. We claim that this is due to the abstraction the simulator provides us; the simulator is not a complete model of the hardware. Gibson et. al. [67] have done a similar experiment using different simulators and concluded that bugs and omissions in system simulators may render accuracy tuning difficult.

To improve our results, we adjusted the model specification such that it *performed* similar to the hardware, i.e., programs executed in about the same number of clock cycles. Even if the processor implementation would have been completely transparent, it would still be hard to leverage the use of a multi-architecture computer simulator. Features such as fast-loop mode found in certain ARM cores must have a corresponding implementation in the simulator for complete correctness; it would be infeasible to include such details in a general simulator.

We scripted the simulator to run a wide variety of configurations, about 50 in total. We evaluated their performance by comparing simulated execution time to execution time on real hardware. Real execution time was inferred by measuring power during program executions and identifying when the CPU was working. The final CPU configuration is shown in its entirety in section B.1.

4.2.2 gem5 Memory Model

The memory system is an important part of a system simulator when doing performance estimations. At the time of writing, gem5 will not easily work with an out-of-order CPU model together with the GEMS Ruby memory system. Lacking other methods and considering our resources, the simple memory system will provide events that PET can use to determine memory and memory bus communications. The simple memory model was tuned as shown in section B.6.

4.3 Multi-objective Weight Optimization

With an accurate CPU model, the event weights used by PET must now be tuned to match measured power consumption on real hardware. We formulated this as a multi-objective optimization problem.

We started by creating a set of training workloads, essentially computer programs designed to hold certain characteristics compiled to a native ARMv7[1] binary. The design and selection of these will be elaborated in the next subsection. We executed the binaries in the gem5 simulator with the CPU model from last section to obtain

[1]ARMv7 is the instruction set architecture supported by ARM Cortex-A9.

Algorithm 4.1 Algorithm used to evolve a set of event weights.

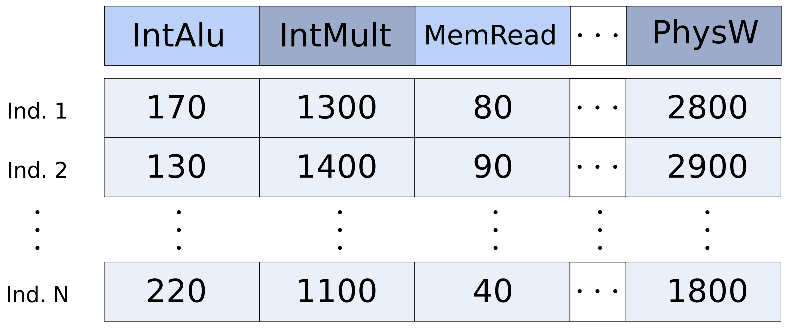
```
individuals = createIndividuals()
generations = 6000

best = None
for 1 to generations:
    for ind in individuals:
        evaluate(ind)
        if ind betterthan best:
            best = ind
    individuals = mutate(ind)

print best
```

a file containing `(time, event)` tuples from the (modeled) execution, just like in Listing 3.1. The next challenge was to assign each of these events a cost.

We attack this problem by running a multi-objective optimization algorithm. We picked a subset of event types that is believed to impact energy consumption, as we described in subsection 3.2.2. Choosing too many events could give us overfitting issues, but taking too few out could lead to lack of detail in our model. We experimented with dozens of optimization algorithms and ended up combining a $1 + \lambda$ evolutionary strategy with simulated annealing. The evolutionary part would make sure that our algorithm was explorative enough (i.e., it covered large parts of the candidate solution space), while the simulated annealing part made the algorithm more aggressive to start with. Using DEAP [68], a Python framework for evolutionary algorithms, we were able to prototype our ideas rapidly. Algorithm 4.1 describes the final algorithm.



	IntAlu	IntMult	MemRead	···	PhysW
Ind. 1	170	1300	80	···	2800
Ind. 2	130	1400	90	···	2900
⋮	⋮	⋮	⋮	⋮	⋮
Ind. N	220	1100	40	···	1800

Figure 4.1: Sample individuals.

Each individual is a set of CPU events, each mapped to an energy cost. Figure 4.1 illustrates what individuals in the population look like. The top row enumerates all event types; e.g., the cost of an integer multiply event for individual 2 is 1400. The unit here is not really important, but roughly corresponds to $Ampere \cdot cycles$, with static drain deducted. The evolutionary algorithm starts by generating 10 individuals by using the algorithm shown in Algorithm 4.2. It assigns most weights a random value in range $\{0, \ldots, 999\}$, while guiding some weights to a known value. E.g., the static energy contribution is known to lie around 96 mA by physical measurements, so its weight is believed to be close to that number. Please note that there is not necessarily a direct mapping between the GA (*Genetic Algorithm*) weights and measured milliamperes; it is up to the evolutionary algorithm to figure out what causes a best fit.

Algorithm 4.2 Algorithm used to generate individuals.

```
# Properties kept by each individual
individualProperties = {
    'IntAlu', 'IntMult', 'IntAlu', 'IntMult', 'MemRead', 'MemWrite',
    'SimdFloatMisc', 'L1IR', 'L1IW', 'L1DR', 'L1DW', 'L2R', 'L2W',
    'PhysR', 'PhysW', 'Static', 'Idle'}

# Generate random individual
def createIndividual():
    ind = EmptySet
    for prop in individualProperties:
        ind[prop] = random()*1000
    ind['Static'] = 96*(random()+0.5)
    ind['Idle'] = 50*(random()+0.5)
    return ind

# Generate 10 random individuals
def createIndividuals():
    individuals = EmptyList
    for 1 to 10:
        individuals.add(createIndividual())
    return individuals
```

To calculate the fitness of an individual, we run PET with the genome weights on a set of workloads and compare the energy profiles with measurements on hardware. The two sets are then compared using Algorithm 4.3, resulting in the individual's fitness – lower is better.

We emphasize that the use of an evolutionary algorithm is an arbitrary choice. Technically, any algorithm could have done this for us. When the weights have evolved to match the hardware measurements, the method used to obtain the weights is of no importance.

Algorithm 4.3 Algorithm used to evaluate an individual.

```
# Evaluate individual
tests = {'trend', 'submul', 'sha2', 'pi'}

# Find RMS distance between two data sets
def distance(graph1, graph2):
    diff = 0
    for points in graph1, graph2:
        diff += (points[1] - points[2]) ** 2
    return sqrt(diff / minsizeof(graph1, graph2));

# Find RMS distance between PET estimate and measured power consumption
def runTest(ind, test):
    data = runPET(getWeights(ind), test)
    measure = getMeasureData(test)
    return rmsDiff(data, measure)

# Find all distances and return result as RMS
def evaluate(ind):
    fitness = 0
    for test in tests:
        testFit =  runTest(ind, test)k
        fitness = fitness + (testFit ** 2)
    return sqrt(fitness)
```

4.4 Choosing Workloads

When running a genetic algorithm, it is critical to lead the evolution in the correct direction. In our case, this is done by providing a reasonable set of workloads (i.e., ARMv7 programs) that stresses distinct modules in the processor. For instance, a memory intensive workload will have high density of memory-related events from the simulator, and will support the genetic algorithm in determining cost for memory accesses. It is important for the set of workloads that are chosen to be diverse and stress many conditions the processor can operate in, e.g., mixes of compute intensive and memory intensive programs. A poorly chosen set of workloads will not give a fair judgment on which genomes that fit well. A bad workload might be too biased towards a few parameters, neglecting the rest, or even mislead the GA into a local optimum [48]. All training programs are compiled with soft-floats to keep simulation complexity low. Another worry is that within the training set, there will most likely exist multiple Pareto optimal solutions [69], but only one of these can truly match the real power consumption.

We came up with the following four workloads.

Pi

This test calculates Pi using Monte Carlo simulation. It includes floating point multiply and division. It runs for a fixed amount of iterations.

SHA-512

The SHA-512 algorithm is a hashing algorithm used in cryptography. It includes a mix of integer operations and memory usage. Implementation from [70].

Trend

This test has two parts. It starts with a tight add loop, and then continues with extensive memory allocation. Presumably, this will create a shift in energy consumption between the two stages.

SubMul

The SubMul test borrows ideas from the previous program, but instead of testing ALU and memory, this test compares subtract and multiply (both ALU).

We claim that the workloads used in this experiment spans the most common instruction types while being simple enough to be simulated in gem5 on reasonable time.

4.5 Results

We have now discussed how PET must be tuned to work with satisfying accuracy. Here we present the results obtained by tuning towards the ARM Cortex-A9.

4.5.1 gem5 CPU Model Accuracy

We modeled different CPU configurations for gem5 and achieved runtime in milliseconds as tabulated in Table 4.1. Each test was run with the command in Listing 4.1, with `$CPU` changed to `exynos_4412p`, `arm_detailed` and `timing`.

```
$ build/ARM/gem5.opt --remote-gdb-port=0 -d m5out
    configs/example/se.py -c bin --cpu-type=$CPU
    --mem-type=LPDDR2_S4_800_x32 --sys-clock=440MHz
    --cpu-clock=1700MHz --num-l3caches=0 --caches --l2cache
    --l2_assoc=16 --l2_size=1MB --l1d_size=32kB
    --mem-size=2048MB --l1d_assoc=4 --l1i_assoc=4
```

Listing 4.1: gem5 Command Line.

As can be seen, we were able to get the runtime of our modified O3 model (`exynos_4412p`) fairly close. With ordinary workloads Pi and SHA-512, the execution time differed

	Pi	SHA-512	Trend	SubMul
Real hardware	13.500	22.600	14.600	28.200
exynos_4412p	13.790	22.819	11.898	23.978
arm_detailed	6.708	10.368	4.344	8.653
timing	19.564	40.659	20.503	41.964

Table 4.1: gem5 runtime accuracy (O3 with classic memory system). In milliseconds.

```
Idle 437
IntAlu 239
IntMult 256
L1DR 54
L1DW 28
L1IR 15
L1IW 129
L2R 97
L2W 785
MemRead 542
MemWrite 56
PhysR 0
PhysW 0
SimdFloatMisc 1648
Static 6
```

Listing 4.2: Final weight configuration.

by less than 2.2 %. Trend and SubMul, the synthetic tests, differed by 22.7 % and 17.6 %, respectively.

4.5.2 Optimization using $1 + \lambda$

PET has been optimized by a genetic inspired algorithm, so it was expected to perform very well on the training data sets. Listing 4.2 lists out the final weights found by the GA. The weights have reasonable values for the ALU operations (`IntAlu, IntMult, MemRead, MemWrite, SimdFloatMisc`) compared to the results found in [17]. The values for the cache hierarchy (`L1DR, L1DW, L1IR, L1IW, L2R, L2W`) does also seem adequate. `PhysR` and `PhysW` are set to zero because the physical memory of the Exynos 4412 Prime is not believed to be powered by V_{core}. However, peculiar values are set for `Idle` and `Static`. Our theory is that CPU idle time is often caused by high I/O activity; this would explain how an idle CPU could use much power. However, we can not be certain how the events match to the hardware. The GA only optimizes the event weights and ignores the intended meaning of them. One must also remember that the `Idle` events are calculated by a very simple model, and the cost can be high if too few idle cycles are recognized by PET.

Results for each training set after the training session are displayed in Figure 4.2, Figure 4.5, Figure 4.3 and Figure 4.4. Each figure represents one of the workloads explained in section 4.4. The red line in each figure represents the prediction by PET, while the green line is a direct plot of readings from the test setup. To remedy the mismatch in execution time, the prediction done by PET is stretched.

Training Set: `Pi`

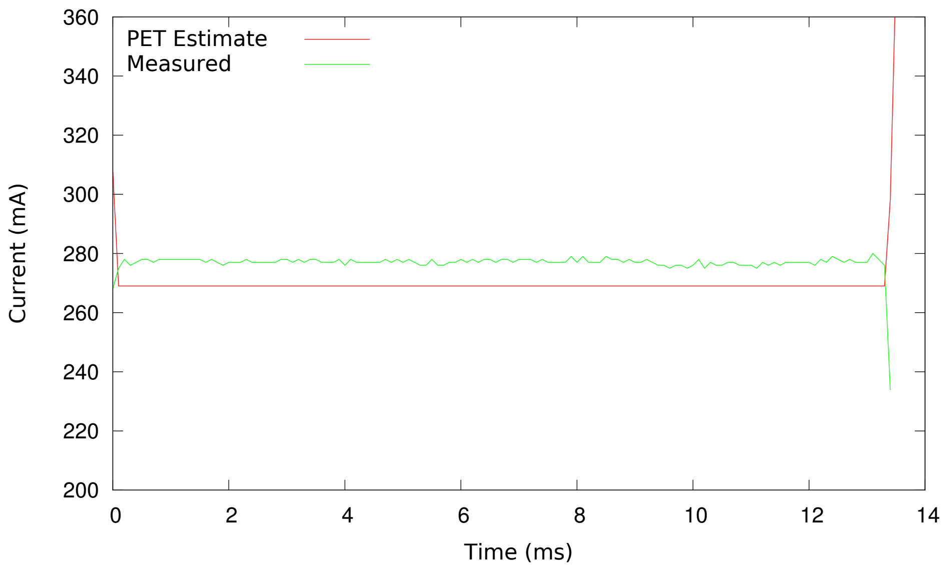


Figure 4.2: Overlay of PET training results (red) and training data (green) for the Pi test.

The `Pi` test uses floating point and random values with a static seed. The program is compiled with soft-float, so it can be viewed as an integer stress test. We obtain rather good accuracy with this test, in general the error is a below 10 mA. As it represents a common workload, this is a very promising result. It is interesting that the beginning and the end of the program are the most power consuming pieces. This is probably due to the simplistic method idle time is calculated, and the out-of-order nature of the processor makes the genetic algorithm overestimate the cost of idle.

Training Set: `Trend`

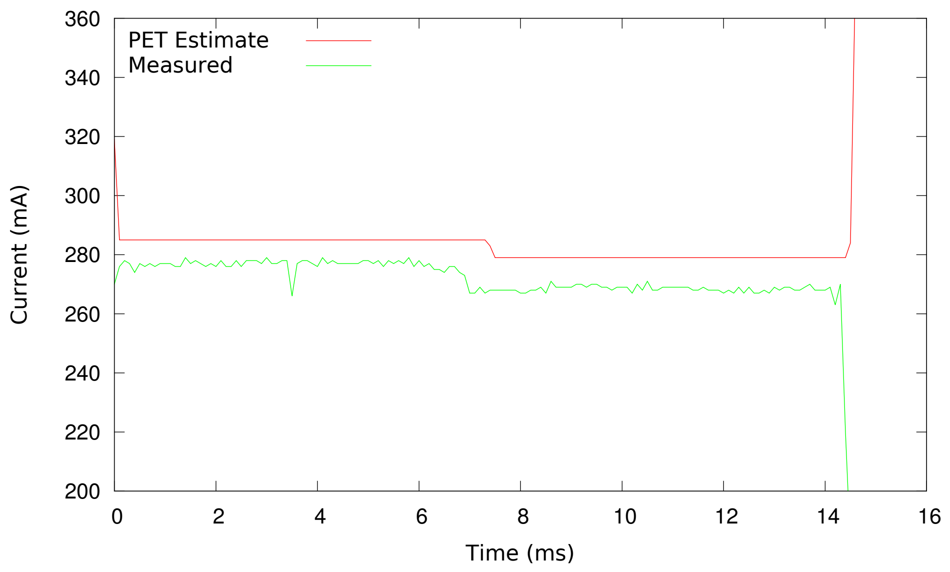


Figure 4.3: Overlay of PET training results (red) and training data (green) for the Trend test.

The `Trend` test checks if PET can follow the flow from an ALU intensive program to a memory intensive program. The accuracy is close to the Pi training set, but the most important factor with this test is that the drop at 7 *ms* is resembled in the prediction. Given that PET recognizes this drop, we know that PET adjusts correctly for the change in instruction flow. The fact that the sudden drop in power consumption seems to come a bit too slow is most likely due to discrepancies between the gem5 model and the real hardware.

Training Set: `SubMul`

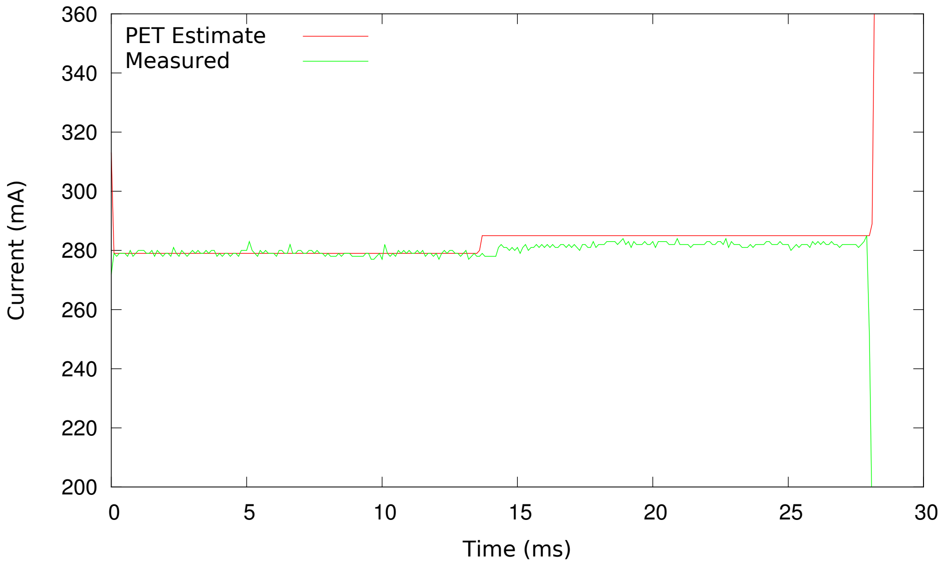


Figure 4.4: Overlay of PET training results (red) and training data (green) for the SubMul test.

Further, the `SubMul` test is created to follow the shift from simple integer operations to a multiplication-dominated loop. PET again follows the trend, but overestimates the cost of multiply. Again, the trend is the most important factor. It should also be noted that even tough the loop contains a lot of multiply instructions, the assembled binary still uses a lot of basic integer operations for program flow, storage, registers, addresses, and so on. Note that the shift now happens earlier in PET than in the hardware, the opposite of `Trend`. This is likely due to runtime variatons in gem5 versus real hardware.

Training Set: `SHA-512`

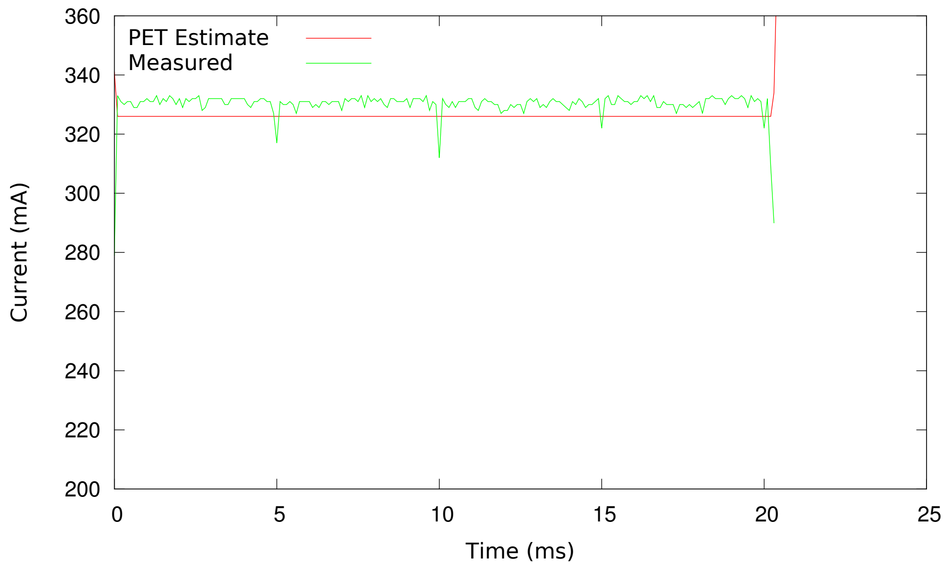


Figure 4.5: Overlay of PET training results (red) and training data (green) for the SHA-512 test.

The last test in the performance tuning algorithm calculates a SHA-512 hash sum. The test results are shown in Figure 4.5 and differs from the other tests by being more memory intensive at the same time as it is an extensive user of both multiply and simpler integer operations. This test is well within 3 % of measured and is yet another example of accurate prediction on general workloads.

Chapter 5
Experiments and Results

We have built PET and trained it to match our reference hardware. Now, a control test is used to verify that the result is good for the general problem, not only the specific instances used for training. This chapter will describe and discuss the test benches used for evaluation as well as the results. It will also discuss challenges in simululation level power estimation.

5.1 Test Environment

PET has already proven that it is able to do the job of power estimation for the training data. The set of workloads has been split in training sets and test sets. This separation is necessary in order to achieve a good result from the genetic algorithm [40, 71].

The test data set consists of a short Dhrystone run and a synthetic add loop, each representing different use of the processor. The Dhrystone test aims to measure overall performance of the general processor, while the add loop will stress the ALUs, leaving most of the other parts of the CPU idle. The Dhrystone test is run for 100 000 iterations. This is too short for benchmarking performance, but it is long enough to measure current drain and short enough to simulate on a simple workstation.

There are sources claiming that gem5 is very accurate [58, 72], but these claims are done with focus on overall performance of long-running benchmarks, not the correctness of the architectural events. This renders a problem for PET, which needs correct architectural events to happen in order to calculate the power profile. Yet another important issue is that the gem5 processor model is not identical, but merely similar to the Exynos 4412. Properties like the fast-loop mode are not implemented in gem5, and parameters for the branch-predictor are not publicly available. Further, this means that some discrepancy between the measurements and PET's prediction is inevitable.

5.2 Power Estimation Challenges

Creating PET has revealed a number of challenges of simulator supported power estimation. The goal of any power estimation tool would be to deliver the correct answer of how much power a certain architecture would use, but many caveats makes this almost impossible with current methods. A non-complete list of things that are important with respect to power consumption, but are hard to get correct in a simulator are described below.

Cache and memory state: The cache and memory systems, together with various cache prefetchers, branch predictors, fast-loop queues and so on will make it difficult to assure that the simulated system are in the exact same state as the physical system.

Interrupts: It is hard to predict when interrupts occur. Interrupts causes context switches, and thus a change in system behavior and power drain.

Undisclosed CPU and system specifications: Unless the whole system is built in-house, there will often be certain specifications that are not available to the tester. PET was built against a CPU where many details were undisclosed; it was not trivial to find sufficient details to configure the simulator.

The above list is by no means a show stopper. However, half-measures must be taken if a non-complete simulator is used. Nevertheless, fruitful results can still be obtained. The best power estimator that could be built around a simulator would be the one that reflected the ideas of the simulated hardware in the best possible way. This estimator would give a hint about final power consumption and its trend over a set of test programs, and would be usable for application specific power optimization.

5.3 Results and Discussion

PET has been benchmarked by simulating the test workloads and comparing measured current drain from the ammeter with the output from PET. This is similar to how fitness is calculated from the training data sets and their corresponding measurements in section 4.3.

5.3.1 Presentation of Test Sets

We now present the two test sets used to evaluate PET performance, `Dhrystone` and `Add`.

Evaluation Set: `Dhrystone`

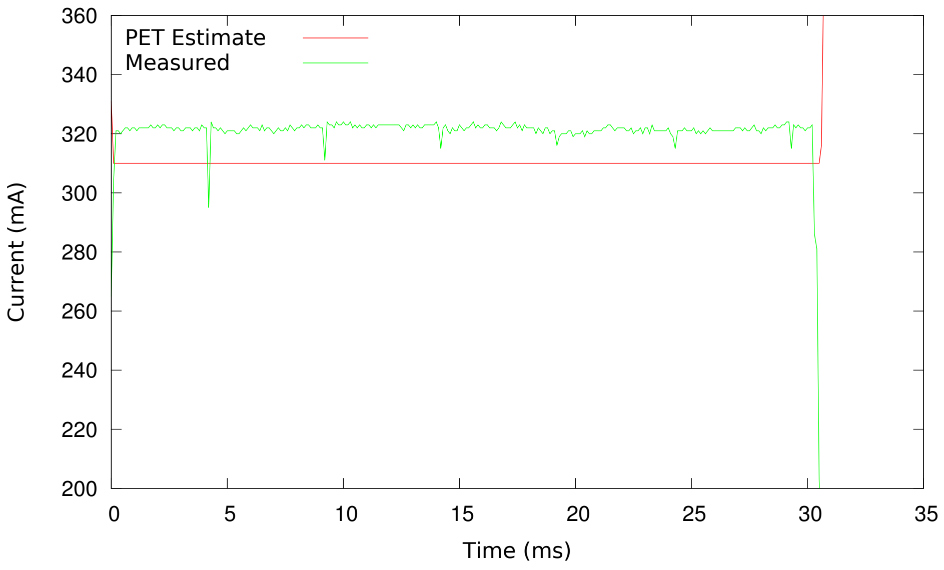


Figure 5.1: Overlay of PET training results (red) and training data (green), `Dhrystone` test.

The `Dhrystone` test results drawn in Figure 5.1 is chosen as one of the final tests for PET as it utilize a wide range of both the integer parts and memory system of the processors. There is a chance that the values are weighted wrongly, but still matches the sum of the real power drain. We claim that decent hits on all four training sets and the Dhrystone test indicates that the GA found a proper set of weight for the ARM Cortex-A9 processor. Please note that the y axis in Figure 5.1 starts at 200 mA, and that the error is less than 3.9 %.

Evaluation Set: `Add`

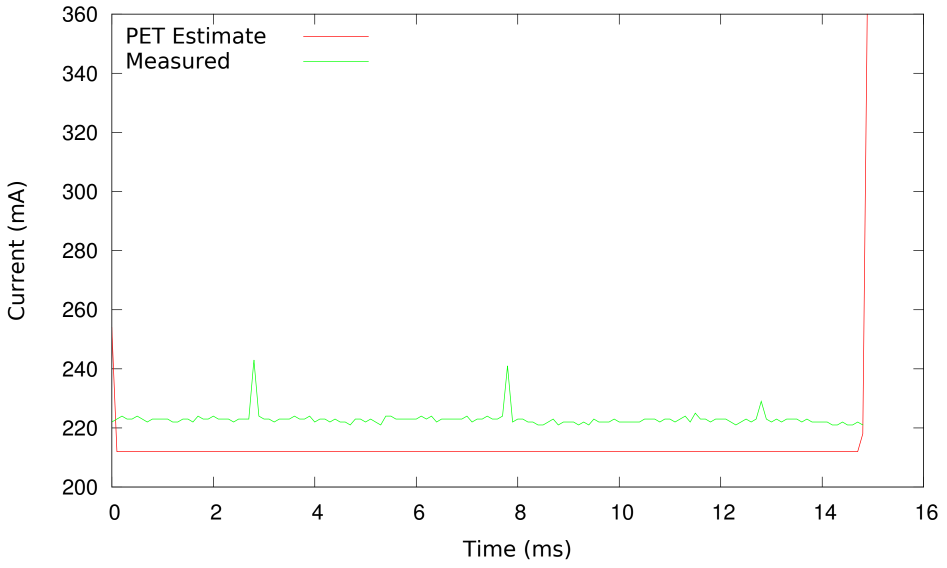


Figure 5.2: Overlay of PET training results (red) and training data (green), `Add` test.

The next test is `Add`, which is displayed in Figure 5.2. It is particularly interesting because it utilized the fast-loop in the Cortex-A9. This implies that it would not need its L1-cache nearly as much as the simulator, as the simulator does not implement fast-loop. However, the real hardware would keep its ALUs more active than the simulated results, thus PET will predict a lower power drain, as it thinks that the caches are in use. This is a problem one must be aware of when the simulator and realized hardware does not completely correspond to each other. However, the main purpose of PET is to estimate changes visible at the simulation level. Thus, this is not a real problem in the ordinary scenarios.

5.3.2 Explanations and Errors

The estimates given by PET is created from a very high level of abstraction, and will never contain enough information to exactly resemble the power profile of the realized chip. However, given that the power consuming events are carefully selected and properly weighted, the current results indicate that this method works satisfactory.

In Figure 5.3, recreated from Figure 8 in [17] which uses exactly the same power measurement setup, it is clear that the measured current drain varies over time seemingly uncorrelated to both on-chip and ambient temperature. Each data point in the figure represents the results from a test run of the `Add` test with about one

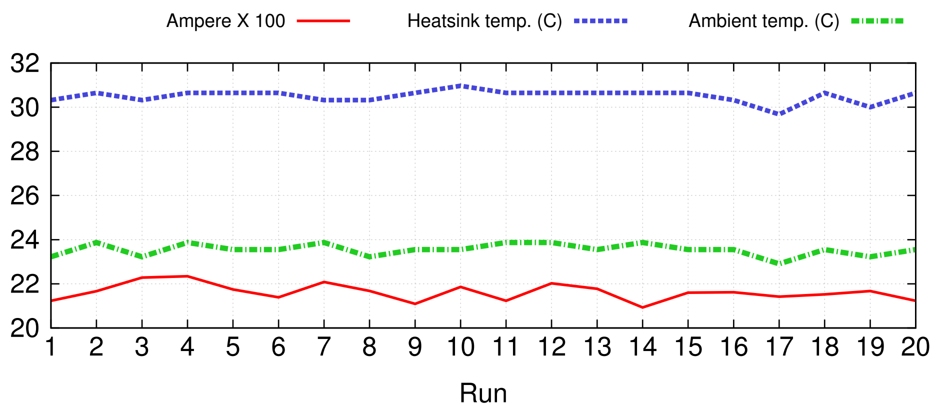


Figure 5.3: Variations in measurements.

hour between each run. The results ranges from 208 mA to 224 mA and averages to 216 mA. The results can be written as:

$$I_{add} = 216\ mA \pm 8\ mA = 216\ mA \pm 3.8\ \%$$

With a measured error of ± 3.8 % it is not unreasonable to expect a 7.6 % error in all measurements. This means that the weights found using measurements and a GA might be wrong since each training set might have slight discrepancies between each other. This again renders it impossible to get correct results. By chance, some genome could fit all training set even though it contains errors, but the weights would most likely be unequal to the "correct" weights.

Most search algorithms are prone to a phenomenon called overfitting [40]. Overfitting happens because the genetic algorithms will find any kind of obscure patterns in the data sets, e.g., if a high power drain was seen randomly, but a specific event often happens at that particular time, the algorithm would try to blame the event for it.

It is clear from Table 4.1 that the CPU model used in gem5 is not exactly equal to the Cortex-A9 core used in the Samsung Exynos 4412 Prime SoC, thus each graph in both training and results are stretched to match each other. This is certainly a source of error, but from `Trend` in Figure 4.3 and `SubMul` in Figure 4.4 it is reasonable to believe that the scaling works, as the change in program flow is shown not far from each other in the predicted graph and real measurement graph.

5.3.3 PET Processing Performance

To test the performance of PET, we ran a simple benchmark on a system consisting of an Intel Core i7 4820, 32 GB DDR3 SDRAM and reading trace log files from a software RAID Level-0 consisting of two Western Digital Caviar Black 750 GB disks.

The results shows that reading the trace log file is not the bottleneck, and that the program itself is CPU-bound. It is easy to feed at least 8 cores when the log file is hosted on a reasonable fast drive. PET running with 8 threads on this particular system is consuming log files at a rate of 133 MB/s, regardless of whether the log files resides in RAM or on disk. The benchmark used a log file of 5458 MB and took 40.871 seconds to process.

Chapter 6 Conclusion and Further Work

In this thesis, we have taken advantage of hardware measurements on real hardware and an architectural simulator to create a power estimation tool. The following sections will highlight our contribution, discuss use cases and relate it to existing solutions. We will also provide a review of interesting topics for further work.

6.1 Conclusion

PET, a power estimation tool, is a software tool for estimating power consumption on existing as well as non-existing computer architectures. It uses output from gem5 together with a set of weighted parameters to estimate energy consumption of a program running on a given hardware model. The weighted parameters are selected by investigating the pipeline of an ARM Cortex-A9 processor. We have run a set of workloads on the hardware platform and logged their current drain over time. Further, the results were used as input for a genetic algorithm that mapped the correct energy usage to each architectural event in the simulator.

PET is not designed to be as accurate as possible, but to assist hardware developers as early in the design stage as possible. As opposed to classical methods, PET can be applied to a design already when only a simulation model exists. Well known tools such as Wattch [30] or McPAT [25, 26] also utilizes a simulator, but requires more knowledge about the final hardware, e.g., RTL and process technology. Regardless of such information, PET is able to estimate current drain with a margin of error within 5 % when testing against the ARM Cortex-A9 processor.

Because PET is a tool meant to be used early and rapidly in the design phase, it has to be fast and easy to use. PET will predict power usage from log files and is tested to evaluate about 133 MB of log files per second on a commodity computer. Even with log files expanding tens of gigabytes, running PET takes less time than running a low-level power estimator.

An excessive amount of time has been used for tweaking PET, gem5 and the genetic algorithm to match our reference hardware platform as precisely as possible. Still, we believe that the effort needed to port our methods to a new hardware platform or architecture is much less. The genetic algorithm was in our case able to find good weights within few hours, and with carefully selected power consuming events it is likely that this is the same for other architectures. Unrealized hardware still needs to derive weights from similar hardware.

Our observation is that PET allows evaluation of the big picture easier and earlier in the design stage than existing solutions, simply because it estimates power with less hardware details. We hope that PET will be useful when developing both tiles for SHMAC architecture and processors in general.

All in all, using PET or other tools built from the same concept of weighting architectural events is possible for a set of scenarios. How exact the model is will depend on the simulator tuning and the genetic algorithm used to match ammeter measurements. The process of settings weights for PET seems cumbersome, but for most practical settings the most important thing is to have the weights reasonably proportioned among themselves.

6.2 Further Work

PET has proven to work quite well with the ARM Cortex-A9, but no effort has been put into verifying that the concept works for multiple architectures. Testing against other both realized and non-realized processors is yet to be done. An other important section where PET could prove useful, is in the development of memory hierarchies. PET has not been tested very well for this purpose as it is not clear from the provided data sheets how the main memory is powered on the ODROID-X2 platform. A similar experiment on fully disclosed hardware would most likely provide much more accurate information for the idea of event based power consumption prediction. An interesting experiment would be to compare the accuracy of PET to tools that work on the RTL-level, even though the major purpose of PET is rough estimation rather than correct predictions.

When it comes to the SHMAC project, an important piece of work will be to use PET and its ideas early in the design phase. Taking advantage of this can can shorten the path to more energy efficient tiles.

References

[1] Gordon E Moore et al. Cramming More Components onto Integrated Circuits, 1965.

[2] Andrew S Tanenbaum. *Structured Computer Organization*. Prentice Hall PTR, 1984.

[3] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of Ion-implanted MOSFET's with Very Small Physical Dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[4] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[5] Chuck Moore. Data Processing in Exascale-class Computer Systems. http://www.lanl.gov/orgs/hpc/salishan/salishan2011/3moore.pdf, 2014.

[6] John L. Hennessy David A. Patterson. *Computer Organization and Design*. Morgan Kaufmann, 4 edition, 2012.

[7] Nikola Rajovic, Paul M Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with Commodity CPUs: are Mobile SoCs Ready for HPC? In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 40. ACM, 2013.

[8] M Bruno, A Macii, and M Poncino. RTL Power Estimation in an HDL-Based Design Flow. In *Computers and Digital Techniques, IEE Proceedings-*, volume 152, pages 723–730. IET, 2005.

[9] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. AccuPower: An Accurate Power Estimation Tool for Superscalar Microprocessors. In *Proceedings of the conference on Design, automation and test in Europe*, page 124. IEEE Computer Society, 2002.

[10] Jahre, Magnus. The Single-ISA Heterogeneous MAny-core Computer (SHMAC). http://www.ntnu.edu/ime/eecs/shmac, 2014.

[11] Wayne Wolf. *Computers as Components*. Morgan Kaufmann, 2 edition, 2008.

[12] David Nguyen, Abhijit Davare, Michael Orshansky, David Chinnery, Brandon Thompson, and Kurt Keutzer. Minimization of Dynamic and Static Power Through Joint Assignment of Threshold Voltages and Sizing Optimization. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 158–163. ACM, 2003.

[13] Nam Sung Kim, Todd Austin, D Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage Current: Moore's Law Meets Static Power. *Computer*, 36(12):68–75, 2003.

[14] Steven M Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Lower Power Microprocessors under Dynamic Workloads. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 721–725. ACM, 2002.

[15] Spiridon Nikolaidis, Nikolaos Kavvadias, Theodore Laopoulos, Labros Bisdounis, and Spyros Blionas. Instruction Level Energy Modeling for Pipelined Processors. *Journal of Embedded Computing*, 1(3):317–324, 2005.

[16] Harry Nyquist. Certain Topics in Telegraph Transmission Theory. *American Institute of Electrical Engineers, Transactions of the*, 47(2):617–644, 1928.

[17] Terje Runde and Stian Hvatum. Exploring Instruction Level Energy Efficiency. 2013.

[18] The ARM Cortex-A9 Processors. http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf, 2013.

[19] Inc. Agilent Technologies. Agilent 34410A and 34411A multimeters data sheet. http://www.home.agilent.com/en/pd-692834-pn-34410A/digital-multimeter-6-digit-high-performance, April 2013.

[20] Hardkernel. ODROID-X2. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G135235611947, 2014.

[21] Samsung Electronics Co., Ltd. Samsung Exynos 4 Quad. http://www.samsung.com/global/business/semiconductor/minisite/Exynos/products4quad.html, June 2014.

[22] Trevor E. Carlson, Wim Heirman, Lieven Eeckhout, Kenzo Van Craeynest, and Ibrahim Hur. The Sniper Multi-Core Simulator. http://snipersim.org/w/The_Sniper_Multi-Core_Simulator.

[23] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. 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*, page 52. ACM, 2011.

[24] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sampled Simulation of Multi-Threaded Applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, April 2013.

[25] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[26] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):5, 2013.

[27] Doug Burger, Todd M Austin, and Steve Bennett. *Evaluating Future Microprocessors: The Simplescalar Tool Set.* University of Wisconsin-Madison, Computer Sciences Department, 1996.

[28] Naraig Manjikian. Multiprocessor Enhancements of the SimpleScalar Tool Set. *ACM SIGARCH Computer Architecture News*, 29(1):8–15, 2001.

[29] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[30] David Brooks, Vivek Tiwari, and Margaret Martonosi. *Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*, volume 28. ACM, 2000.

[31] Daniel Bartholomew. QEMU: a Multihost Multitarget Emulator. *Linux Journal*, 2006(145):3, 2006.

[32] Shye-Tzeng Shen, Shin-Ying Lee, and Chung-Ho Chen. Full System Simulation with QEMU: An Approach to Multi-View 3D GPU Design. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3877–3880. IEEE, 2010.

[33] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[34] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. The M5 simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[35] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[36] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[37] The gem5-dev mail archive. http://www.mail-archive.com/gem5-dev@gem5.org/.

[38] The gem5 Simulator System. http://gem5.org/Main_Page.

[39] Ali Saidi and Andreas Hansson. Simulating Systems not Benchmarks, HiPEAC 2012. http://gem5.org/dist/tutorials/hipeac2012/gem5_hipeac.pdf.

[40] Stuart Russell and Peter Norvig. *Artificial Intelligence.* Pearson, third edition edition, 2010.

[41] Claus Wilke. File:Fitness-landscape-cartoon.png. http://en.wikipedia.org/wiki/File:Fitness-landscape-cartoon.png, June 2005.

[42] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A Field Guide to Genetic Programming.* Lulu. com, 2008.

[43] Sanford Weisberg. *Applied Linear Regression*, volume 528. John Wiley & Sons, 2005.

[44] David C. Lay. *Linear Algebra and Its Applications.* Pearson International, 2011.

[45] Raymond H Myers. *Classical and Modern Regression with Applications*, volume 2. Duxbury Press Belmont, CA, 1990.

[46] Peter JM Van Laarhoven and Emile HL Aarts. *Simulated Annealing.* Springer, 1987.

[47] David B Fogel. Evolutionary Algorithms in Theory and Practice, 1997.

[48] Marek Obitko. Introduction to Genetic Algorithms. http://www.obitko.com/tutorials/genetic-algorithms/.

[49] Jahre, Magnus. SHMAC: An Infrastructure for Heterogeneous Computing Systems Research. http://www.ntnu.edu/documents/139931/80945963/eecs-shmac-hipeac-csw.pdf, 2013.

[50] Leif Tore Rusten and Gunnar Inge Sortland. Implementing a Heterogeneous Multi-Core Prototype in an FPGA, 2012.

[51] The Boost Community. boost C++ Libraries. http://www.boost.org/, May 2013.

[52] William J Song, Sudhakar Yalamanchili, Arun F Rodrigues, and Saibal Mukhopadhyay. Instruction-based Energy Estimation Methodology for Asymmetric Manycore Processor Simulations. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, pages 166–171. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2012.

[53] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[54] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[55] Tim Blechmann. Chatper 18. Boost.Lockfree. http://www.boost.org/doc/libs/1_55_0/doc/html/lockfree.html, 2011.

[56] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C Programming Language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[57] The Boost Community. Boost Test: Introduction. http://www.boost.org/doc/libs/1_55_0/libs/test/doc/html/intro.html, 2007.

[58] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. Accuracy Evaluation of GEM5 Simulator System. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–7. IEEE, 2012.

[59] Anthony Gutierrez. Re: [gem5-users] Cortex-A15 Simulation. https://www.mail-archive.com/gem5-users@gem5.org/msg05670.html, October 2012.

[60] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. A Detailed Analysis of Contemporary ARM and x86 Architectures. *Report, UW-Madison Technical*, 2013.

[61] ARM. Cortex-A9 Technical Reference Manual revision r3p0. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388g/DDI0388G_cortex_a9_r3p0_trm.pdf, 2014.

[62] Wikimedia Foundation. Exynos @ Wikipedia. http://en.wikipedia.org/wiki/Exynos, June 2014.

[63] Wikimedia Foundation. ODROID @ Wikipedia. http://en.wikipedia.org/wiki/Odroid, 2014.

[64] Geek Land USA. Geekland. http://www.geekland.co/ARM-Cortex-A9-Exynos-Quad-Core-4412-Android-Development-Board-GK-4412P.htm, 2014.

[65] 7-CPU.com. ARM Cortex-A9 @ 7-CPU. http://www.7-cpu.com/cpu/Cortex-A9.html, 2014.

[66] ARM Holdings plc. Cortex-A9 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a9.php?tab=Specifications, 2014.

[67] Jeff Gibson, Robert Kunz, David Ofelt, Mark Horowitz, John Hennessy, and Mark Heinrich. FLASH vs.(simulated) FLASH: Closing the simulation loop. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 49–58. ACM, 2000.

[68] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary Algorithms Made Easy . *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[69] Kalyanmoy Deb. Multi-objective Optimization. In *Search methodologies*, pages 403–449. Springer, 2014.

[70] Aaron D. Gifford. Secure Hash Algorithm (SHA). http://www.aarongifford.com/computers/sha.html, 2014.

[71] Karmen Rajer-Kanduč, Jure Zupan, and Nineta Majcen. Separation of Data on the Training and Test Set for Modelling: A Case Study for Modelling of Five Colour Properties of a White Pigment. *Chemometrics and intelligent laboratory systems*, 65(2):221–229, 2003.

[72] Anthony Gutierrez Joseph Pusdesris, Ronald G Dreslinski Trevor Mudge, Chander Sudanthi Christopher D Emmons, and Mitchell Hayenga Nigel Paver. Sources of Error in Full-System Simulation. 2014.

[73] Randall Munroe. XKCD: Ohm. http://xkcd.com/643/.

Appendix A
TDT4501: Exploring Instruction Level Energy Efficiency

In the pages to follow, the paper from our fall specialization project (TDT4501) is attached. Raw data and source code used to create that project can be found at github.com and can be cloned with:

```
git clone https://github.com/terjr/arm-project.git
```

The fall project solved part 1 of the problem as described in the problem description:

> *Investigate the power/energy consumption of simple benchmark programs on real hardware, i.e., create benchmark programs and evaluate performance by measurements.*

Hardware and test setup is the same as used in the thesis.

Exploring Instruction Level Energy Efficiency

Terje Runde and Stian Hvatum
TDT4501 - Computer Science, Specialization Project
Department of Computer Science, NTNU

{terjr,hvatum}@stud.ntnu.no

***Abstract*—Modern microprocessors are limited by power density and new designs must emphasize energy efficiency to become successful. Building energy efficient hardware requires better understanding of how the ISA and its implementation relates to energy efficiency. This paper investigates the instruction level energy efficiency of an ARM Cortex-A9 and presents current drain and pipeline utilization for different instructions. These numbers reveal information about how instructions are executed and how energy efficient their implementations are in this proprietary architecture.**

The testbench used in the experiments relies on accurate energy measurements. This is achieved by measuring voltage drop over a shunt resistor placed between the CPU core voltage supply pins and an external power supply. Memory usage is avoided by exploiting instruction buffers and omitting loads/stores. This renders the memory hierarchy unused, isolating the core as much as possible.

The results shows that the ARM ISA is well balanced. Commonly used instructions such as `add`, `sub` and `mul` seems to be energy efficient. Unfortunately, the ISA suffers from an inefficient handling of conditional execution and status flag updates. Such instructions seems to force synchronization, which then leads to inefficient utilization of the otherwise computationally strong core.

***Index Terms*—Energy efficiency, microprocessor chips, performance analysis**

I. Introduction

MAKING processors burn less energy while at the same time increasing performance is currently one of the greatest challenges hardware designers are facing. Performance alone can be improved by cramming more components onto integrated circuits [1] utilizing new process technologies. However, due to the end of Dennard scaling [2], power density on chip will increase linearly with transistor count. Computer designers are forced to employ novel techniques mitigating this issue, such as shutting down parts of the chip [3] and designing specialized hardware.

As processors grow more sophisticated, it becomes harder to reason about their energy efficiency. Even RISC processors, which traditionally were designed to be simple[4], have seen a steep increase in complexity during the last decade[5]. Features that previously only existed in CISC processors are now entering the RISC domain; more complex operations are done per clock cycle. Current RISC designs may have deep pipelines, increased component complexity, advanced branch predictor units and a high degree of instruction level parallelism. They include features that aims to reduce energy consumption and increase throughput in return for added complexity. Moreover, processors are increasingly designed to integrate seamlessly with external components such as accelerators and the memory system.

The need for energy efficient processors is increasing. To better understand how execution of different instructions contribute to energy consumption we propose a method to measure energy efficiency at the instruction level of a processor. Being able to model and monitor energy consumption has recently got the industry's attention: Semiconductor companies are making software and hardware targeting the embedded market providing a monitor for energy consumption on-chip, giving application engineers the opportunity to optimize for energy.

With the emergence of performance counters, it is now possible to detect how different pipeline stages affects the overall power consumption. Previous research correlates power drain seen from the wall outlet with performance counters on the CPU[6]–[8]. Others look at energy usage under different workloads[9].

In this paper, we analyze the instruction level energy efficiency of a modern RISC architecture. We isolate as many architecture components as possible by correlating performance counters with observed current drain for specific instructions. We also show that it is feasible to get a per instruction energy overview of an existing architecture by understanding the hardware and writing benchmark programs. We look at simple single-cycle instructions as well as the most complex instructions using multiple cycles on our target processor. When each relevant instruction has been measured, we compare their normalized energy consumption and discuss properties of different instructions.

This work is motivated by and related to the SHMAC project[10]–[12] at NTNU. The goal of the SHMAC project is to build an energy efficient heterogeneous many-core computer architecture: multiple processing cores with different capabilities share a single ISA and can be put on the same die to create a processor tailored for a specific application. Exploring instruction level energy efficiency gives us a better view on how different ISA implementations perform at different tasks[13]. This information can be useful in the design phase of novel computer architectures such as SHMAC.

Another use for this kind of information is that compilers can optimize code for energy efficiency and not only performance. Further, it can enable simulators to estimate energy consumption of a given program and even compare energy efficiency on different cores, as shown to be effective in [13].

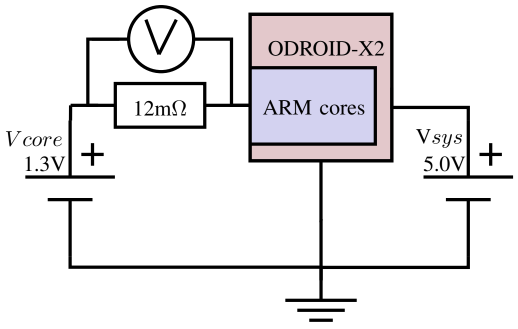


Fig. 1: Experiment setup

Manufacturer	Hardkernel
Platform	ODROID-X2
SoC	Samsung Exynos 4412 ”Prime”
CPU Core	ARM Cortex-A9 (r3p0)
Number of Cores	4
Clock Freq.	1.7 GHz
Core Voltage	1.3V
OS	Debian GNU/Linux Testing (“jessie”)
Kernel	Linux 3.8 (custom)
Voltmeter	Agilent 34410A
Power Supply	Agilent E3631A
Shunt Resistor	Thermovolt AB 5697 0002 12mΩ

TABLE I: System specifications

II. Methodology

A. Test Environment

In our experiments, we are using the ODROID-X2 [14] developer platform, which has an Exynos 4412 ”Prime” System-on-Chip with four ARM Cortex-A9 processor cores. We disable three of the cores through *sysfs*, leaving only one core available to the scheduler. The processor runs at a fixed frequency of 1.7 GHz. The test environment is sketched in Figure 1 and the details are summarized in Table I.

The Cortex-A9 is a 32-bit out-of-order dual-issue speculative RISC processor, and even though its primary use is in mobile and embedded applications, it shares many features with current desktop processors [15], [16]. It can issue two instructions per cycle and branches its pipeline into four lanes, as depicted in Figure 2. Most instructions can execute in either of the two general ALU’s, but multiply instructions must execute in the ALU with a hardware multiplier. The processor core also has separate units for floating point operations (the NEON co-processor) and address manipulation, but these will not be further considered in this paper.

In general, energy consumption of a processor varies with respect to the workload; the harder it has to work, the more energy it uses. In this paper, we seek to achieve the highest possible ALU throughput the processor can offer. To accomplish this, we are required to gain knowledge of the pipeline and other components within the CPU.

Official documentation of the pipeline structure is limited to the “Cortex-A9 Technical Reference Manual” [18] and “The ARM Cortex-A9 Processors” whitepaper [17]. However, by

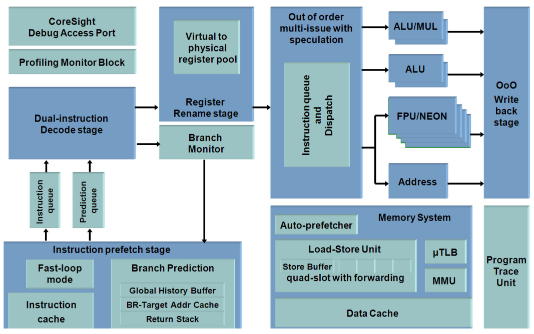


Fig. 2: ARM Cortex-A9 pipeline and peripherals[17]

running some architectural experiments and consulting the performance counters we are able to infer some details.

B. Architectural Experiments

The A9 processor has 58 distinct events[1] that each can be mapped to one of six generic event counters in the Performance Monitor Unit (i.e. only six generic events can be tracked simultaneously). It also has a separate cycle counter. By comparing execution unit counters for the two ALUs and the cycle counter, we obtain detailed statistics about the pipeline activity. For example, we run `add` instructions with and without hazards, and verify that the core is able to share the work between execution units in the latter case. Note that these performance counters are approximate due to the speculative core, and only serves as a guideline and sanity check for our assumptions.

Using performance counters as above, we are able to confirm a feature on the A9 processor that is very vaguely documented; fast-loop mode. As the name suggests, this feature enables rapid execution of small loops. It does so by fetching instructions from the instruction cache only at the first loop iteration, effectively voiding time and energy spent on instruction lookups between iterations. However, which loops that falls into this category is not documented, but by using performance counters we are able to determine this with confidence. We disable the L1 cache, penalizing runs that do not fit in fast-loop, making it easy to distinguish between runs within and outside fast-loop. We find that for a loop to be executed in fast-loop it must hold one subtle property: the loop in its entirety must fit within the first 60 bytes of a 64 byte cache line. Consequently, the loop body must be 13 instructions or less (15 including `sub` and `bne`). Loops without this property will cause code to be executed outside fast-loop and get a significant decrease in performance.

Furthermore, executing code within fast-loop limits the number of cache mispredicts to two, independent of the iteration count. We confirm this by looking at the cache mispredict performance counter.

[1]A complete overview can be seen in table A.18 in [18]

```
    label:
    instruction
    ... ; repeats 13X
    instruction
    subs
    bne label
```

Fig. 3: Instruction loop

C. Benchmarks

As a first approximation, the benchmark programs consists of an infinite series of identical instructions. Since the A9 core runs at a fixed frequency and we are providing a fixed core voltage, so energy usage per instruction can be deduced from the continuous power drain during instruction runs. In general, the power drain related to a particular instruction can be calculated as following:

$$P_{instruction} = I_{instruction} \cdot V_{core} \tag{1}$$

The fixed core clock cycle gives a fixed amount of time per cycle. Thus, we use clock cycles as our base time unit. Energy is then given as

$$E_{instruction} = P_{instruction} \cdot cycles \tag{2}$$

We are unable to measure core power directly with our equipment. However, voltage and frequency are assumed constant, which gives a linear relation between current and consumed energy. Instead of providing numbers in Joule per instruction, we measure the current drain and multiply with the number of cycles the instruction lies within any of the processors functional units. This gives us the unit of Ampere-cycles, which in our environment maps directly to energy. We neglect the voltage drop over the shunt resistor, which is in the order of a few mV.

This simple setup does not take the memory system into account; we are undoubtedly not able to feed the processor instructions at no cost in terms of access speed and – more importantly – memory system energy usage. Thus, we enhance our setup by running all benchmark code within fast-loop. To explicitly feed the processor instructions without the overhead of interrupts, we write Linux kernel modules that once inserted, execute a loop similar to the one shown in Figure 3. Note that the `subs` and `bne` are used to generate a loop body small enough to fit in fast-loop, and at the same time allows us to terminate the program after some number of iterations.

It is stated in [18] that branching to immediate locations does not use clock cycles. Our micro-benchmarks branches to immediate locations, but it does so conditionally. We assume that the calculation of the branch condition, the `subs`, takes its normal execution time, while the following branch instruction is invisible.

D. Power Measurements

To measure energy consumption, we use an Agilent 34410A multimeter[19] to sense the voltage drop over a shunt resistor, set up as shown in Figure 1. The multimeter is configured to sample at its highest sampling rate of 10 kHz. This yields one sample every 170,000 instructions with an error of at most $1mV$. It is obvious that we are unable to observe inter-cycle fluctuations with this equipment, but as we run the same instruction practically indefinitely we extract the average. The loop for each instruction runs for about 20 seconds and we gather 50,000 samples (over a period of 5 seconds) in the middle of this loop. Observational errors are accounted for by running the power measurement loop many times for each instruction. We also sleep 30 seconds in between runs to dilute the effect of temperature variations. Running over the entire testbench takes about 100 minutes and we average the medians for each instruction run to get a single value.

We separate power consumption on the ARM cores and the development board by modifying the ODROID-X2 and providing a separate power supply for the A9 cores. They get powered by an external power supply giving $1.3V$ DC, while the rest of the board is powered from a another power supply at $5.0V$, as depicted in Figure 1. We cannot verify that CPU cores sit alone on the $1.3V$ power rail, but we observe a strong degree of correlation between core activity and V_{core} power drain.

Certain instructions use more than one cycle to complete their work, so the energy usage has to be normalized. An instruction that occupies the pipeline for two cycles is believed to use approximately twice as much energy. By normalizing, we can convert point-in-time current drain in terms of Amperes to energy per instruction in Ampere-cycles.

E. Pitfalls

We measure the current drain of different instructions separately, so we need to fix as many parameters as possible. We must acknowledge that some factors affects power consumption and produces noise in our data.

One obvious such factor is the chip temperature: it is known that power consumption increases at higher core temperatures. We explore the boundaries by physically applying cooling spray and notice that our measurements on average gets 4% higher with a temperature increase from 9°C to 63°C . The `mul` instruction had the greatest leap and used 7% more energy at 63°C. In our experiments, only one of the four available cores are used. Stressing a single core over time did not increase temperature by more than 7°C (from idle at 47°C to 54°C at load) and reached an equilibrium where temperature remained constant. Assuming that it is generally true that a single core cannot heat the entire SoC significantly, and that the increase in power consumption is at most 10% over 50°C, we get

$$P_{inc} = P_{orig} \cdot T_{inc} \cdot \frac{0.10}{50} = P_{orig} \cdot T_{inc} \cdot 0.002 \tag{3}$$

Assuming the trend is close to linear, output will increase by 0.2% per °C increased. Also, we start our measurements several seconds after the benchmarks, giving the core plenty of time to reach work temperature. For our purpose, the time used to reach work temperature was pretty much instant. Note that this temperature logging was done with a different kernel as it required support for Dynamic Voltage and Frequency Scaling (DVFS), which we disabled in the test setup in order to fix the clock frequency.

Energy consumption is almost certainly affected by the amount of bit flipping within the core. In all the tests, the instruction arguments are static. This means that the results could be different if we changed the arguments. To mitigate this, we used as equal arguments as possible. Still, different instructions contain and use arguments differently, so we cannot guarantee complete fairness between instructions.

We are running Linux as the base environment for our tests, which makes it simpler to run our micro-benchmarks. However, running an entire operating system beneath our benchmark programs implies that there is much going on where we have no direct control. To mitigate the artifacts originating from the operating system, we disable all the maskable interrupts and run our benchmark programs entirely uninterrupted as a kernel module.

As explained in section subsection II-B, we utilize the fast-loop mode of the processor to avoid memory access latency. We disable the L1 cache to easier detect when we are outside the fast-loop mode, and thus we are certain that there is no memory access going on.

III. Results

A. Introduction

In this section we present data gathered from our experiments on the ARM Cortex-A9. A brief description of each instruction can be found in the ”ARM and Thumb-2 Instruction Set Quick Reference Card”[20]. First, we discuss performance counters from experimental testbench runs. Together with the sparse official documentation, it enables us to make some assumptions about how different instructions are executed in the processor. We then discuss the results from the per instruction energy analysis.

B. Decomposing the Core

Instructions executed in the processor will utilize a subset of all the available core components. By combining the components depicted in Figure 2 with the performance counter data listed in Table II and Table III, we can deduce which instructions that trigger what parts. We can also see how frequently each part of the pipeline is used, as a fraction of cycle count and the given component event counters.

All results in Table II and Table III are gathered by running each instruction included in our experiments using the template shown in Figure 3. The cycle count (*Cycles*) tells us how long time, in terms of clock cycles, it took for the processor to execute the $252 \cdot (13 + 2) = 3780$ instructions. The loop has room for 13 test instructions, while the last 2 is the loop head consisting of `subs` and `bne`. *Main Ex.* is the number of cycles where the main execution pipeline is active, labeled ALU/MUL in Figure 2. *Second Ex.* is for the second execution pipeline, labeled ALU. All instructions in our test bench have a correct branch prediction count of 251 (*Pred.*). This is most likely because the first and the last iteration of

Instr.	Cycles	Main Ex.	Second Ex.	Pred.	Mis pred.	No disp.	Issue Empty
adc	1976	1762	1758	251	2	89	89
adcs	3599	1762	1756	251	2	1693	1690
add	1976	1762	1758	251	2	89	89
addeq	6594	1635	1883	251	2	4709	4709
addne	3349	1762	1758	251	2	1463	1463
adds	3598	1761	1757	251	2	1712	1712
and	1976	1762	1758	251	2	89	89
ands	3599	1762	1756	251	2	1693	1690
asr	1976	1762	1758	251	2	89	89
asrs	6361	2135	1385	251	2	2968	204
bic	1976	1762	1758	251	2	89	89
bics	3599	1762	1756	251	2	1693	1690
clz	2104	1824	1699	251	2	151	88
cmn	3599	1761	1757	251	2	1713	1713
cmp	3598	1761	1757	251	2	1712	1712
cpsid	14627	3516	1	251	2	11110	11110
eor	1976	1762	1758	251	2	89	89
eors	3600	1762	1756	251	2	1694	1691
lsl	1976	1762	1758	251	2	89	89
lsls	6361	2135	1385	251	2	2968	204
lsr	1976	1762	1758	251	2	89	89
lsrs	6362	2135	1385	251	2	2969	205
mov	1976	1762	1758	251	2	89	89
movs	3599	1762	1756	251	2	1693	1690
mvn	1976	1762	1758	251	2	89	89
mvns	3600	1762	1756	251	2	1694	1691
nop	3604	3268	251	251	2	84	84
orr	1976	1762	1758	251	2	89	89
orrs	3599	1762	1756	251	2	1693	1690
pkhbt	1976	1762	1758	251	2	89	89
pkhtb	1976	1762	1758	251	2	89	89
qadd	3598	1761	1757	251	2	1712	1712
qdadd	3600	1885	1633	251	2	1712	1588
qdsub	3600	1885	1633	251	2	1712	1588
qsub	3598	1761	1757	251	2	1712	1712
rev16	2104	1824	1699	251	2	151	88
rev	2104	1824	1699	251	2	151	88
revsh	2105	1824	1699	251	2	152	89
ror	1976	1762	1758	251	2	89	89
rors	6361	2135	1385	251	2	2968	204
rrx	1976	1762	1758	251	2	89	89
rrxs	3599	1762	1756	251	2	1693	1690
rsb	1977	1762	1758	251	2	90	90
rsbs	3598	1761	1757	251	2	1712	1712
rsc	1976	1762	1758	251	2	89	89
rscs	3599	1762	1756	251	2	1693	1690
sbc	1976	1762	1758	251	2	89	89
sbcs	3599	1762	1756	251	2	1693	1690
sel	2100	1761	1758	251	2	337	89
setend	14627	3516	1	251	2	11110	11110
ssat16	3598	1761	1757	251	2	1712	1712
ssat	3598	1761	1757	251	2	1712	1712
sub	1977	1762	1758	251	2	90	90
subs	3598	1761	1757	251	2	1712	1712
sxtab16	6443	3021	3761	251	2	1832	77
sxtab	4481	2932	2344	251	2	666	81
sxtah	6443	3021	3761	251	2	1832	77
sxtb16	2104	1824	1699	251	2	151	88
sxtb	2104	1824	1699	251	2	151	88
sxth	2104	1824	1699	251	2	151	88
teq	3599	1762	1756	251	2	1693	1690
tst	3600	1762	1756	251	2	1694	1691
usat16	3598	1761	1757	251	2	1712	1712
usat	3598	1761	1757	251	2	1712	1712
uxtab16	6443	3021	3761	251	2	1832	77
uxtab	6443	3021	3761	251	2	1832	77
uxtah	6443	3021	3761	251	2	1832	77
uxtb16	2105	1824	1699	251	2	152	89
uxtb	2104	1824	1699	251	2	151	88
uxth	2104	1824	1699	251	2	151	88

TABLE II: Performance counter data from 252 iterations of all tested instructions, excluding multiply

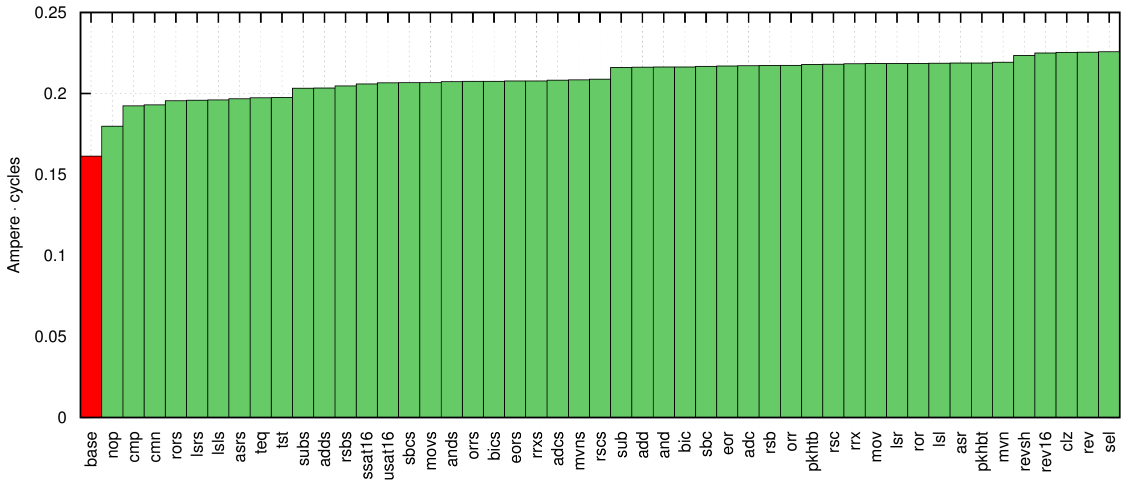


Fig. 4: Energy profile of single-cycle instructions, excluding multiply

the loop is mispredicted (*Mis pred.*). *No disp.* is the number of cycles where there the processor was unable to dispatch any instructions to any execution lane. *Issue Empty* is the number of cycles where there was no instructions in the instruction queue. Note that we can see how many cycles the processor is stalling by looking at the *No disp.* and the *Issue Empty* counters. When the *No disp.* number is higher than *Issue Empty*, it means that the processor had to stall due to hazards or intended flushing (e.g. `setend` flushes the pipeline as all further issued instructions must follow the new endianess). In special cases, such as our baseline instruction `setend`, we see that the amount of *No disp.* is very high, which again means that the CPU is mostly stalling. It seems to be a strong relation between low power usage and high stall numbers.

C. *Instruction Level Energy Efficiency*

We distinguish between single-cycle instructions and multi-cycle instructions because they behave differently in and around the execution pipelines. Instructions using only one cycle are fairly easy to reason about as there is no need to normalize energy consumption with respect to the cycle count (i.e. time). However, it is important to also recognize CPU capabilities such as dual issuing which are present on the processor: most single-cycle ALU instructions execute pairwise in parallel – one in each ALU – giving a peak performance of two instructions per clock cycle. Multi-cycle instructions needs to be carefully considered. Typically, multi-cycle instructions divide work which can be done in a subset of the available ALUs (e.g. one) over several cycles, and can therefore introduce bottlenecks in the execution path. This again makes the processor do less, lowering the average current drain. For all these reasons, we partition the measured data in two data sets; one for single-cycle instructions and one for multi-cycle instructions.

Figure 4, 6, 7 and 5 displays our results from measuring current drain for each instruction. The instructions is sorted in increasing order by Ampere cycles. Green bars represents single-cycle instructions, light blue are two-cycle instructions, while dark blue represents three-cycle instructions. The red bar to the left on each graph shows the baseline for current measurement. The baseline is an alias for the least power-consuming instruction we could find, which is the `setend`-instruction. This instruction sets the endianness for all memory operations to either big or little endian [21], and has a current drain of only $161.3mA$ when executed repeatedly. This is expected because it would force pipelines to be empty most of the time.

During measurements, V_{core} was kept stable at $1.3V \pm 50mV$, well within the specifications of the processor. The pipelines were kept as full as possible, avoiding hazards and instruction loading. This implies that instructions utilizing large parts of the processor will most likely be more energy consuming than those using only few components. This statement is supported by the fact that the `setend`-instruction has little pipeline activity at the same time as it has a low continuous current drain.

D. *Single-Cycle Instructions*

On our target CPU, 70 of the 115 tested instructions[2] use a single cycle, while the remaining 45 uses 2 or 3. Nearly half of the instructions are multiply instructions, so these will be discussed separately. Figure 4 displays a comparison of the 50 non-multiply single-cycle instructions.

The results in Figure 4 shows that the ordinary single-cycle instructions do not differ very much. An interesting result is how instructions bearing the `s`-flag seems to have a lower consumption than their non-`s` companion. These instructions updates status flags and will likely force in-order execution. According to the performance counters in Table II there is reason to believe that the processor has to stall one cycle

[2]119 including conditionals

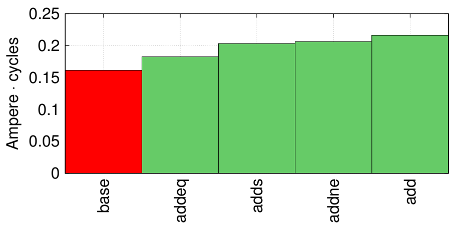


Fig. 5: Energy profile showing conditional execution.

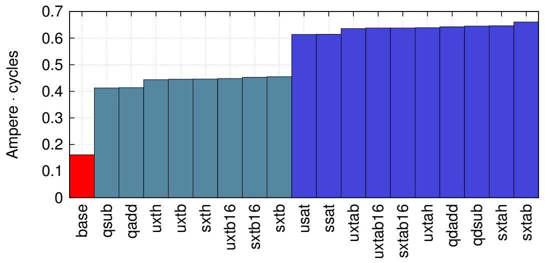


Fig. 6: Energy profile of multi-cycle instructions, excluding multiply.

between each issue. From our results, it seems that this saves energy. However, the instructions needs longer time to complete, which is indeed less energy efficient.

The results from the conditional-executed instructions are also subject to forced in-order execution. We can see from Figure 5 how variations of `add` compares. In the figured test, `addne` is committed every time, while `addeq` never has its results committed. It is interesting to see that even though `addeq` is never committed, it uses almost as much power as the other `adds`. By looking at Table II we see that `addeq` and `addne` introduces a lot of both *No disp.* and *Issue Empty*. We do not know exactly why, but it is reasonable to believe that one must assure that the previous instruction did not alter the status flags before the results are committed or discarded. In an in-order single-issue processor, conditional execution provides a framework to avoid unnecessary jumps, while in an out-of-order core, conditional execution is most likely much harder to implement. Also note that this test is very synthetic and the ISA is likely to be unoptimized for such activity. In a real world workload, it is possible that the required synchronization is hidden.

Further, Figure 4 shows that the `nop`-instruction has a rather low power consumption. This is a bit misleading, as the `nop`-instruction assembles to `mov r0, r0`, having both read-after-write and write-after-write hazard on itself. This makes the `nop`-instruction serialize itself, and it is hard to fill the pipeline with this instruction. Knowing this, it makes sense that `nop` works in this way, as it is often used to fill out clock cycles with non-destructive work. It would not make sense to optimize the `nop` instruction, as it then would fail to complete it's goal as a space-and-time filler.

Generally, when accounting for the number of cycles used by the different instructions, we see that the least current demanding single cycle instructions are `add` and `sub` at $216.2mA$, while `rev` and `sel` consumes slightly more with a drain of $225.4mA$. The measurements have a standard deviation of $4mA$ and $3.7mA$, respectively. Overall, the standard deviation ranges from $2mA$ to $7mA$, which we consider to be more than good enough.

E. *Multi-Cycle Instructions*

45 of the instructions that was compared used 2 or 3 cycles to complete their results. 18 of these instructions are non-multiply. Non-multiply instruction power measurements are displayed in Figure 6. A selection of the performance counter results are shown in Table II. We see that the unsigned extend instructions(`ux*`) are slightly cheaper than signed extend (`sx*`). This might indicate that some hardware is left idle when not needing sign extension. The instructions are normalized according to their stated cycle count in the table B-5 in [18].

F. *Multiply*

The ARM Cortex-A9 contains a single multiply pipeline, but has two general ALUsFigure 2. The multiply instructions are queued up waiting to execute through the same pipeline. This implies that multiply instructions would have a lower continuous power drain because it does less useful work and will seemingly use less energy compared to instructions utilizing both pipelines at its full potential. We have not compensated for this matter other than multiplying the power drain with the number of cycles used to finish one multiply instruction. It is unknown how the different multiply instructions utilize the pipeline(s). As we can see from Table III, there is reason to believe that at least some of the multiply-accumulate instructions utilize both pipelines[22]. This means that some instructions are able to utilize more hardware while still queuing up through the multiply-enabled main pipeline.

By looking at Figure 7 we see that the single-cycle multiply instructions are quite similar, but those using two or three cycles are more interesting. We do not know why the results are as stated, as most of the internal architecture are not available for the public. According to Table B-5 in [18], some multiply instructions uses more time than others before the result is available.

From the performance counters in Table III, we see that instructions are treated differently by the architecture. We have not considered all the tested instructions in detail, but it is evident to us that there is a strong negative correlation between performance counters (*No Disp.* and *Issue Empty*) and processor power drain. The results in Figure 7 shows power drain in Amperes multiplied by the cycle counts. The values are not normalized according to the performance counter values.

G. *Evaluation*

Each instruction was measured 41 times. We found small variations in power consumption between testbench runs, but all results shows the same trend. As stated in subsection II-E,

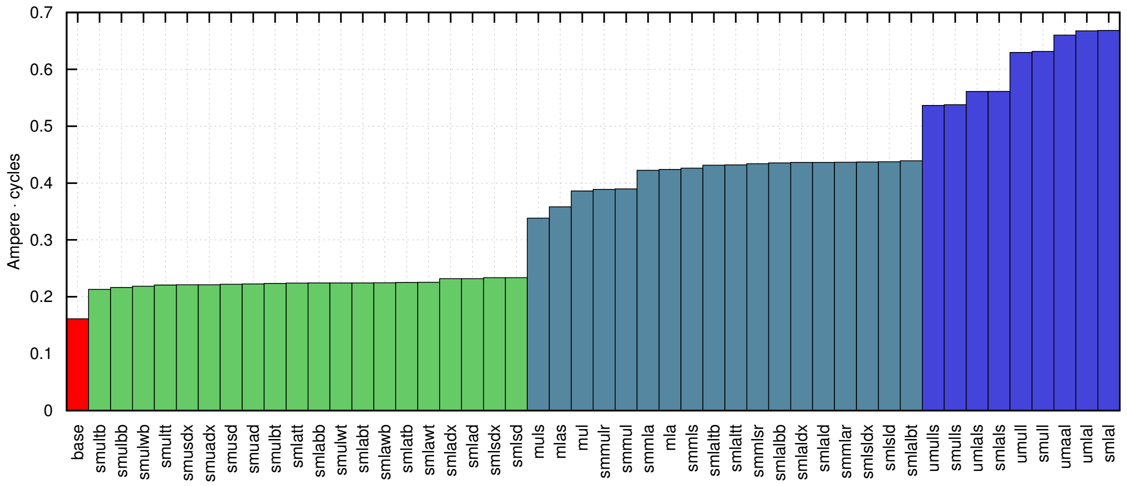


Fig. 7: Energy profile of multiply instructions.

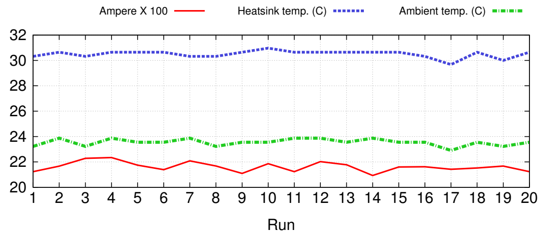


Fig. 8: Changes in heat and energy consumption for `add` at different runs together with heatsink and ambient temperature

the power consumption is not easily pushed by temperature. Figure 8 shows how the change in power consumption of the instruction `add` over different runs combined with the ambient temperature and the heatsink temperature. According to these results, we assume that the change in power consumption was not due to heat. We did not log temperature for all test runs, but assume that the results from Figure 8 holds, and that this small change in temperature is at least not solely responsible for variations in the current drain measurements.

IV. Conclusion

We have explored the inner workings of the ARM Cortex-A9 processor core and measured energy consumptions for various instructions. We found that the energy consumption for simple single-cycle instructions are rather equal. This RISC processor also includes a range of more advanced instructions that needs more than a single cycle to complete. We have looked into all multi-cycle instructions related to multiply and multiply-accumulate, along with a few register level data movement instructions.

Our main observation is that those instructions that are unable to fully fill the pipelines comes out as more energy efficient on the current readings. This is most likely because near empty pipeline consume less energy than a full pipeline. We must emphasize that these instructions are not more energy efficient than their counterparts, only slower in producing their intended results. Apart from this, the numbers tell that the most efficient instructions are `sub` and `add`, followed by common logical functions. This is expected as all these instruction are both easily implemented and commonly used.

It is also seen that the instructions executing conditionally and those settings status flags are subject to a less efficient instruction dispatching. We assume that synchronization is needed for this kind of instructions. Conditional executing is most likely better idea in a simple in-order CPU than in advanced out-of-order CPU cores. We also notice that instructions that should not be committed is issued, executed and then discarded.

For the multi-cycle instructions, we observed that even though the processor datasheet[18] states a number of cycles for each instruction to complete its result, different pipelining schemes apply to the different instructions. Multiply can only be done in the main execution unit, while accumulate is seemingly executed in the second pipeline. This means that even though `mul` introduces queueing for access to the main pipeline, multiply-accumulate (`mla`), is equally fast, see Table III.

A. *Further Work*

Our results comes from completely synthetic benchmarks, and we do not yet know how this would differ from real world workloads. The synthetic tests fill the pipeline with equal instructions, while common workloads would at least contain a few different instructions simultaneously.

The results was normalized according to numbers found in the CPU datasheet. We believe that more informative results would emerge if the performance counter data was used to adjust the measured current drain, rather than number of cycles used. This is after all a multiple-issue pipelined processor core.

Instr.	Cycles	Main Ex.	Second Ex.	Pred.	Mis pred.	No disp.	Issue Empty
mla	6608	6530	4895	251	2	76	76
mlas	15639	6529	12548	251	2	8857	73
mul	6602	6525	252	251	2	3336	76
muls	15617	6526	252	251	2	12100	61
smlabb	3604	3518	3265	251	2	83	83
smlabt	3604	3518	3265	251	2	83	83
smlad	3604	3518	3265	251	2	83	83
smladx	3604	3518	3265	251	2	83	83
smlal	7106	7028	5020	251	2	575	76
smlalbb	7102	7021	5019	251	2	3582	76
smlalbt	7102	7021	5019	251	2	3582	76
smlald	7102	7021	5019	251	2	3582	76
smlaldx	7102	7021	5019	251	2	3582	76
smlals	15888	6529	12548	251	2	9106	73
smlaltb	7102	7021	5019	251	2	3582	76
smlaltt	7102	7021	5019	251	2	3582	76
smlatb	3604	3518	3265	251	2	83	83
smlatt	3604	3518	3265	251	2	83	83
smlawb	3604	3518	3265	251	2	83	83
smlawt	3604	3518	3265	251	2	83	83
smlsd	3604	3518	3265	251	2	83	83
smlsdx	3604	3518	3265	251	2	83	83
smlsld	7102	7021	5019	251	2	3582	76
smlsldx	7102	7021	5019	251	2	3582	76
smmla	6608	6530	4895	251	2	76	76
smmlar	6608	6530	4895	251	2	76	76
smmls	6608	6530	4895	251	2	76	76
smmlsr	6609	6530	4895	251	2	77	77
smmul	6602	6525	252	251	2	3336	76
smmulr	6603	6525	252	251	2	3337	77
smuad	3602	3266	252	251	2	334	334
smuadx	3602	3266	252	251	2	334	334
smulbb	3353	3267	252	251	2	84	84
smulbt	3353	3267	252	251	2	84	84
smull	6857	6779	6774	251	2	326	76
smulls	15637	6528	15558	251	2	8856	72
smultb	3353	3267	252	251	2	84	84
smultt	3353	3267	252	251	2	84	84
smulwb	3353	3267	252	251	2	84	84
smulwt	3353	3267	252	251	2	84	84
smusd	3602	3266	252	251	2	334	334
smusdx	3602	3266	252	251	2	334	334
umaal	7106	7028	5020	251	2	575	76
umlal	7106	7028	5020	251	2	575	76
umlals	15888	6529	12548	251	2	9106	73
umull	6857	6779	6774	251	2	326	76
umulls	15637	6528	15558	251	2	8856	72

TABLE III: Performance counter data from 252 iterations of all tested multiply instructions.

Also, we have not yet dived into how instruction arguments affects the energy usage on modern processors. We believe that instruction patterns that causes a high degree of bit toggling would yield higher energy usage, due to the amount of energy used to charge and release the transistors. A problem rising is the fact that we do not know how the processor schedules or distributes the different instructions, thus one has to be very careful when writing the benchmarks.

When selecting instructions for our benchmarks, we have omitted the set of floating-point instructions. This is because in the ARM Cortex-A9, the floating point unit (NEON) is considered a co-processor[18], and thus out of our scope. Investigating the energy efficiency of co-processors versus processors that embed such functionality would add value to our results.

There are also room for improvements regarding the experimental setup. Ultimately, one would like to be able to measure each instruction individually, but according to the Nyquist-Shannon theorem[23], this would require a sampling rate of at least 3.4 GHz. We could not simply go slower on the clock, as a clock frequency reduction will affect the energy efficiency, possibly in the negative direction[24].

Compilers, simulators and synthesis tools would benefit from this kind of information, and one could possibly generate output that is more energy optimized than currently available.

REFERENCES

[1] G. E. Moore *et al.*, “Cramming more components onto integrated circuits,” 1965.

[2] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong, “Device scaling limits of Si MOSFETs and their application dependencies,” *Proceedings of the IEEE*, vol. 89, no. 3, pp. 259–288, 2001.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[4] S. P. Dandamudi, *Guide to RISC Processors for Programmers and Engineers*. Springer, 2005, ch. 3.

[5] A.-E. Bogen, “Risc versus cisc,” http://alfbogen.com/2013/06/16/risc-versus-cisc/.

[6] S. A. M. Karan Singh, Major Bhadauria, “Real Time Power Estimation and Thread Scheduling via Performance Counters,” May 2009.

[7] R. B. et.al., “Decomposable and responsive power models for multi-core processors using performance counters,” June 2010.

[8] Bircher and John, “Complete system power estimation using processor performance events,” April 2012.

[9] A. Carroll and G. Heiser, “An analysis of power consumption in a smartphone,” in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010, pp. 21–21.

[10] NTNU, “The Single-ISA Heterogeneous MAny-core Computer (SHMAC),” http://www.ntnu.edu/ime/eecs/shmac, Desember 2013.

[11] Y. Umuroglu, “SHMACsim : A Cycle-accurate Simulation Infrastructure for the Heterogeneous SHMAC Multi-Core Prototype,” p. 111, 2013.

[12] L. T. Rusten and G. I. Sortland, “Implementing a heterogeneous multi-core prototype in an fpga,” Ph.D. dissertation, Norwegian University of Science and Technology, 2012.

[13] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, “Single-isa heterogeneous multi-core architectures: The potential for processor power reduction,” in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 81–92.

[14] Hardkernel, “hardkernel.com,” http://www.hardkernel.com/renewal_2011/products/prdt_info.php?g_code=G135235611947.

[15] J. L. H. David A. Patterson, *Computer Organization and Design*, 4th ed. Morgan Kaufmann, 2012.

[16] D. A. P. John L. Hennessy, *Computer Architecture, A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[17] “The ARM Cortex-A9 Processors,” http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf.

[18] “Cortex-A9 Technical Reference Manual revision r3p0,” http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388g/DDI0388G_cortex_a9_r3p0_trm.pdf.

[19] I. Agilent Technologies, “Agilent 34410A and 34411A multimeters data sheet,” http://www.home.agilent.com/en/pd-692834-pn-34410A/digital-multimeter-6-digit-high-performance, April 2013.

[20] “ARM and Thumb-2 Instruction Set Quick Reference Card,” http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf.

[21] “ARM Compiler toolchain Version 4.1 Assembler Reference,” http://infocenter.arm.com/help/topic/com.arm.doc.dui0489c/DUI0489C_arm_assembler_reference.pdf.

[22] R. Radhakrishnan, “Integer pipeline description for cortex a9.” http://gcc.gnu.org/ml/gcc-patches/2009-10/msg01858.html, October 2009.

[23] C. E. Shannon, “Communication in the presence of noise,” *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949.

[24] T. D. Burd and R. W. Brodersen, “Energy efficient cmos microprocessor design,” in *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, vol. 1. IEEE, 1995, pp. 288–297.

Appendix B Modifications to gem5

All file paths are relative to the root of gem5. Diffs are based of gem5-stable revision `aaf017eaad7d`.

B.1 configs/common/Exynos_4412P.py

```
# Copyright (c) 2012 The Regents of The University of Michigan
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
    are
# met: redistributions of source code must retain the above copyright
# notice, this list of conditions and the following disclaimer;
# redistributions in binary form must reproduce the above copyright
# notice, this list of conditions and the following disclaimer in the
# documentation and/or other materials provided with the distribution;
# neither the name of the copyright holders nor the names of its
# contributors may be used to endorse or promote products derived from
# this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
# OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
# Authors: Ron Dreslinski


```

```python
from m5.objects import *

# Simple ALU Instructions have a latency of 1
class Exynos_Simple_Int(FUDesc):
    opList = [ OpDesc(opClass='IntAlu', opLat=1) ]
    count = 1

# Complex ALU instructions have a variable latencies
class Exynos_Complex_Int(FUDesc):
    opList = [ OpDesc(opClass='IntMult', opLat=4, issueLat=1),
               OpDesc(opClass='IntAlu', opLat=1),
               OpDesc(opClass='IntDiv', opLat=12, issueLat=12),
               OpDesc(opClass='IprAccess', opLat=3, issueLat=1) ]
    count = 1


# Floating point and SIMD instructions
class Exynos_FP(FUDesc):
    opList = [ OpDesc(opClass='SimdAdd', opLat=4),
               OpDesc(opClass='SimdAddAcc', opLat=4),
               OpDesc(opClass='SimdAlu', opLat=4),
               OpDesc(opClass='SimdCmp', opLat=4),
               OpDesc(opClass='SimdCvt', opLat=3),
               OpDesc(opClass='SimdMisc', opLat=3),
               OpDesc(opClass='SimdMult',opLat=5),
               OpDesc(opClass='SimdMultAcc',opLat=5),
               OpDesc(opClass='SimdShift',opLat=3),
               OpDesc(opClass='SimdShiftAcc', opLat=3),
               OpDesc(opClass='SimdSqrt', opLat=9),
               OpDesc(opClass='SimdFloatAdd',opLat=5),
               OpDesc(opClass='SimdFloatAlu',opLat=5),
               OpDesc(opClass='SimdFloatCmp', opLat=3),
               OpDesc(opClass='SimdFloatCvt', opLat=3),
               OpDesc(opClass='SimdFloatDiv', opLat=3),
               OpDesc(opClass='SimdFloatMisc', opLat=3),
               OpDesc(opClass='SimdFloatMult', opLat=3),
               OpDesc(opClass='SimdFloatMultAcc',opLat=4),
               OpDesc(opClass='SimdFloatSqrt', opLat=9),
               OpDesc(opClass='FloatAdd', opLat=4),
               OpDesc(opClass='FloatCmp', opLat=5),
               OpDesc(opClass='FloatCvt', opLat=5),
               OpDesc(opClass='FloatDiv', opLat=9, issueLat=9),
               OpDesc(opClass='FloatSqrt', opLat=33, issueLat=33),
               OpDesc(opClass='FloatMult', opLat=5) ]
    count = 1


# Load/Store Units
class Exynos_LS(FUDesc):
    opList = [ OpDesc(opClass='MemRead',opLat=1),
               OpDesc(opClass='MemWrite',opLat=1) ]
    count = 1
```

```


# Functional Units for this CPU
class Exynos_FUP(FUPool):
    FUList = [Exynos_Simple_Int(), Exynos_Complex_Int(),
              Exynos_LS(), Exynos_FP()]

# Tournament Branch Predictor
class Exynos_BP(BranchPredictor):
    predType = "tournament"
    localPredictorSize = 512
    localCtrBits = 2
    localHistoryTableSize = 512
    globalPredictorSize = 2048
    globalCtrBits = 2
    choicePredictorSize = 8192
    choiceCtrBits = 2
    BTBEntries = 2048
    BTBTagSize = 18
    RASSize = 16
    instShiftAmt = 2

class Exynos_3(DerivO3CPU):
    LQEntries = 4
    SQEntries = 4
    LSQDepCheckShift = 0
    LFSTSize = 1024
    SSITSize = 1024
    decodeToFetchDelay = 1
    renameToFetchDelay = 1
    iewToFetchDelay = 1
    commitToFetchDelay = 1
    renameToDecodeDelay = 1
    iewToDecodeDelay = 1
    commitToDecodeDelay = 1
    iewToRenameDelay = 1
    commitToRenameDelay = 1
    commitToIEWDelay = 1
    fetchWidth = 2
    fetchBufferSize = 16
    fetchToDecodeDelay = 2
    decodeWidth = 2 # syslevel benchmark
    decodeToRenameDelay = 2
    renameWidth = 2
    renameToIEWDelay = 1
    issueToExecuteDelay = 1
    dispatchWidth = 4 # syslevel benchmark
    issueWidth = 2 # marketing
    wbWidth = 2
    wbDepth = 2
    fuPool = Exynos_FUP()
    iewToCommitDelay = 1
    renameToROBDelay = 1
    commitWidth = 4
    squashWidth = 2
    trapLatency = 37
    backComSize = 5
    forwardComSize = 5
    numPhysIntRegs = 56
    numPhysFloatRegs = 192
    numIQEntries = 16
    numROBEntries = 40

    switched_out = False
    branchPred = Exynos_BP()

# Instruction Cache
class Exynos_ICache(BaseCache):
    hit_latency = 2 # 7cpu
    response_latency = 2 # 7cpu
    mshrs = 6
    tgts_per_mshr = 8
    size = '32kB'
    assoc = 4
    is_top_level = 'true'

# Data Cache
class Exynos_DCache(BaseCache):
    hit_latency = 3 # 7cpu
    response_latency = 8 # 7cpu
    mshrs = 16
    tgts_per_mshr = 8
    size = '32kB'
    assoc = 4
    write_buffers = 16
    is_top_level = 'true'

# TLB Cache
# Use a cache as a L2 TLB
class ExynosWalkCache(BaseCache):
    hit_latency = 7 #7cpu
    response_latency = 7 # 7cpu
    mshrs = 16
    tgts_per_mshr = 8
    size = '2kB'
    assoc = 2
    write_buffers = 16
    is_top_level = 'true'


# L2 Cache
class ExynosL2(BaseCache):
    hit_latency = 37 # 7cpu?
    response_latency = 37 # 7cpu?
    mshrs = 32
    tgts_per_mshr = 8
    size = '1MB'
    assoc = 16
    write_buffers = 8
    prefetch_on_access = 'true'
    # Simple stride prefetcher
    prefetcher = StridePrefetcher(degree=1, latency = 2)
```

B.2 configs/example/se.py

```
diff --git a/configs/example/se.py b/configs/example/se.py
--- a/configs/example/se.py
+++ b/configs/example/se.py
@@ -104,7 +104,7 @@
         idx += 1

     if options.smt:
-        assert(options.cpu_type == "detailed" or options.cpu_type == "inorder")
+        assert(options.cpu_type == "arm_detailed" or options.cpu_type == "inorder")
         return multiprocesses, idx
     else:
         return multiprocesses, 1
@@ -219,7 +219,7 @@
     system.cpu[i].createThreads()

 if options.ruby:
-    if not (options.cpu_type == "detailed" or options.cpu_type == "timing"):
+    if not (options.cpu_type == "arm_detailed" or options.cpu_type == "timing"):
         print >> sys.stderr, "Ruby requires TimingSimpleCPU or O3CPU!!"
         sys.exit(1)

@@ -255,4 +255,5 @@
     MemConfig.config_mem(options, system)

 root = Root(full_system = False, system = system)
+print options
 Simulation.run(options, root, system, FutureClass)
```

B.3 configs/common/CacheConfig.py

```
diff --git a/configs/common/CacheConfig.py b/configs/common/CacheConfig.py
--- a/configs/common/CacheConfig.py
+++ b/configs/common/CacheConfig.py
@@ -55,9 +55,22 @@

         dcache_class, icache_class, l2_cache_class = \
             O3_ARM_v7a_DCache, O3_ARM_v7a_ICache, O3_ARM_v7aL2
+    elif options.cpu_type == "exynos_4412p":
+        try:
+            from Exynos_4412P import *
+        except:
+            print "exynos_4412p is unavailable. Did you compile the O3 model?"
+            sys.exit(1)
+
+        dcache_class, icache_class, l2_cache_class = \
+            Exynos_DCache, Exynos_ICache, ExynosL2
+
     else:
         dcache_class, icache_class, l2_cache_class = \
             L1Cache, L1Cache, L2Cache
+    print dcache_class
+    print icache_class
+    print l2_cache_class

     # Set the cache line size of the system
     system.cache_line_size = options.cacheline_size
@@ -71,8 +84,9 @@
                                    size=options.l2_size,
                                    assoc=options.l2_assoc)

+        print system.cpu_clk_domain
         system.tol2bus = CoherentBus(clk_domain = system.cpu_clk_domain,
-                                     width = 32)
+                                     width = 64)
         system.l2.cpu_side = system.tol2bus.master
         system.l2.mem_side = system.membus.slave
```

B.4 configs/common/CpuConfig.py

```
diff --git a/configs/common/CpuConfig.py b/configs/common/CpuConfig.py
--- a/configs/common/CpuConfig.py
+++ b/configs/common/CpuConfig.py
@@ -116,6 +116,13 @@
 except:
     pass
 
+
+try:
+    from Exynos_4412P import Exynos_3
+    _cpu_classes["exynos_4412p"] = Exynos_3
+except:
+    pass
+
 # Add all CPUs in the object hierarchy.
 for name, cls in inspect.getmembers(m5.objects, is_cpu_class):
     _cpu_classes[name] = cls
```

B.5 src/arch/arm/linux/process.cc

```
diff --git a/src/arch/arm/linux/process.cc b/src/arch/arm/linux/process.cc
--- a/src/arch/arm/linux/process.cc
+++ b/src/arch/arm/linux/process.cc
@@ -66,7 +66,7 @@
 
     strcpy(name->sysname, "Linux");
     strcpy(name->nodename, "m5.eecs.umich.edu");
-    strcpy(name->release, "3.0.0");
+    strcpy(name->release, "3.10.2");
     strcpy(name->version, "#1 Mon Aug 18 11:32:15 EDT 2003");
     strcpy(name->machine, "armv7l");
```

B.6 scr/mem/SimpleDRAM.py

```
diff --git a/src/mem/SimpleDRAM.py b/src/mem/SimpleDRAM.py
--- a/src/mem/SimpleDRAM.py
+++ b/src/mem/SimpleDRAM.py
@@ -258,6 +258,62 @@
     tXAW = '50ns'
     activation_limit = 4

+# A single LPDDR2-S4 x32 400MHz interface (one command/address bus)
+class LPDDR2_S4_800_x32(SimpleDRAM):
+    # 1x32 configuration, 1 device with a 32-bit interface
+    device_bus_width = 32
+
+    # LPDDR2_S4 is a BL4 and BL8 device
+    burst_length = 8
+
+    # Each device has a page (row buffer) size of 1KB
+    # (this depends on the memory density)
+    device_rowbuffer_size = '1kB'
+
+    # 1x32 configuration, so 1 device
+    devices_per_rank = 1
+
+    # Use a single rank
+    ranks_per_channel = 1
+
+    # LPDDR2-S4 has 8 banks in all configurations
+    banks_per_rank = 8
+
+    # Fixed at 15 ns
+    tRCD = '15ns'
+
+    # 8 CK read latency, 4 CK write latency @ 533 MHz, 1.876 ns cycle time
+    tCL = '15ns'
+
+    # Pre-charge one bank 15 ns (all banks 18 ns)
+    tRP = '15ns'
+    tRAS = '42ns'
+
+    # 8 beats across an x32 DDR interface translates to 4 clocks @ 400 MHz.
+    # Note this is a BL8 DDR device.
+    # Requests larger than 32 bytes are broken down into multiple requests
+    # in the controller
+    tBURST = '7.5ns'
+
+    # LPDDR2-S4, 16Gbit
+    tRFC = '210ns'
+    tREFI = '3.9us'
+
+    # Irrespective of speed grade, tWTR is 7.5 ns
+    tWTR = '7.5ns'
+
+    # Activate to activate irrespective of density and speed grade
+    tRRD = '10.0ns'
+
+    # Irrespective of density, tFAW is 50 ns
+    tXAW = '50ns'
+    activation_limit = 4
+
 # A single WideIO x128 interface (one command and address bus), with
 # default timings based on an estimated WIO-200 8 Gbit part.
 class WideIO_200_x128(SimpleDRAM):
```

REMEMBER: WITH GREAT POWER COMES GREAT CURRENT SQUARED TIMES RESISTANCE.
OHM NEVER FORGOT HIS DYING UNCLE'S ADVICE.

From XKCD [73].