INVESTIGATING POWER MANAGEMENT SCHEMES IN OUT-OF-ORDER MICROPROCESSORS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2017

By
Yaman Cakmakci
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

INVESTIGATING POWER MANAGEMENT SCHEMES IN
OUT-OF-ORDER MICROPROCESSORS
Yaman Cakmakci
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2017

Limitations imposed by the the end of Dennard Scaling have led to a significant increase in the power density of chips. This has elevated power efficiency to a first order design constraint.

This thesis proposes a novel microbenchmark generator, the Generator of LLVM Assisted Microbenchmarks (GLAM), a tool which enables the generation of architecture agnostic microbenchmarks by exploiting the LLVM Intermediate Representation. GLAM can be used for design space exploration, as a tool to aid in power model generation, and has been used to evaluate the power management scheme described in this thesis.

Furthermore, a novel power management scheme, Cyclical Power Gating (CPG), is proposed. CPG exploits a state-retentive power-gating technique that allows power consumption to be scaled linearly without reducing the supply voltage. CPG works by turning the core on and off over a parameterised period and duty cycle. The low switching overhead of CPG can be used to apply it at the granularity of program function level, and provides Energy Delay Product (EDP) gains when compared to nominal operation.

A simulation-based comparison of CPG with equivalent voltage and frequency levels of a system equipped with Dynamic Voltage and Frequency Scaling shows that CPG provides 6.27% better Energy-Delay-Product than Voltage and Frequency Scaling.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

Spending the last four years in Manchester made me realise that no matter how dark a city is, it is actually the people that brings in the light. First, and foremost I am thankful to my supervisor Mikel Lujan for accepting me as a PhD student and making these splendid four years possible. His guidance in times where I felt lost, along with his patience will be remembered. I would like to thank Will Toms for his assistance in times of need, and for all the technical discussions that would at times context switch to non-technical gibberish during our lunch breaks.

I had the chance to work with very bright people within the APT. I would like to thank Christos, John Mawer, Andy Nisbet, James Clarkson, John Goodacre, Guillermo, Bernard, Swapnil, Mireya, Andrey, Thanos, Seckin, Mohsen, Farideh, Andrew Leeming, Geoffrey, Matt, Jim Garside, Guangda, Andi, and Antoniou for being great colleagues. A special thanks to Ioanna, Serhat, Cosmin, Ozan, Emre and Sebastian for all the beers we had and their support during the stressful writing period. The MLO folk also deserve a shout-out for all their humour and cheerfulness. Thanks to Nikos, Henry, Sarah, and Kostas for all the lunches we had in the business school, along with the weekend activities that resulted in occasional hangovers. I wish all the best for Kostas and Idoia. I would like to thank Anatoli for being a great gig buddy, and a friend. I would like to thank Erhan, and Babis for being great flatmates, and having to endure the terror of sharing a house with me. A special thanks to my best mate Yalim, for all the hilarious conversations we had over gtalk throughout the writing of this thesis.

I have been lucky to establish friendships here that I am sure will last forever. I would like to thank Asan for all that he has taught, and all the open conversations we had. I wish him the brightest future. Paris deserves a special thanks for his politically correct sense of humor, and all the coffees that we shared for slacking purposes. Wish you a great future with Laura and the kid! A big thanks to Evangelos for proving his friendship once more by taking the time to proof-read the thesis. I hope our paths will

intersect in the future! Finally, I would like to thank Esin for all her love and laughter.

For all their support and affection, a big thanks to my family to whom this thesis is dedicated to.

If I have forgotten anyone, please excuse me.

Chapter 1

Introduction

Microprocessors are devices that can be programmed to solve a wide variety of problems with different inputs. For example, they can be programmed to run critical autopilot algorithms that control aeroplanes, or for applying not so critical digital filters to photographs shared over the internet. The fundamental building blocks of microprocessors are called transistors. Transistors are used as switching circuits to build logic gates, which are then used to build logic blocks such as adders, decoders and registers, that make up a microprocessor. Microprocessors are manufactured using Complementary Metal Oxide Semiconductor (CMOS) technology and have been increasing in an exponential scale since the 1960s.

An important factor on the subject of CMOS evolution and transistor scaling is Dennard scaling. Dennard laid the scientific foundations of scaling [DGR$^+$74]. A simplified interpretation of Dennard scaling is that each new generation of transistors results in a two fold decrease in chip area and power consumption, while the performance is doubled. The main benefit that came from shrinking transistor sizes were the inherent performance improvements that came with the new technology.

A CMOS IC such as a microprocessor has two main contributors to the power it dissipates. Static power, also called *leakage*, is dissipated as long as the circuit is powered on [WHB05]. Leakage is dependent on supply voltage, and increases exponentially with temperature [WHB05]. Dynamic power only contributes to power dissipation when the transistors are switching. The feature size of a transistor specifies the length between the source and drain terminals of a transistor. With 90-nm transistors, the direct benefits of Dennard Scaling started to diminish mainly due to the increase in leakage power [DCK07].

Moore explained a trend in his 1965 paper [Moo65] that has remained valid for

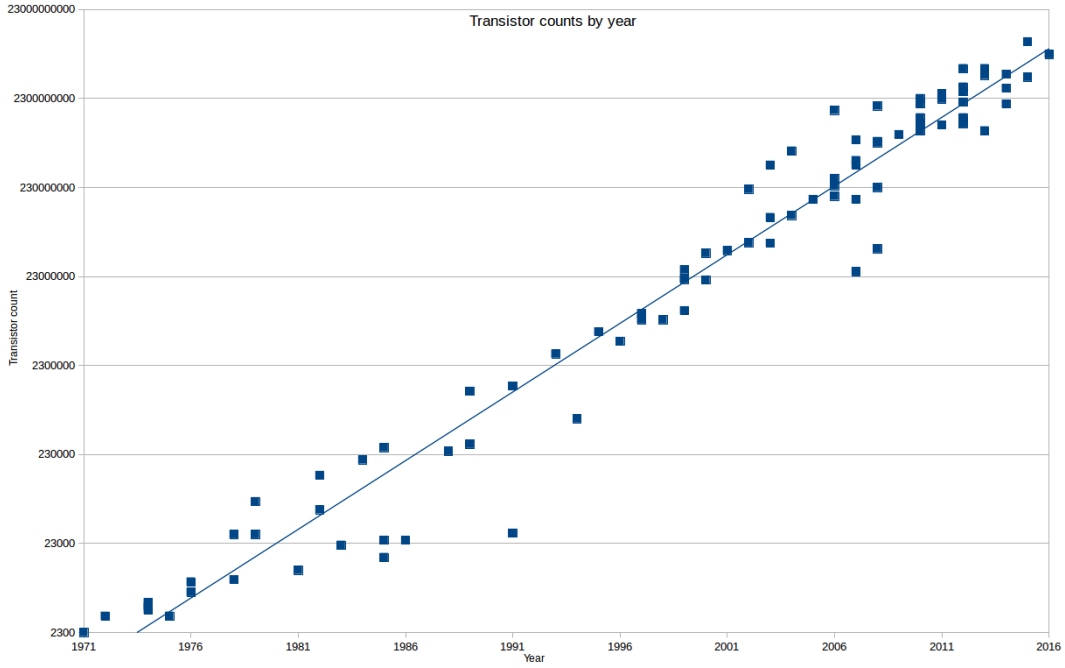


Figure 1.1: Trend showing the exponential increase in processor transistor counts. [Data taken from [DKM+12]]

around half a century. He wrote: "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase." He also added that there is no reason why this would not continue in the long term. A simplified rephrasing of Moore's previous statement can be interpreted as the number of transistors that can be deployed per area will double every two years. Figure 1.1 shows how this observation applied to transistor counts for production processors marketed from 1971 up until 2016 [DKM+12].

It has been projected that Moore's law will continue beyond 7 nanometers by introducing new technologies that will replace the current silicon CMOS devices in the long term, and by perfecting the manufacturing process in the short term [TP06, Tou16]. As of 2016, Intel Corporation announced that it will be shipping the 10 nm Cannonlake architecture by 2017 [Eas16].

Although the manifestation of Moore's Law has provided an increase in transistor counts, the breaking of Dennard Scaling has resulted in underutilisation of these transistors. The increase in transistor counts without the means of decreasing power characteristics in proportion leads to an increase in the power density of microprocessors. This can result in thermal issues, which in turn limit the operating capabilities

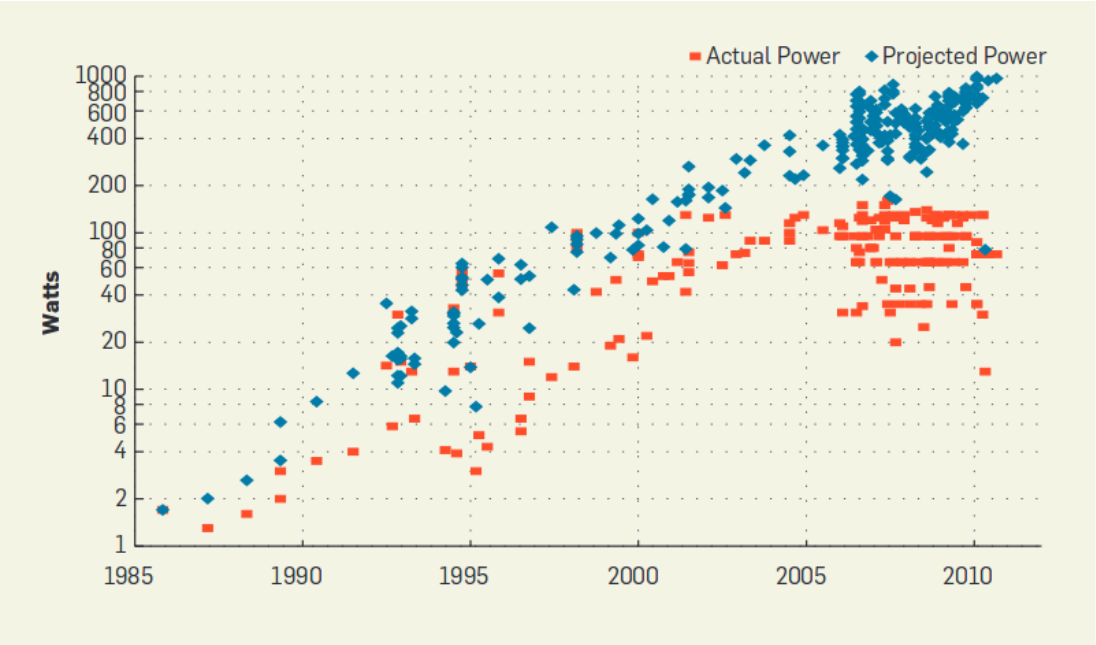


Figure 1.2: Microprocessor power dissipation projections against actual power dissipation (Figure taken from Danowitz *et al.* [DKM+12])

of these chips. The problem is illustrated in Figure 1.2, where the proportional scaling between actual and projected power breaks around 2005. The inability to retain the power density of chips has lead to efficient power management of computing systems becoming a primary design requirement.

There are two power management schemes that are being used extensively in computing systems; power gating, and Voltage and Frequency Scaling (VFS). Power gating is used to cut off the voltage supply to unused blocks in a system, and enable the voltage supply back on when that block is required for execution. VFS is utilised online by a technique called Dynamic Frequency and Voltage Scaling (DVFS), where the voltage and frequency of a block is reduced in parallel at times when the expected performance of the block is low.

The methodology for performing empirical research in the domain of computer architecture, including power management schemes, are either by implementing the Register Transfer Level (RTL) level code on Field Programmable Gate Arrays (FPGA) using hardware description languages, or through simulators written in high-level programming languages. As the intellectual property for common instruction sets such as ARM or x86 are not always available to the research community, system simulations are used as a cost-effective way for performing research. A major disadvantage of

using simulators is long simulation times due to computational requirements of cycle-accurate simulation and program size of benchmarks contained in typical benchmark suites. Current methodologies to reduce simulation times mark regions of interests based either on program phases, or hardware events. These regions are simulated at a higher level of detail, while periods that are not in these regions are simulated at a lower level of detail with minimal degradation in accuracy [GEE10].

This dissertation aims to tackle some of the challenges posed by computer architecture research methodologies, and the end of Dennard scaling. There are two main contributions that appear in this dissertation. These contributions, presented in Chapter 4 and Chapter 5, are:

- Generator of LLVM Assisted Benchmarks (GLAM), an architecture independent synthetic workload generator that generates user-defined benchmarks at the Low-Level Virtual Machine (LLVM) [LA04] Intermediate Representation (IR) level. GLAM is utilised to generate synthetic workloads aimed at exploring architectural proposals. This approach enables faster evaluation of research ideas by reducing the simulation time compared to running actual benchmarks (Chapter 4).

- By first presenting the shortcomings of VFS, a novel runtime configurable power-gating method called Cyclical Power Gating (CPG) is proposed for decreasing leakage, and temperature. CPG works by switching between on and off (power-gated) cycles within a given period while storing the state of the power-gated blocks for quick wake-up purposes. CPG is proposed as an alternative to VFS, and it is shown that CPG can outperform equivalent VFS states in terms of Energy Delay Product. (Chapter 5)

1.1 Challenges with computer architecture simulation

Simulating microprocessor based systems enables design space exploration without prototyping at the hardware level, but long simulation times is the major downside of this approach. A given research idea is tested by inserting the proposed changes directly into the simulator source code or through software modules that are interfaced with the simulator. These modifications are then evaluated against a baseline configuration by executing a subset of a benchmark suite that is convenient to evaluate the implemented idea. There are two issues with performing simulations to evaluate an

idea by executing benchmarks.

Firstly, an implemented research idea is generally focused towards a certain block within a microprocessor. On the other hand a selected benchmark from a suite may not be continously stressing the targeted hardware block, which leads to a requirement to single out and replay the phase that is intended to stress that particular block. One solution to this problem is available through means of *offline phase classification*, where a program binary is divided into chunks of instructions and these chunks are grouped together based on a similarity metric using a clustering algorithm [HPLC05]. Although offline analysis leads to reduced simulation times, the assumptions made during offline analysis might fall short since execution also depends on runtime data.

The second issue lies in the fact that most benchmarks have long execution times, and running a benchmark execution until it terminates may take a long time to simulate. Most of the published work limits benchmark execution up until some number of instructions from the beginning of execution, and this may lead to an inaccurate evaluation of that benchmark since all of the program behaviour is not captured in the evaluation. As an example, simulation of the *mcf* benchmark from the SPEC CPU2006 benchmark suite [Hen06a] on the customised gem5 simulation platform described in Chapter 3 would take approximately a week, and this would only cover the first 50 seconds of the execution.

A possible solution to the previously mentioned problems on benchmark executions is using *microbenchmarks* to evaluate architectural changes in a simulated environment. A microbenchmark is a small segment of code that specifically targets a certain set of blocks within a microprocessor. Using microbenchmarks greatly decrease execution time without affecting the validity of the evaluation, since the same architectural block is utilised with lesser number of instructions [CAP+15, JEJI08].

The Generator of LLVM Assisted Microbenchmarks (GLAM) described in Chapter 4 introduces a novel way of producing architecture independent microbenchmarks by targeting the LLVM Intermediate Representation.

1.2 Challenges with power management

With transistor counts increasing and Dennard Scaling reaching an end, the scaling down of transistors has given rise to reliability and power-related research problems [SABR04]. This means that if the number of transistors per area were to keep on doubling with each new generation of transistors in the absence of proportionally scaled

operating voltages, power density of newer chips will be so high that it would not be possible to run them at the highest performance profile that transistor specifications allow. The power characteristics of the transistor technology used directly affects the power dissipation of a chip.

Equation 1.1 gives the total power dissipated by a chip [WHB05]:

$$P_{total} = P_{dynamic} + P_{static}. \tag{1.1}$$

Static power, defined by Equation 1.2, consists of the product of supply voltage (V_{dd}) for the device and the current (I_{static}) passing through [WHB05]

$$P_{static} = I_{static} V_{dd}. \tag{1.2}$$

Dynamic power is directly proportional to circuit clock frequency and the square of voltage, and C is the amount of charge required to charge the load capacitance (Equation 1.3). The parameter α defines the ratio of switching going on on the chip at a given frequency cycle. For example, when half of the transistors on a chip would be switching at a given cycle then α would be 0.5 [WHB05].

$$P_{dynamic} = \alpha C V_{dd}^2 f. \tag{1.3}$$

Although it still is technically possible to manufacture smaller transistors, the operating voltages can not be lowered as aggressively as before because they are on the boundary of a physical lower limit [FDN+01]. Operating voltages of transistors hitting a lower limit also means that the contribution of leakage power to system power dissipation is showing an increasing trend.

As transistors shrink they become more susceptible to reliability issues such as ageing. Transistor *wear-out*, also known as *transistor ageing*, is the gradual degradation of transistors from their initial properties in terms of operation. This may eventually lead to the total failure of the transistor. The common failure mechanisms [SWV+09] for transistors are Time Dependent Electric Breakdown (TDDB), Negative-Bias Temperature Instability (NBTI), Hot Carrier Induction (HCI) and electromigration.

Almost all of the reliability problems are caused by operation of the device at a high temperature [Ala03], which is a function of power dissipation. Schemes that provide finer grained power management in terms of response time should be developed in order to have more reliable and power efficient systems.

Power-efficient computing techniques can be classified into three categories; temporal, spatial and redundancy reduction. *Temporal* techniques are mainly concerned with decreasing the throughput by employing methods such as execution throttling [UKM02] and Dynamic Voltage Frequency Scaling (DVFS) [HM07]. *Spatial* techniques improve power-efficiency by constraining the computation into a smaller area at the expense of increased power density. The concept of memory hierarchy can be given as a primary example of this technique [HP12]. *Redundancy reduction* techniques are optimisations applied to the system in order to reduce redundant information storage or messaging such as disabling of cache coherence when memory blocks are known to be private to a core [CRG+11], or application aware adaptive cache replacement policies [JNaS+12].

When system performance is a concern, a major contradiction in requirements for power-efficient computing and temperature decreasing methods lies in the fact that power-efficient techniques aim to reduce the spatial or temporal space that an operation takes place in, potentially creating hotspots, whereas keeping the temperature lower requires the distribution of the operation over a wider area or period. In the case that temperature becomes the primary concern, temporal power-efficiency techniques are more suitable for tackling the issue.

Dynamic power is dissipated when there is transistor activity, whereas static power (leakage) is the power dissipated as long as a transistor is powered. Two techniques are applied in order to reduce power in circuits; Voltage and Frequency Scaling (VFS) and power-gating.

Dynamic Voltage and Frequency Scaling (DVFS) is the most commonly used technique for power-efficiency and thermal control in microprocessors [HDH+10, KM08]. DVFS relies on decreasing voltage and frequency of the on-chip transistors based on decisions coming from the related layer such as the operating system. It is important to note that voltage and frequency are dependent on each other, and once voltage is decreased the upper bound of the frequency that a transistor can run at is also decreased. DVFS can be controlled by an algorithm that selects an appropriate pair from a set of available voltage/frequency pairs, and provides the opportunity to dynamically switch between performance (high voltage/frequency) and low-power modes (low voltage/frequency).

There are two main issues concerning the usability of DVFS in the future. Firstly, the contribution of static power to the total power consumption is increasing as scaling down of supply voltages has limitations due to reaching physical boundaries. The

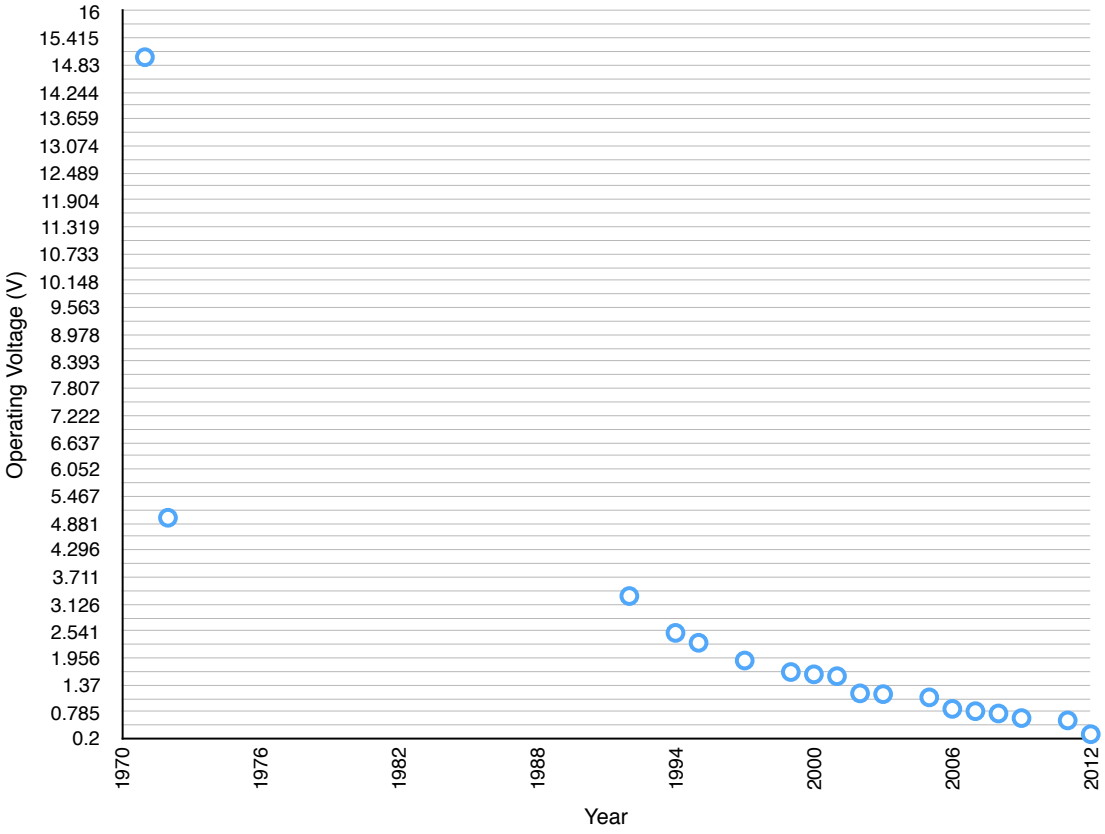


Figure 1.3: Trend showing transistor operating voltages by year. [Data taken from [DKM+12]]

meaning of this is that it is not possible to utilise these transistors in their highest capabilities. *Utilisation wall* [VSG+10] caused by the end of reduction in operating voltages is one of the major challenges in the field of computer architecture and VLSI research. Operating voltages of microprocessors by year is given in Figure 1.3. Various approaches are being applied to go beyond the utilisation wall such as circuit-level techniques, approximate computing, introducing on-chip accelerators and larger on-chip caches [Tay12]. A second issue is related to lowering of supply voltages. As previously discussed, DVFS relies on voltage/frequency pairs. The number of possible pairs decreases as the gap between nominal and threshold voltages are lowered with each newer transistor generation, thus resulting in a reduction of the effectiveness of the scheme [LSH10].

Another power-saving technique commonly used in microprocessors is *power gating*. Power gating is used to turn off blocks that are not in use to reduce power. One major advantage is that it eliminates static power completely as there is no voltage being applied to the transistors within the block that is gated off. The downside of this is that all the information in the power gated block must be copied to somewhere else in the system or be lost. State-retentive power-gating provides an adjustable balance point between sleep/wake-up speed and power savings. There is research on state-retentive power gating that power gates a whole processor core [JKK+12] or only the register file [RRK11] while waiting for memory accesses in order to save power.

In light of the previous paragraphs, a computing system requires a decision mechanism in order to switch between a performance driven mode and a power-efficient mode that would help in preventing thermal and reliability issues. This can be further enhanced by proposing a computing stack that contains the appropriate elements in each of the layers ranging from circuit-level to the application layer. This PhD thesis proposes a **a simulation-based investigation for an alternative power management technique, called Cyclic Power Gating (Chapter 5)**. Cyclic Power Gating can be utillised as a power management scheme that is able to overcome the shortcomings of VFS on future technologies, and provide better leakage management as it is a form of power gating.

The contributions to the field and the structure of the thesis is summarised in the following sections.

1.3 Publications

The material from Chapter 5 appears in the following journal publication:

- Y. Cakmakci; W. Toms; J. Navaridas; M. Lujan, "Cyclic Power-Gating as an Alternative to Voltage and Frequency Scaling," in IEEE Computer Architecture Letters, 2015

Other publications during time of studies:

- Project Beehive: A Hardware/Software Co-designed Stack for Runtime and Architectural Research. Christos Kotselidis, Andrey Rodchenko, Colin Barrett, Andy Nisbet, John Mawer, Will Toms, James Clarkson, Cosmin Gorgovan, Amanieu d'Antras, Yaman Cakmakci, Thanos Stratikopoulos, Sebatian Werner, Jim Garside, Javier Navaridas, Antoniu Pop, John Goodacre and Mikel Lujan, MULTIPROG-2016

1.4 Thesis structure

The organisation of the thesis is as follows:

- **Chapter 2:** Background material intended to provide the fundamentals of the material presented in this thesis will be covered in this chapter. Topics included are out-of-order microprocessors, power dissipation in microprocessors, power management schemes, workload generation, and performance-power tradeoffs. A review of the literature for power management schemes with a focus on DVFS and power gating is provided. Previous work on power models, and workload generation is also explained.

- **Chapter 3:** Experimental infrastructure used to obtain results within this work will be provided in detail including the methodology used to calculate power and temperature.

- **Chapter 4:** Synthetic workload generation is a technique used to generate benchmarks for design space exploration, computer architectural evaluation, and fault detection. Generator of LLVM Assisted Microbenchmarks (GLAM) is introduced as a novel way of generating synthetic workloads. The phases that encompass workload generation are: code specification, code generation, harness

generation and execution. Each of these phases are explained in detail along with examples.

- **Chapter 5:** This chapter introduces Cyclic Power Gating (CPG) as a novel power management scheme. The state-retentive architecture that is used is presented, followed by an explanation of how the power-gating overheads are modelled. Then CPG period selection is discussed, and a comparison of CPG with VFS is provided. The results are compared with an equivalent VFS setup.

- **Chapter 6:** A summary of the thesis is presented, along with possible future work.

Chapter 2

Background

2.1 Introduction

A *compute stack* is a multi-layered abstraction that splits the hardware and software based on system design requirements. A highly generalised composition would include hardware, systems software, and application layers. System blocks that operate at a digital logic level such as processors, GPUs, interrupt controllers stand in the hardware layer. Since hardware resources are limited, a mediating layer between the application layer and hardware is required in order to allocate these resources for execution. Systems software is the layer that allocates the system resources to applications so that their correct execution is guaranteed. Operating systems, hypervisors, and virtual machines reside in the systems software stack. Although predictions that heterogeneous systems [CMHM10] will be increasingly prevalent in the future to overcome the power wall, contemporary computing systems are still dominated by multiple issue Out-of-Order cores with little or no heterogeneity.

Initially, power and energy efficiency have been a concern for embedded systems. After the benefits of technology scaling in terms of performance increases and power reductions started to diminish, combined with the emergence of energy-hungry data warehouses, power and energy efficiency become a first order design constraint for all computer architectures.

This chapter provides a primer for the material presented in Chapter 4, and Chapter 5. Pipeline stages that constitute the operation of an out-of-order microprocessor will be described. This will be followed by a description of power dissipation in microprocessors that is enough to understand the problems that arise from scaling down of transistors without going into VLSI-level details. The relationship between power,

energy and performance will also be discussed to provide a perspective on how these metrics are inter-related. A list of metrics used throughout this dissertation will also be provided along with their definitions.

2.2 List of related metrics

This section provides a list of metrics used throughout the dissertation.

- **Instructions Per Cycle (IPC):** Provides the number of instructions committed per cycle. IPC is a common metric used to measure the throughput of a processor.
- **Cycles Per Instruction (CPI):** CPI provides the average latency of instructions, in cycles, that have been executed. It is the inverse of IPC.
- **Power:** Power is the energy used per unit time, and is measured in Watts.
- **Average Power:** Average power over a period T.
- **Peak Power:** Maximum dissipated power in a period T.
- **Energy:** Integral of power over a period T.
- **Energy Delay Product (EDP):** EDP is the multiplication of the energy it takes to complete a unit of work by the time it takes to complete that work.
- **Instruction Level Parallelism (ILP):** Average number of independent instructions within an instruction stream.

2.3 Out-of-order microprocessors

Pipelining is an execution technique used to increase the throughput of a processor in terms of instructions executed per unit of time. Pipelined microprocessors were initially *scalar processors* meaning that the execution was limited to a maximum throughput of one instruction per cycle [HP12, SL13]. *Superscalar processors* enabled the issuing of multiple instructions per cycle thus leading to an increase in instruction throughput. Instruction Level Parallelism (ILP) is a metric of how many instructions an application can run in parallel within an instruction window without being stalled

by dependent instructions. The number of instructions that can be issued in parallel is dependent on the ILP inherent in that stream.

The parallelisation of an instruction stream is limited by pipeline hazards [HP12, SL13]. There are three types of pipeline hazards; structural, control, and data hazards.

Structural hazards occur when there are more instructions to be executed than functional units to execute on, leading to the execution of these instructions to be stalled until the functional unit is available. When an instruction is stalled, all the other instructions that depend on the output of the stalled instruction are also stalled. This leads to the stalling of execution for that instruction and the following instructions that are dependent on it until the execution units are freed.

Control hazards are introduced by the inability to predict which branches will be taken resulting in stalls until the branch that will be taken is known. Branch prediction is a method used to overcome control hazards, but branch predictors still lack the accuracy to correctly predict the control flow without errors.

Data hazards occur when the availability of a source operand for an instruction depends on the destination operand of a previous instruction. Given an instruction stream that consists of totally independent instructions, it would be possible to execute all the instructions in the stream in a single cycle in the presence of an unlimited number of functional units. On the contrary, if the instruction stream is structured so that each instruction in the stream is dependent on the previous in order to execute, then it would take a cycle for each instruction to be processed. Classification of instruction dependencies can be grouped into two main classes; true and false dependencies. A true dependency occurs when an instruction's source operand is dependent on the previous instruction's destination operand. This is called a Read-After-Write (RAW) dependency. False dependencies occur in the form of Write-After-Read (WAR) or Write-After-Write (WAW). Although it is not possible to parallelise instructions that contain true dependencies, it is possible to get rid of false dependencies using *out-of-order execution techniques* [SL13] .

An *out-of-order microprocessor* [HP12, SL13] is a superscalar microprocessor that employs mechanisms to resolve false dependencies between instructions and exploit Instruction Level Parallelism (ILP). In this section, a description of how an out-of-order microprocessor pipeline is structured will be presented.

The following subsections will detail out each pipeline stage for an out-of-order core that is given in Figure 2.1.

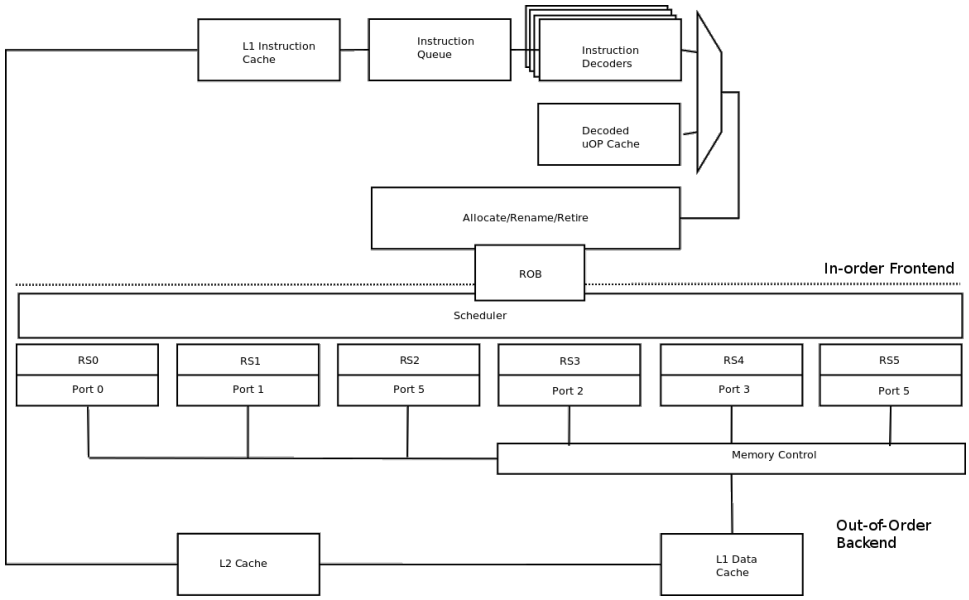


Figure 2.1: Generic Out-of-Order pipeline with in-order fetch,decode and retire stages and out-of-order execution backend.

2.3.1 Overview of an out-of-order pipeline

Figure 2.1 provides a block diagram for an out-of-order pipelined processor. Pipelined processors that can dynamically schedule instructions based on operand availability are able to execute instructions out of program order, but the executed programs are retired from the pipeline in program order. A program is fetched, decoded and issued in-order, but depending on inter-instruction dependencies, the execution of the instructions can be out-of-order. While the solution to a control hazard, in the form of a branch predictor, has a direct effect on where the next instruction will be fetched from, structural and data hazards cannot be resolved during the in-order stages as the information required to detect these hazards is not available at the early stages of the execution [SL13]. The details of the each pipeline stage will be provided in the rest of this subsection. These details are not microarchitecture specific, and are intended to provide a generic introduction.

Fetch

The main activity that occurs in the fetch stage is the filling of the instruction queue. An instruction queue is a FIFO structure that contains the sequence of instructions that may potentially be executed. If a branch predictor is used, instructions are fetched from a Level-1 instruction cache into the instruction queue depending on the estimation of

the next address to fetch instructions from. Otherwise the fetch unit stalls until the next address to fetch from is calculated. Some architectures also include a mechanism that shuts off the fetch stage on detection of loops. This mechanism is known as Loop Stream Detector (LSD) in the Intel architecture. Once a loop is detected, the instructions are cached in a limited size buffer, called a *loop buffer* [Sin08].

Decode

Instructions are read from the instruction queue and decoded so that the operand requirements and executions units are known in advance. Operand requirements consist of knowing whether an instruction produces an output, and the number of source operands. If the operation will produce an output a physical register file entry is generated for the instruction to write its result. In case of an Instruction Set Architecture (ISA) with instructions of variable lengths, the decode logic becomes rather complicated, resulting in a higher contribution to the overall energy consumption of the core. An entry in the Reorder Buffer (ROB), physical register file, or load/store queue is allocated at this stage for the instruction depending on the resources it requires.

Register rename

Register renaming is the assignment of physical registers to eliminate false dependencies. False data dependencies are removed by dynamically assigning different physical registers for the same logical (architectural) registers. The lifetime of a physical register can be summarised in four phases: Allocation, writeback, dependent reads, and deallocation. When an instruction is decoded, a free physical register is allocated for that instruction to write its result. The value produced by executing the instruction is written to the allocated physical register. Subsequent instructions that depend on the architectural register read the value from the renamed physical register. When there are no more references left to the physical register in the pipeline the allocated physical register is freed.

Issue

The issue stage marks the beginning of the execution for an instruction. At this stage, an instruction that is queued in a *reservation station* is issued on to a functional unit for execution. A reservation station, also known as an *issue queue*, is a microarchitectural structure where instructions wait until their source operands or a functional

unit becomes ready. Reservation stations are either centralised or distributed. Despite complexity at the hardware logic level, centralised reservation stations offer the advantage of higher functional unit utilisation as all the information on instructions to be executed is available [SL13]. Distributed reservation stations require less logic, but do not offer as high utilisation as centralised reservation stations, as once an instruction is in a reservation station, it is stuck in that specific queue for the specific execution unit. A good balance between the two is offered by clustered reservation stations where a reservation station covers several execution units of the same type. A dispatch stage precedes the issue stage depending on the implementation of the reservation type. It is responsible for submitting decoded instructions to the reservation stations. There is no need for a dispatch stage in the case of a centralised reservation station implementation.

Execute

Instructions are executed in the related functional unit in this stage. The types of available functional units are ISA dependent. Most ISAs support integer arithmetic through an Arithmetic Logic Unit (ALU), floating point arithmetic through a Floating Point Unit (FPU), memory operations through a Load/Store Unit (LSU). There are also some architectures that require functional units for various Single Instruction Multiple Data (SIMD) instructions.

Completion

The physical register file is updated by writing the result of the recently executed instruction to the allocated register. The Reorder Buffer (ROB) is needed at this stage in order to retire the instructions in-order. The order is maintained by the initial allocation of an entry in this FIFO structure during the decode stage. The value produced by the completed instruction is copied over from the physical register file to the architectural register file. This marks the end of execution for the instruction being executed, and this is reflected in the architectural state of the system.

2.4 Microprocessors as CMOS circuits

Transistors are the building blocks of all the digital devices. Metal Oxide Semiconductor Field Effect Transistors (MOSFET) are the type of transistors that are used to build

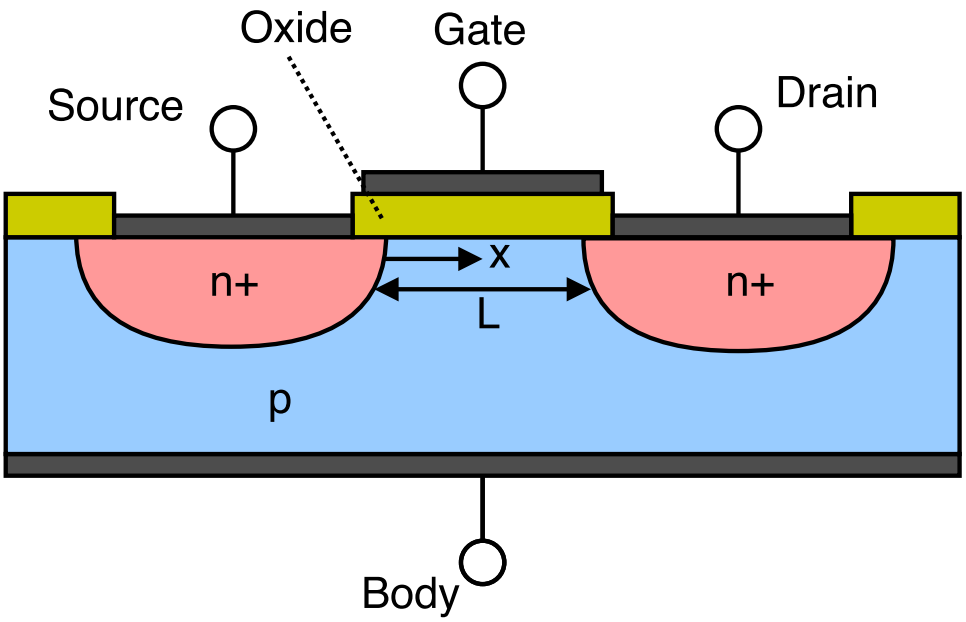


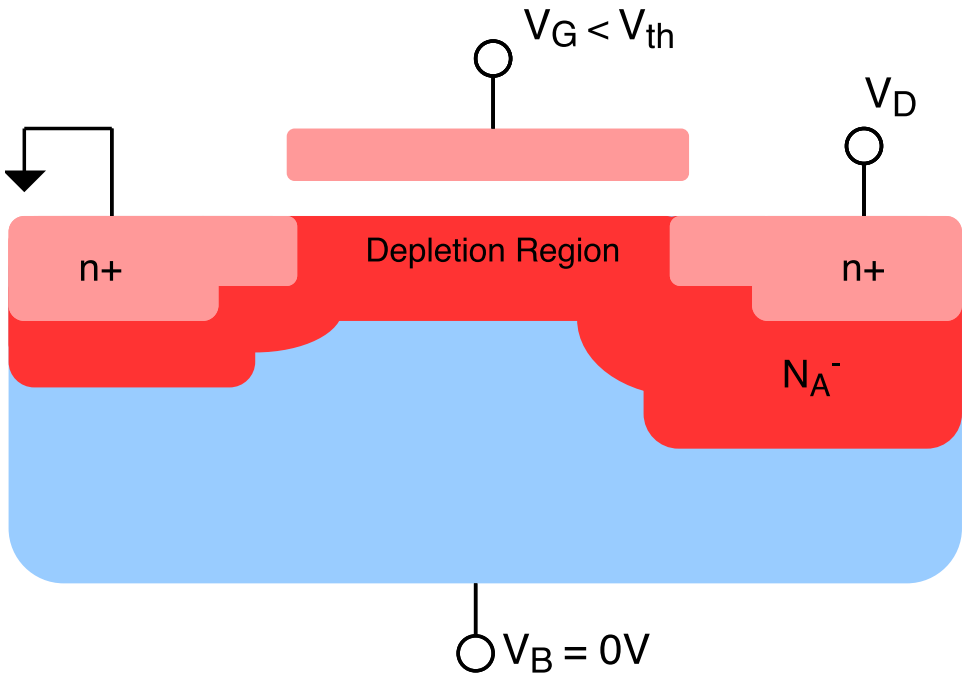
Figure 2.2: Lateral view of an n-type MOSFET transistor, where L is the gate length, and x is the direction of the flow from source terminal to the drain terminal.

Complementary Metal Oxide Semiconductor (CMOS) circuits that make up contemporary microprocessors [WHB05]. MOSFETs are used to control current flowing from one terminal, known as the source onto the end terminal called the drain by applying voltage on a third terminal called the *gate*. An oxide layer below the gate terminal, called *gate oxide*, is used as a dielectric layer insulating the gate from the substrate and other terminals. A diagram showing a cross-section of a MOSFET is given in Figure 2.2.

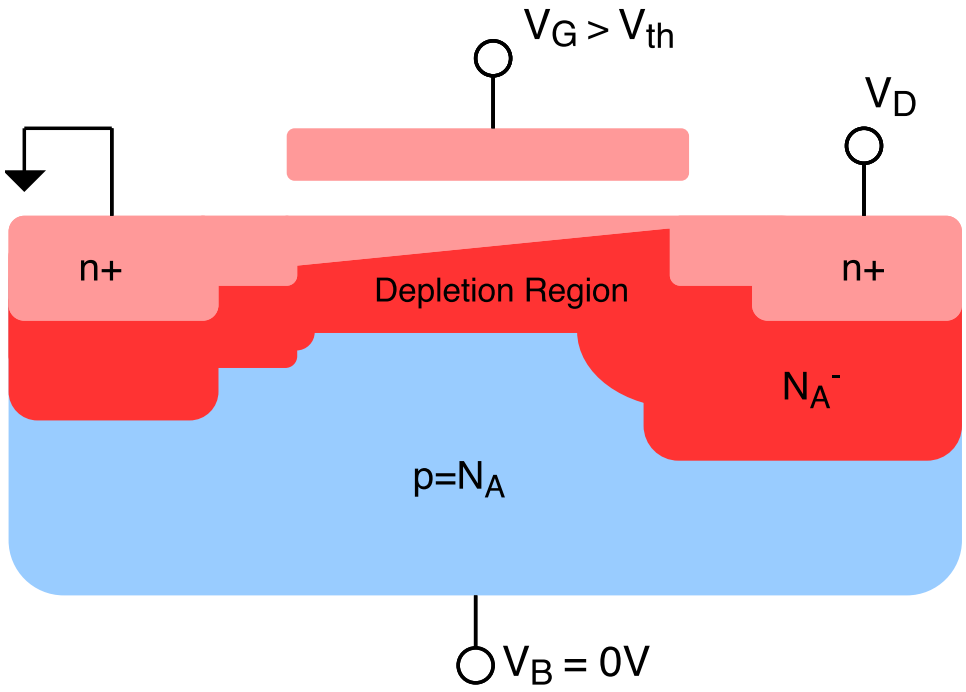
MOSFETs come in two types differentiated by whether the channel induced on the substrate is of positive or negative type. A *p-type* MOSFET carries the current using positive charged particles called holes, whereas a *n-type* MOSFET the current is carried using electrons.

The voltage applied on the gate is known as V_G, V_{th} is the minimum voltage that needs to be applied to enable a conducting path between source and drain, and V_D is the voltage at the drain terminal.. When V_G is lower than V_{th} (Figure 2.3a), the transistor is said to be in an *insulating state* as the resistance between the source and drain is too high for having current flow in between the two terminals. When V_G is greater than V_{th} (Figure 2.3b), current starts to flow from source to drain, and the transistor acts as a closed circuit.

A CMOS circuit is a composition of p-type and n-type MOSFET devices connected

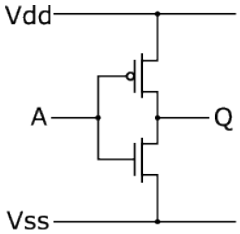


(a) $V_G < V_{th}$

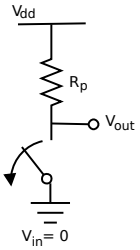


(b) $V_G > V_{th}$

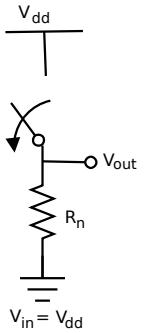
Figure 2.3: Insulating and conducting states of a transistor. Where the gate voltage (V_G) is lesser than threshold voltage (V_{th}), the transistor is in an insulating state. A conductive path is formed between the source and drain terminals as gate voltage becomes higher than threshold voltage.



(a) CMOS Inverter consisting of an n-type transistor and a p-type transistor, where A is the input voltage and Q is the output voltage.



(b) $A = 0$



(c) $A = Vdd$

Figure 2.4: CMOS inverter with configuration in on and off states.

in serial and in parallel to implement the required behaviour of the circuit. A CMOS inverter is shown in Figure 2.4a to provide an example. The inverter is composed of a n-type MOSFET, and a p-type MOSFET connected serially. Figure 2.4b shows the circuit model of the inverter when 0V is applied at input A. At that state the n-type MOSFET is *on* and the p-type MOSFET is *off*, so the current flows between positive supply (V_{dd}) and Q thus asserting the output high. When the input voltage, V_{SS} is equal to drain voltage V_{dd}, the p-type MOSFET is *on*, and the n-type MOSFET is *off*. Q is connected to the ground in this configuration, asserting the output low.

2.5 Power dissipation in microprocessors

Power dissipation of a microprocessor is the sum of static and dynamic power dissipated as given in Equation 2.1. Static power, given by Equation 2.2, is the product of supply voltage (V_{dd}) for the device and the leakage current ($I_{leakage}$) passing through. Dynamic power is directly proportional to circuit clock frequency and the square of voltage, and C is the amount of capacitance required to charge the load capacitance (Formula 2.3). α defines the ratio of switching going on on the chip at a given frequency cycle. For example, when half of the transistors on a chip would be switching at a given cycle then α would be 0.5.

$$P_{total} = P_{dynamic} + P_{static} \tag{2.1}$$

$$P_{static} = I_{leakage} V_{dd} \tag{2.2}$$

$$P_{dynamic} = \alpha C V_{dd}^2 f \tag{2.3}$$

The contribution of static power to overall power consumption is becoming an increasing proportion of overall power as transistors shrink [FDN+01]. Two main factors that make up static power are gate leakage and sub-threshold leakage. Sub-threshold leakage is the power dissipated even when a transistor is below V_{th}, translating into the power dissipation of a transistor when the gate is off. Gate leakage is caused by the direct tunnelling of electrons through the gate insulator.

Supply voltage (V_{dd}) decreases with each new transistor technology in order to obtain a reduction in power density. A decreased supply voltage means that the transistor has to operate at a lower frequency, assuming that the threshold voltage (V_{th}) remains the same. Since low frequency operation is not something chip designers want, a reduction in V_{th} is also made. V_{th} is proportional to V_{dd} and with smaller V_{th}s leakage increases so voltage scaling is no longer possible [BMMF02].

The problem with gate leakage current is that, as the gate oxide gets thinner with newer technology nodes, gate leakage current increases. High-K insulators have compensated this issue at some technology nodes [FASB04].

Dynamic Voltage and Frequency Scaling (DVFS), and power gating will be discussed in the following subsections as the most common means of achieving power and energy efficiency in microprocessors.

2.5.1 Dynamic Voltage and Frequency Scaling

DVFS works by scaling the supply voltage and the operating frequency of the core in parallel. It is important to note that voltage and frequency are dependent on each other, and once voltage is decreased the upper bound of the frequency that a transistor can run at is also decreased. DVFS enabled processors provide a number of voltage and frequency pairs called *operating points* along with a software interface allowing the operating system, or the userspace applications to select operating points based on a criteria such as performance or power saving.

There are two main issues concerning the usability of DVFS in future technologies. The first issue is the contribution of static power to total power consumption as scaling down of the supply voltages is becoming an issue. This means that the importance of the time it takes to finish a task is not the only limiting factor for performance anymore. Although DVFS provides a decrease in power dissipation, this gain is shadowed

by the fact that the actual amount of time it takes to complete a given task takes longer, and this results in an increase in the energy required to complete the task due to higher leakage energy. Since the contribution of leakage power to the total power dissipation increases as a new technology node is adopted, the utility of DVFS comes into question. Secondly, process variability in newer technology nodes, and transistor wear-out results in variant threshold voltage and this also has a limiting affect on the operational range of DVFS [HW04, Bor05].

2.5.2 Recent advancements in DVFS management

This subsection aims to present a subset of the recent research within the domain of DVFS prediction and control. DVFS governors are used to make predictions on system performance by looking at certain metrics within the system such as performance counters, load, or power consumption. These predictions enable the governor to set the processor voltage and frequency to a level that do not introduce a considerable performance hit, while increasing energy savings.

It is possible to control DVFS enabled computing systems in various temporal granularities. DVFS can be controlled by the hardware, operating system, or application layers. The distance between the controlling layer and the DVFS controller is the determining factor in temporal granularity used in decision making for transitioning between different voltage and frequency levels since each layer in between introduces an extra overhead in terms of time. The scale of the controller can be at microprocessor, server, or datacenter levels. Since control can only be achieved through an ability to estimate what is likely to happen, accurate prediction techniques are required to effectively utilise DVFS. High reaction time of the DVFS controllers presented in this subsection can justify the requirement for a low-latency power management scheme. An alternative approach to DVFS have been presented in the Figure 5.2 of this dissertation.

Performance and power prediction

Simplistic performance and power models tend to split execution behaviour into pipeline and memory intervals without taking the variability in latency incurred by memory accesses into account. Eyerman *et al.* [EE10] models execution using pipelined and nonpipelined fractions. A *pipelined* phase consists of the cycles that the out of order pipeline is executing instructions without being stalled. A *nonpipelined* phase begins

when either an instruction or data load miss occurs on the last level cache (LLC) and pipeline execution can not progress until the related cache misses complete. Two energy prediction models are proposed based on whether clock gating is implemented in the system or not. The work assumes that static power consumption of the processor at a given voltage is known in advance, and a hardware counter that provides energy consumption is present. A similar approach is present in the work of Keramidas *et al.* [KSK10] where the pipelined phase is named *steady state*, and the *stall cycles* account for the nonpipelined phase. A new hardware performance counter that counts the total cycles for a *leading load* is presented in [RLSdS11]. A leading load begins when a load instruction misses at the LLC. All the other subsequent loads that issue and complete before the leading load are not counted by the hardware counter as the leading load already covers that period. Stores are not taken into account since most modern microprocessors contain a write buffer that is implemented to hide memory access latencies that are incurred from store misses. The leading load counter is activated by an additional bit that is inserted for each load/store queue entry. Since the leading load counter has been proposed using simulated research, Su *et al.* devises a methodology to implement an equivalent counter on commodity hardware using a combination of five performance counters [SGG$^+$14, SGS$^+$14]. The number of memory cycles is extracted by a combination of counters that track Miss Status Handling Registers (MSHR) occupancy in cycles and cache miss counters. The pipelined fraction is obtained by using the executed cycles performance counter.

Although the models presented in the previous paragraph are accurate in the presence of memory controllers with constant access time, contemporary systems utilise Dynamic RAM where access time depends on the location of stored data. Another issue also arises in the presence of memory prefetchers, which are hardware blocks intended to hide memory latency. Prefetchers fetch data from memory based on a prediction of the memory location that the pipeline will request. Miftakhutdinov *et al.* [MEP12] addresses these issues by introducing a memory access critical path measurement methodology which also takes memory bandwidth into account.

Dynamic concurrency throttling (DCT) is an adaptive execution technique used for scheduling multi-threaded workloads on a system with multiple processing units. The number of threads that can be executed on a multi-threaded architecture are throttled from an application-level scheduler that is distinct from the operating systems scheduler using a prediction model. The application level scheduler provides the number

of hardware threads that it will be needing, and the operating system halts the unused hardware threads if the number required by the application level scheduler is less than the number of available hardware threads, which in turn leads to energy savings. A performance counter events-driven userspace approach for applying DCT has been provided in [CMDAN06]. Useful instructions per cycle, *uIPC*, is defined as the IPC over a phase without the synchronisation and parallelisation instructions. This metric is combined with hardware performance counter events triggered over that phase to provide insight into the amount of computational resources that may be required. Curtis *et al.* [CMSB+08] extends this work by showing that EDP can be decreased significantly by introducing a prediction model that can estimate near-optimal setting for selecting the most beneficial DVFS and concurrency throttling pair. An important point when taking DVFS into account is that the IPC increases when frequency is lowered, while the execution time of the program increases. This is due to the relative completion of memory operations taking shorter time in terms of cycles as core cycle periods increase. This leads to the adoption of instructions per second metric instead of IPC in order to retain the accuracy of the prediction model.

A summary of the predictors is provided in Table 2.1 based on memory models, hardware overheads, and the operational scope of the predictor in terms of spatiality and temporality.

DVFS Governors

A DVFS governor is a mechanism that decides what the most suitable operating point is based on predetermined criteria such as low power, EDP or temperature control. It can either be implemented in hardware or software layers of the compute stack.

A chip can be partitioned into different Voltage Frequency Islands (VFI) enabling each separate VFI with a island-wise operating point. This reduces the granularity of DVFS so that per-core or memory system wide operating points can be set, leading to higher flexibility in system-wide DVFS at the cost of complexity. Various DVFS governor algorithms have been compared for Chip Multiprocessors (CMP) over the execution of multi-threaded benchmarks [HM07]. Results show that having many VFIs may not be worth the additional circuit complexity as the overheads overshadow the gains. In contrast to this analysis, which utilises off-chip voltage regulators, it has been shown that on-chip voltage regulators can provide significant energy savings due to an order of magnitude faster operating point switching speed [KGWB08]. While off-chip regulators can switch operating points at a granularity of microseconds, on-chip

Table 2.1: An overview of DVFS performance/power predictors

Predictor	**Memory model**	**Requirements**	**Granularity**
[EE10]	Constant latency	Stall cycle counter, first pending miss register, power meter, static power values at different voltage levels	Per-core, runtime overhead not provided
[KSK10]	Constant latency	Steady state counter, miss-event interval counter	Single core, 150M cycles prediction window, runtime overhead not provided
[RLSdS11]	Constant latency	Leading loads counter, one bit per load/store queue entry	Single core, Interval based prediction window
[SGS+14]	Constant latency	None	Per-core, 200ms prediction window, negligible runtime overhead
[MEP12]	Variable latency	Global critical path counter per request, Global DRAM slack counter, Per DRAM bank slack counters, Prefetch stall counter	Single core, 100K instruction prediction window, runtime overhead not provided
[CMSB+08]	Not discussed	Dynamic Concurrency throttling algorithm, 9 counters/thread	Multi-threaded, Prediction window not specified

regulators can switch between operating points at nanosecond scale. Since the aim of this work is to present a comparison between the overheads of on-chip and off-chip voltage regulators, an offline DVFS algorithm that calculates the optimal operating point before runtime is used. The results show that for a single power domain, the fast switching provided by on-chip voltage regulators is compensated by the switching overheads. When a similar evaluation is made for four power domains, with each domain representing a core in the CMP, it is shown that on-chip regulators can provide significant improvements.

Power capping is a hierarchical power management method where a peak power limit is specified at a higher level such as a datacenter, with smaller structures such as servers and microprocessors adapting their runtime power budget in coordination with the requirements of the higher level structures. Cochran *et al.* [CHCR11] propose an offline classifier using performance counters, temperature, and power sensors for picking up suitable DVFS voltage and frequency under a specified power budget. The operating point data from the offline classifier is distilled into a lookup table that enables the selection of suitable operating points at runtime. This achieves high energy savings with minimal deterioration in performance.

A model that predicts the system performance by employing hardware implemented Artificial Neural Networks is proposed in [CWL11]. A global resource manager is used to keep track of per-application resource allocations for a chip multiprocessor. Hardware-based ANNs learn to make better allocation decisions through the performance response of each application resulting in higher performance.

A control theoretic DVFS controller that targets temperature control was studied in [WMW09]. The algorithm works by using temperature sensors and an online power estimator to determine DVFS level required to fulfill power budget. Temperature prediction is used to avoid reaching a maximum temperature rather than reducing power consumption.

A fraction of CMPs are equipped with a DVFS controller for the memory system. Coordination of the core DVFS, and memory-system DVFS seperately is inefficient. *CoScale* [DMB+12] proposes an algorithm that executes on the operating system scheduler to enable the coordination of core and memory-system DVFS.

Compile-time methodologies for dynamic power management have been studied extensively in previous work. Dynamic compilation is a run-time technique, where the machine code that executes on the processor is generated on-the-fly at run-time. Wu et

al. [WMC+05, WMC+06] propose a dynamic compilation system that acquires information about the resource requirements of the application that it compiles, and drives the DVFS to lower frequencies if idle time is detected. Multi-threaded workload performance decreases when there are more threads executing in parallel than the number of processing units.

2.5.3 Power gating

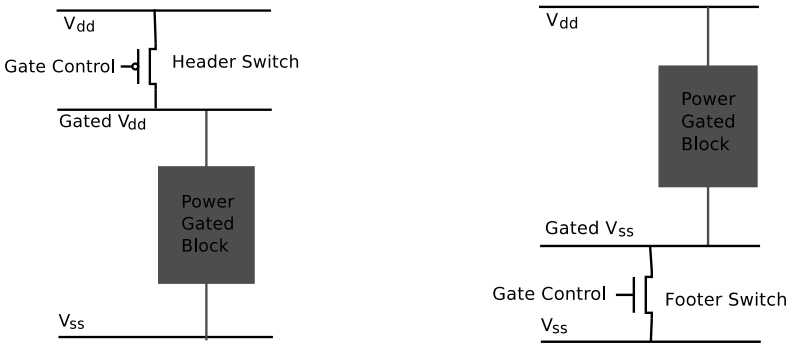
Power gating is a circuit level power reduction technique that cuts off the voltage to the block that is being power gated. Power gating is used to turn off blocks that are not in use to reduce static power. A major advantage of power gating from a power management perspective is that it eliminates static power dissipation completely as there is no voltage being applied to the transistors within the blocks that are gated off, thus the power dissipation of the power gated block becomes non-existent. The downside of this is that all the information in the power gated block must be copied somewhere else in the system or be lost. There is research on state-retentive power gating that power gates a whole processor core [JKK+12] or only the register file [RRK11] while waiting for memory accesses in order to save power.

A sleep transistor is used to cut off and enable voltage on the block to be power-gated. Power gating could be implemented in a coarse granularity using a single sleep transistor to cover a whole logic block. A more fine grained method can be followed by inserting a sleep transistor for each standard cell (Multi Threshold CMOS). The trade-off between these two approaches is between area and control over the power-gated cells. It is possible to power-gate a circuit block using two different configurations using either a header switch (Figure 2.5a), or a footer switch (Figure 2.5b).

The header switch configuration is established by placing the sleep transistor between V_{dd} and the block to be power-gated. A PMOS transistor is used in header switch configurations. When a footer switch is used, the sleep transistor is placed between the block to be power-gated and V_{ss}. Footer switches are implemented using NMOS transistors [Kea07].

Power gating schemes for in-order cores

Detecting pipeline stalls in in-order cores is a trivial task, since the only factor that leads to a pipeline stall is high-latency memory accesses. Power gating for in-order



(a) Header Configuration (b) Footer Configuration

Figure 2.5: Power Gating scheme where the power gating of the circuit is controlled at positive (V_{dd}) or negative supply(V_{ss})

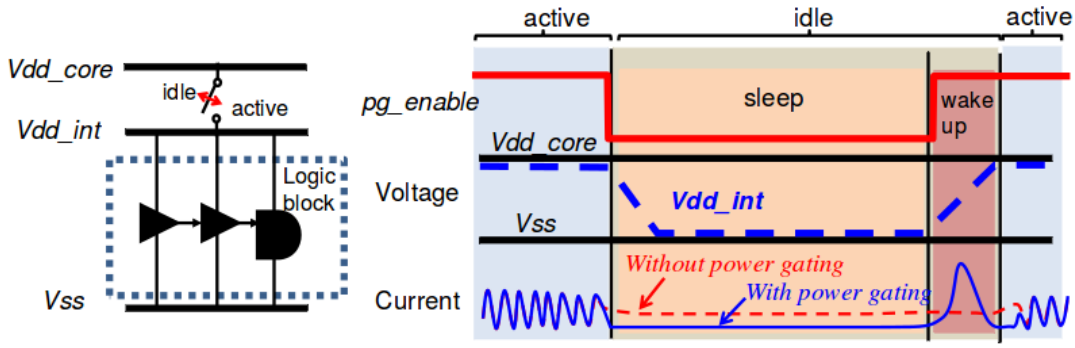


Figure 2.6: Power gating scheme (Taken from [JKK+12]). When pg_enable is asserted low, the supply voltage to the power gated block is cut off.

cores has been examined in [RRK11, ISK+09].

An investigation of state-retentive power gating during memory accesses in in-order multi-threaded architectures was carried out in [RRK11]. A two level sleep configuration that power-gates the register file is proposed with the first mode being highly-responsive at the expense of lesser leakage savings, and the other being a high wake-up latency high leakage saving mode. The first mode of sleeping is applied when an L1 miss is detected, whereas the second mode is activated after the detection of an L2 miss.

Jeong *et al.* [JKK+12] provides a low-overhead method for reducing leakage power dissipation in in-order cores during long memory-stalls by introducing a state retentive power gating architecture called Memory Access Power Gating (MAPG). MAPG is composed of a Programmable Power Gating Switch (PPGS), and a controller designed to power-gate the core when the execution is not progressing due to memory-stalls.

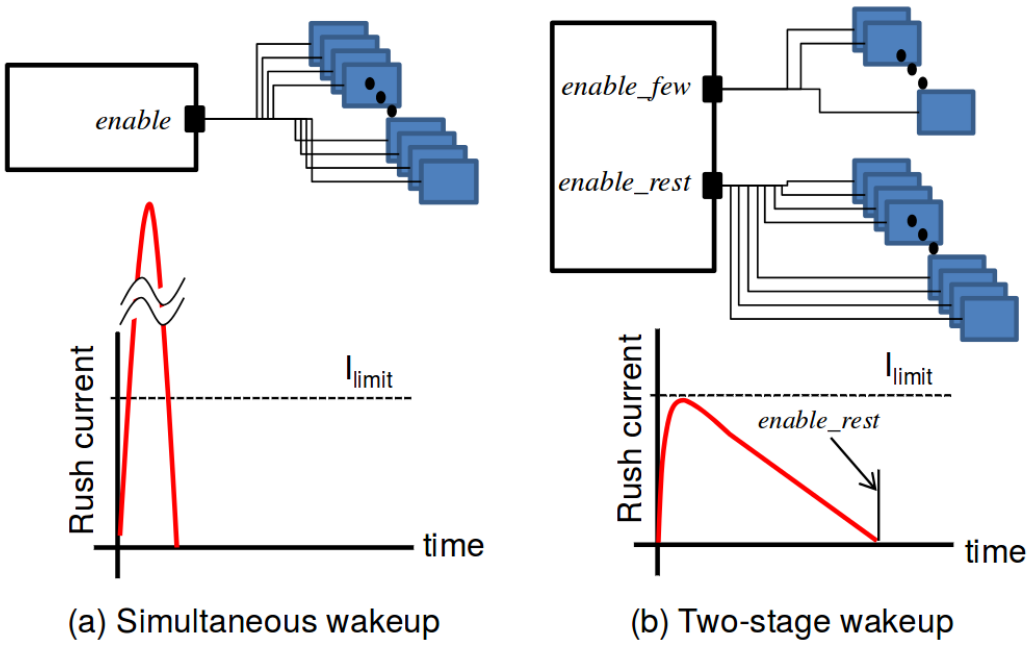


Figure 2.7: Handling of in-rush current (Taken from [JKK+12])

The operation of a simplified power gating scheme, as shown in [JKK+12], is given in Figure 2.6. The scheme works by dividing the current paths between supply voltage (V_{dd_core}), and ground (V_{ss}). When the signal to enable power gating (*pg_enable*) is asserted, the supply to the virtual voltage domain (V_{dd_int}) is cut off, and all the transistors within that domain lose their state along with their static power dissipation. The difference between a state-retentive power gating design and a simplistic power gating design is the introduction of state-retentive registers, that retain the state of the logic block during sleep, and restore the state before wake-up. An important design consideration to keep in mind while implementing power-gating is the handling of in-rush current. *In-rush* current is defined as the rushing of current into an electrical circuit once it is turned on. A larger current could result in dysfunctional behaviour or may lead to the burning out of circuit elements. In order to overcome these risks the header switches are partitioned into groups and powered up sequentially, and the partitioned voltage domains are powered up in groups as shown in Figure 2.7. The guarantee of reliability that comes from decreasing the current comes at the price of *wake-up latency*. The time (T_{charge}) taken to charge the capacitive elements (Q) of a circuit is a function of the in-rush current (I_{limit}) as given by Equation 2.4:

$$T_{charge} = Q/I_{limit}. \tag{2.4}$$

PPGS divides the core into two power domains; *collapsible* and *non-collapsible*. The non-collapsible domain is responsible for powering of the state retention registers, and clamp circuits that enable retention of data. Another issue that adds to the wake-up and sleep delays is the time it takes to save and restore the data from the state-retentive registers. This is done by gating off of the core clock, and then asserting the clamp signals to enable data retention. The inverse of this process is followed when restoring the state, where the data is restored back after the gated power domain is reenabled.

Calculation of safe wake-up modes for the core are performed through the construction of a Power Delivery Network (PDN) model for cores using 22nm and 32nm technologies with high performance and low-operating power devices. The total charge for core logic, excluding caches, and interconnect capacitance is modeled using Equation 2.5, where Q_{core} is the total charge, C_{logic} is the device capacitance, and C_{int} is interconnect capacitance. With the given parameters, the minimum wake-up latency for a single power domain is given by Equation 2.6. The wake-up latency induced by the two-stage wake-up methodology then becomes $2 * T_{min-charge}$.

$$Q_{core} = (C_{logic} + C_{int}) \times V_{dd\_core} \tag{2.5}$$

$$T_{min-charge} = Q_{core}/I_{limit} \tag{2.6}$$

The authors conclude the work by introducing MAPG controller, which is a state machine with three states: active, stall, and idle. *Active* state is when the core is in execution mode, or waking-up from a power-gated state. *Stall* state is when the execution is not progressing forward due to the handling of memory dependencies. Memory dependencies are easily detected by setting a timer each time a memory instruction is being executed, and observing whether the time passed is greater than the latency of a last-level cache miss (i.e. memory access). This method is convenient for in-order cores that MAPG is implemented on, but would not be feasible for out of order execution since memory requests can be executed in parallel, and efficiently estimating memory stalls in this case is not trivial. The *idle* state is when a core is halted as there is nothing to execute for at least 100 microseconds. This approach is shown to provide energy savings of 20% on average compared to nominal execution.

Power gating in superscalar cores

The general approach when power gating certain blocks in a superscalar microprocesor relies on mechanisms to detect the idleness of that particular block. Other situations

such as avoiding thermal emergencies are also taken into account in various schemes as described below.

In a system with fewer threads than cores, a thread in a hot-core can be migrated to a cooler one and the migrated core can be power-gated for thermal management purposes using a software controlled approach [CCF+07]. Thread migration for forced power gating becomes a problem, when the number of tasks that are running is more than the number of cores available in the system due to performance considerations. Vega *et al.* [VBB13] proposes applying Per Core Power Gating (PCPG) through the use of an operating system scheduler that employs a heuristic which measures the difference between tasks to cores mappings by looking at the ratio of per mapping performance (IPC of all running threads) over per mapping power dissipation.

Idle-cycle injection (ICI) is a method that is used to force a core into an idle state. A control algorithm that injects idle cycles for temperature reduction is proposed in [SMC+13].

Decreasing the instruction fetch bandwidth of a superscalar core using fetch throttling was found to provide energy savings due to a decrease in speculative execution penalties [UKM02]. A Branch prediction guided technique is proposed for the power-gating of floating point units in [HBS+04]. Battle *et al.* [BHHR12] propose a register file power gating methodology using a reference counting register file that is modified to detect unused registers.

An application for the power gating of unused register subarrays in a GPU is proposed in [JRKA15]. Power gating in superscalar cores requires mechanisms to detect stall prediction. Arora *et al.* [AMP+15], provide a comprehensive idle-time analysis for benchmarks that utilise CPU and the GPU simultaneously. The findings indicate an absence of long idle times in most of the benchmarks, and claims that there are no situations in which conventional power-gating would lead to energy savings.

Hu *et al.* [HBS+04] proposes architectural techniques for reducing leakage power by power-gating execution units. They first provide analytical equations for estimating the *break-even point*, which is the point where the overhead of power-gating becomes equal to the energy saved by power-gating, and then introduce power-gating schemes where the convenient times for sleeping of execution units are detected by looking at idle periods and branch mispredictions.

The time points that indicate important power-gating intervals in this work are given in Figure 2.8.

The decision to power-gate the circuit is taken when an idle interval is detected

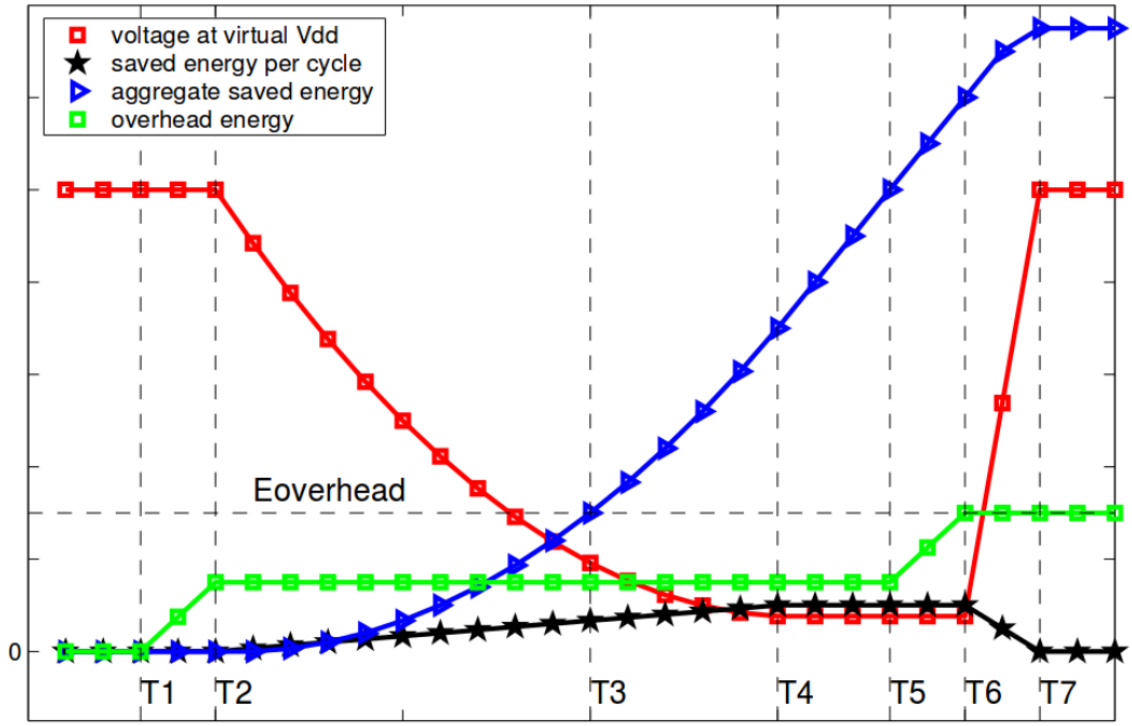


Figure 2.8: Power gating cycle intervals (x-axis represents time, and y-axis represents energy)(Taken from [HBS+04])

at the time between $t = 0$ and $t = T_1$. The leakage energy up until T_1 amounts to the energy while the unit to be power-gated is in active mode. The sleep signal is asserted at T_1, and is distributed to the header switch responsible for cutting off the supply voltage to the unit to be power-gated. T_2 marks the time when the sleep signal reaches the gate of the header switch. The energy consumed between T_1 and T_2 is $E_{overhead_1}$.

When the sleep signal reaches T_2, the virtual supply voltage is cut off and the charge within the power-gated block starts to decrease as a result of the decreasing voltage. The virtual supply voltage drops superlinearly until T_4, where the voltage drops to almost zero resulting in an increase in leakage energy savings.

An upcoming busy interval is detected at T_5 through the use of control logic. The deasserted sleep signal leads to an energy overhead ($E_{overhead_2}$) caused by the propogation of the signal. The signal reaches the header switch at T_6, thus leading to the charging of the virtual supply voltage. The leakage energy savings per cycle decrease until it becomes zero when the virtual supply voltage is fully charged at T_7.

The break-even point T_3 is the point where the overhead of power-gating ($E_{overhead} = E_{overhead_1} + E_{overhead_2}$) is equal to the energy saved by power-gating. Aggregate energy

Table 2.2: An overview of power-gating schemes

Scheme	**Core type**	**Event**	**Power-gated Blocks**	**Wake-up overhead**
[RRK11]	In-order	Cache miss	Register file	5 cycles
[JKK$^+$12]	In-order	Cache miss	Core	$\sim$10ns
[VBB13]	Out-of-order	Migration heuristic	Core	20ms
[SMC$^+$13]	Out-of-order	Thermal	Core	Varies by processor
[HBS$^+$04]	Out-of-order	Execution unit idle	Execution units	9 cycles
[BHHR12]	Out-of-order	Register file bank idle	Register file bank	17 cycles
[JRKA15]	GPU	Register subarray unused	Register subarray	10 cycles

savings are equal to the energy overhead for the switching on and off of the header device at this point.

The results show that the time based technique can power-gate the floating point unit for 28% of the execution time, where as the branch prediction guided scheme can power-gate the floating point unit for 40% of the time. Performance degradation for both schemes is shown to be at 2%.

A summary of power gating schemes for different architectures covering different microprocessor blocks is provided in Table 2.2. As previously mentioned, power gating an in-order core based on a predefined event is trivial since it can easily be known *a priori* which events will lead to pipeline stalls. This problem is transformed into a non-trivial problem when the core type that is to be power-gated is changed into an out-of-order core. This is mainly due to the inability to accurately predict whether a memory access will lead to a pipeline stall, and the period of that stall. Memory stall prediction remains an open problem for out-of-order cores, and even state-of-the-art predictors applied for DVFS control are accurate to 65% [MEP12]. Another issue with performance event-centric power-gating is in their inability to be utilised as actuators for control algorithms if thermal control was required. Cyclic power gating proposed in Chapter 5 can be used in such a scenario, since it is equipped with a low switching

overhead, and is not tied to a certain event by design.

2.6 Performance and power tradeoffs

Different approaches for power management can be taken depending on the organisation of the compute stack. It is possible to implement approaches that are hardware based, software driven or a combination of both.

It is useful for the processor to be aware of the class of running applications in order to take power management decisions. When the software stack is provided with an interface to the hardware layer, power reduction and dynamic power management techniques are utilised more efficiently. A holistic approach where all the layers have the appropriate interfaces to access system-wide information about the runtime state has proven beneficial in many of the previous research [JNaS+12, DMB+12, CRG+11, SKK11, LMC+11].

Despite differences in pipeline depth and pipeline stages, all superscalar microprocessors contain both in-order and out-of-order pipeline structures. Most contemporary commodity superscalar processor designs contain a backend that executes out-of-order, and an in-order frontend that contains the fetch and retire stages. Structures that are required to maintain program order at fetch and retire stages. Out-of-order structures are designed to exploit instruction level parallelism (ILP) that is inherent in sequential programs. ILP is a metric of how many instructions an application can run in parallel within an instruction window without being stalled by dependent instructions. Instruction dependencies are classified in three types; write-after-write (WAW), read-after-write (RAW), write-after-read (WAR). WAW and WAR are called false dependencies, and these types of dependencies can be omitted by register renaming where a different physical register is assigned for the same logical register in order to break the dependency [HP12, SL13].

The maximum achievable ILP is determined by the largest sequence of independent instructions, once false dependencies have been removed. Listing 2.1 shows two different instruction streams with Listing 2.1a showing a stream where no ILP exist, and Listing 2.1b showing a stream with an ILP of 4. Another way of thinking about ILP, is looking at the distance between the last write on a register, and the first subsequent read on that register. This is also called *instruction dependency distance* [SL13].

There are false dependencies that can be handled by compiler techniques (i.e. instruction scheduling), and hardware techniques (i.e. register renaming) [SL13]. True

```
r8 = r9 + r10
r6 = r7 + r8
r4 = r5 + r6
r2 = r3 + r4
r0 = r1 + r2
```

a: ILP=1

```
r0 = r1 + r2
r3 = r4 + r5
r6 = r7 + r8
r9 = r10 + r11
r12 = r1 + r0
```

b: ILP=4

Listing 2.1: Instruction streams with differing ILPs

dependencies are the cause of serialisation within the pipeline, and they are one of the main causes that lower the ILP of an instruction stream. In a given instruction stream, that executes on an ideal out of order pipeline, instructions are only serialised in the presence of true dependencies (Read After Write dependencies). ILP provides a theoretical upper limit for a given program bound by a certain ISA. However the issue and retire width of a processor bounds the maximum ILP that can be exploited. An optimal execution on a given processor design would be when the number of instructions executed per cycle is equal to the issue/retire width of the processor specification. In reality this is not possible mainly due to two reasons; memory access latency, and structural hazards.

The stalls in a pipeline due to memory access latency are called *memory stalls*. In an ideal memory subsystem, where data locality is not an issue, instructions in the pipeline would not be stalled while waiting for memory as all the memory requests would be filled immediately. What happens in an actual system is that each memory request issued through load/store instructions in the pipeline is forwarded into the memory hierarchy. The request is served within a time period depending on the layer of memory hierarchy the requested location resides in. This can range from few cycles if the request was filled from the L1 cache upto hundreds of cycles if it needs to be fetched from memory.

Another type of stall is the *pipeline stall*, which happens when application requirements could not be met by the underlying processor (i.e. lack of parallel execution units for a given task, low decode throughput, instruction dependencies, or a combination of these) [SL13].

While out-of-order execution is able to mask some of the memory access latency, there is not much the core can do in the case of heavily memory-bound program phases where most of the time spent during execution is spent within the memory hierarchy.

This latency provides opportunity for exploiting memory-bound phases for energy-efficient execution. There are different techniques for memory latency reduction in order to decrease the memory access time, or reducing core performance during these phases that both result in energy savings. Compute-bound workloads make heavy use of the backend and the frontend, whereas most of the execution time is spent outside the core in memory-bound workloads.

An *application phase* is defined as a period of execution with a degree of homogeneity with respect to the behaviour of the processor. A phase is deemed memory-bound when most of the execution time is spent in the memory system as opposed to being spent performing computation in-core. The optimal execution speed for compute-bound workloads is the maximum core frequency available [AA14] in the system. Memory-bound workloads on the other benefit from power savings on a lower core frequency [SKK11].

The degree to which a program is memory-bound is highly dependent on the data set that is provided as input to that program. Although it is possible to know whether a piece of code interacts with the memory system by only looking at its instructions, it is not generally possible to know in advance which layer of the memory hierarchy will respond to these requests.

While in memory-bound phases most of the work is being done by the memory system, compute-bound phases complete their work within the core. This means that the related subsystem that the work is being done on is the main contributor to power dissipation.

The time it takes to bring data from higher levels of the memory hierarchy is usually much higher than out-of-order execution can compensate for, resulting in the pipeline being stalled most of the time waiting for data. This means that the core does not need to run at high frequency, as memory latency becomes the bottleneck of execution. From an energy consumption perspective, execution of memory-bound workloads on lower core frequencies lead to higher energy savings compared to higher frequency execution.

When the memory system is not involved, in an ideal compute-bound phase, the most energy-efficient way of execution occurs when the program being executed runs uninterrupted at the maximum available frequency. Various layers in the compute stack can incur overheads by interrupting program execution. As an example, the operating system level interruption of the execution can be caused by a context switch. This might result in the instructions of the program residing in the instruction cache being

flushed leading to an unnecessary batch of memory system operations. Running at slower frequencies (in the presence of DVFS) will also result in higher execution times, and this performance hit may lead to higher static energy dissipation.

2.6.1 Power and performance modelling

Modelling of power and performance is useful during early design space exploration of new processor designs, and for runtime execution profile (such as power saving, high-performance, ...) selection. This subsection provides several approaches that can be used at runtime or early design time. The work presented here can be used in tandem with the novel microbenchmark generation tool described in Chapter 4 to generate power models, and be exploited to provide online decision mechanisms for the power management scheme described in Chapter 5.

Accurate and stable runtime power modelling for mobile and embedded CPUs

Walker et al. [WDH+16] provides a methodology for building runtime power models for mobile and embedded devices by using hardware performance counters (HWPC). The methodology consists of four steps that are: data acquisition, HWPC selection, model formulation, and a CPU voltage model.

The data acquisition step is where data for different HWPCs are collected by running a variety of workloads for the mobile domain. The infrastructure built for data acquisition takes different voltage/frequency pairs available within the processor for more precise measurements.

Most processors have a limit on the HWPCs that can be monitored simultaneously. The authors propose a novel statistical HWPC selection methodology in order to overcome this problem. This work also points out that the CPU voltage varies based on the workload being executed, and claims that it is important to model the voltage regulator for high precision modelling of static power. Finally, the authors compare the results produced by their models with the actual measurement data, and find that their models accurately represent the power for ARM Cortex-A7 and ARM Cortex-A15 cores, with an error of 3.8% and 2.8% respectively.

Roofline model

Williams *et al.* [WWP09] proposes a performance-based roofline model that is motivated by the fact that the work done on any Von Neumann-based processor, despite

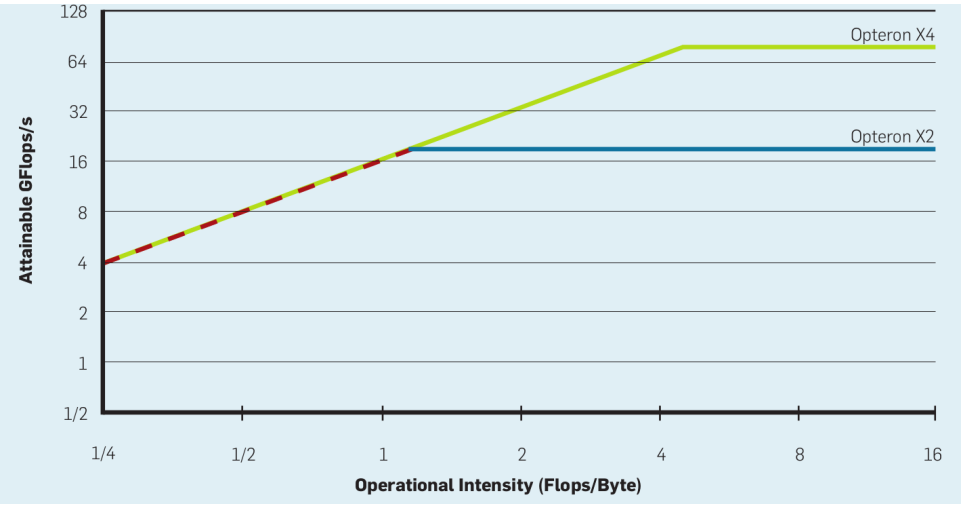


Figure 2.9: Performance Roofline Model (taken from [WWP09]).

architectural differences, is composed of memory and numerical operations with constraints on peak performance dictated by the architecture. The authors argue that the higher level of accuracy that comes with stochastic analytical models, and statistical performance models rarely provides the means for understanding how a program can be optimized, thus limiting their usability to only a group of experts within the domain. Performance-based roofline models connect the floating-point performance W(Flops) with the memory performance Q(bytes) using a metric called *operational intensity* defined as I (Flops/Byte).

Operational intensity can be thought of as a scaling factor, with a higher value meaning higher scalability. Figure 2.9 shows an example for two different versions of AMD Opteron, with the difference in both processors only being the number of cores, the X2 being a dual-core system, whereas the X4 is a quad-core that has the same microarchitecture as X4. From the model's point of concern they both have the same memory bandwidth, and only differ in floating point operation bandwidth. The x-axis here represents the operational intensity (per operation memory bandwidth), while the y-axis represents the attainable floating-point performance. Any possible implementation of an algorithm is represented by a vertical line drawn on the x-axis. The roofline given in the figure provide the upper bound for these two architectures, and means that any algorithm implemented on these machines can not be optimised to achieve values above this roofline.

[CP14] extends the described roofline model by including microarchitectural parameters for the cache hierarchy and out-of-order execution.

McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures

McPAT is a hierarchical software modelling tool that enables power, area and timing aspects to be modelled at a detail ranging from architectural to wire level [LAS+09]. All the components that are available in current microprocessors such as interconnects, multithreading, caches, and memory controllers can be included in the models. It is also possible to have McPAT communicate with performance simulators, temperature and reliability modelling tools using the XML based interface. This enables evaluation of systems with power management schemes such as DVFS. The hierarchical structure of McPAT lets researchers work at the level of abstraction that is required of the research idea. Unspecified parameters can be inferred by the optimiser. The power model relies on capacitance values since dynamic power is a function of load capacitance and voltage, whereas the timing model calculates the RC delays requiring wire resistance values. Area models are calculated by taking memory structures, and combinational logic structures like decoders into account.

2.7 Workload generation

Synthetic workloads, also called microbenchmarks, are dummy programs aimed to exhibit execution behaviour that is targeted to generate activity within a subset of microprocessor blocks which enables testing of the microprocessor in situations such as peak power or reliability issues. These workloads also enable evaluation of architectural proposals in a faster manner in terms of execution time when compared with regular benchmarks. This section introduces the state-of-the-art in synthetic workload generation. The presented content aims to provide the basis to explain why a new microbenchmark generator tool (GLAM), as described in Chapter 4, is required. GLAM is proposed to enable microbenchmark generation at the intermediate representation level in an architecture independent fashion by leveraging the LLVM IR. A comparison of GLAM against the presented material will also be presented at the end of the section.

CERE: LLVM-based codelet extractor

CERE [CAP+15] is a codelet extractor that generates an LLVM IR representation of instrumented binaries. *Code isolation* is a method for capturing the essence of the work

a given computer program completes, and recreating that for reducing the time and resources it takes for benchmarking and compiler evaluation purposes. The challenges in code isolation include different language support, repeatability on different architectures, and accuracy at both computation and memory levels. [CAP+15] propose CERE as an LLVM IR level code isolator that can outline hotspots within a program written in any programming language that has a LLVM supported frontend. CERE captures the execution behaviour of a given program, and generates an output that is a highly accurate representation of it.

Automated microprocessor stressmark generation

Joshi et al. [JEJI08] propose an automated way for generating synthetic workloads aimed to stress the processor (stressmarks) for maximum power dissipation, maximum temperature and maximum change in current. This is accomplished by initially creating an abstract workload model (AWM) by collecting statistics over an actual benchmark execution. The AWM incorporates features related to instruction mix, instruction-level parallelism (ILP), instruction/data footprint, and data access patterns. Instruction mix represents variable latency integer and floating point instructions, along with memory and branching instructions from the dynamic instruction stream of an executed workload. The ILP feature for the model is created by generating a cumulative distribution of instruction dependency distances spread over eight buckets. Instruction footprint is acquired by measuring unique instruction references, while the data footprint is generated by investigating the unique address references within the program. Data stride is modelled as a distribution of localised data strides as introduced in [Jos07]. The modeling technique is motivated by the fact that quantifying data locality using a simple distribution is inadequate for workload synthesis, thus a characterisation that contains various data access patterns at instruction granularity is required. The stride value for each load and store instruction is measured, and then the most frequent value for each individual instruction is recorded along with the percentage of that stride within the measured individual instructions' accesses. If an instruction is observed to have over 80% stride similarity, it is labelled as *strongly strided instruction*. A cumulative distribution of the most frequent stride patterns is generated along with the ratio of the memory accesses that each of these patterns represent. Branching behaviour is modelled by generating a branch transition rate, which is the ratio of switches between taken/not-taken branches divided by the total number of branch instructions. A moderate branch transition rate indicates an irregularity

within branching behaviour, which decrease the likelihood of correct branch prediction.

A workload is synthesised from an AWM through five steps. In the first step, a linear block chain is generated based on the average basic block size, where the instruction footprint obtained from the AWM is used to calculate the size of each block along with the population of the individual blocks using the obtained instruction mix characteristics and ILP metric. A second step involves the modeling of memory access patterns, where each memory access instruction is assigned a stride value from the stride distribution explained in the previous paragraph. Each memory access walks through a circular buffer in strides specified by its assigned stride value, where the size of the buffer is dependent on the ratio of the program's data footprint divided by the total number of load/store instructions within that program. The branching behaviour is modelled in the third step by assigning a transition rate for each branch instruction that was inserted in the first step. The fourth step assigns registers to each register based on the instruction dependency distances from the AWM. Finally, code is generated by emitting assembly code into a C source file skeleton that targets a specific ISA. Stressmark generation involves the utilisation of a genetic algorithm in conjunction with the described workload synthesis methodology, where a benchmark is synthesised from an AWM in iterations until a benchmark that stresses the objective function (maximum change in current, maximum temperature, etc) the most is found.

The workload synthesis method is evaluated to be accurate with an average error of 10.9% for replicated SPEC CPU2000, and Specjbb2005 benchmarks. Generated stressmarks are evaluated to meet the requirements in generating the maximal amount of stress defined by the objective function.

Systematic energy characterization of CMP/SMT processor systems via automated microbenchmarks

Bertran *et al.* [BBG+12] propose MicroProbe as an automated microbenchmark generation framework, and shows its uses for application phase grained power consumption by calculating the contribution of each component for an IBM POWER7 CMP system with multithreading capabilities. The tool is also utilised for measuring energy per instruction (EPI), and generating stressmarks that target maximum power dissipation. The framework consists of architecture, code generation and design space exploration (DSE) modules driven by user-defined policies intended for validation of research ideas. The generation process is driven by a python interface which provides

the user with a high-level of flexibility. The architecture module provides portability by decoupling the architectural properties from the code generation module by having the user input the architectural details using text files. It implements three functions: an ISA definition for the PowerPC architecture, the microarchitectural definition for the POWER7, and a set-associative cache model. The ISA definition enables the generation of assembly code for the defined ISA, and can be input by the user in a text format. Microarchitecture definitions contain information on processor components like functional units, caches, and layout. The partial information contained within the microarchitectural definition is completed using an automatic bootstrapping scheme that generates instruction level profiling information such as latency and instruction to block mappings. EPI is also measured during this phase using the available on-chip power sensors. The microarchitectural models are used to reduce the time it takes for DSEs by providing the related microarchitectural information, thus avoiding the need to apply an exhaustive search to infer these properties. A sample cache model proposed in this work is the set-associative cache model that defines the cache hierarchy for the POWER7 microarchitecture. Using the model definition makes it possible for the code generator to model the exact cache behaviour of the architecture that the microbenchmark will execute on. Each cache level in the hierarchy is assigned a single cache set responsible for generating the hit/miss ratio for the given level. The code generation module follows a five-step process for generating the target microbenchmark. These steps are: skeleton definition, instruction distribution, memory behaviour modeling, branching behaviour, and ILP modeling by defining the instruction dependency distance. The framework also makes it possible for the user to add new passes as new requirements arise. The authors claim that abstract workload models do not expose the ability for instruction type selection, whereas the compiler-like code generation module of MicroProbe allows this by introducing an instruction selection pass. The DSE module enables the user to specify a design space, along with the algorithms to search within it. It is required for finding benchmarks that meet certain criteria dictated by the research idea that is being investigated. It allows for exhaustive searching within the DSE, as well as genetic algorithm searches. The integration of DSE support with the code generator makes it more convenient for implementing a feedback-driven generation method. A performance counter-based power model for the POWER7 is also proposed. This model is built in a bottom-up fashion, defining system power as the sum of the power consumption of individual components. They break down the total power consumption into four dimensions defined over idle power dissipation.

These dimensions are dynamic power dissipated per hardware thread, power overhead of utilising symmetrical multi threading (SMT), power overhead of running multiple cores simultaneously, and the uncore power dissipation. The work is concluded by claiming an average error below 2.3% for estimating the power consumption for the SPEC CPU2006 benchmarks, and stressmarks that are 10.7% power-inefficient over the observed SPEC CPU2006 benchmark runs.

Workload synthesis: Generating benchmark workloads from statistical execution profile

Kim et al. [KLJR14] generates an execution profile for a given workload by utilising hardware performance counters, and outputs a representative synthetic workload with a reduced execution time compared to the actual workload. The work targets the mobile application space, where it is a burden within the research community as benchmarking Android applications are non-trivial due to their closed source nature combined with their inherent interactivity. A two-step methodology that consists of a sampling stage and a regeneration stage is used. The application to be replicated is profiled by sampling performance counters periodically, where each interval is named a *workload slice*. The period for each workload slice is defined as *slice length*. By taking workload slices through an execution, a *tracing log* that indicates performance counter results per slice over time is constructed. The profiled program is regenerated by concatenating code templates (*kernel functions*), that are representative code sequences for a given performance counter class for each workload slice. This is achieved by making use of a synthesis solver, which basically maps kernel functions to workload slices based on the execution profile of each workload slice. The distinct classes of kernel functions consist of: ALU, branching, memory, idle, branch miss and cache miss. The shortcoming of the kernel functions for memory functions is that they only utilise L1 hits and misses. The replicated workload has a 18% discrepancy in IPC compared to the actual workload.

2.7.1 A comparison of GLAM with the presented tools

From an operational and functional point of view all of the presented materials in this section bear certain similarities with GLAM. The LLVM based codelet extractor

[CAP+15] is similar in that it operates at the LLVM IR level to isolate code structures from a larger program body that could later be executed in a standalone fashion, thus reducing execution time while still retaining the execution properties of the isolated section, but the isolated code sections are still too coarse-grained to be labelled microbenchmarks. A second issue was the limitation of this tool to the Intel x86 architecture at the time this dissertation was written. The workload synthesisers [JEJI08, KLJR14, BBG+12] aim to generate microbenchmarks with similar execution properties as GLAM, since both of these tools target maximal utilisation of selected microprocessor blocks. Where they differ is their approach in doing so, in which GLAM microbenchmarks are generated directly from a user provided definition, whereas the automated microprocessor stressmark generator relies on an abstract workload model generated through runtime profiling. In addition these tools require a backend for each new output architecture, as opposed to GLAM, where generation for the LLVM IR can easily be transformed into all of the LLVM supported architectures. An in-depth description of GLAM can be found in Chapter 4.

2.8 Summary

This section introduced the basics of out-of-order execution, power dissipation in microprocessors, power management schemes, the relationship between power dissipation and performance, and automated workload generation. The current landscape in these areas were briefly explained, and the opportunities for improvement have been pointed out.

The concerns that underlie the utility of DVFS in the future are mainly due to the end of transistor scaling. Increased leakage power, and the reduced difference between supply and threshold voltages are the main reasons that lead to these concerns. Power gating is a sure way of reducing leakage power, and has been applied widely within research and commercial domains. Recent developments revolving around these concepts were provided. A novel power management scheme, and how it compares with a recent technology node, is provided in Chapter 5. Since research on microarchitecture level proposals require a simulator, and the execution of benchmark suites on simulation environments take too long, a novel microbenchmark generator that is mainly used to create microbenchmarks to enable quick prototyping on simulators is proposed in Chapter 4.

In Chapter 3, the experimental infrastructure used to investigate the work that is

proposed in this thesis will be presented.

Chapter 3

Experimental Infrastructure

Chapter 2 reviewed existing power management schemes and their limitations. Modifications at a microarchitectural-level and their interaction with the higher layers of the compute stack makes it mandatory to build a simulation infrastructure where the effect of microarchitectural modifications can be evaluated on performance, power and temperature. It is possible to evaluate the performance of applications including low-level software such as operating systems or virtualisation layers on real hardware [PZW+07]. Most commodity processors provide built-in counters that provide information on average power consumption, or system temperature [Spr02, Han12]. Microarchitecture level modifications can be evaluated in two ways, (i) by implementing RTL level code on FPGAs, or (ii) via simulators written in a high level programming language with the capability to simulate microarchitectural details. Although there are open-source efforts to create RTL level experimentation platforms [LWC+16], the intellectual property for implementations of common ISAs such as ARM or x86 remains largely out of reach for the research community. This makes it necessary to use simulators to evaluate ARM and x86 ISA based systems.

The experimental infrastructure used to determine the efficacy of material presented in Chapter 5 is briefly explained in this chapter. The tool flow shown in Figure 3.1 is introduced in order to evaluate microarchitectural ideas and their consequent power and temperature implications. For this reason, the available tools in the research domain were integrated together in order to establish a simulation environment required to evaluate the novel power management approach proposed in Chapter 5 that is intended to address the limitations of the current power management schemes, as described in Chapter 2, in terms of performance, power, and temperature. Each of the blocks will be explained in further detail within their respective sections.

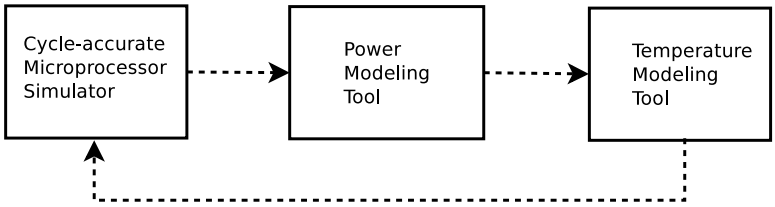


Figure 3.1: Experimental Infrastructure Tool Flow.

3.1 Processor simulation

Simulation of microprocessors and memory hierarchies is a cost-effective way of evaluating architectural ideas. Simulation makes it possible to quantitatively compare a baseline architecture against different modified versions of that architecture before going to the costly and time consuming task of verifying an RTL implementation. Simulators differ in parameters including simulation model, instruction set architecture (ISA) support, memory hierarchy and interconnect configurations, multi-core support, source code availability, and performance.

The level of detail captured by a simulation environment is determined by the amount of accuracy required in terms of representing an actual system. The most widely used modes of simulation are *functional*, *cycle-accurate*, and *interval-simulation*.

Functional simulation is used to verify the behaviour of a given design without taking temporal aspects of that design into account [Eec08]. A common use case of functional simulation is verification of the hardware/software interface. This makes it impossible to simulate microarchitectural aspects that are representative of actual hardware such as the pipeline stages of a superscalar microprocessor. Although it is technically possible to decouple memory hierarchy simulation from the simulation of a core, simulating a highly accurate memory system with functionally simulated cores still leads to incorrect simulation results due to the skew in memory access timings introduced by the low level of detail that is inherent in the functional core model [GPD$^+$14].

Cycle-accurate modelling increases the level of detail that is simulated at a functional level at the cost of simulation time. While functional models usually execute at a single instruction per cycle model without taking pipeline details into account, cycle-accurate models typically model each pipeline stage with a relatively higher level of accuracy by simulating pipeline stages and their structures with individual clocks assigned to each of them. This results in a very steep increase in simulation time. There are two approaches taken to decrease the simulation time of a cycle accurate simulation

which are co-simulation using field programmable gate arrays (FPGA) [CSK+07], and interval simulation [GEE10] [EEKS09].

Co-simulation using FPGAs involve the offloading of execution of a part of the simulated system onto previously built Intellectual Property (IP). Although the speedup gained from this approach can be orders of magnitude greater [TWA+10], it requires the development of the hardware block using a hardware design language (HDL), and an extra effort in verifying the implemented block. Although verification is required for simulators written in high level languages, the time it takes to verify hardware designs is significantly higher compared to software designs. Since FPGA device sizes are also limited, another issue arises with the implementation of many-core systems on FPGAs as it is not possible to fit the whole design space into them.

The other alternative, interval simulation, speeds up the simulation by only simulating the regions of interest in a highly detailed fashion, while simulating the remaining execution regions using an analytical model at a higher level of abstraction. The analytical model is implemented as a superloop that increments simulation statistics based on occurences of events, rather than simulating the hardware blocks that process these events in a detailed manner. The regions of interest can be set by a list of events that trigger a detailed simulation such as branch mispredicts or last level cache (LLC) misses. It is also possible to mark regions of interest from the code being executed, so that the simulator is notified to increase the level of detail being simulated.

In the following subsection an overview of available simulators in the computer architecture domain will be presented along with a justification on the selection of the simulator used within this dissertation.

3.1.1 An overview of available simulators

Table 3.1 provides a list of the most widely used simulators for architectural research. The initial investigation on the scope of research being done in the area of microarchitecture from a power management perspective, with emphasis on both the compute and memory blocks, lead to the following general requirements for building an infrastructure that enables users to experiment in the most flexible fashion:

- Multi-architecture support was a key requirement in order to evaluate different architectures and different ISAs on the simulation platform.
- Memory System Flexibility in terms of configuration and ease of prototyping memory system blocks related to cache hierarchy and coherence.

- Source code availability along with user community support in order to make modifications in unforseen circumstances as the API for a closed-source simulator may be limited for specifying required behaviour

- Support, along with a user community behind the tool would be helpful in times when problems took a long time to solve.

- Full system simulation was a key requirement in order to include the operating system behaviour in evaluations

The advantages and shortcomings of the listed simulators will be discussed below, along with a description of why gem5 was chosen as the simulator to be used in this work.

gem5

The gem5 simulator [BBB+11] is a highly modular simulation infrastructure that enables experimentation of research ideas at microarchitecture and system level including microprocessors, interconnects, memory hierarchy and cache coherence. It can perform functional and cycle-accurate simulation, and can switch between functional and cycle-accurate simulation modes using a fast forwarding mechanism for performing interval simulations. There are two different ways of simulating the memory system within gem5; *General* and *Ruby*. *General* memory system simulates a snooping bus with parameters for adjusting bandwidth and latency. *Ruby* [MSB+05] is a highly

Table 3.1: List of architectural simulators

Simulator	**Supported architectures**	**Memory System Flexibility**	**Simulation Modes**	**Licence Type**	**Comm. support**
gem5	x86-64, armv8, aarch64, mips, sparc, ppc	Flexible	System call, full system	BSD-like	High
Sniper	x86-64	Limited	System call	MIT	High
Z-sim	x86-64	Limited	System call	GPLv2	Low
Simics	x86-64, armv7, ppc, mips, sparc	Limited	System call, full system	Proprietary	Low
Marss	x86-64	Limited	System call, full system	GPLv2	Low

detailed memory system simulation framework where the user can explore highly detailed memory hierarchy configurations and on-chip networks with different topologies. Ruby also comes with a domain specific language (DSL), named *Specification Language for Implement Cache Coherence (SLICC)* that makes it possible to define cache coherence protocols that can be plugged into the cache hierarchy without any extra effort. Cache controllers, memory controllers and the protocol that is used when messaging between these controllers can be modeled using SLICC. All the level of detail comes at the price of increased execution time. Measured execution time for highly memory-bound workloads is around 40k instructions per second, whereas this number is increased to 400k instructions when the execution is completely compute-bound. The average error for simulation of gem5 utilising the ARM ISA has been measured to be in the range of 20% on average when compared with an ARM Cortex-A15 [GPD+14]. The causes of errors in simulation have been summarised as; (i) architectural equivalence do not necessarily model microarchitectural equivalence, (ii) loss of information due to abstraction, (iii) core models that are not representative of the actual core being simulated.

Sniper

Sniper [HCE12] is a multicore x86-64 simulator that executes using a mixture of interval [GEE10] and cycle-accurate simulation. Only regions of interests that are marked by certain hardware events are simulated using the cycle-accurate engine, while the rest of the simulation is carried out using a simplified analytic model that has been shown to be sufficiently accurate [EEKS09]. Sniper's visualisation support makes data analysis easier than with other simulators. It can only simulate directory based cache coherence with MESI, MESIF and MSI protocols. The McPAT power modelling tool is fully integrated into the simulator, making it straightforward to obtain power estimates for the overall simulation, or periodically. A user can easily obtain stacked plots that demonstrate the fraction of cycles spent, power plots over time, and profiling information that contains a call graph. It is the fastest simulator relative to the level of detail that it simulates, peaking at around several millions of instructions per second (MIPS). However, the lack of multiple architecture support ties this simulator to the x86-64 architecture making simulating mobile domain applications difficult. An evaluation of the Sniper simulator finds that the average error across a range of multi-threaded applications was found to to be 25% [CHE11] when validated against an Intel XEON X7460 multi-core server.

ZSim

ZSim [SK13] is a detailed many-core simulator, that can simulate 1000s of cores. Parallel simulation is enabled using a two phase bound-weave algorithm. The algorithm works by first generating a list of memory interactions through a higher-abstraction simulation. The second phase consists of detailed simulation making use of the timings obtained from the first to minimise the synchronisation overheads that other parallel simulators incur. Acceleration of Out-of-Order instruction execution is achieved by using Dynamic Binary Translation (DBT). DBT enables simulation of executed instructions in one to two orders of magnitude compared to other cycle-accurate simulators. It is the fastest simulator that is available with an open source licence, with simulation performance reaching hundreds of MIPS for detailed out-of-order cores. It has a modular cache configuration system, that makes the integration of custom implementations for coherence and replacement policies easy. Unfortunately, it lacks support for mobile architectures, full-system simulation and power modelling. The average absolute error for ZSsim has been measured as 11.2% for multi-threaded workloads, and lower than 10% for single-threaded workloads when validated against an Intel XEON L5640 multi-core processor.

Simics

Simics [MCE+02] is a proprietary licensed simulator that can be integrated with the Ruby memory system simulator. It has an ISA simulator that can perform functional and cycle-accurate simulations for ARM, MIPS, Intel and Sparc ISAs. It enables users to implement and integrate their microarchitectural ideas through a microarchitectural interface (MAI). Being able to configure the memory system using Ruby provides all the blocks needed to implement detailed memory system simulations. Executions can be rewound for further examination of the simulated system making debugging easier using *reverse simulation*. Another distinguishing feature of Simics is the fault injection mechanism that enables injection of faults to simulated systems. The proprietary nature, and fairly low usage within the computer architecture community are the main shortcomings of this simulator.

Marss

Microarchitectural and System Simulator for x86-based Systems (marss) [PACG11] is a multicore x86-64 simulator that comes with a detailed Out-of-Order execution model

that can simulate hundred thousands of instructions per second. It includes write-back and write-through cache implementations, MESI and MOESI cache coherence protocols, and a simple DRAM model. It lacks in modelling of detailed cache models, therefore making implementations for experimental cache replacement models not possible without extra effort. Lacking support for ISAs other than x86-64, and the lack of a modular simulation infrastructure required for the purposes of this dissertation has been the main reason for ruling out the possibility of using this simulator.

Based on the evaluation of the parameters listed in Table 3.1, the gem5 simulator was chosen based on its high level of modularity, community support, and the open source licence. The software architecture and the modifications made to gem5 to build the required experiment infrastructure will be detailed in the following subsections.

3.1.2 gem5

A simplified block diagram encapsulating the execution of gem5 is provided in Figure 3.2. The simulator infrastructure is an event-driven architecture that triggers events when the event *tick* is reached. This approach negates unnecesseray computation for simulation periods that are idle. Simulation is driven from a python interface that consists of wrappers for the C++ objects constructed through a system definition provided at the beginning of the simulation. A system configuration describes the type of cores, level of details for timing purposes, memory hierarchy configuration, network-on-chip configuration, root filesystem and operating system kernel to execute.

Software architecture

The simulator is built using an event-driven architecture, where each event triggers an introspection or a change in the simulation objects that are associated with the handling of the event. The quantum of simulation in terms of time is called a *tick* in gem5. Events are triggered when a specified tick is reached, and the triggering of an event can also result in scheduling of other events. For example, the first thing an initialised core does is schedule a *fetch* instruction event at a given address. Once the tick for the scheduled fetch is reached, this fetch event requests the instruction residing at the address the program counter points to, which is not cached at the beginning of execution, leading to an instruction cache miss event that is scheduled based on the latency provided in the simulator configuration. The fetch request event that originated from the execution of the fetch pipeline stage leads to several other events that eventually

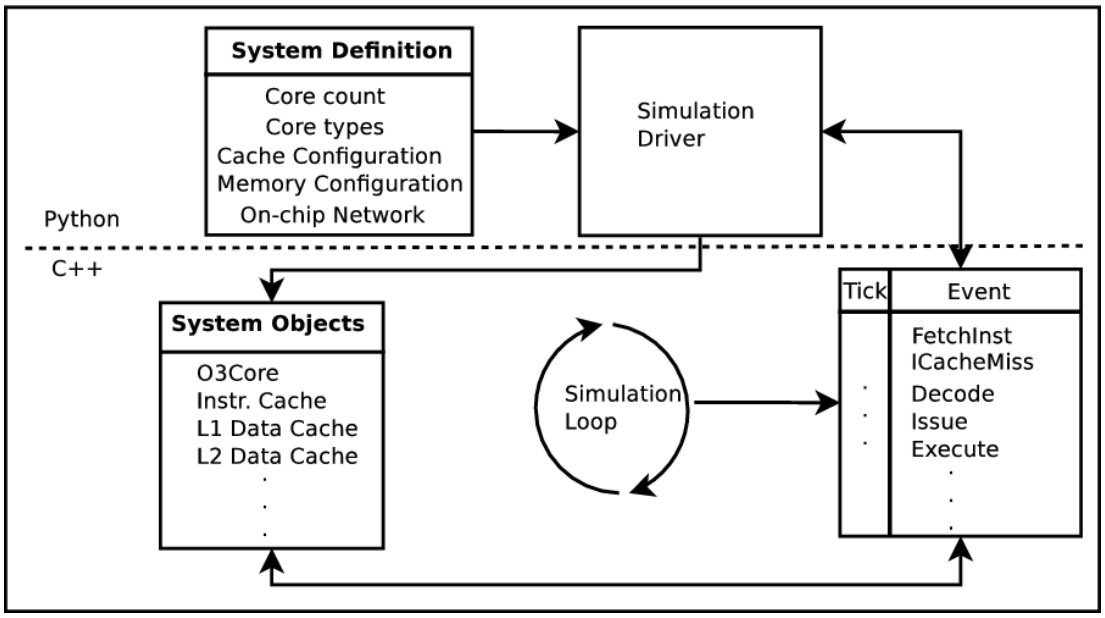


Figure 3.2: gem5 block diagram.

lead to the filling of the cache line that the requested instruction address corresponds to, and the subsequent filling of the instruction queue. All the blocks that take part in this chain of events are either modified or introspected during this process, and this is how events that are temporal in nature inflict changes in the spatial configuration of the simulated logic blocks. These simulated logic blocks are called SimObjects in gem5. The class hierarchy for simulation objects is given in Table 3.3.

A SimObject represents a physical component that is being simulated, and it also makes it possible for the simulated physical component to be configured prior to simulation. It is inherited from the EventManager, Serializable, and Drainable classes. EventManager class contains member functions that enables an object to schedule and deschedule events onthe event queue. Drainable class is used to put simulation objects into a consistent state when switching between CPU models or checkpointing. Serializable class helps with the serialization of simulation objects so that they can be written to memory when taking checkpoints. *Clocked objects* (ClockedObject) are simulation objects that inherit the SimObject class with extensions that enable synchronised timing of events. Another type of object is the *memory object* (MemObject) that is an extension of the ClockedObject. The MemObject class provides master and slave ports on top of the timing functionality provided by ClockedObjects. The introduction of these ports enable MemObject instantiations to be connected to each other in a way that precisely represents the behaviour of memory system blocks such as load/store

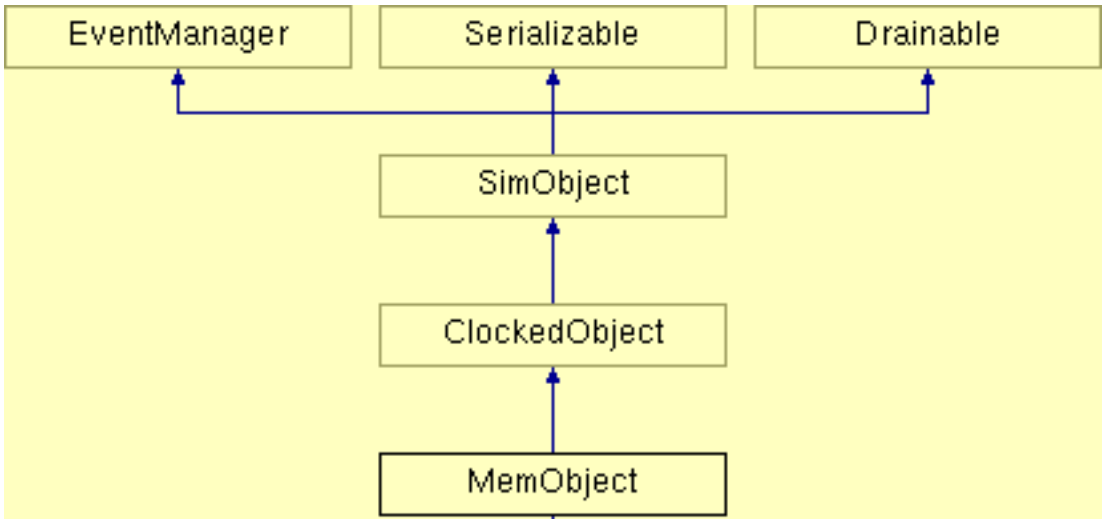


Figure 3.3: Simulation object class hierarchy.

queues, or instruction, and data caches.

There are four CPU models that can be simulated using gem5. These are *atomic*, *timing*, *out of order*, and *minor*. The atomic CPU model acts as a functional model that targets correct ISA execution with constant timing behaviour in memory accesses that are defined using latency parameters provided at configuration time. The timing CPU model simulates an in-order core and the timing for its memory accesses in a more detailed way that represents the behaviour of an actual memory system. The minor CPU model is a timing CPU model developed to accurately represent an ARM Cortex A9 core. The Out-of-Order model implements a superscalar CPU model that exploits instruction level and memory level parallelism. The details of the Out-of-Order model will be provided in the following paragraph as it is the core model that has been used throughout this work.

The Out-of-Order CPU model contains a 5-stage pipeline that consists of fetch, decode, register rename, issue-execute-writeback (IEW), and commit stages. The fetch stage is responsible for filling up the instruction queue, and can be configured to work with branch prediction. The decode stage resolves the operands of the fetched instruction, and determines to which functional unit operands are sent for execution. The register renaming stage resolves false dependencies for the instruction operands. In the IEW, a reservation is made at the related functional unit for the execution of the instruction, the instruction is executed. Then the result is written back to the register file. Finally, the commit stage retires the instructions that have executed in the order that they were fetched.

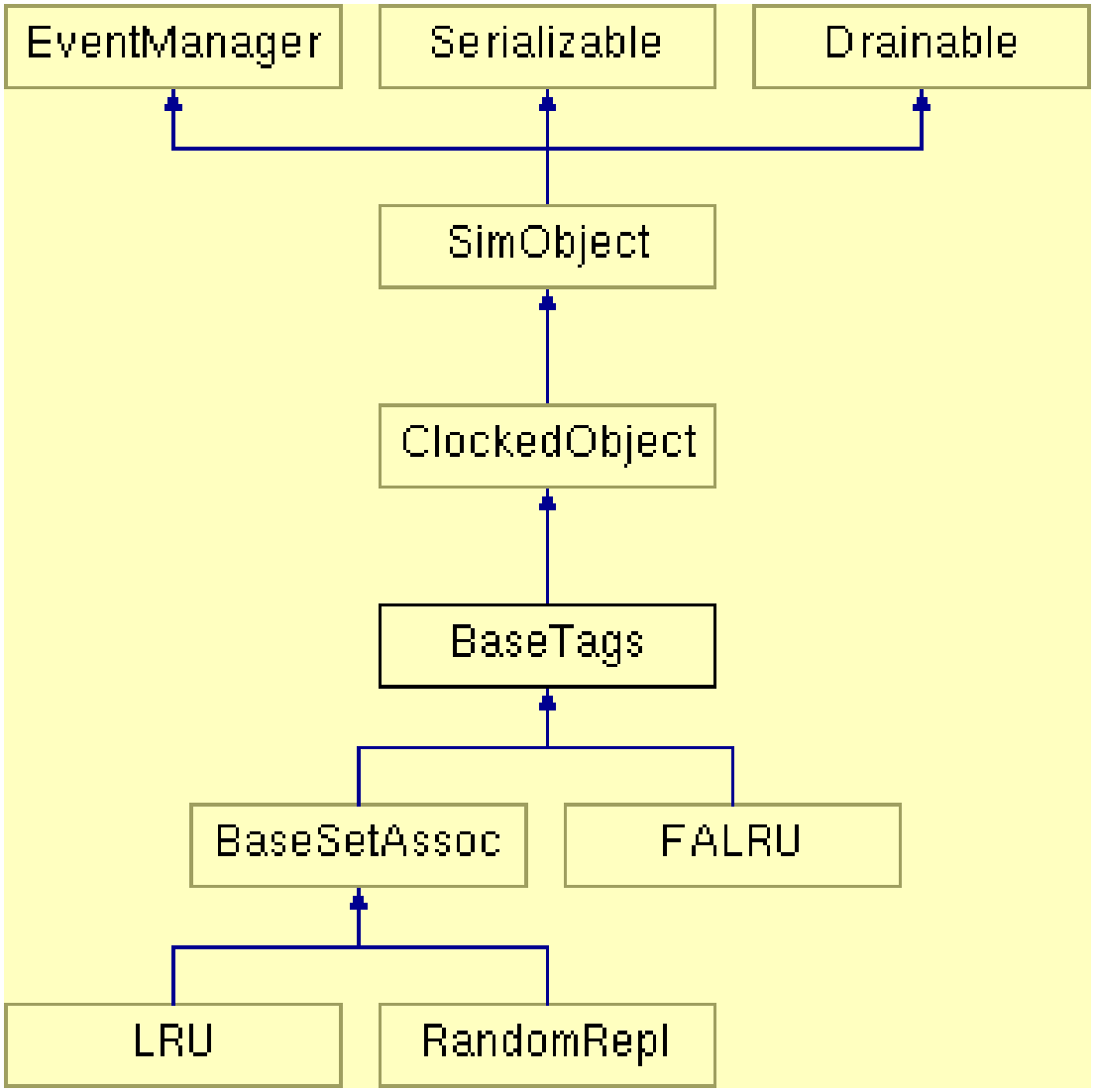


Figure 3.4: Cache replacement policy class hierarchy.

Memory system configuration involves the definition of the memory hierarchy depth by specifying the levels of caching in the system along with parameters for the size and replacement policy of the caches residing on each level. All the caches are inherited from the BaseCache class, whose base class is the MemObject. BaseCache contains CPU and memory side port connections, and functions used for receiving requests based on the timing model being simulated. Cache objects are connected with the neighbouring levels in the memory hierarchy using Port objects. Each instantiation of a cache is a specialisation of the Cache class using a *cache tag store* that is inherited from the BaseTags class. BaseTags contain properties such as the cacheline length, cache size, access latencies and number of blocks. Replacement policies are implemented by inheriting from the BaseTags class. A replacement policy declares how and where new blocks will be inserted and replaced within a cache. The class hierarchy for cache replacement policies and representation is given in Figure 3.4.

A memory request in gem5 can be issued by any simulation object that contains a master port, whereas the responses are handled using slave ports contained within simulation objects. For instance, a CPU core contains two separate ports for making requests from the memory system through fetch queue or the load/store queue. Messaging between master and slave ports is established by forwarding Packet objects until they reach the destination where the requested data is located.

3.2 Power modelling

Power models are linked into the simulation using a modified version of the Multicore Power, Area, and Timing (McPAT) tool [LAS+09]. The choice of McPAT has been made on the basis that it already had partial integration with the gem5 simulation environment, with the bits for periodically obtaining power statistics missing, which has been added during the construction of the simulation infrastructure. McPAT enables users to model power, area and timing in a holistic manner for in-order and out-of-order cores, and memory hierarchies. It can model dynamic and static power dissipation seperately for Complementary Metal Oxide Semiconductor (CMOS), Silicon On Insulator (SOI), and double-gate devices. It is built on top of CACTI [MBJ09], which is a tool that models timing, area, leakage, and dynamic power for cache hierarchies.

The block diagram of McPAT is given in Figure 3.5. McPAT is configured through an XML file, where the user specifies architectural, manufacturing process, and timing level details such as number of cores, supply voltage, physical register file size, or

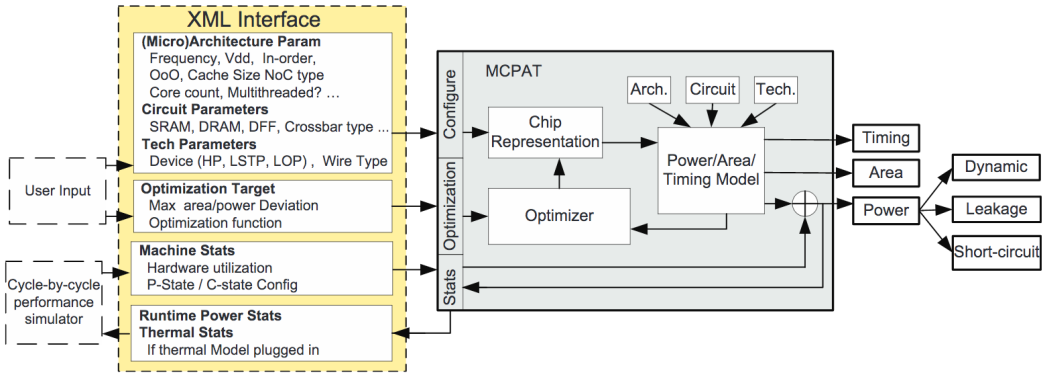


Figure 3.5: McPAT block diagram (Taken from [LAS+09]).

clock frequency. A cycle accurate simulator, or the performance counter results obtained over execution on real hardware, can be used to generate the dynamic inputs concerning the machine state.

3.3 Temperature modelling

Thermal evaluation of the simulated architecture is accomplished using the HotSpot [HGV+06] temperature modelling tool. HotSpot generates an RC model of the system that is being simulated. An RC model is a way of representing the circuit as a network of resistors and capacitors. It requires a floorplan that specifies each architectural block in terms of width, height, and an origin point in a two dimensional representation. Chip and heat sink specifications in terms of thickness and thermal conductivity are also provided at configuration time. Initial temperatures of each block that have been defined in the floorplan can also be configured.

HotSpot requires power traces as input at each sampling period to calculate temperature. The provided traces should provide the component-level power dissipation for each architectural block that has been specified in the floorplan file.

The thermal behaviour of chips can be modelled at different granularities. As illustrated in Figure 3.6, it is possible to choose between functional block, regular grid sizes or functional blocks divided into regular grids as simulation output.

A temperature map generated by HotSpot through running a sample application on a single core microprocessor is given in Figure 3.7. It can be seen that the sample application stresses the execution unit blocks heavily, causing them to heat up to 360 Kelvin.

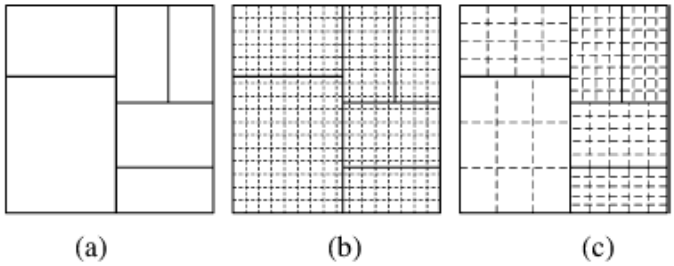


Figure 3.6: HotSpot modelling granularity (a) functional blocks (b) grid (c) functional blocks with grids (Figure taken from Huang *et al.* [HGV+06]).

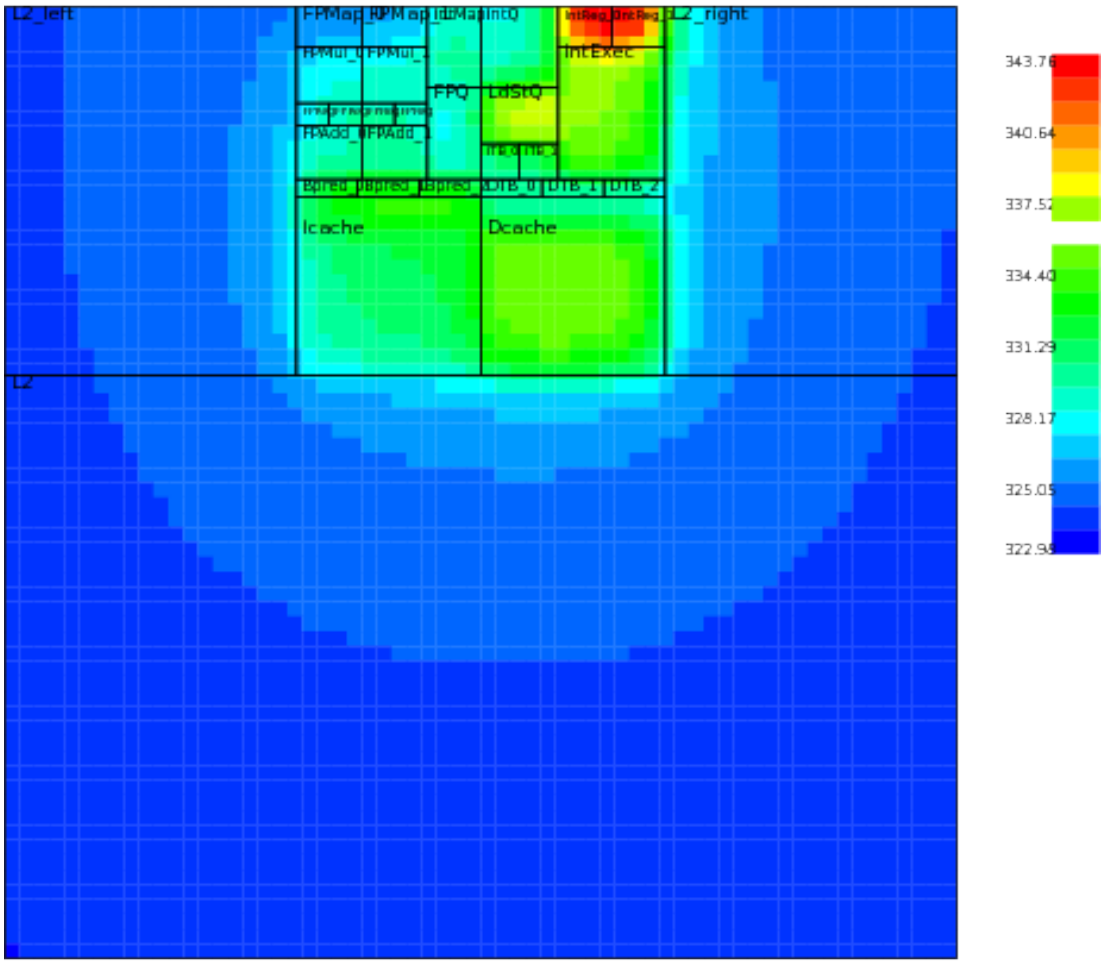


Figure 3.7: Sample HotSpot output.

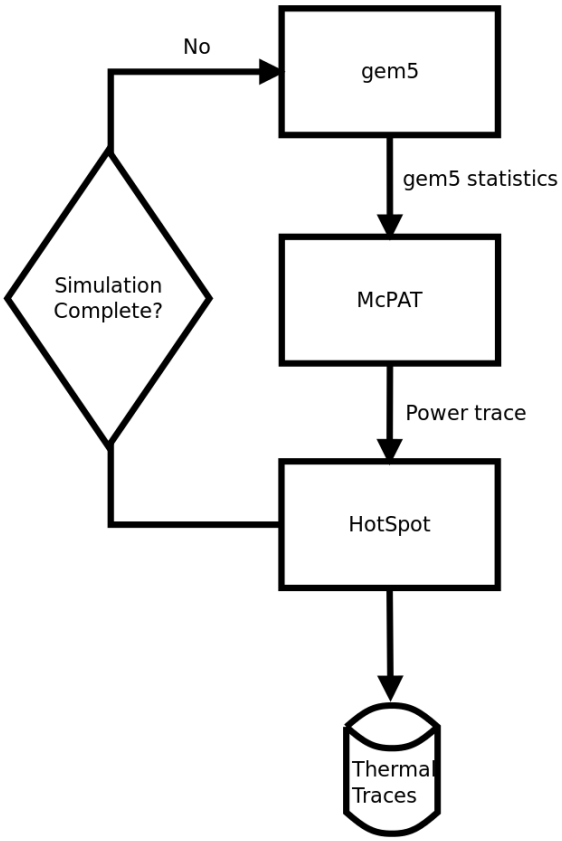


Figure 3.8: Simulation tool flow

This modelling tool was chosen on the basis that it was the only available temperature modelling tool that was being maintained during the course of this dissertation.

3.4 Simulation toolflow

It was indicated in the beginning of the chapter that there was a requirement for a tool flow that would enable microarchitectural research evaluation with power and temperature aspects. The tool flow presented in Figure 3.8 has been developed in order to make such research possible.

The tool flow presented involves gem5 statistics being input into McPAT at a configurable period, and McPAT generating the required power traces for HotSpot to calculate the temperature of each block based on activity dictated by the application that is being simulated.

3.5 Simulation models

This section provides the configuration of the simulation environment for the research presented in Chapter 5.

3.5.1 Architectural model

The simulated architecture used within this dissertation models an ARM A57 that is an Out-of-Order microprocessor utilising the ARMv8-A ISA [ARM]. This core was specifically chosen as it was the state-of-the art out-of-order ARM core that was available in the market at the writing of this dissertation. The in-order fraction of the pipeline has a three instruction wide fetch unit that includes a dynamic branch predictor with a Branch Target Buffer (BTB) containing a Global History Buffer (GHB). BTB acts as a cache that stores the address of branch instructions. At each fetch cycle the Program Counter (PC) is compared with the entries in the BTB, and when a hit occurs the instructions from the address of either the taken or not taken branch are fetched based on the majority of past branch decisions as stored in GHB. The decode stage is 8-wide enabling upto eight micro-instructions to be issued. The issue queue has 32 entries. Execution units contained within the core are: a 16-Entry load/store unit for memory operations, two integer units, and an FPU. The integer physical register file contains 192 entries, and the floating point register file contains 128 entries. There is a 40-entry reorder buffer for tracking the fetched instruction order. The The memory hierarchy consists of a 3-way set associative 48KB instruction cache, a 32KB 2-way set associative L1 data cache, a 512KB 16-way set associative L2 cache, and a 8MB 16-way set associative L3 cache.

3.5.2 Technology model

FinFET transistors provide better leakage properties than their CMOS counterparts. As of this writing, the latest technology node was 14nm. A state-of-the art 14nm Fully Depleted Silicon on Insulator Fin Field Effect Transistor (FinFET) model similar to [WJMH15] was used to simulate power and temperature effects.

3.5.3 Floorplan

The floorplan of the simulated architecture for temperature modeling is given in Figure 3.9. The floorplan is a modified version of the out-of-order core floorplan that is

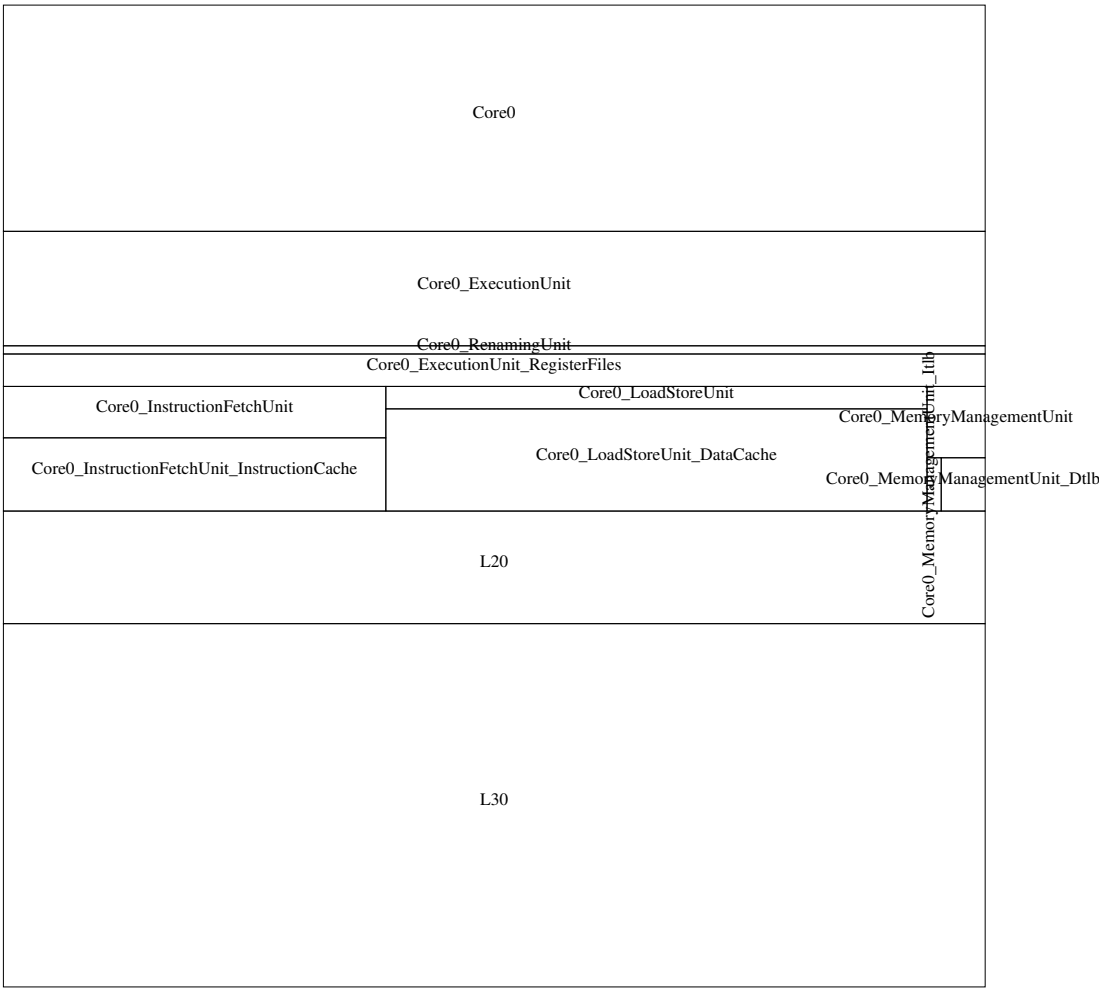


Figure 3.9: Simulated ARM A57 floorplan

provided by HotSpot [HGV+06]. This has been achieved by modifying the relevant blocks such as instruction and data caches in accordance with the architectural model specifications described previously.

3.6 Hardware Experiments

This section describes the hardware infrastructure that is used to conduct the experiments in Chapter 4. The methodology used for the experiments is the same although different architectures, namely ARM and x86_64, are used in the experiment. A block diagram showing how a benchmark is profiled is given in Figure 3.10. Power and performance event counters are available for both of the architectures. This eliminates the need to setup an external power management infrastructure, leading to opportunities in

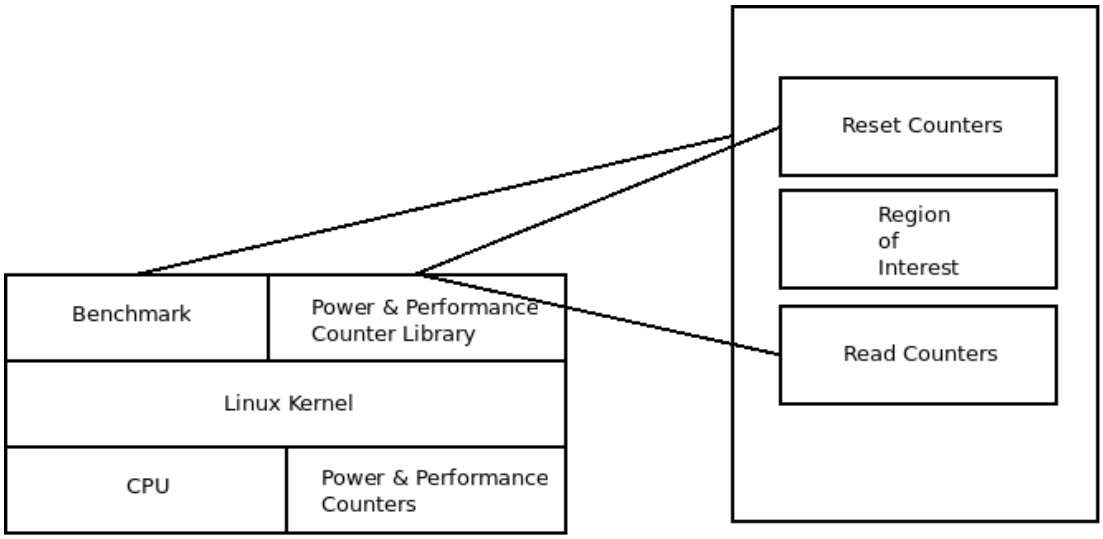


Figure 3.10: Hardware Event Profiling

software-only measures for profiling of software with respect to power consumption. Measurement of events is accomplished by resetting the related performance counters before the region of interest in a benchmark, and reading the counters at the end of the region of interest.

3.7 Summary

An experimental infrastructure composed of microprocessor, power, and temperature is a key requirement for evaluating microarchitectural techniques. An overview of the simulator landscape was provided, along with a justification of why the described setup was used. This chapter presented the simulation infrastructure that is used for the experiments made in Chapter 4 and Chapter 5 .

Chapter 4

Generator of LLVM Assisted Microbenchmarks

Microbenchmarks are segments of code that are aimed to target a single operation within a processor in order to isolate the effects of that single operation while nothing in parallel is being executed within the system. The granularity of the operation can be as simple as the execution of a single block of ALU instructions, or can trigger relatively more complicated execution as in the case of a dual-threaded synchronisation code segment that executes on two-cores and stresses the cache coherency mechanism of the processor. The most common use cases for microbenchmarks are: (i) evaluating processor behaviour in terms of power, thermal, reliability and performance [WJY+07], (ii) verification of a processor design [LFOK11], (iii) generating representative workloads to reduce execution times [CAP+15], and (iv) fault detection [KPFG14].

An overview of the available microbenchmark and workload generation tools, and their shortcomings have been provided in Chapter 2. This chapter will introduce a novel automated benchmark generator, named Generator of LLVM Assisted Microbenchmarks (GLAM) for automatically generating microbenchmarks independent of architecture by generating microbenchmarks from a user given specification that is then translated into an intermediate language (IR), where the generated IR is consequently compiled into native code.

Manual generation of microbenchmarks each time for a different architecture is a cumbersome process in terms of the process being open to errors while writing assembly code and the time spent for rewriting a piece of code that essentially executes

in a very similar way to previously written code. Even though previous work on automated benchmark generation [BBG+12] [JEJI08] decouples code generation from benchmark requirements, thus providing modularity, a new backend for code generation needs to be written in order to utilise this decoupling. GLAM, on the other hand, translates user specifications into LLVM IR, enabling it to be compiled into any architecture supported by the LLVM compiler infrastructure. *Architecture agnosticism* that is enabled by generating the microbenchmarks into LLVM IR is the main contribution of GLAM. The main motivation behind implementing this tool was the need for a way of quickly examining the properties of compute-bound execution and memory-bound execution in the simulation environment described in Chapter 3. The chapter will first explain the building blocks, and system architecture of GLAM, going into detail on how microbenchmarks are generated from user specification to machine code generation. This will be followed by showing the utility of GLAM by generating a comparison for two different architectures that execute the same microbenchmark specifications, providing researchers a practical way of comparing different architectures. Finally, how this tool can be used to provide power characterisation on an instruction granularity is shown.

4.1 GLAM: Generator of LLVM Assisted Benchmarks

The tool flow of GLAM is given in Figure 4.1. High-level microbenchmark specifications by the user can be provided using a JSON formatted specification file. The flexibility provided by the specification format enables the user to generate either streamed or looped instruction sequences and link them to each other. This makes it possible to generate binaries ranging from microbenchmarks to larger scale benchmarks that can mimic the behaviour of actual applications. Options to include profiling code using performance counters, and support for different architectures can also be included using the specification. Repeatability is enforced by design as all the code that is generated, and the experiments conducted are based on a single JSON file. More about the capabilities that can be exploited using this structured way of specifying code generation will be explained in the related subsection.

Once the user generates these specifications, the GLAM framework transforms these specifications into LLVM IR. How the IR is generated depends on the requirements of the user. GLAM is able to generate IR code that takes microarchitectural properties into account through a microarchitectural definition. Information provided

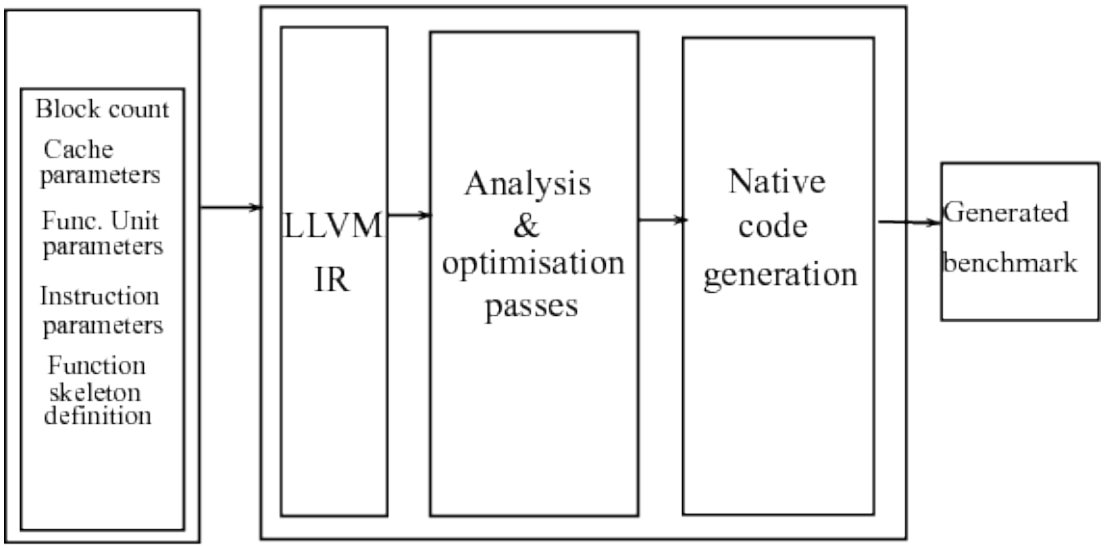


Figure 4.1: GLAM tool flow

such as the details of the cache hierarchy or the number of functional units within the core helps the framework tailor the generated code for the architecture that is being experimented on.

After the generation of an intermediate level representation, the IR code can then be translated into native assembly code into any architecture that is supported by the LLVM Static Compiler (llc). If the user requires the generated code to be profiled, a harness that includes hardware performance counter profiling code is used to automatically wrap each of the generated assembly routines. The output for each of the function executions are written to a single Comma Seperated Values (CSV) format for ease of data processing. This makes it possible for the user to easily evaluate different instruction mixes with varying memory behaviour from a performance and power perspective.

In the following subsections details on program specification, code generation, architectural definition, and profiling code insertion is provided along with a sample use-case of the tool and the chapter will be concluded.

4.1.1 Code specification

GLAM uses the JSON data interchange format, which is a context free grammar commonly used in the web application domain. GLAM specification format has a hierarchical structure letting the user define multiple numbers of modules that may contain multiple functions, and functions that may contain multiple numbers of code blocks.

It enables the specification of the programs to compile the generated IR into native code. Function prototypes can be customised along with their parameters. This makes it possible to pass parameters like loop counters through the harness during execution. The most noteworthy definition within the specification is the block entry. Block entry lets the user specify homogeneous instructions streams that are sequenced or looped along with the instruction distances between them. This makes it possible to mimic the behaviour of any program phase as blocks can be concatenated in any granularity and behaviour that a program can have.

Listing 4.1 provides a sample GLAM benchmark specification. A top-level representation named *experiment* is used to define everything that will be common for the modules that will be built. All the information provided for the *experiment* is used to automatically generate a harness for each specified architecture. It is also possible to provide the architecture definition files that provide specific information such as cache hierarchy details to guide in the benchmark generation process. The architectural definition is most useful when generating memory operations. The execution harness can be generated to run on actual hardware with Hardware Performance Counter (HWPC) based profiling enabled per function generated, or to run on a simulator with the HWPC profiling code replaced by the related simulator hooks that let the user enable/disable statistics dumping or tracing. The number of times an experiment is to be repeated can also be specified, so that a statistically significant execution of the generated code can be established.

All the information required to generate the IR code is defined in the *module* representation. *Module* is a top-level container for all the functions that are specified within it. *Function* specifications provide the input parameters, the data types associated with the parameters and the return type of the function. There are three types of parameters that could be specified; data, data pointer, and loop counter. Data declares parameters of the specified data type to be passed as values. Data pointer, as the name suggests, is a pointer to the input parameter type and is used in conjunction with the memory related operations. Loop counter parameter type is used to control the looping behaviour.

The most essential part of the GLAM benchmark specification is the *block* specification. Code blocks that can perform various operations can be defined within a function, and can be appended after each other making it possible to generate functions with a high level of variety in their control flow. Each block definition can only contain a single operation. These operations can be grouped as; integer arithmetic, floating

point arithmetic, and memory operations. Arithmetic operations can be addition, subtraction, multiplication, and division. Memory operations are used to generate memory accesses that hit or miss on different levels of the cache, using different strides, with or without thrashing behaviour. The count of the instructions to be generated within the block, and a homogeneous instruction dependency distance for these instructions can also be specified.

Appendix B can be consulted for an exhaustive description of the GLAM keywords.

4.1.2 Code generation

The first step of code generation is generating the related LLVM modules, functions and blocks as empty stubs. GLAM keeps an internal representation of all the LLVM elements in the form of wrapper functions, making it possible to map the GLAM specifications to the LLVM elements.

After this process, an entry block that is responsible for the initialisation is generated. The entry block is used to allocate space on the stack for loop counters or data parameters passed onto the current stack frame, and generating enough instruction distance for the next blocks instruction distance requirements to be met.

Each of the generated blocks are connected to each other using unconditional branches. All the blocks are generated based on name, instruction dependency distance, operation and instruction count properties that are provided from the GLAM specification file. A loop flag that indicates whether looping should be implemented within the block can be set to true if looping is required. The operation property specifies, which LLVM function to use, which boils down to the specific functional unit usage of the current block.

The output of the code generation module consists of an assembly file per module per architecture that conform to the initial JSON specifications provided by the user. An example of the outputs from the generation sequence is given in Listing 4.2. A block of 8 add instructions with an instruction dependency of 2 is specified as shown in Listing 4.2a. An LLVM IR block is generated based on the specifications in Listing 4.2b, and it is compiled into x86-64 assembly using llc in 4.2c. The output of the same code block for aarch64 architecture can be seen in Listing 4.2d. Then the function in the assembly file is wrapped by a C++ source program that is tailored to call the generated piece of code with the specified inputs. The generation of the C++ harness will be explained in the following subsection.

```
{
    "experiment": {
        "name": "glam_base_experiment",
        "descr": "Experiments for int, float, mem",
        "repeat" : "10",
        "arch" : ["x86-64", "aarch64" ],
        "archfile" : ["xeon-e5-2670.json", "none"],
        "simulate" : [false, false],
        "hwpc" : ["Instructions", "Cycles", "L1_Access",
                  "LLC_Loads", "LLC_Load_Miss"],
        "module": [{
            "name": "iaddsample",
            "descr": "integer alu test",
            "output" : "output/intalutests.ll",
            "function": [{
                "name": "iadd1",
                "descr": "iadd1",
                "prototype":{
                    "returns": "int64",
                    "input": ["int64","int64"],
                    "value": ["0", "0"],
                    "type": ["loopctr","data"]
                },
                "block": [
                    {
                        "name": "b0",
                        "descr": "block b0",
                        "distance": "1",
                        "operation": "add",
                        "instrcount": "256",
                        "loop":["PARAM","0"]
                    }]
            }]
        }]
    }
}
```

Listing 4.1: A sample microbenchmark specification

4.1.3 Execution harness generation

A C++ execution harness is automatically generated for each generated LLVM IR module, once the module is compiled into native assembly as shown in Figure 4.2.

```
"block": [
{
  "name": "b0",
  "descr": "block b0",
  "distance": "2",
  "operation": "add",
  "instrcount": "8"
}]
```

a: GLAM Block Specification

```
b0:
  %15 = add i64 %12, %13
  %16 = add i64 %13, %14
  %17 = add i64 %14, %15
  %18 = add i64 %15, %16
  %19 = add i64 %16, %17
  %20 = add i64 %17, %18
  %21 = add i64 %18, %19
  %22 = add i64 %19, %20
```

b: Generated LLVM IR

```
addq    %rdi, %rax
addq    %rcx, %rdi
addq    %rax, %rcx
addq    %rdi, %rax
addq    %rcx, %rdi
addq    %rax, %rcx
addq    %rdi, %rax
addq    %rcx, %rdi
```

c: Generated x86-64 assembly

```
add     x8, x8, x12
add     x12, x12, x14
add     x14, x14, x8
add     x8, x8, x12
add     x12, x12, x14
add     x14, x14, x8
add     x8, x8, x12
```

d: Generated ARM assembly

Listing 4.2: Code generation outputs

Each function defined within the LLVM IR module is wrapped with code for profiling the function. The wrapper code is responsible for initialising HWPCs, and the power measurement setup if required. The generated harness drives the execution as specified in the GLAM specification, inserting the profiling hooks for the specified HWPCs, and printing out the results in CSV format.

In the following section, three use cases for GLAM will be shown. The first use case shows the execution of the benchmarks on different architectures, which are generated from the same specification. The second use case provides an example of using GLAM to examine trade-offs between performance and power for compute bound and memory bound workloads. The third use case is an example of how GLAM can be used to measure Energy Per Instruction.

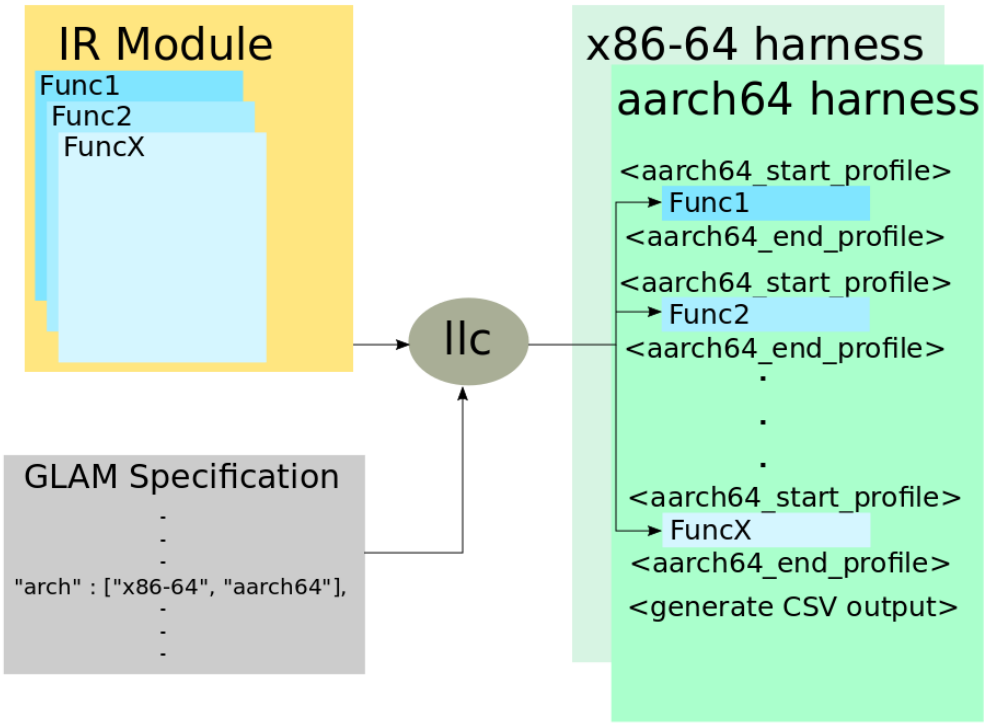


Figure 4.2: GLAM Harness Generation

4.2 Evaluation

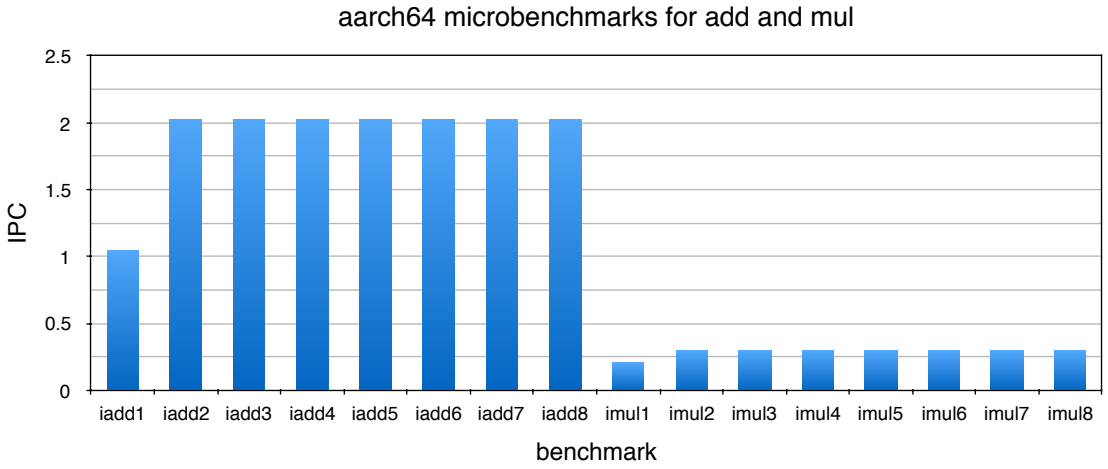
In this section three use cases for GLAM will be shown. Generation of microbenchmarks from the same specification will be presented first. This will be followed by an analysis of performance and power on memory-bound and compute-bound microbenchmarks. Finally, a methodology to measure the EPI using GLAM will be detailed out.

4.2.1 Comparison of microbenchmarks on different architectures

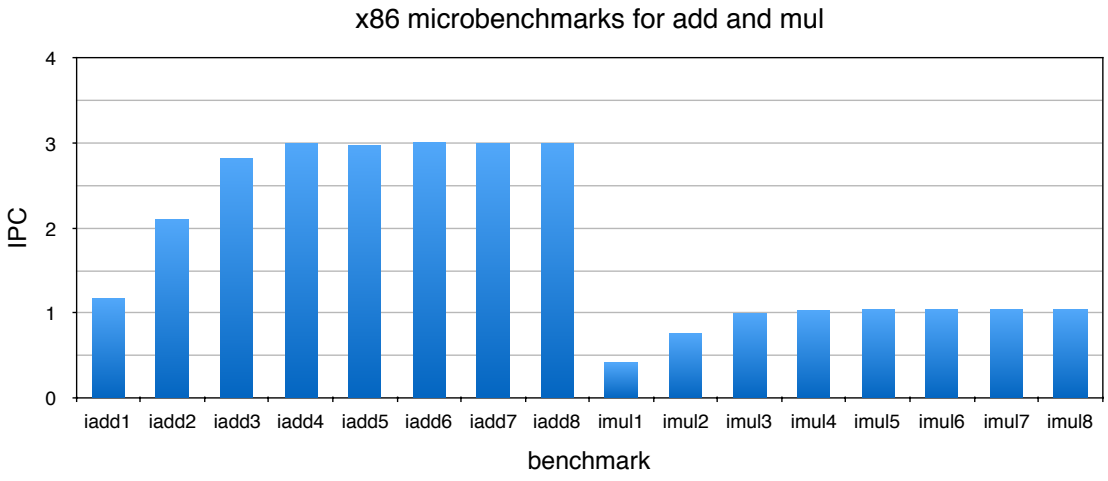
The first use case of GLAM is intended to show how a generated LLVM IR can be used to evaluate the behaviour of ALU units in microprocessors that have different architectural and microarchitectural configurations. For this reason, a GLAM specification was prepared to generate looped blocks of code that contains sequences of addition and multiplication instructions (Listing B.1). A different microbenchmark is generated for each case with instruction dependency distances ranging from one to

eight. Then the generated LLVM IR was automatically compiled into native assembly code for AArch64 and x86_64 architectures as specified in the GLAM specification. The harnesses for each architecture were automatically built to read Hardware Performance Counters (HWPC) for instructions retired, and the number of cycles for each of the function generated.

The results for microbenchmarks generated to stress the ALU of the underlying processor are presented in Figure 4.3.



(a) AArch64



(b) x86_64

Figure 4.3: Integer microbenchmarks

The results for the aarch64 architecture (Figure 4.3a) were obtained from the A57 core on an ARM Juno Development platform that contains a big.LITTLE architecture with four A53 cores, and two A57 cores. The 86_64 results (Figure 4.3b) were

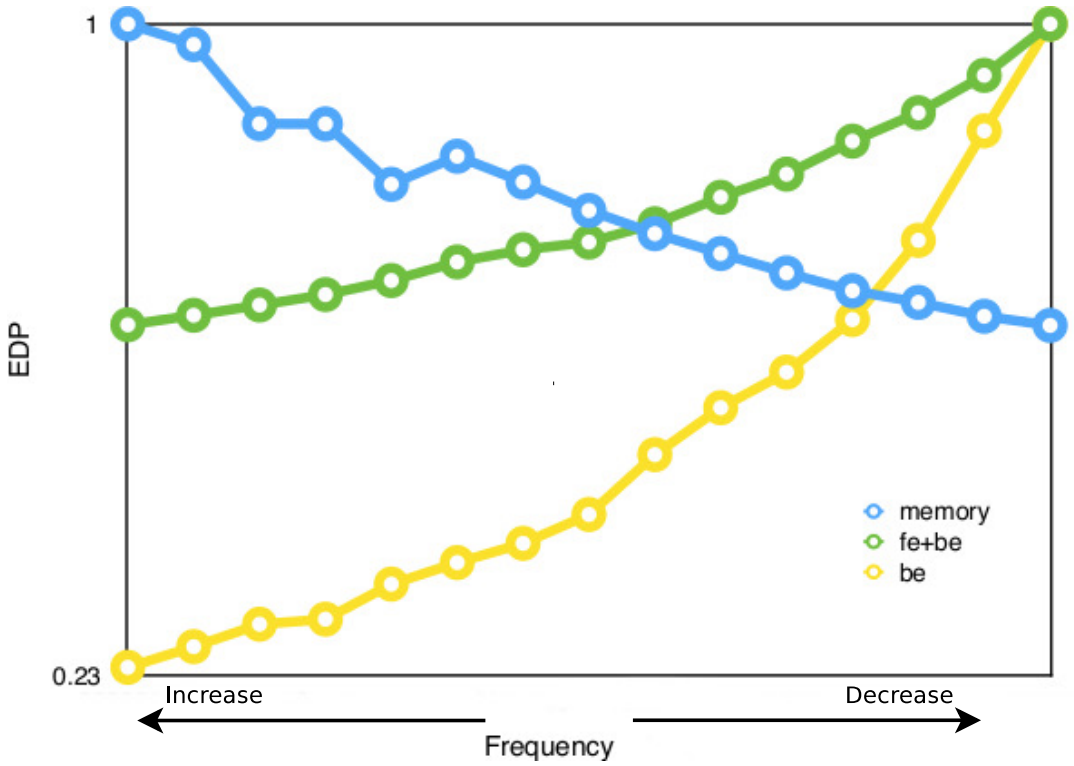


Figure 4.4: EDP scaling of memory-bound vs compute-bound application

taken from a 16-core Intel Sandy Bridge Xeon E5-2690 server. The obvious differences between the execution of the same microbenchmark stem from the underlying architectural details such as the number of functional units, and the number of cycles a multiplication operation completes in. It can be seen from the figures that the IPC of iadd microbenchmarks do not increase after two for the AArch64 (Figure 4.3a) as the A57 core only has two ALUs available for integer arithmetic, whereas the Xeon core (Figure 4.3b) has three ALUs thus an upper bound of 3 for the IPC. This subsection has been presented to show a use-case of GLAM for comparing code for different ISAs executing on different architectures.

4.2.2 Performance and power trade-offs

GLAM can also be used to exploit HWPCs for investigation of performance and power trade-offs. This use case aims to look into the relation between EDP and runtime frequency for compute bound and memory bound applications. Figure 4.4 shows the energy delay product (EDP) of three classes of applications running on an Intel Sandy Bridge Xeon E5-2690; memory-bound, compute-bound, and compute-bound

with front-end disabled. The memory-bound workload is a load stream designed to miss on LLC with each load instruction, whereas the compute-bound workloads are different versions of the same ALU instruction stream, similar to the experiment described in the previous subsection, where no instruction dependencies exist. The back-end only version takes advantage of the loop stream detector (LSD) within the Sandy Bridge microarchitecture in order to shut down the frontend (fetch and decode) while executing small loops [DHJS10]. These two-cases show the extreme ends of the program phase spectrum from the perspective of the processor core. Every other possible program phase lies within these two extremes, and can be classified based on where they stand in the spectrum. The interesting point here is that memory-bound program phases do not require the core to run at the highest frequency, as most of the work is actually being done by the memory system, outside of the core. Contrary to this, the best EDP for compute-bound program phases requires the program to run on the highest frequency available.

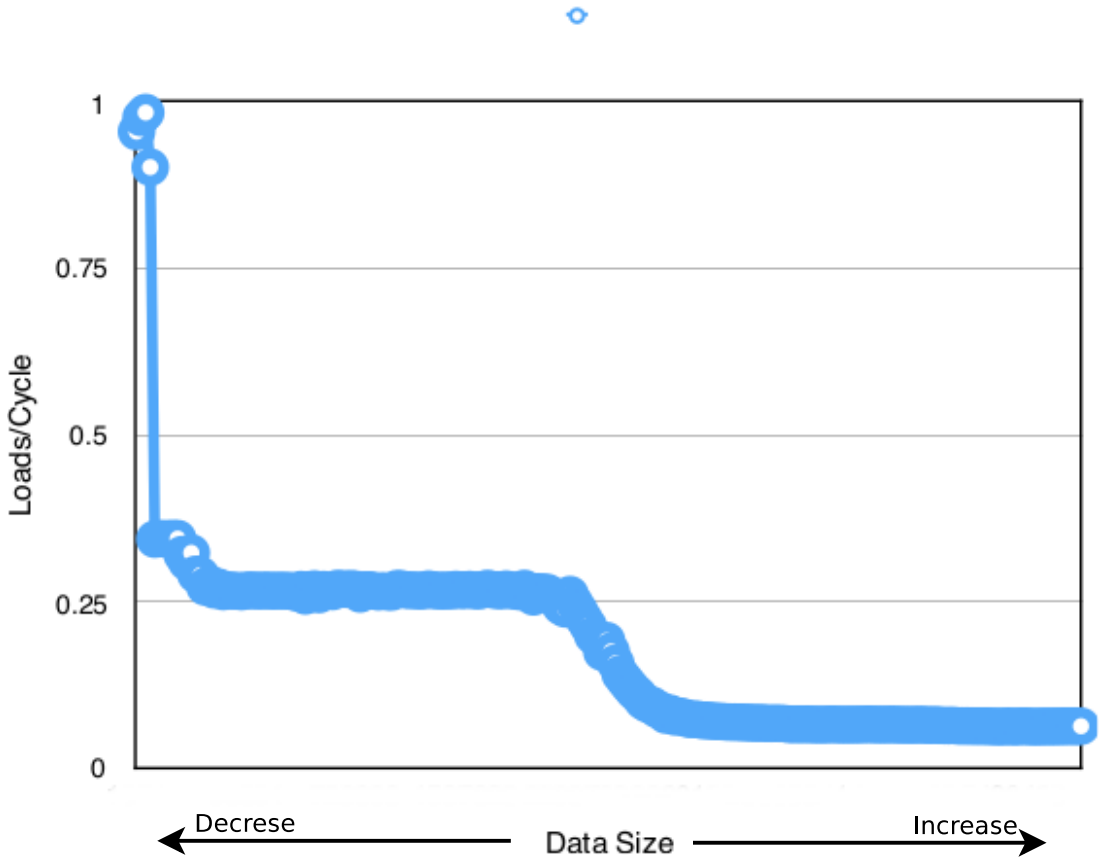


Figure 4.5: Effect of memory latency

Figure 4.5 shows how the average load instructions executed per cycle for a loop of

load instructions with a streaming behaviour vary based on the size of the input data. It can be observed from the figure that as the data size increases the number of loads committed per cycle decreases. This is solely due to the increase in time it takes for a memory access to be filled as the data block accessed by the loop cease to fit in a higher level of the memory hierarchy as data size increase.

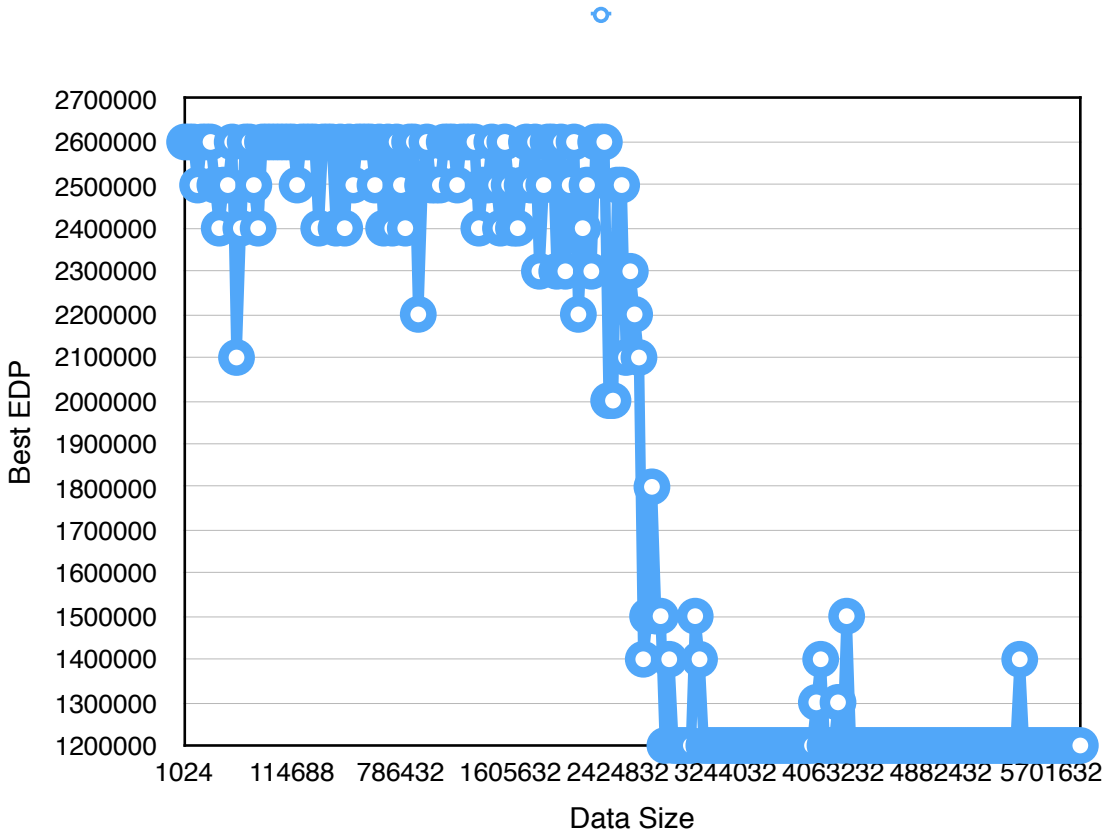


Figure 4.6: Best EDP based on frequency selection

Figure 4.6 shows the running frequency that provides the best EDP for the aforementioned looping load stream routine. From the figure, it is evident that the best EDP is the lowest frequency when the size of the input data is larger than the last level cache size of the processor, that is 20 MBs in this case. The GLAM specification file used in this subsection for the memory-bound microbenchmark is provided in Listing B.2, whereas the compute-bound specification is given in Listing B.1.

4.2.3 Measuring Energy Per Instruction

Energy per instruction (EPI) is a metric used to indicate the dynamic power consumption of a single instruction throughout the execution. It is possible to use GLAM to

Table 4.1: Energy Per Instruction Measurements

Instr. / **Core**	Integer add	Int multiply	Single-FP add	Single-FP multiply	Double-FP add	Double-FP multiply
A53	73pJ	100pJ	86pJ	81pJ	121pJ	380pJ
A57	249pJ	545pJ	351pJ	358pJ	485pJ	820pJ
Xeon i7-4770	1206pJ	3494pJ	5344pJ	6492pJ	6708pJ	10924pJ

measure the EPI in an architecture independent manner. In this subsection, a methodology for measuring EPI using GLAM on different architectures will be shown. The total energy consumption for a processor can be decomposed into idle and dynamic factors (Equation 4.1). The idle power amounts to the static energy, also called leakage. E_{idle} is the constant energy consumed to retain the transistor supply voltage. The dynamic power, given by $E_{dynamic}$, is the energy consumed while there is switching activity in the processor blocks that are being utilised.

$$E_{total} = E_{idle} + E_{dynamic} \tag{4.1}$$

In order to measure the EPI, the average idle power dissipation when the processor is not doing any work is firstly measured. After the idle power is measured, sequences of instructions for the type of instruction to be measured is generated. For this purpose functions that consist of looped sequences of 1024 instructions ($N_{instructions}$) for integer add, integer mul, single-precision floating point add, single-precision floating point mul, double-precision floating point add, and double-precision floating point mul are generated. The instruction dependency distance has been set to four in order to fill all of the available functional units (N_{FU}) for the instruction being measured. After the execution of the generated instruction sequences, EPI can then be calculated as in Equation 4.2:

$$E_{instruction} = \frac{E_{total} - E_{idle}}{N_{FU} \times N_{instructions}} \tag{4.2}$$

The EPI results for measurements taken on an ARM Juno Development platform and an Intel Xeon E5-2670 processor are provided in Table 4.1. The methodology used for the EPI measurements have been explained in Chapter 3 (Figure 3.10). The numbers given in Table 4.1 reflect a single execution of the experiment since variation from execution-to-execution was observed to be negligible.

Juno platform is a big.LITTLE architecture with four A53 cores, and two A57

cores. The idle power was measured beforehand with DVFS disabled in order to prevent fluctuations in energy consumption that might be caused by changes in voltage and frequency. A53 cluster frequency was set to 850MHz, which is the maximum available frequency for the little cluster. A57 cluster frequency was set to 1.1GHz, again being the maximum available frequency for the big cluster. Idle energy consumption over a period of one second for the A53 cluster has been measured as 0.124J, and 0.19J for the A57 cluster. Each microbenchmark was pinned to a single A53, and A57 core in separate executions and the execution energy was measured over each run.

Intel Xeon E5-2670 is a high-end server processor with eight cores. The idle energy over a second was measured to be 0.75J. The measurements were obtained using the on-chip RAPL counters with DVFS and hyperthreading disabled, and the core frequency set to 2.4GHz.

4.3 Conclusion

GLAM has been introduced in this chapter as a way of generating michrobenchmarks for an intermediate representation that provides enough proximity to machine language semantics while also retaining the portability offered by being a high-level abstraction. Three use-cases for GLAM were provided, and another use-case where GLAM generated microbenchmarks are used for faster architectural evaluation on simulators will be shown in Chapter 5. In the next chapter Cyclic Power Gating will be introduced as a novel method of power management in out-of-order microprocessors. The tool is set to be released with an open source licence in 2018.

Chapter 5

Cyclic Power Gating

The increase in power dissipation per chip area caused by Moore's law providing denser chips combined with the end of Dennard scaling can lead to the under-utilisation of available transistors within a chip [HGV+06, EBA+11]. It became evident that it might not be possible to operate these transistors at a high frequency as this mode of operation required higher supply voltages. But having densely packed chips with transistors supplied at higher voltages causes thermal problems, that in turn lead to reliability problems. Since thermal issues are rooted in increased power density, power efficiency has become a first order design constraint.

This has steered researchers in the domain of computer architecture towards introducing power management techniques at different granularities to overcome power density-related problems. Initially this led to the rise of symmetrical multi-core and symmetrical multi-threaded designs [BDM09]. After the realisation that the scaling of symmetric multi-core systems would come to a halt due to limiting issues such as cache coherence, power dissipation, and memory wall, different multi-core designs began to emerge [EBA+11]. General Purpose GPU (GPGPU) systems tried to tackle the memory wall problem by introducing a many-thread architecture through smaller but faster memory hierarchy designs. On the other hand, asymmetric multi-core processors [Gre11] started to emerge, driven by the application requirements for smartphones in particular.

There have also been propositions to tackle this problem using circuit level techniques that are independent of architecture and topology, such as Voltage and Frequency Scaling (VFS) and power gating.

Voltage and Frequency Scaling (VFS) works by scaling the supply voltage and the operating frequency of the core in parallel. Dynamic Voltage and Frequency (DVFS)

is a technique to drive VFS either through a software, or hardware interface. DVFS can provide significant energy savings when computation requirements for an application are not intense [SKK11]. As discussed in Chapter 2, voltage and frequency are dependent on each other, and that the maximum frequency that a circuit can run at is decreased when voltage is lowered. There are two main issues concerning the usability of DVFS in future technologies. Firstly, the contribution of static power to the total power consumption is becoming important due to the increasing transistor densities as described in the first paragraph of this chapter. Secondly, process variability in sub 65nm nodes [HW04], and transistor wear-out results in variation in the threshold voltage of transistors [Bor05] and this affects the operational range of DVFS. It has been suggested that there would be a standard deviation of approximately 33% in threshold voltages of the mean threshold voltage [MSZ+11].

Power gating works by cutting off the supply voltage to a circuit block, such as the whole core or a microarchitectural block such as the register file [HBS+04, RRK11]. Although power gating is a feasible way of decreasing static power dissipation, it is mainly applicable at times when a circuit block is detected to be idle.

When a system is power gated all internal state is lost. Therefore, power-gating is conventionally applied once a core is detected to be idle, when the core frontend is shut off and all the instructions in-flight are committed. There are several different methods to preserve the state of the core making it possible to power-gate while the core is active.

There have been alternative proposals to exploit power-gating by injecting idle-cycles from the software layer, thus driving the core into an idle state to achieve thermal control over the system [SMC+13]. The shortcoming of this approach is the need to switch contexts for the control algorithm to work and issue idle cycles, which amounts to a granularity of tens of milliseconds.

State-retentive power-gating is an approach that can be used as a means of achieving finer granularity, at the expense of leakage savings. This is due to the introduction of extra state-retentive logic into the circuit. State-retentive power-gating on a processor core enables the power-gating of circuit blocks within the core by saving the run-time state of the power-gated circuit block to state-retentive registers. This enables the processor to be power-gated regardless of whether the processor is idle or not. Since the run-time state is stored closer to the processor, the wake-up time incurred by state-retention is lesser compared to conventional power gating.

In this chapter an aggressive form of state retentive power gating, called Cyclic

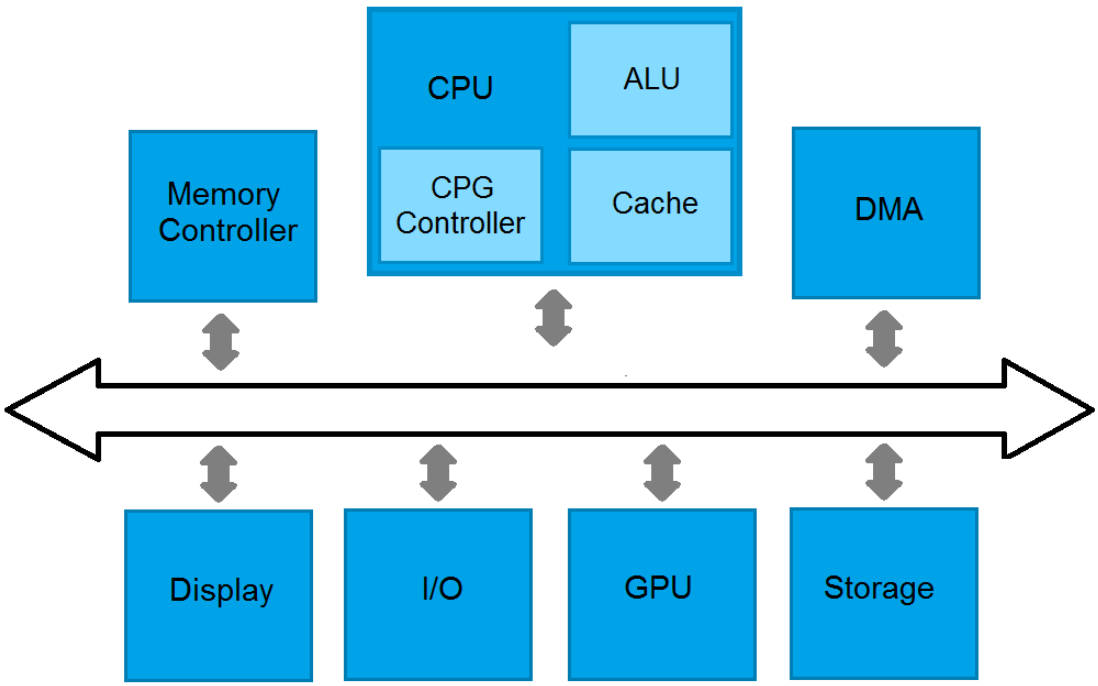


Figure 5.1: Block Diagram of a System-on-Chip with a CPG Controller.

Power Gating (CPG) is proposed, and evaluated. CPG power-gates the core on and off over an adjustable period whilst retaining the run-time state of the core for faster wake-up and sleep time transitions. In designs where VFS is not available, CPG can be utilised as an alternative to DVFS with lesser static power dissipation and finer-granularity for switching frequencies. The effective frequency of a processor can be set by changing the off rate, which is the ratio of off and on periods within a single power-gating period. A comparison of temporally equivalent nodes to a VFS configuration has been included in the evaluation section of this section in order to show the feasibility of the idea. CPG can also be employed alongside VFS since there are no technical limitations that would prevent those two approaches operating together, possibly leading to higher power savings, although this is out of scope regarding this dissertation. A state-retentive power-gating method, that is able to save and restore the processor state during sleep and wake-up periods, is employed in order to minimise the performance impact of power-gating. A controller that drives the CPG needs to be implemented in the core, in order to provide higher responsiveness. Figure 5.1 shows how the CPG controller fits in a System-on-Chip (SoC) architecture.

CPG operates as shown in Figure 5.2. It works by power-gating the core on (retaining voltage supply to the virtual voltage domain) and off (by cutting off the voltage

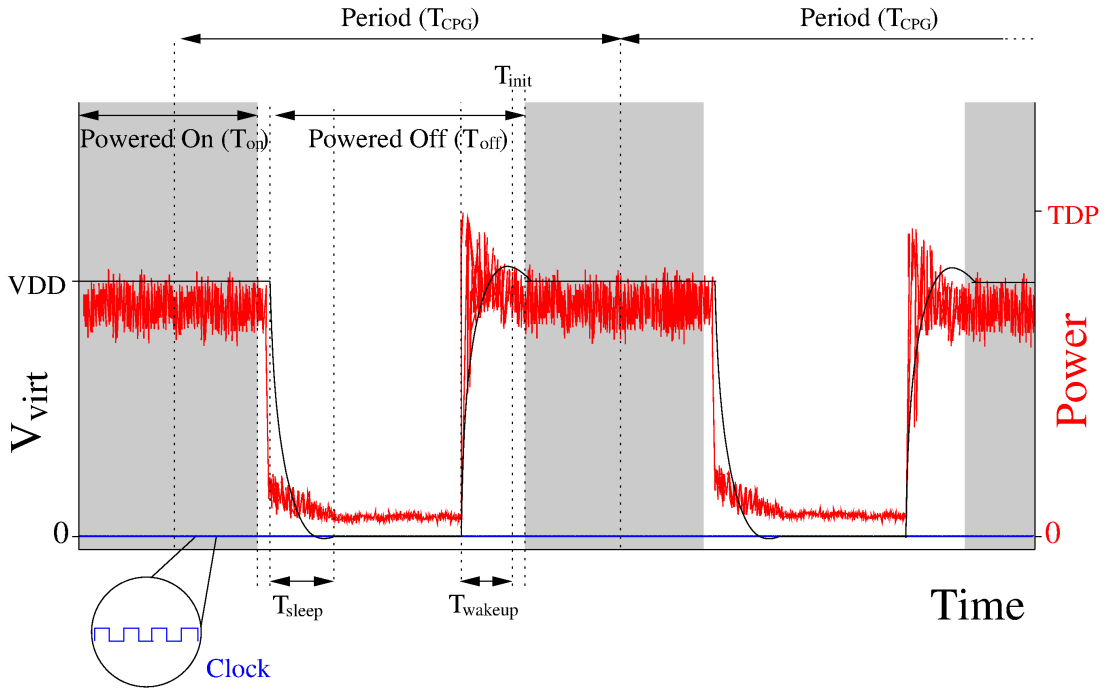


Figure 5.2: Cyclic Power Gating.

to the virtual voltage domain), effectively changing the operating frequency while the supply voltage and clock frequency is constant. A more detailed description of this figure is given in Section 5.2.

The effective operating frequency of the core is given by the ratio of on cycles and the sum of on and off cycles throughout the fixed period, T_{CPG}. This can be calculated using Equation 5.1.

$$CPG_{duty\_cycle} = \frac{T_{CPG} - T_{off}}{T_{CPG}} \tag{5.1}$$

T_{off} is defined as the number of off cycles within T_{CPG}. A timer block in the CPG controller counts the off cycles given by T_{off}, and this count is reinitialised when a new CPG period begins. This makes it possible to change the duty-cycle (i.e. the effective core frequency), between any two CPG cycles without incurring any extra overheads other than the sleep and wake-up processes. The duty-cycle can theoretically be assigned any value between $[0:1]$. This means that the effective frequency of the core can be scaled to any value that is lower than the operating frequency. This provides greater flexibility in scaling the frequency than DVFS where the number of possible voltage and frequency pairs for a given technology node is limited.

Throughout the rest of this chapter, the CPG architecture is presented in detail,

providing an evaluation of the overheads incurred, and providing results for the evaluation of how it compares with VFS. The detailed evaluation of the overheads provides the identification of a break-even time, that is the minimum number of cycles to sleep for which the state-retentive power-gating methodology applied consumes the same amount of energy as it saves.

5.1 State-Retentive architecture

The state-retentive power-gating design that is employed in the proposed Cyclic Power Gating (CPG) architecture is presented in this section.

The block diagram for the state-retentive architecture is given in Figure 5.3. All of the memory structures within the architecture are implemented using SRAM cells, with the exception of pipeline registers that are implemented as D-type Flip Flops (DFF). The cell-array of each SRAM is connected to a data-retention voltage supply that retains the power supply during the sleep periods of the core. Elements that are part of the memory hierarchy, L1 caches, and the TLBs, are not power-gated to allow outstanding memory requests to be completed while the core is sleeping. This enables the exploration of memory access latency-induced pipeline stalls for saving energy. There is also additional overhead incurred for clamping and retaining data and initialising the latches and clocks [JKK+12]. The modelling parameters will be provided in detail in the experimental methodology section.

5.2 Power-Gating overheads

The sleep and waking-up processes do not occur instantly, as can be seen from Figure 5.2 for the T_{sleep} and T_{wakeup} periods. This means that power is still dissipated during these periods, which can lead to power-inefficient execution for cases where the power dissipated during sleep and wake-up phases is greater than the power saved during the rest of the CPG cycle. Power-gating overheads during T_{sleep} and T_{wakeup} are caused by the charging and discharging of the Power Delivery Network (PDN).

Figure 5.4 shows how the power-gated sections are connected to a virtual power supply, V_{virt}, which is connected to V_{dd} through header switches. The header switches are turned off once the core is power-gated, causing the energy stored in the PDN to be discharged through the leakage current of the transistors. When the header switches are back on, the core wakes up, the PDN is recharged and V_{virt} is connected back to

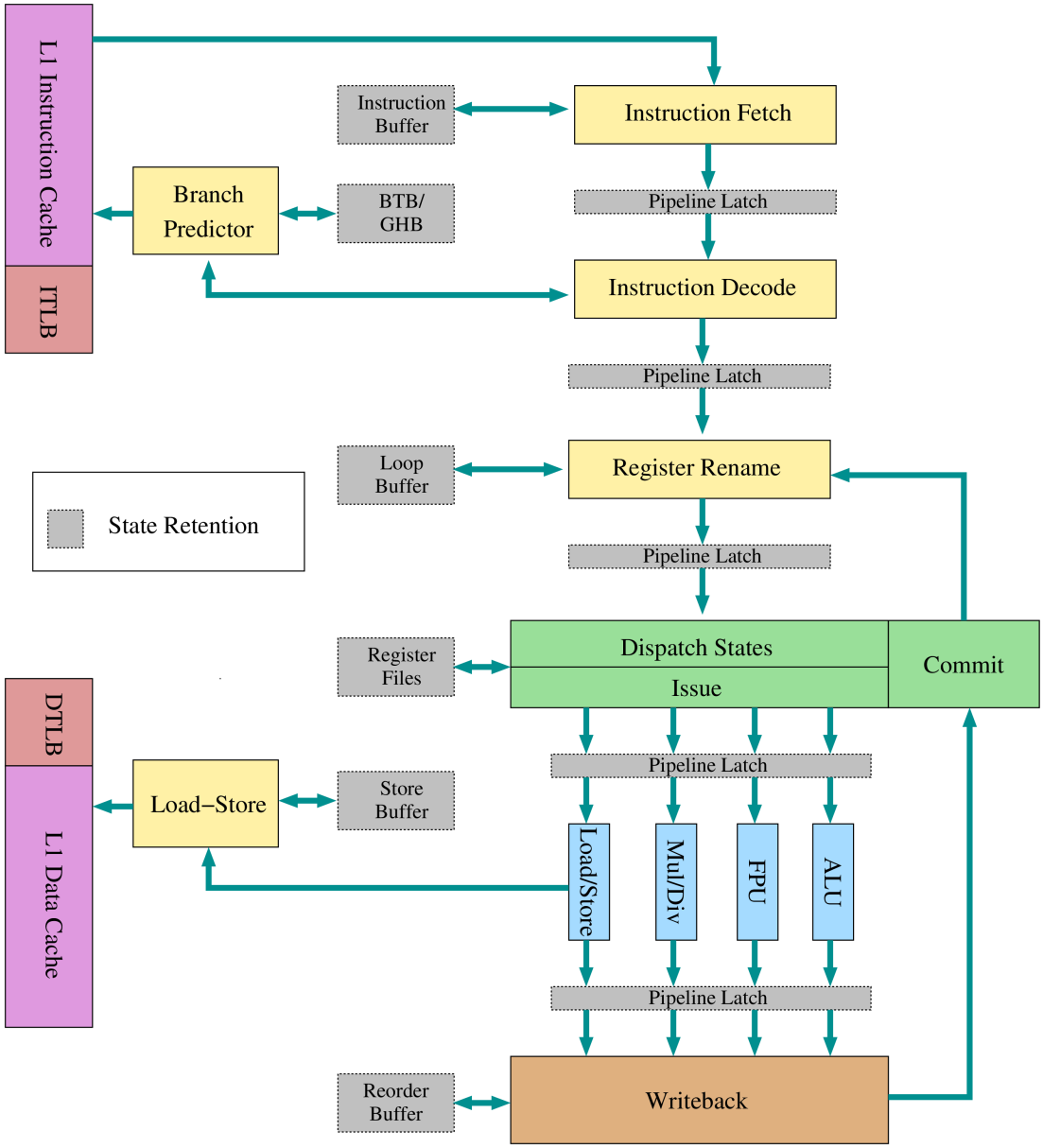


Figure 5.3: State-retentive Architecture

V_{dd}. The time and energy required to transition V_{virt} from zero to V_{dd} is determined by C_D, which consists of the capacitances of the transistors connected to the power supply C_s, the capacitances of the header switches C_h and the power supply decoupling capacitance C_d. The energy that is consumed to charge V_{virt} is given by Equation 5.2:

$$E_{charge} = \frac{1}{2} C_D V^2. \tag{5.2}$$

In-rush current is the current that flows through a circuit when the circuit is first

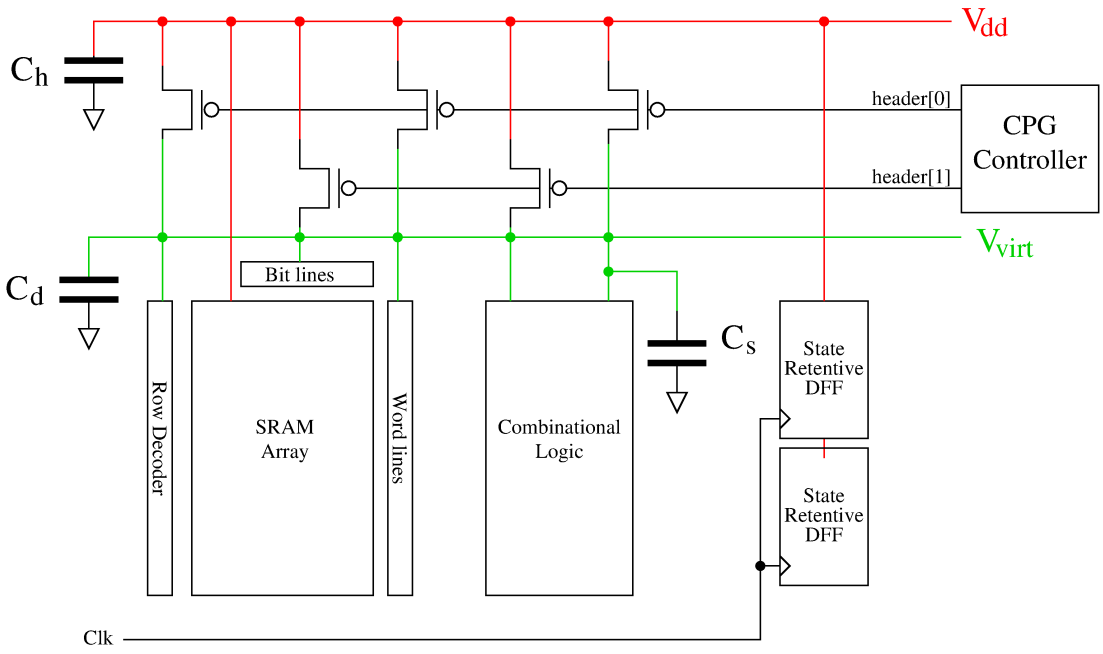


Figure 5.4: State-retentive Power Gating

powered on, and is larger than the steady state current. A high in-rush current during the waking up of the core can result in voltage droop, which may lead to potentially erroneous behaviour. The header switches shown in Figure 5.4 are partitioned into two groups (header[0] and header[1]) and activated sequentially to prevent possible faults from a large in-rush current. The number of headers in header[0] has been chosen so that the limited in-rush current drawn does not exceed the current limit (I_{TDP}) bound by the Thermal Design Power (TDP) as described in [JKK+12].

With the limited current, the time taken to charge the PDN is as given in Equation 5.3:

$$T_{charge} = \frac{C_D V}{I_{TDP}}. \tag{5.3}$$

The header switches in header[1] are turned on as soon as V_{virt} is charged to 90% of V_{dd}. The total time it takes for the charging process to complete is given by $T_{wakeup} = 2 \times T_{charge}$.

The overheads of power-gating have been examined through a state-retentive architecture model generated using the McPAT power estimation tool [LAS+09]. The McPAT power modelling tool has been modified to sum up all the capacitances inside the circuit in order to estimate C_D, using the assumptions as explained in [HBS+04] where $C_h = 0.1C_s$ and $C_D = 0.2C_s$. I_{TDP} has been obtained from the TDP calculation performed by McPAT.

In-rush currents might also cause voltage noise events that might potentially harm the circuit. It has been ensured using the Voltspot PDN modelling tool [ZWM+14] that the calculated power-gating in-rush currents do not cause any voltage noise greater than the safe margins, set around 5% of V_{dd}.

When C_D is discharged, the core is in a steady power-gated state. The only power dissipated at this point is due to the leakage current of the state-retentive SRAM components. To calculate the leakage for state-retentive elements, leakage currents of the state-retentive cell arrays for each SRAM component have been decoupled from the rest of the leakage of the core within McPAT. It is possible to calculate the energy consumed during the power-gated state using these decoupled values of capacitance, leakage, and TDP current.

A key value is the *break-even time*, T_{be}, which is the time below which power-gating consumes more energy than it saves. A *simplified* estimate of break-even time can be calculated in terms of cycles of the CPU running at nominal frequency by using the per-cycle TDP energy:

$$E_c = \frac{TDP}{f_{max}} \tag{5.4}$$

The energy saved while the CPU is power-gated is related to the ratio of current while the CPU is active (I_{TDP}), to state retention leakage (I_{off}^{pg}). Therefore, the number of cycles the CPU needs to be power-gated to offset power-gating overhead is:

$$c_e = (\frac{E_{wakeup}}{E_c} + c_{overhead}) * \frac{I_{TDP}}{(I_{TDP} - I_{off}^{pg})} \tag{5.5}$$

where $c_{overhead}$ is the number of cycles for draining/filling and initialization:

$$c_{overhead} = (T_{prep} + T_{init}) * f_{max} \tag{5.6}$$

The actual time taken to perform the power-gating operation is:

$$c_{pg} = c_{overhead} + T_{wakeup} f_{max} \tag{5.7}$$

where f_{max} is the nominal clock-frequency. The break-even time is therefore:

$$c_{be} = c_e + c_{pg} \tag{5.8}$$

For the target architecture presented in this section this value is ∼50 cycles (22ns).

5.3 CPG Power and Evaluation Strategy

Power dissipation of the CPG scheme is given by Equation 5.9:

$$P = P_{on} * duty\_cycle + (P_{off} * (1 - duty\_cycle)) + \frac{E_{pg}}{T_{CPG}}. \tag{5.9}$$

It can be seen that the power dissipation is inversely proportional to T_{CPG}, which is the period for a single on-off phase. This is due to the power gating overhead given by E_{pg} being constant for a single on-off phase, as the core sleeps and wakes up only once. The power dissipated while the core is awake is given by P_{on}, whereas the sleeping power dissipation is given by P_{off}. The overheads incurred by E_{pg} includes overheads for pipeline filling and draining along with initialisation. Since the caches are not power-gated during the period the core sleeps, their contribution to the overall power dissipation is not tied to CPG. These aspects will be inspected in the analysis for the memory-bound workload in the evaluation section.

An initial evaluation of CPG is done by executing two microbenchmarks that represent the extreme ends of the application phase spectrum. *Instruction dependency distance* is the distance between the first read of the register that a previous instruction has written to. The compute bound microbenchmark is a series of Arithmetic Logic Unit (ALU) adds with an instruction dependency distance of four that accesses the L1 data cache at a rate of 0.05 per instruction. The choice of a dependency distance higher than the integer ALU issue width of the processor was made in order to minimise inter-instruction dependencies which would lead to a slowdown in execution. The memory bound instruction consists of a sequence of load instructions that are looped over, and it generates an LLC miss rate of 0.78 per instruction. The details of the microbenchmarks will be provided along with how they execute with the CPG scheme will be provided in the following subsections.

5.4 Experimental methodology

To investigate CPG, a simulator platform consisting of gem5 simulator modified to execute periodic calculations for power and temperature using McPAT and Hotspot was built. The cycle-accurate gem5 [BBB+11] was used since an implementation of the CPG scheme affects the timing behaviour of the system.

The event-driven architecture of the gem5 makes it possible to implement new simulation objects that can issue events into the system event queue, and take action when

events are triggered. CPG events were implemented by making use of the gem5 events system. CPG events schedule core sleep and wake-up events based on the CPG period and off rate parameters that are provided at simulation configuration time. One problem this approach causes is in interrupt handling. A received interrupt needs to be reposted if it was received while the core is asleep. This functionality has been implemented in order to avoid deadlocks, and erroneous behaviour that might be caused by the missing of an interrupt by the operating system.

The simulation tool flow is given in Figure 5.5. A CPU similar to the ARM A57 at a 14-nm technology node was modelled. The simulation parameters can be seen in Table 5.1. Chapter 3 can be consulted for a detailed description of the simulation infrastructure.

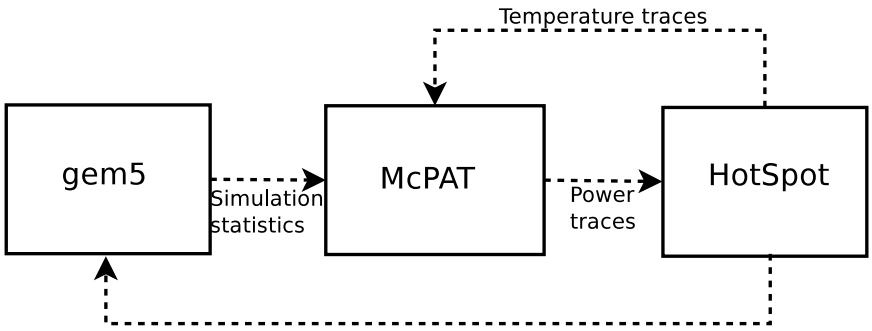


Figure 5.5: Tool flow used for simulations.

Section 5.4 presented the experimental methodology used to evaluate the material presented in this Chapter. The evaluation of the CPG scheme for memory-bound and compute-bound workloads will be presented in Section 5.5. The applicability of the scheme will be shown by evaluating it on two microbenchmarks generated using the Generator of LLVM Microbenchmarks (GLAM) tool presented in Chapter 4. This will be followed by a comparison of CPG with VFS for SPEC CPU2006 benchmarks in Section 5.6. Section 5.7 will present how CPG can be applied at a program function granularity, and the chapter will be concluded.

Table 5.1: Simulation parameters

Parameter	**Configuration**					
Architectural Parameters						
Processor	ARMv8					
Machine width	3-wide (issue upto 8 uops)					
L1 instruction cache	48KB 3-Way Set-Associative					
L1 data cache	32KB 2-Way Set-Associative					
L2 private cache	512KB 16-Way Set-Associative					
L3 shared cache	8MB 16-Way Set-Associative					
Memory Size	8GB					
Physical Register File	128-Entry Integer, 192-Entry Float					
Load/Store Queue	16-Entry					
Reorder buffer	40-Entry					
Issue Queue	32-Entry					
Physical Parameters						
Area	$2.2mm^2$					
TDP	3W					
C_D	17.3nF					
Wake-up Energy	7.98nJ					
Wake-up Time	5.27ns					
Sleep Time	8.3ns					
Break-Even Time	20.7ns					
Voltages (V)	1.00	0.96	0.92	0.88	0.84	0.80
Frequencies (GHz)	2.00	1.74	1.50	1.26	1.00	0.73
Equivalent off rate	0.0	0.13	0.25	0.37	0.50	0.63

5.5 Evaluation

This section will provide an evaluation of the CPG scheme by investigating how it performs when executing programs with differing amount of time spent on core. This evaluation is made on the premise that the benefit of CPG is inversely proportional with the time of execution spent in-core, meaning that CPG will prove beneficial for

memory-bound workloads.

5.5.1 Compute bound evaluation

The compute bound microbenchmark is a loop containing 16 add instructions, with an instruction dependency distance of four, that operate on 64-bit registers with their values stored in the processor registers at all times. The number of instructions were chosen in order to minimise the contribution of data cache misses that are introduced from the execution of the loop. The only data cache accesses that are made during the execution is for the loop counters since they are stack-allocated. An instruction dependency distance of four was chosen to generate highly instruction level parallel code to ensure a high utilisation of the out-of-order core being simulated. The benchmark is executed until the CPU commits 4.75 billion instructions after a 250 million warm-up window. The number of committed instructions have been selected arbitrarily, and a change in this parameter would not lead to any changes in results since the executed code is homogeneous. The execution of the application is compared with nominal CPU operation at 2GHz and CPG periods of 100, 200, 300, 400, and 500 nanoseconds with off rates of 10%, 20%, 30%, 40%, 50%. Each bar in the x-axis represents an execution of the compute-bound benchmark with each x-axis label indicating the CPG period, and the off cycle percentage. For example, 500_30 represents the execution of the benchmark with a CPG period of 500 nanoseconds, and the ratio of the off period to the on period as 30% (i.e. on rate of 70%, or an off period of 150 nanoseconds).

Performance results can be seen in Figure 5.6. Each bar in the figure represents the time in seconds that it takes for the microbenchmark to complete its execution. Nominal operation where CPG is not applied is indicated by the noCPG bar. The fastest and most energy efficient way of completing a compute-bound benchmark is through race-to-idle, where a task is executed at the highest available frequency and without any interruptions [AA14]. Since all the work required for the application execution occurs on the core, the execution time increases monotonically as the core is throttled down more aggressively (i.e. a higher off rate). Comparison of energy consumption for the CPG scheme with various throttling and periods against noCPG is given in Figure 5.7. The plots show a monotonically increasing trend, which is linked to the explanation for performance. The difference in energy added on top of the noCPG execution is the sum of static power and the dynamic overhead of switching that was dissipated during the times that the core was power-gated. The energy overhead shown in the figure matches with the race-to-idle explanation given. Since temperature is a

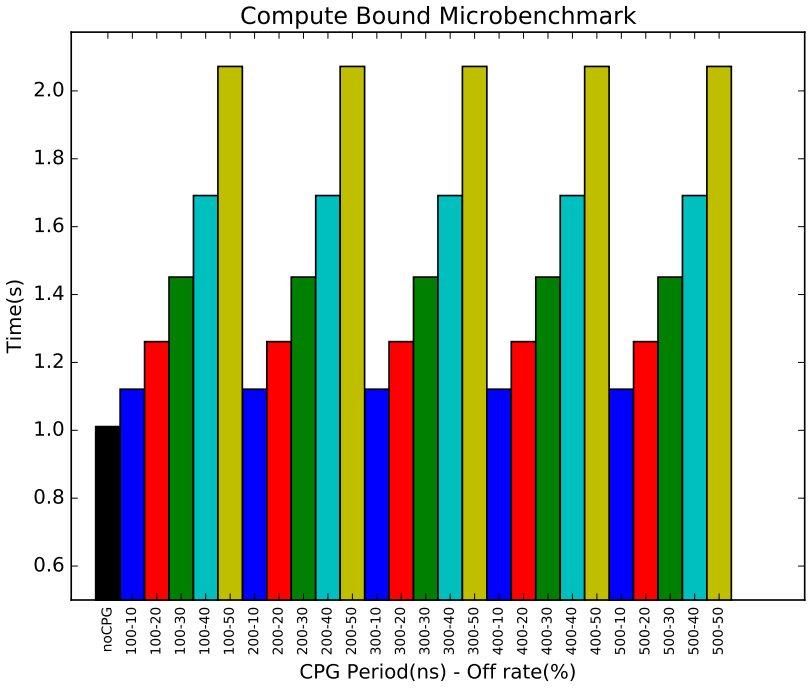


Figure 5.6: Execution Time for a Compute Bound Microbenchmark.

function of average power, the outcome of consuming less power as off-rate increases (Figure 5.8) is a lower final temperature as shown in Figure 5.9. Although a reduction in power do not directly result in energy-efficiency, the technique might provide useful in controlling temperature for cooling purposes when the thermal design power (TDP) limit of the processor has been reached.

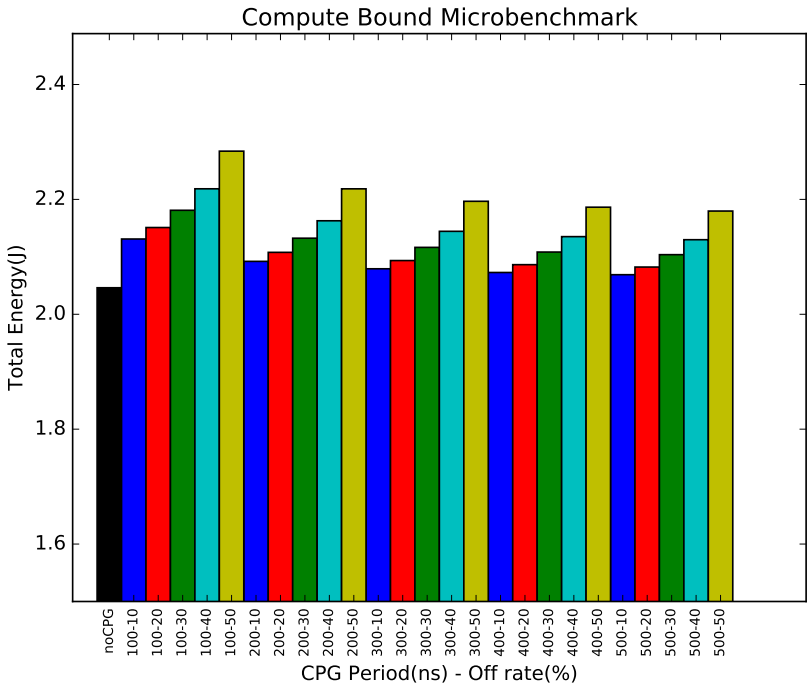


Figure 5.7: Total Energy for a Compute Bound Microbenchmark.

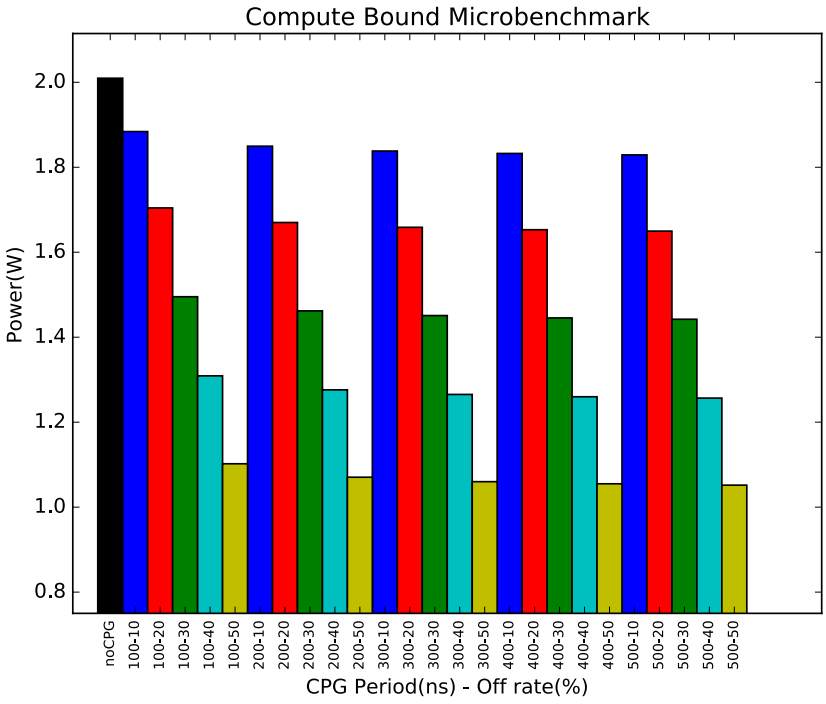


Figure 5.8: Average Power for a Compute Bound Microbenchmark.

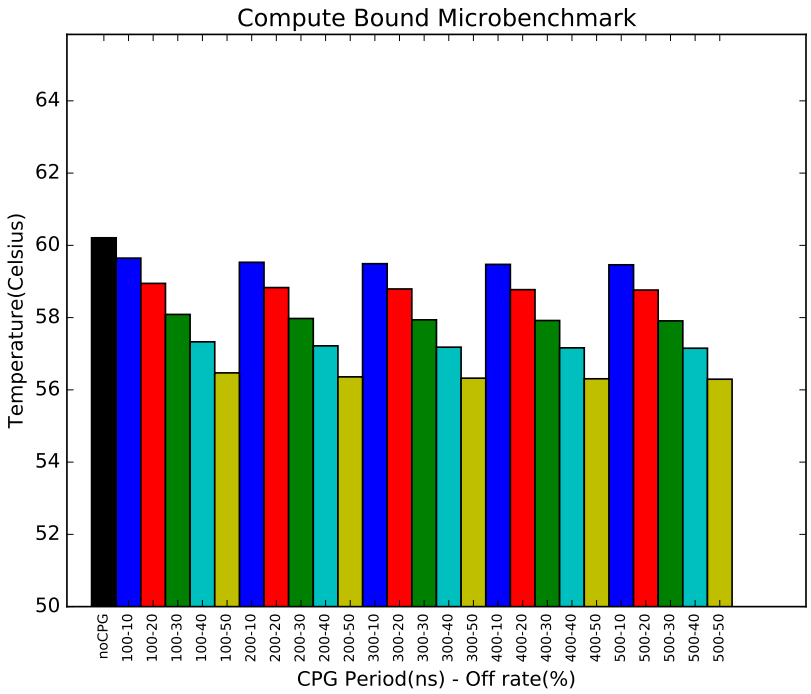


Figure 5.9: Final Temperature for a Compute Bound Microbenchmark.

For compute bound executions, the performance of throttled cores running under CPG is always worse than nominal operation and therefore the power reductions do not lead to EDP savings (Figure 5.10). To conclude, compute bound workloads are highly sensitive to interruptions in execution resulting in reduced performance and energy efficiency. In this scenario CPG may still provide fine-grained temperature control, which may prove its utility in thermal control, and reliability issues.

5.5.2 Memory bound evaluation

The memory bound microbenchmark is a loop containing 32 load instructions that operate on 64-bit registers. The addresses that the data are loaded from are incremented in 64 byte increments causing each load instruction to miss in LLC cache, and the overall miss rate per instruction is 0.78%. Overall miss rate is decreased due to the existence of loop counters that are stack allocated. The choice of 32 load instructions have been made in order to counter this. The microbenchmark has an IPC of 0.02, making it represent a case of a memory-bound program phase. The benchmark is executed until 52 million instructions are committed by the core. The execution of the application is compared at CPG periods of 100, 125, 150, 175, and 200 nanoseconds that are power-gated 10%, 20%, 30%, 40%, and 50% of the time, against a 2GHz (noCPG) frequency

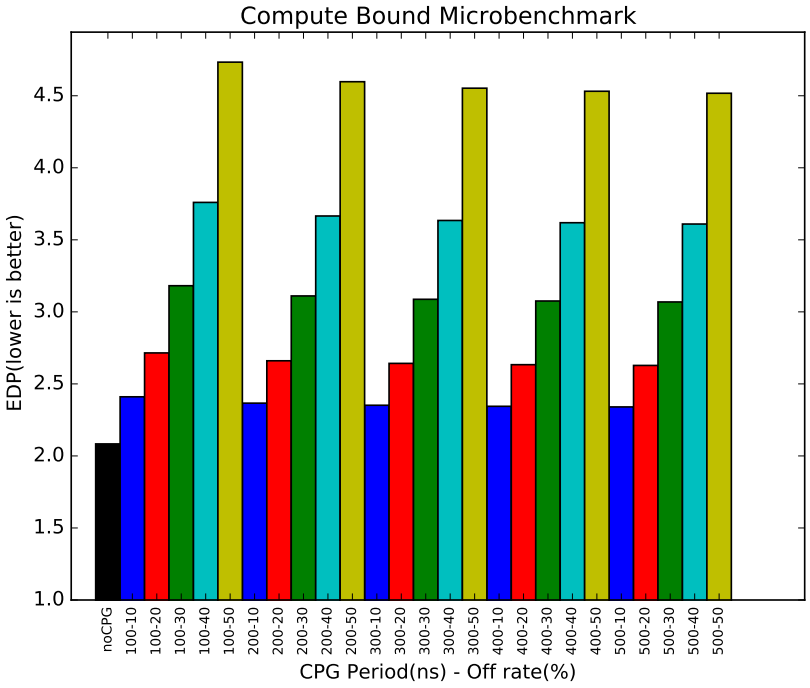


Figure 5.10: EDP for a Compute Bound Microbenchmark.

execution that is not throttled. Smaller increments in period selection have been selected for memory bound evaluation as having high stall periods greater than L3 miss periods would lead to unneccessary power-gating of the core.

As previously stated, the executing code is looping over a block of independent load instructions. The order of execution can be simplified as a cycle of fetch from predicted branch and load. Since the loop counter is not dependent on the load instructions, the loop execution can progress independent of the load instructions. Given a reorder buffer, and reservation stations of unlimited size, this could go on indefinitely in the presence of a branch predictor. Because the load instructions have a long latency to reach the commit stage, any instructions that are executed out-of-order are squashed and re-fetched. This is known as a *replay trap* [ERB+95]. Performance results for the memory bound microbenchmark are given in Figure 5.11. Each bar in the figure represents the time in seconds that it takes for the microbenchmark to complete its execution. A monotonic trend is observed although the differences on the lower CPG periods are negligible. There are two factors contributing to the slowdown; serialisation of load instructions, and stalling of the fetch stage during off periods. Stalling of the fetch stage does not affect execution as much as it does in the compute-bound workload, as there are always long-latency requests that are continuing while the core

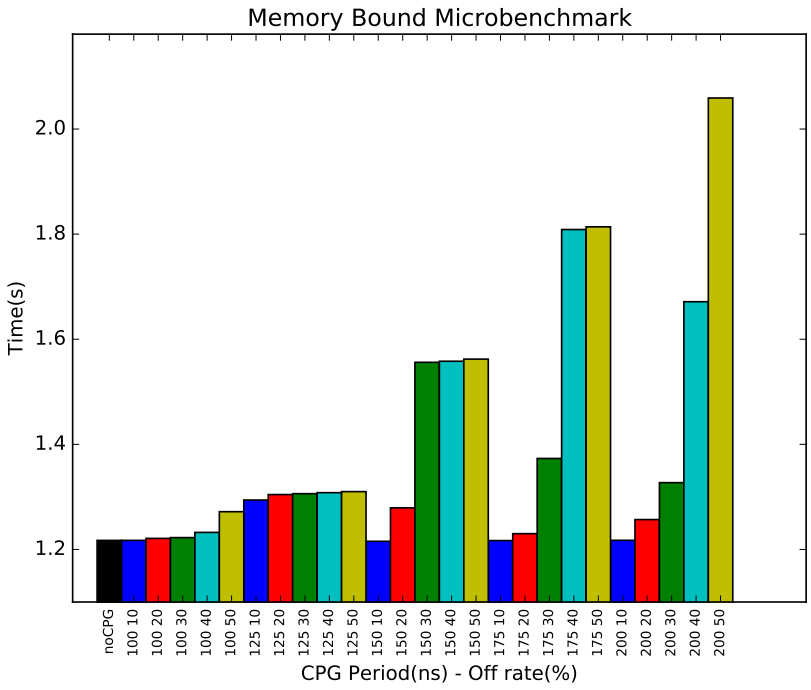


Figure 5.11: Execution Time for a Memory Bound Microbenchmark.

is asleep. Memory requests can be handled in parallel, but the power-gating prevents new requests being issued, resulting in unnecessary delays in starting memory operations. One possible solution to this is equipping the CPG controller with a decision mechanism on when to sleep in the presence of memory instructions.

The CPG scheme has advantage in terms of energy efficiency when the execution is memory-bound. This is reflected in Figure 5.12, where all off rates with all the CPG periods have lower energy footprints compared to nominal execution. This is because most of the execution time of the microbenchmark is spent on the memory system. Contrary to the compute bound results, the plots show a monotonically decreasing trend for values of CPG periods of 125ns and 100ns. Delaying of the instruction fetching due to throttling has an advantage, but once the reorder buffer is full, while there is room for execution, it starts creating inefficiencies in the pipeline. There is also the effect on delaying the issuing of new load requests due to the forced backend stalls especially in the CPG instances with higher periods. Average static power monotonically decreases in a consistent manner as shown in Figure 5.13. Although average power, and average static power (Figure 5.14) are much lower in most instances, the reduction in performance affects execution time leading to discrepancies in overall energy. All of the CPG instances have lower energy consumption compared to the nominal operation.

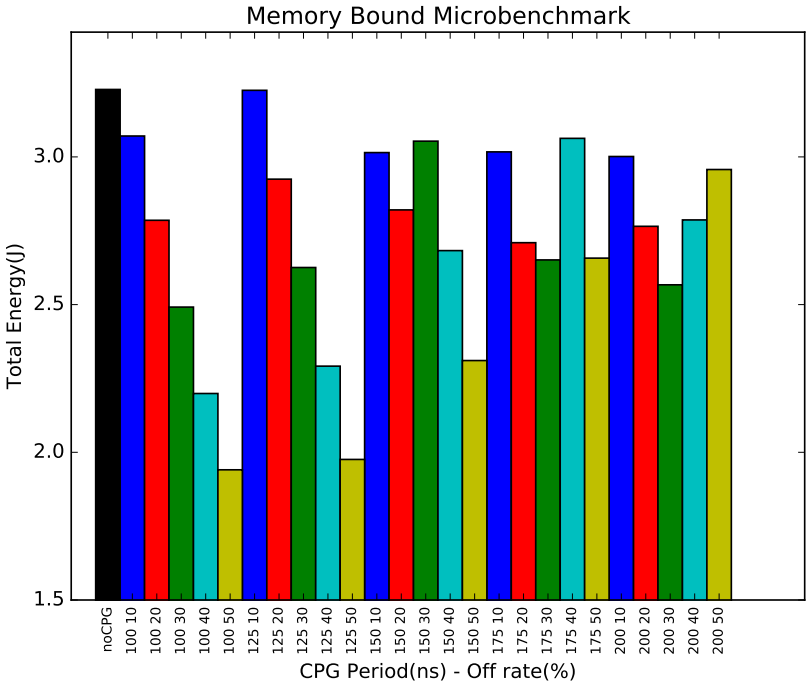


Figure 5.12: Total Energy for a Memory Bound Microbenchmark.

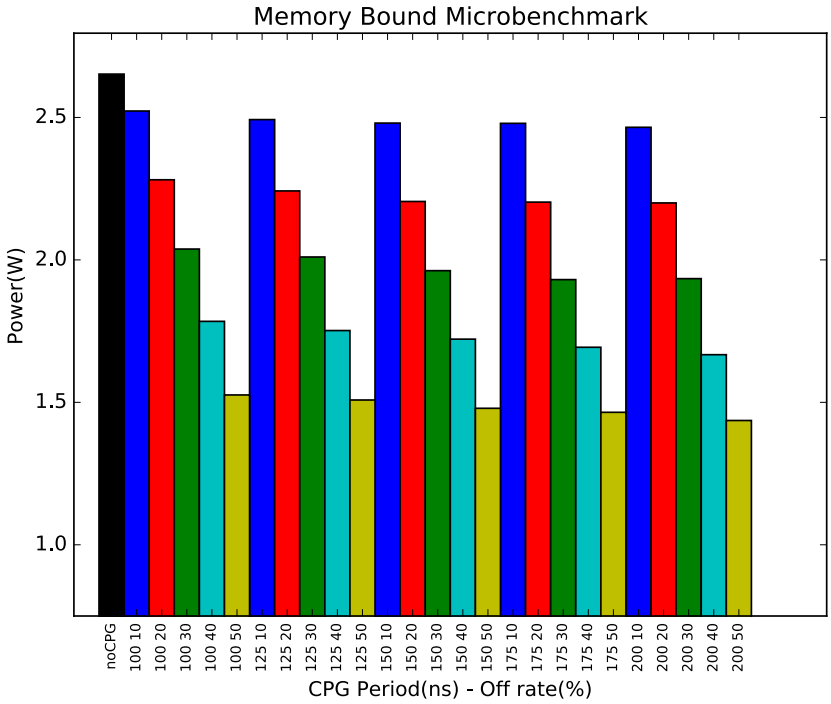


Figure 5.13: Average Power for a Memory Bound Microbenchmark.

Final temperatures are similar to that of compute-bound microbenchmark executions, again showing CPGs utility in temperature control. The plots for final temperatures

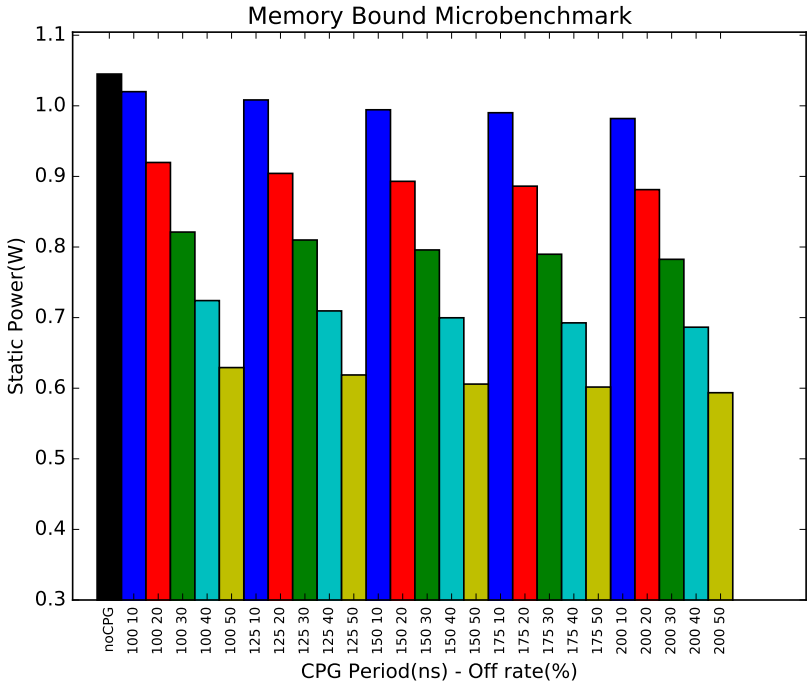


Figure 5.14: Average Static Power for a Memory Bound Microbenchmark.

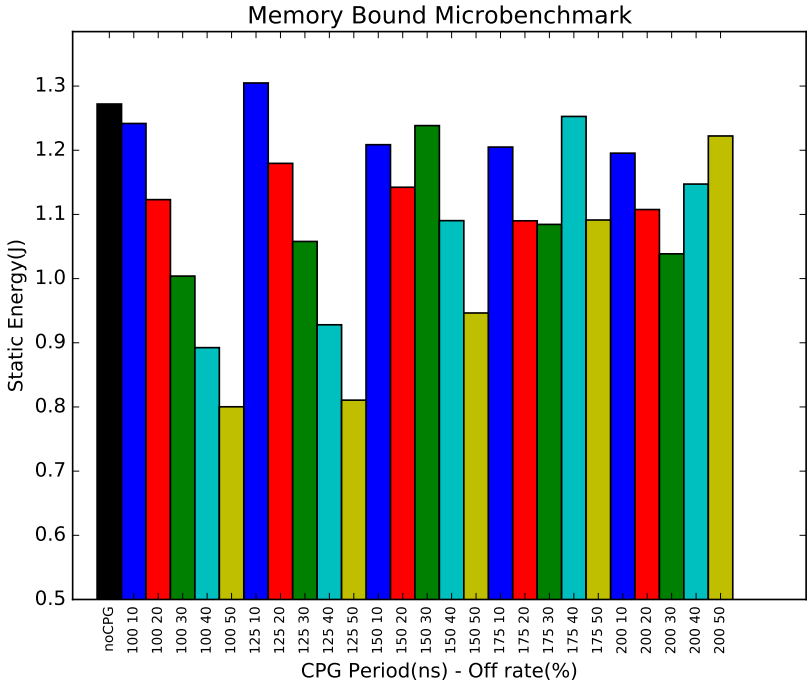


Figure 5.15: Static Energy for a Memory Bound Microbenchmark.

are given in Figure 5.16, and they show a monotonically decreasing trend that is consistent with the trend in average power dissipation. Temperature being a function of

average power, applying CPG in a memory-bound phase shows that thermal control may be achieved while not affecting the performance. This is due to the loosened requirements for core-bound computation, since most of the work is being done in the memory system. Applying CPG in a memory-bound phase is useful in where energy can be saved with negligible performance degradation, with the added advantage of lower thermal dissipation.

Finally, the EDP results are given in Figure 5.17. It is clear from the plots that they do not show a clear pattern for CPG periods higher than 125, which is a result of the previously explained interactions between the front-end and the back-end of the pipeline. For lower CPG periods, it is possible to achieve better EDP than nominal execution (noCPG), which shows that applying CPG on memory-bound workloads can be beneficial in terms of energy savings with minimal degradation in performance.

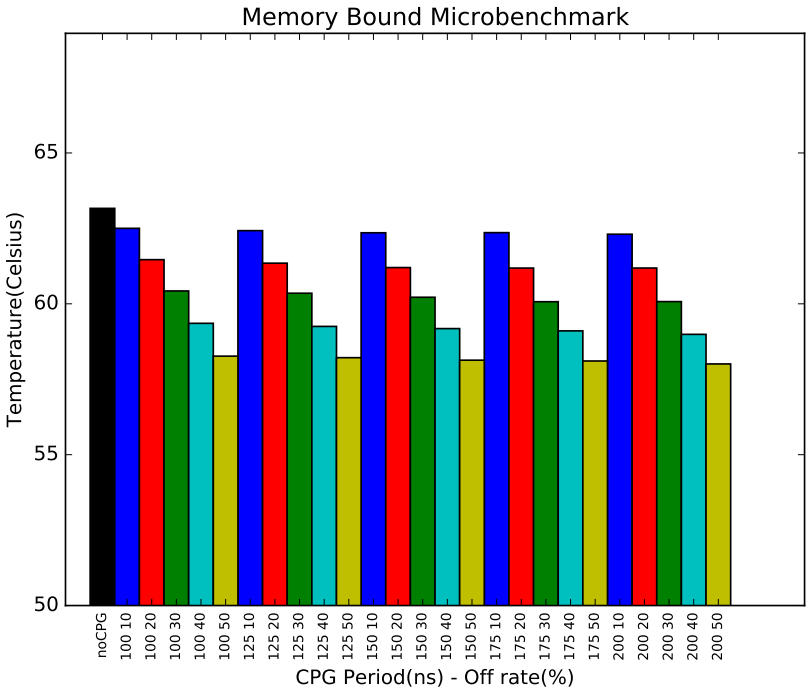


Figure 5.16: Final Temperature for a Memory Bound Microbenchmark.

5.5.3 An analysis of CPG for varying levels of memory intensity

Figure 5.18 shows the optimal EDP gains that can be achieved for program phases that lie in between the compute-bound and memory-bound microbenchmarks that have

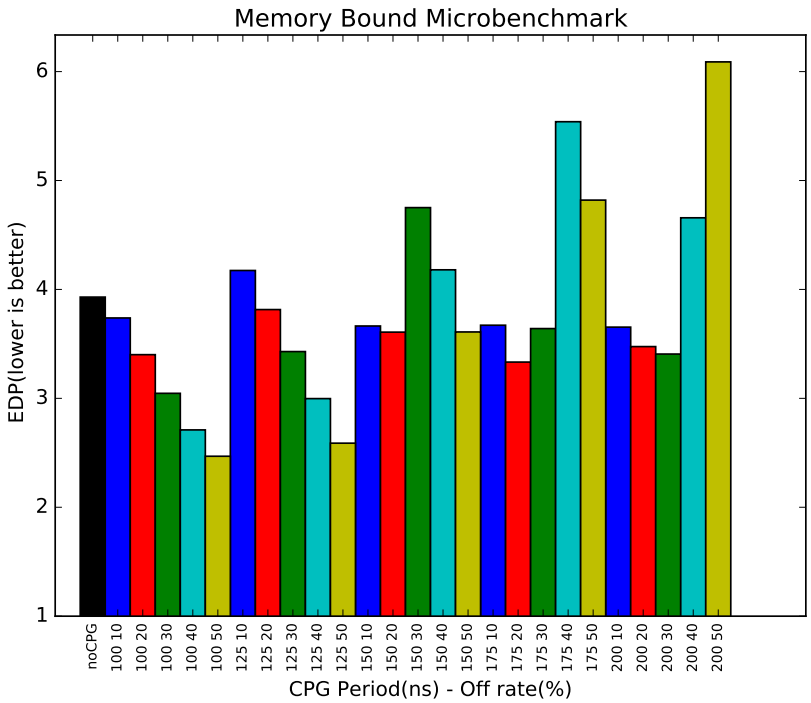


Figure 5.17: EDP for a Memory Bound Microbenchmark.

been presented in the previous subsections. Last Level Cache Misses Per Kilo Instructions (LLCMPK) is a metric that can be used to denote the memory intensity of a program. Microbenchmarks with LLCMPK values of 1, 5, 10, 25, 50, and 100 have been created using the GLAM tool described in Chapter 4. These microbenchmarks have been executed with CPG periods 100, 125, 150, 175, 200 and with off periods of 10%, 20%, 30%, 40%, 50%. Each bar in the figure represents the lowest EDP obtained from the results of these executions normalised against nominal operation. It can be seen from the figure that EDP is improved in all instances except LLCMPK_1, which can be classified as compute-bound.

In the next section a comparison of CPG will be made to show how it evaluates against VFS. Although these schemes can be used together, CPG might prove useful in cases where per-core on-chip voltage regulators are not feasible on chips, or with low-voltage devices where there is no possible means of having multiple voltage and frequency pairs due to lack of enough difference between nominal and threshold voltages.

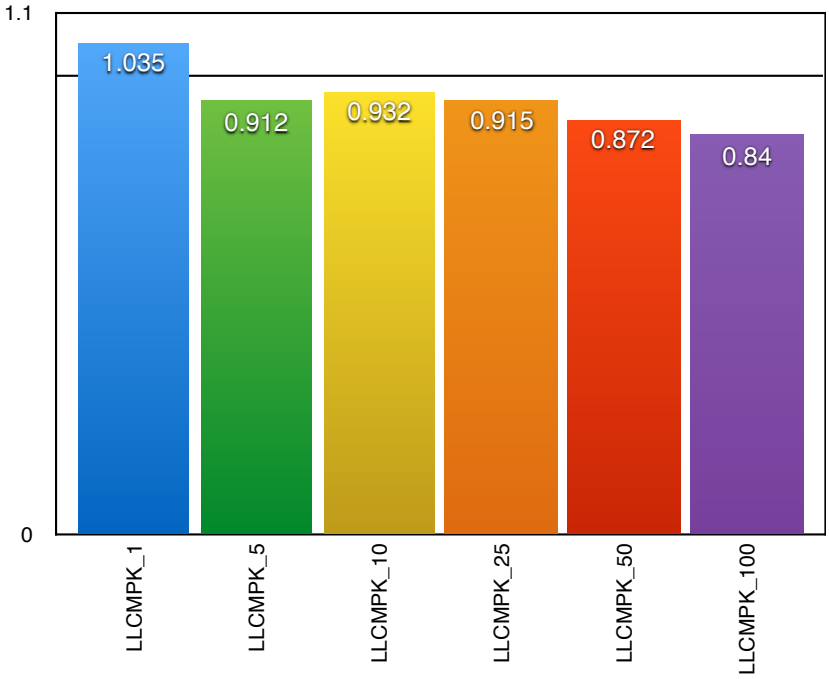


Figure 5.18: EDP winners of CPG executions accross benchmarks with differing LLCMPK.

5.6 Comparison with VFS

A comparison of VFS and CPG in terms of performance, power and energy-delay-product (EDP) is presented in this section. These two power management approaches were compared for benchmarks from the SPEC CPU2006 benchmark suite [Hen06b].

SPEC CPU2006 benchmarks has a wide range of application profile such as memory intensive graph traversals, core-bound computation, and irregular branching behaviour. The descriptions of the benchmarks are given in Table 5.2. Each benchmark has been executed until five billion instructions are committed by the CPU. Extremely memory-bound application behaviour is covered by the *mcf*, *omnetpp*, *libquantum*, and *GemsFDTD* benchmarks. For example, the mcf benchmark has forty-eight last level cache misses per kilo instructions and an IPC of 0.10. From the selected benchmarks *hmmer*, *gobmk*, *h264ref*, *tonto*, and *soplex* represents compute-bound applications. As an example *hmmer* and *gobmk* has IPCs of 2.7, and 1.2 respectively. Although their IPCs differ, these two benchmarks have similar L1 data cache (dcache) access rates at about approximately 0.04 misses per cycle for both of them. When instruction cache (icache) miss rates are examined, *hmmer* exhibits a much lower rate at 0.000695 misses per cycle, whereas *gobmk* has an instruction cache miss rate of 0.019.

Benchmark	Programming language	Application domain	Description
astar	C++	AI	Path-finding algorithms for graph and map data structures
bwavess	Fortran 77	Computational Fluid Dynamics	Navier-stokes equation solver
bzip2	C	Compression	File compression using the Burrows-Wheeler algorithm
calculix	Fortran, C	Structural mechanics	Solves partial differential equations using the finite element method
dealII	C++	Partial differential equations	Solves partial differential equations using the adaptive finite element method
GemsFDTD	Fortran	Computational Electromagnetics	Solver for Maxwell's equations
gobmk	C	AI	Plays and analyses games of Go
h264ref	C	Compression	Video compression algorithm that implements the H.264/AVC standard
hmmer	C	Search	Gene sequence search in a database
lbm	C	Computational fluid dynamics	Implementation of the Lattice Boltzmann Method for incompressible fluid simulation
leslie3d	Fortran	Computational fluid dynamics	Solver for a computational fluid dynamics problem using Linear-Eddy Model
libquantum	C	Quantum computing	Quantum computer simulation library

mcf	C	Combinatorial optimisation	Solver for a single-depot vehicle scheduling in public transportation
milc	C	Quantum chronodynamics	Simulation of 4D lattice gauge theory
namd	C++	Molecular dynamics	Simulates large biomolecular systems
omnetpp	C++	Discrete event simulation	Simulates a large Ethernet network
povray	C++	Computer graphics	Implements a ray tracer
sjeng	C	AI	Chess playing algorithm
soplex	C++	Linear programming	Solves a linear program reduced to triangular equations using LU-factorisation
tonto	Fortran	Quantum chemistry	Determines the atomic structure of crystals
xalancbmk	C++	Data representation	Transforms an XML file with an XSLT description into HTML
zeusmp	Fortran	Physics	Simulation of astrophysical phenomena

Table 5.2: Description of SPEC CPU2006 benchmarks

The nominal frequency of the CPU is set to 2 GHz. Each VFS node is compared to a CPG scheme with an off-period that gives an equivalent scaling in frequency. For example an off-period of 50% is taken as equivalent to clock frequency of 1GHz. CPG periods that have been executed start from 100 nanoseconds up to 200 nanoseconds with increments of 25 nanoseconds. The CPG period that provides the best result in its related metric is shown in the charts presented in this chapter in order to maintain clarity. Each bar in the x-axis represents a benchmark execution, with the benchmark name, CPG period, and the off cycle percentage. For example, *hmmer-150_37* represents the execution of the *hmmer* benchmark with a CPG period of 150 nanoseconds, and an off period of 37%. Results are divided into four buckets, with each bar in a

group representing the value of a benchmark normalised to the equivalent VFS execution. The normalised execution time for the VFS execution in each group is 1. The benchmarks under the 1.26GHz group are all benchmarks executed with an off period of 37% as an example.

Figure 5.19 shows the performance results of voltage and frequency pairs compared against their equivalent CPG configuration. A value equal and greater than one indicates that the CPG scheme has worse performance than the VFS equivalent. Overall, the CPG scheme has approximate 15% better performance than VFS.

It can be observed that VFS outperforms CPG for *gobmk*, and *sjeng* on the lowest frequency, 1GHz (50%). For *gobmk*, VFS have a relative higher dcache utilisation, which means that instructions are fetched into the pipeline more efficiently. Another difference is in dcache utilisation, where VFS has a higher hit rate compared to CPG. The greatest indicator for the difference in performance times is observed in the Out-of-Order pipeline structures. The number of times reorder buffer, issue queue and load queues had their full signals asserted are approximately 15% higher in the VFS execution, meaning that the VFS run for this benchmark exploits out-of-order more efficiently compared to the CPG scheme at this frequency.

The other benchmark in the 1GHz bucket where VFS outperforms CPG is *sjeng*. The reason for CPG performing worse than VFS for this benchmark is related to the out-of-order core model that gem5 simulates. In this model, when a load or store instruction fails to issue memory requests due to miss status handling registers (MSHRs) not being available, these memory instructions are squashed and replayed. This is seen in the 25% higher number of squashed loads in the CPG execution. In the case of *sjeng*, the number of dcache caused squashes is higher for the VFS execution by 27%, but the difference in the number of cycles dcache is blocked is higher in the CPG scheme by 42%.

Since *mcf* is a highly memory-bound benchmark the difference in performance lies in the data path. The contribution of out-of-order execution to performance in *mcf* is negligible, as indicated by the low IPC of this benchmark. This can be observed by the differences in load store queue (LSQ) full events provided by the simulator, where VFS has a 11% higher LSQ utilisation. Another point that effects the performance is in the out-of-order core model that gem5 simulates, as explained for the case of *sjeng*. This is seen in the 25% higher number of squashed loads in the CPG execution. Although there are more squashed loads in the CPG execution, the miss rates for data caches are lower. This means that the data is brought onto the core more quickly in the

CPG scheme than in VFS. On a highly memory-bound application the performance bottleneck is caused by the memory latency, and CPG is observed to induce lower memory latency in this case.

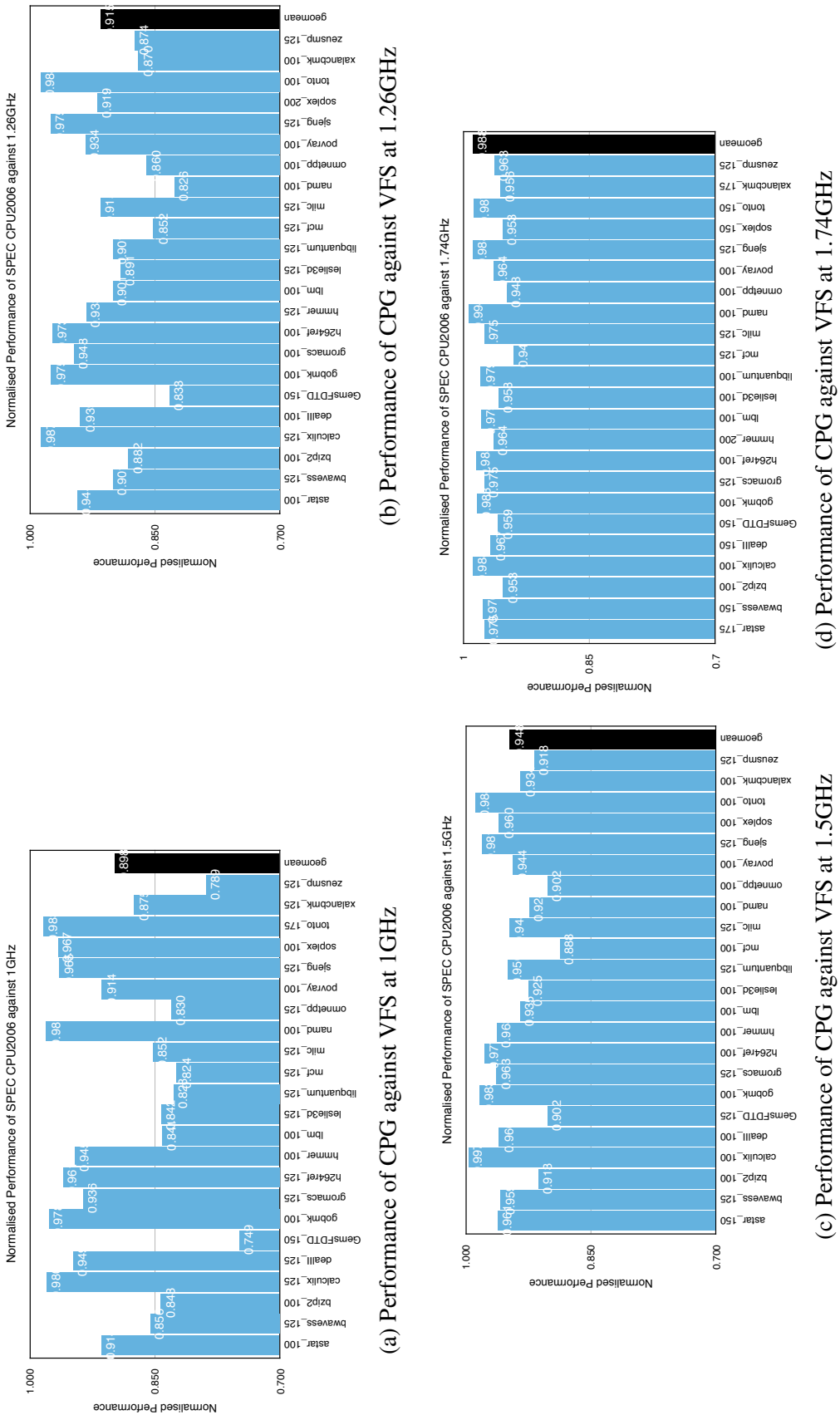
CPG has slightly better performance for the *bzip2* benchmark due to better speculative execution. This is observed in the higher branch prediction rate for CPG execution. The *milc* benchmark has higher irregularities in control flow compared to *bzip2*. This is one of the reasons CPG scheme performs better than VFS, with 2% higher correct branch prediction rate. Although, 2% is not a big difference, this translates as twice as much squashed icache misses.

The analysis provided for the 1GHz bucket applies to the rest of the buckets entirely. Still, a decreasing trend across buckets in terms of the normalised performance of CPG across all benchmarks is observed. This is due to the core executing on nominal frequency without interruptions, whereas VFS is still bound by a smaller frequency. Overall, CPG scheme can perform better than VFS by 6.2%.

The normalised energy consumed for the CPG scheme against VFS is given in Figure 5.20. Each of the bar charts on the x-axis represent the CPG enabled execution of a benchmark with the x-axis label for each plot providing the benchmark name, CPG period, and the off cycle percentage for that specific execution. The label *mcf-100_13%* indicates that the *mcf* benchmark for that sample has been executed with a CPG period of 100 nanoseconds, and an off period of 13% meaning that the core is asleep for 13 nanoseconds after doing 87 nanoseconds of work. There are four buckets; 1GHz, 1.26GHz, 1.5GHz, and 1.74GHz with each bucket representing a voltage and frequency pair for a VFS configuration.

It can be seen from the figure that CPG is almost equally energy-efficient than VFS for all the benchmarks in all the voltage and frequency levels. A greater reduction in energy can be seen in the lower frequencies, 5.1% at 1GHz, especially in the memory-bound benchmarks where the static power contributes more to the total power dissipation of the core as most of the work is being done by the memory system. This is mainly due to the better leakage reduction that CPG provides.

The energy, and performance results as they were explained in the previous chapter showed that CPG can outperform VFS in both of those areas. Energy Delay Product (EDP), is a metric that is used to provide a balance between energy and performance. The EDP results on how CPG compares with VFS are given in Figure 5.21. Since CPG outperforms VFS in energy and performance in lower voltage and frequency levels, the resulting EDP for the CPG scheme is 6.27% better compared to VFS.



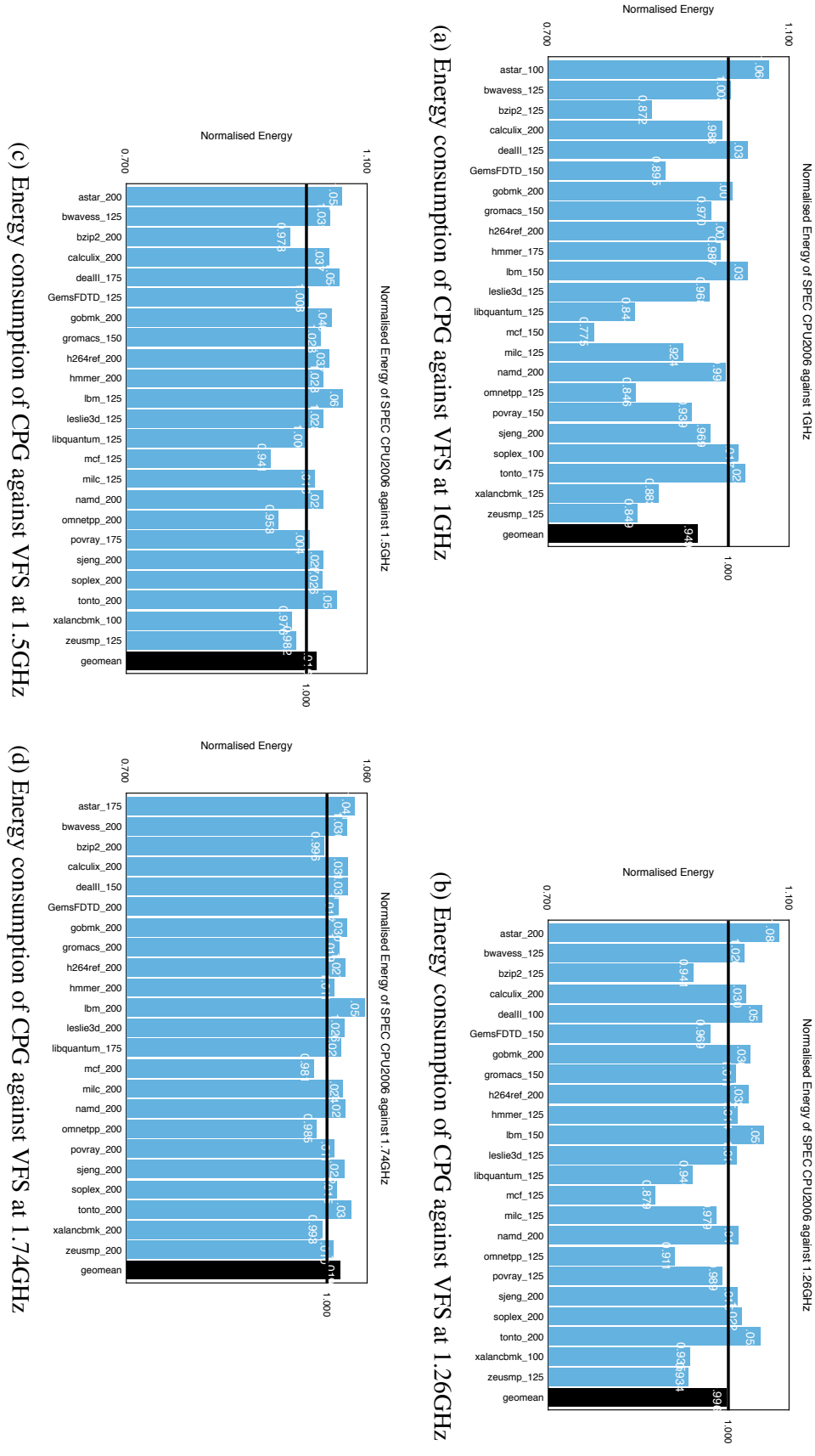
(a) Performance of CPG against VFS at 1GHz

(b) Performance of CPG against VFS at 1.26GHz

(c) Performance of CPG against VFS at 1.5GHz

(d) Performance of CPG against VFS at 1.74GHz

Figure 5.19: Performance of CPG against 4 VFS levels (Lower is better)



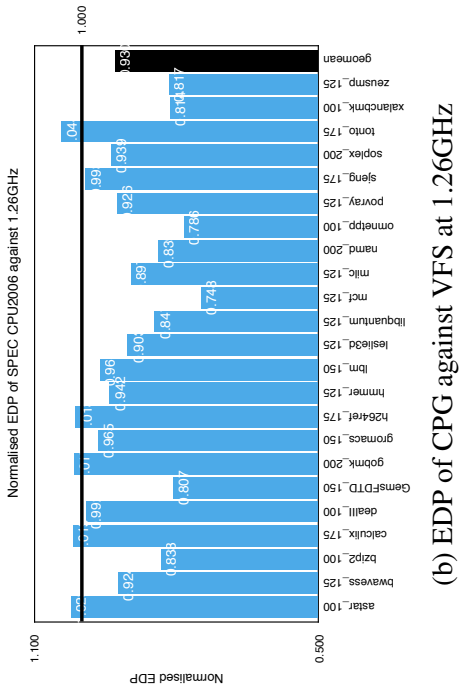
(a) Energy consumption of CPG against VFS at 1GHz

(b) Energy consumption of CPG against VFS at 1.26GHz

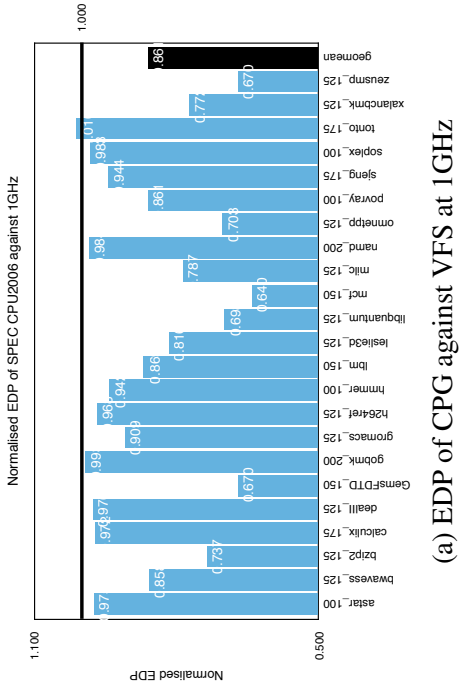
(c) Energy consumption of CPG against VFS at 1.5GHz

(d) Energy consumption of CPG against VFS at 1.74GHz

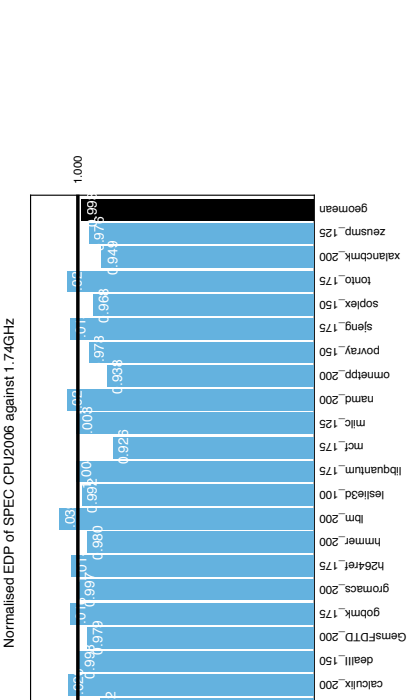
Figure 5.20: Energy consumption of CPG against 4 VFS levels (Lower is better)



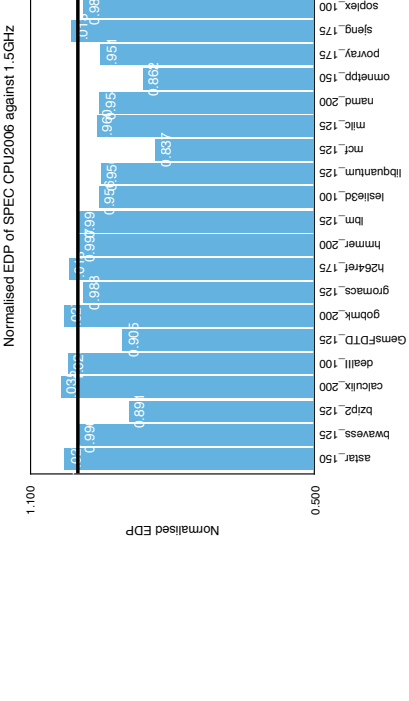
(b) EDP of CPG against VFS at 1.26GHz



(a) EDP of CPG against VFS at 1GHz



(d) EDP of CPG against VFS at 1.74GHz



(c) EDP of CPG against VFS at 1.5GHz

Figure 5.21: EDP of CPG against 4 VFS levels (Lower is better)

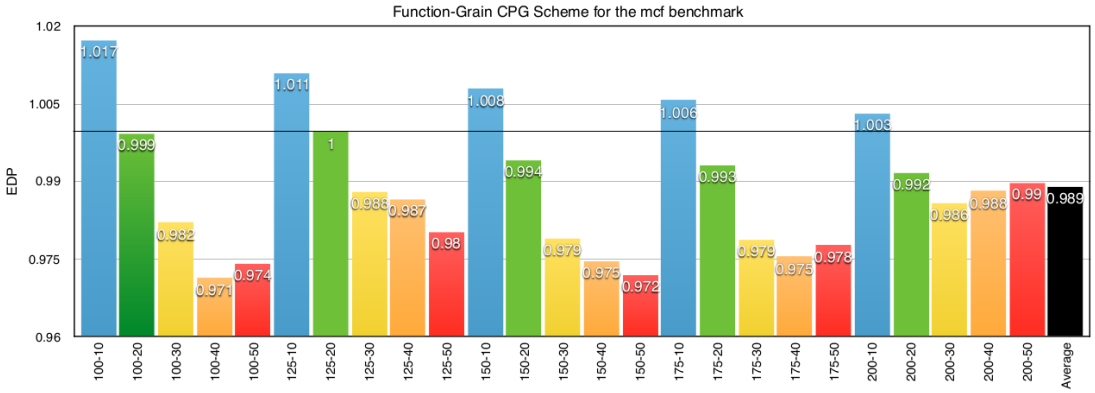


Figure 5.22: EDP of the Function-Grain CPG Scheme for the *mcf* Benchmark (Lower is better).

5.7 CPG at program function granularity

In previous sections, application of CPG was shown for microbenchmarks that exhibit homogeneous phase behaviour, and the SPEC CPU2006 benchmarks that exhibit variable phase behaviour. Actual applications are composed of both memory-bound and compute-bound phases. It was shown that applying CPG during compute-bound phases will be detrimental in terms of EDP. In this section a methodology to apply CPG during the execution of memory-bound functions within the *mcf* benchmark will be presented. For this purpose, an instruction that interfaces the CPG Controller with the software was implemented in the gem5 simulator. Then, the per-function cache miss rates for the *mcf* benchmark was obtained by executing it using *Cachegrind*. Cachegrind is a software tool from the Valgrind Dynamic Binary Instrumentation Toolkit [NS07]. The functions that had a Last Level Cache Miss Per Kilo Instruction (LLCMPK) of over 6 were obtained using this analysis. All the function entries were modified to include the CPG instruction in order to enable CPG at function entry. The CPG instruction was inserted before these functions in order to disable throttling at function exit. The modified version of the *mcf* benchmark was executed on the simulation infrastructure. The EDP results can be seen in Figure 5.22.

Each bar in the x-axis represents an execution of the *mcf* benchmark, with the CPG period, and the off cycle percentage. For example, 125_20 represents the execution of the *mcf* benchmark with a CPG period of 125 nanoseconds, and the ratio of the off period to the CPG period is 20% (i.e. on rate of 80%, or 25 nanoseconds off period).

The provided results are normalised to nominal execution at 2GHz, and it can be

seen that all CPG configurations except the ones with 10% off have better EDP compared to nominal execution. The main reason for this can be explained by these periods being lesser than the break-even time. A 1.1% increase in EDP on average is observed, with the maximum gain in EDP being 2.9%.

5.8 Conclusion

This chapter has presented CPG as a novel power management approach using a state-retentive power gating methodology. The evaluation for a system modelled using 14 nanometer technology parameters showed that CPG can be a strong alternative to VFS in future technologies, where the scaling of frequency and voltage will be limited due to the lower supply voltage future technologies are required to operate on. It has also been shown that applying CPG at a lower granularity can provide better EDP compared to nominal operation. Chapter 6 will provide the concluding remarks, and show future directions that the work presented in this thesis can be steered towards.

Chapter 6

Conclusions and Future Work

Aggressive technology scaling is driving the requirements for novel power management methodologies in order to overcome power density induced problems such as heat dissipation and transistor ageing. Reducing static power dissipation in a controllable manner with minimal degrading in performance is a key criteria for tackling thermal issues and achieving energy-efficient systems. It is possible to maximise the utilisation of power management schemes when the elements within the compute stack can relay the information about their runtime state across the stack. Another aspect that should be considered is the granularity that a power management scheme can work at. High overhead schemes tend to cause performance degradation especially in compute-bound program phases, and this leads to energy inefficiency. This dissertation presented an alternative way for power management, and has shown that it is possible to provide better Energy Delay Metrics compared to nominal operation when the operating granularity was lowered to program function level.

6.1 Summary

Chapter 1 introduced the problems that were born out of aggressive transistor scaling. The challenges in computer architecture simulation about execution of benchmarks on simulators were briefly presented. Furthermore, a brief overview of microprocessor power management and reasons for their limitations were provided.

Chapter 2 provided background material to aid in the comprehension of the following chapters. The operation of an out-of-order microprocessor was explained. Power dissipation and common power management schemes were discussed covering the subject from the transistor level up to the application level. Furthermore, an overview

of the developments within power management, hardware-software interaction, and workload generation was provided.

Chapter 3 described the state-of-the-art in hardware simulation, and explained the operation of the experimental infrastructure used for this thesis. Details of the cycle-accurate simulator used and how it was interfaced with power and temperature modeling tools have been provided. Furthermore, power and performance measurement methodologies on real hardware were provided.

Chapter 4 introduced a portable benchmark generator that generates code for a virtual ISA. Three use cases were presented to demonstrate domains where the tool could be used in. The benchmarks generated used for this tool were used for the evaluation of the Cyclical Power Gating (CPG) scheme that was explained in Chapter 5.

CPG has been proposed as a power management scheme that is fit to tackle the power challenges brought on by aggressive technology scaling. The evaluation of CPG against a DVFS configuration on a 14nm out-of-order microprocessor showed that CPG outperforms DVFS by 6.27% on EDP.

6.2 Future work

6.2.1 CPG Off-period Selection

A key part of the Cyclic Power-Gating scheme is the selection of T_{cpg}, the period over which the CPG recurs and (most critically) T_{off}, the power-gated period. T_{off} is limited by the *break-even time* of state-retentive architecture, below which the cost of turning the CPU on and off outweigh any benefits from power-gating. Above the break-even time, the overheads of the power-gating diminish and the energy-saving increases with sleep-time. However, in order for the CPG scheme to be energy-efficient the time spent power-gated in the pipelined execution phase must be minimized. Non-pipelined regions last as long as the memory-latency of the critical path loads. If T_{off} is too long, any outstanding memory accesses will be completed and the power-gating will needlessly increase execution time. If T_{off} is too short, the CPU will be woken up too early and the CPU will still be stalled (or about to be stalled) and energy will be wasted due additional active cycles spent in non-pipelined execution. The optimal time for the off-period in each *cycle* of a CPG scheme may be different and ideally a custom T_{off} that best suits the CPU execution patterns over a period would be selected. This could be further examined by investigating online phase detection methods, and

combining them with an online CPG governor equipped with the ability to take such decisions over each period.

6.2.2 Exploiting memory stalls for determining CPG sleep periods

In Chapter 5, an evaluation of the CPG scheme over SPEC benchmarks was made followed by a function granularity evaluation for a memory-bound benchmark. It has been shown that it is possible to outperform nominal execution when CPG has been applied at only a fraction of memory-bound functions, measured in terms of the number of last level cache misses per kilo instructions, while the rest of the functions were executing nominally. Appendix A presents the groundwork for detecting possibly memory-bound functions using a supervised machine learning algorithm that is executed at compile-time. Entry and exit into these functions can be hinted by the compiler to the microprocessor through a CPG-enabling instruction, and the processor can apply CPG only during those times. When this is combined with a leading load based memory stall prediction technique as described in Chapter 2, CPG duty cycles can be decided by the microprocessor leading to further reductions in EDP.

6.2.3 CPG enabled compute stack

A computing system requires a decision mechanism in order to switch between a performance boosting power-efficient mode and a cooling mode designed to sustain the expected lifetime of the system. This can be enabled by proposing a computing stack that contains the appropriate elements in each of the layers ranging from circuit-level to the application layer. To increase the utilisation of CPG, a **computing stack that is equipped with appropriate mechanisms on each layer to have a power-efficient high performance computing system with controllable immunity to temperature related faults** can be investigated.

A holistic approach to tackling this problem requires schemes that are able to take action based on knowledge of application character (memory/computation requirements) combined with hardware state (localised temperature estimations). Figure 6.1 shows the proposed CPG-aware stack. As is the case with layered architectures each layer can use services provided by the layer underneath it. A brief description for each layer is provided below:

- **Application Layer**: This layer can contain any Java application that is compiled

into bytecode. These bytecodes are executed by the underlying managed runtime environment (JVM).

- **Managed Runtime Environment**: This layer is a simple stack machine, which is used as a virtual execution environment to abstract an application from the underlying operating system and hardware. The primary responsibility of this layer is generating machine specific code from bytecode. MREs are widely used in datacenters and mobile systems, with the most popular examples being Java Virtual Machine (JVM), and Android Runtime. The main features of an MRE are its automatic handling of memory management, and ability to dynamically compile code. Dynamic compilation has been shown to provide significant advantages in an adaptive runtime environment where the system-wide concern can switch between performance and energy-efficiency during different periods. Temperature control can also be achieved within this layer.

- **Operating System**: The operating system is used to manage the underlying hardware resources (CPUs, memory, I/O devices,...) and provide the higher layers with interfaces to access these resources.

- **Processor**: The blocks of interest in this layer are Dynamic Voltage Frequency Scaling (DVFS) controller, CPG controller and performance counters. Performance counters are architectural registers that track the occurrence of certain events such as last level cache misses, instruction cache misses or L2 cache read requests. They are implemented by processor vendors to provide the users of the platform the opportunity to inspect their software.

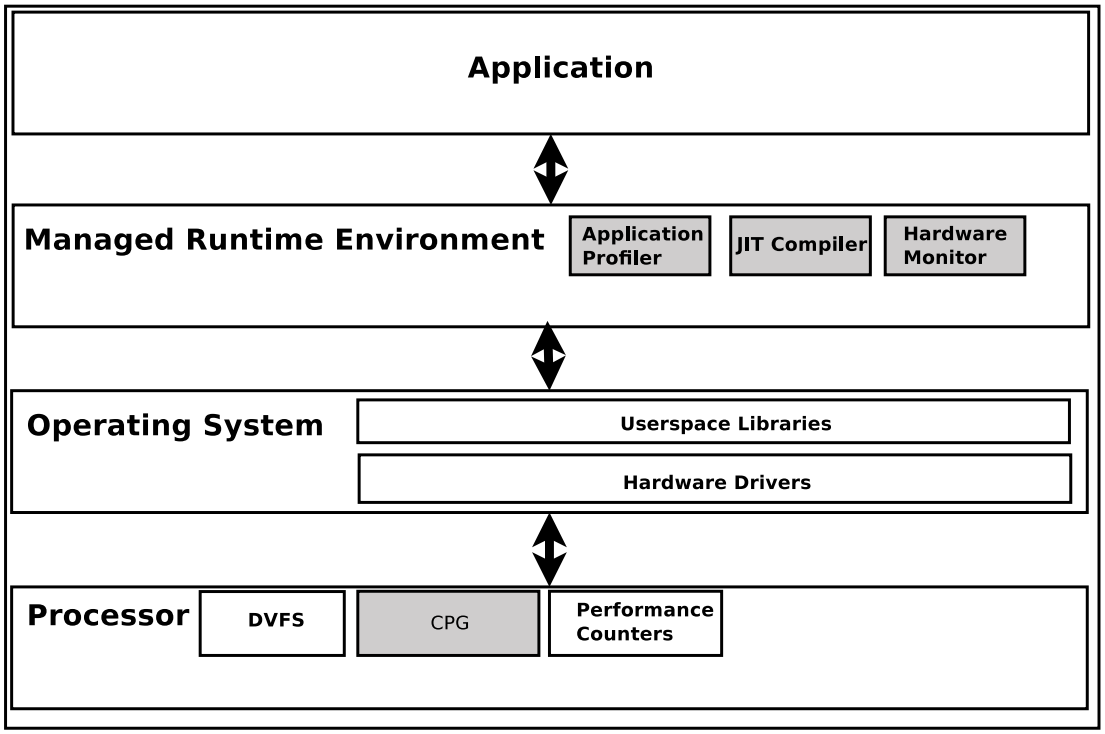


Figure 6.1: CPG Enabled Compute Stack

Bibliography

[AA14] Susanne Albers and Antonios Antoniadis. Race to Idle: New Algorithms for Speed Scaling with a Sleep State. *ACM Transactions on Algorithms*, 10(2):9:1–9:31, 2014.

[Ala03] Muhammad A. Alam. A Critical Examination of the Mechanics of Dynamic NBTI for PMOSFETs. In *Electron Devices Meeting, 2003. IEDM'03 Technical Digest. IEEE International*, pages 14–4. IEEE, 2003.

[AMP+15] Manish Arora, Srilatha Manne, Indrani Paul, Nuwan Jayasena, and Dean M Tullsen. Understanding Idle Behavior and Power Pating Mechanisms in the Context of Modern Benchmarks on CPU-GPU Integrated System. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 366–377. IEEE, 2015.

[ARM] ARM. A57 mpcore processor technical reference manual.

[BBB+11] 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 Computer Architecture News*, 39(2):1–7, 2011.

[BBG+12] Ramon Bertran, Alper Buyuktosunoglu, Meeta S. Gupta, Marc Gonzalez, and Pradip Bose. Systematic Energy Characterization of CMP/SMT Processor Systems via Automated Micro-Benchmarks. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–211. IEEE Computer Society, 2012.

[BDM09] Geoffrey Blake, Ronald G Dreslinski, and Trevor Mudge. A Survey of Multicore Processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009.

[BGDG+10] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. Energy-Efficient Cloud Computing. *The Computer journal*, 53(7):1045–1051, 2010.

[BHHR12] Steven Battle, Andrew D. Hilton, Mark Hempstead, and Amir Roth. Flexible Register Management Using Reference Counting. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2012.

[BMMF02] David Blaauw, Steve Martin, TREVOR MUDGE, and Krisztián Flautner. Leakage current reduction in vlsi systems. *Journal of Circuits, Systems, and Computers*, 11(06):621–635, 2002.

[Bor05] Shekhar Borkar. Designing Reliable Systems From Unreliable Components: The Challenges of Transistor Variability and Degradation. *MICRO, IEEE*, 25(6):10–16, 2005.

[CAP+15] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. CERE: LLVM-Based Codelet Extractor and Replayer for Piecewise Benchmarking and Optimization. *ACM Transactaions on Architecture and Code Optimimization*, 12(1):6:1–6:24, 2015.

[CCF+07] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Henrdrik Hamann, Alan Weger, and Pradip Bose. Thermal-Aware Task Scheduling at the System Software Level. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, pages 213–218. ACM, 2007.

[CHCR11] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & cap: Adaptive DVFS and Thread Packing Under Power Caps. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–185. ACM, 2011.

[CHE11] 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.

[CMDAN06] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online Power-Performance Adaptation of Multithreaded Programs Using Hardware Event-Based Prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 157–166. ACM, 2006.

[CMHM10] Eric S Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236. IEEE Computer Society, 2010.

[CMSB+08] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction Models for Multi-Dimensional Power-Performance Optimization on Many Cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 250–259. ACM, 2008.

[CP14] Victoria Caparros Cabezas and Markus Pschel. Extending the Roofline Model: Bottleneck Analysis with Microarchitectural Constraints. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 222–231. IEEE, 2014.

[CRG+11] Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. Increasing the Effectiveness of Directory Caches By Deactivating Coherence for Private Memory Blocks. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 93–104. ACM, 2011.

[CSK+07] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat.

FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture*, pages 249–261. IEEE Computer Society, 2007.

[CWL11] Ming Chen, Xiaorui Wang, and Xue Li. Coordinating Processor and Main Memory for Efficient Server Power Control. In *Proceedings of the International Conference on Supercomputing*, pages 130–140. ACM, 2011.

[DCK07] Robert H Dennard, Jin Cai, and Arvind Kumar. A Perspective on Todays Scaling Challenges and Possible Future Directions. *Solid-State Electronics*, 51(4):518–525, 2007.

[DGR+74] 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.

[DHJS10] Martin Dixon, Per Hammarlund, Stephan Jourdan, and Ronak Singhal. The Next-Generation Intel Core Microarchitecture. *Intel Technology Journal*, 14(3), 2010.

[DKM+12] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: Recording Microprocessor History. *Communications of the ACM*, 55(4):55–63, April 2012.

[DMB+12] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. Coscale: Coordinating CPU and Memory System DVFS in Server Systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 143–154. IEEE Computer Society, 2012.

[Eas16] Ashraf Eassa. Intel Corp. Confirms First 10-Nanometer Product on Track for 2017 Introduction, 2016.

[EBA+11] 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.

[EE10] Stijn Eyerman and Lieven Eeckhout. A Counter Architecture for Online DVFS Profitability Estimation. *IEEE Transactions on Computers*, 59(11):1576–1583, 2010.

[Eec08] Lieven Eeckhout. Sampled processor simulation: A survey. *Advances in Computers*, 72:173–224, 2008.

[EEKS09] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):3, 2009.

[ERB+95] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, et al. Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-Issue CMOS RISC Microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.

[FASB04] Clemens J Först, Christopher R Ashman, Karlheinz Schwarz, and Peter E Blöchl. The interface between silicon and a high-k oxide. *Nature*, 427(6969):53–56, 2004.

[FDN+01] David J. Frank, Robert H. Dennard, Edward Nowak, Paul M. Solomon, Yuan Taur, and Hon Sum Philip Wong. Device Scaling Limits of Si MOSFETs and Their Application Dependencies. *Proceedings of the IEEE*, 89(3):259 –288, 2001.

[GEE10] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *16th IEEE International symposium on High-Performance Computer Architecture (HPCA-16)*, pages 307–318. IEEE Computer Society, 2010.

[GPD+14] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. Sources of Error in Full-System Simulation. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 13–22. IEEE, 2014.

[Gre11] Peter Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White Paper*, pages 1–8, 2011.

[Han12] Arria V Device Handbook. Features of the cortex-a9 mpu subsystem. 2012.

[HBS+04] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural Techniques for Power Gating of Execution Units. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 32–37. ACM, 2004.

[HCE12] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. Sniper: Scalable and Accurate Parallel Multi-Core Simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*, pages 91–94. High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC), 2012.

[HDH+10] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *2010 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 108–109. IEEE, 2010.

[Hen06a] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[Hen06b] John L. Henning. Spec CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[HGV+06] Wei Huang, Shougata Ghosh, Sivakumar Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R Stan. HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, 2006.

[HM07] Sebastian Herbert and Diana Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 38–43. IEEE, 2007.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.

[HPLC05] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and More Flexible Program Analysis. In *Journal of Instruction Level Parallelism*, 2005.

[HW04] Raymond Heald and Ping Wang. Variability in Sub-100nm SRAM Designs. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, pages 347–352. IEEE Computer Society, 2004.

[ISK+09] Daisuke Ikebuchi, Naomi Seki, Yu Kojima, M Kamata, Lei Zhao, Hideharu Amano, Toshiaki Shirai, Satoshi Koyama, Tatsunori Hashida, Y Umahashi, et al. Geyser-1: A MIPS R3000 CPU Core with Fine Grain Runtime Power Gating. In *Solid-State Circuits Conference, 2009. A-SSCC 2009. IEEE Asian*, pages 281–284. IEEE, 2009.

[JEJI08] Ajay M. Joshi, Lieven Eeckhout, Lizy K. John, and Ciji Isen. Automated Microprocessor Stressmark Generation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 229–239. IEEE, 2008.

[JKK+12] Kwangok Jeong, Andrew B. Kahng, Seokhyeong Kang, Tajana S. Rosing, and Richard Strong. MAPG: Memory Access Power Gating. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1054–1059. EDA Consortium, 2012.

[JNaS+12] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. CRUISE: Cache Replacement and Utility-aware Scheduling. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 249–260. ACM, 2012.

[Jos07] Ajay Manohar Joshi. *Constructing Adaptable and Scalable Synthetic Benchmarks for Microprocessor Performance Evaluation*. ProQuest, 2007.

[JRKA15] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. Gpu register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 420–432. ACM, 2015.

[Kea07] *Power Gating Overview*, pages 33–40. Springer US, Boston, MA, 2007.

[KGWB08] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 123–134. IEEE, 2008.

[KLJR14] K. Kim, C. Lee, J. H. Jung, and W. W. Ro. Workload Synthesis: Generating Benchmark Workloads From Statistical Execution Profile. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 120–129. IEEE, 2014.

[KM08] Stefanos Kaxiras and Margaret Martonosi. Computer Architecture Techniques for Power-Efficiency. *Synthesis Lectures on Computer Architecture*, 3(1):1–207, 2008.

[KPFG14] Manolis Kaliorakis, Mihalis Psarakis, Nikos Foutris, and Dimitris Gizopoulos. Accelerated Online Error Detection in Many-Core Microprocessor Architectures. In *2014 IEEE 32nd VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2014.

[KSK10] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. Interval-based Models for Run-time DVFS Orchestration in Superscalar Processors. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 287–296, New York, NY, USA, 2010. ACM.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[LAS+09] 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 *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480. ACM, 2009.

[LFOK11] Joseph C. Libby, Ashley Furrow, Paddy O'Brien, and Kenneth B. Kent. A Framework for Verifying Functional Correctness in Odin ii. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–6. IEEE, 2011.

[LMC+11] Michael A. Laurenzano, Mitesh Meswani, Laura Carrington, Allan Snavely, Mustafa M. Tikir, and Stephen Poole. Reducing Energy Usage with Memory and Computation-Aware Dynamic Frequency Scaling. In *European Conference on Parallel Processing*, pages 79–90. Springer, 2011.

[LSH10] Etienne Le Sueur and Gernot Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, pages 1–8. USENIX Association, 2010.

[LWC+16] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, et al. An Agile Approach to Building RISC-V Microprocessors. *IEEE Micro*, 36(2):8–20, 2016.

[MBJ09] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. *HP Laboratories*, pages 22–31, 2009.

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[MEP12] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. Predicting Performance Impact of DVFS for Realistic Memory Systems. In

Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-45, pages 155–165, Washington, DC, USA, 2012. IEEE Computer Society.

[Moo65] Gordon E. Moore. Cramming More Components Onto Integrated Circuits, 1965.

[MSB+05] Milo MK 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. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[MSZ+11] Evelyn Mintarno, Joelle Skaf, Rui Zheng, Jyothi Bhaskar Velamala, Yu Cao, Stephen Boyd, Robert W. Dutton, and Subhashish Mitra. Self-Tuning for Maximized Lifetime Energy-Efficiency in the Presence of Circuit Aging. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(5):760–773, 2011.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Sigplan notices*, pages 89–100. ACM, 2007.

[PACG11] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*, pages 1050–1055. ACM, 2011.

[PSZ+07] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *ACM SIGOPS Operating Systems Review*, pages 289–302. ACM, 2007.

[PZW+07] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Kang G Shin, et al. Performance Evaluation of Virtualization Technologies for Server Consolidation. *HP Labs Tec. Report*, 2007.

[RL16] Haris Ribic and Yu David Liu. AEQUITAS: Coordinated Energy Management Across Parallel Applications. In *Proceedings of the 2016 International Conference on Supercomputing*, page 4. ACM, 2016.

[RLSdS11] Barry Rountree, David K. Lowenthal, Martin Schulz, and Bronis R. de Supinski. Practical performance prediction under Dynamic Voltage Frequency Scaling. In *2011 International Green Computing Conference and Workshops*, pages 1–8, July 2011.

[RRK11] Soumyaroop Roy, Nagarajan Ranganathan, and Srinivas Katkoori. State-Retentive Power Gating of Register Files in Multicore Processors Featuring Multithreaded In-Order Cores. *Computers, IEEE Transactions on*, 60(11):1547–1560, 2011.

[SABR04] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177–186. IEEE Computer Society, 2004.

[SGG+14] Bo Su, Joseph L. Greathouse, Junli Gu, Michael Boyer, Li Shen, and Zhiying Wang. Implementing a leading loads performance predictor on commodity processors. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 205–210, Berkeley, CA, USA, 2014.

[SGS+14] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L. Greathouse, and Zhiying Wang. Ppep: Online performance, power, and energy prediction framework and dvfs space exploration. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 445–457, 2014.

[Sin08] Ronak Singhal. Inside intel next generation nehalem microarchitecture. In *Hot Chips*, volume 20, 2008.

[SK13] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 475–486. ACM, 2013.

[SKK11] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. Green Governors: A Framework for Continuously Adaptive DVFS. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8. IEEE, 2011.

[SKZ08] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy Aware Consolidation for Cloud Computing. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, pages 1–5. USENIX Association, 2008.

[SL13] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, 2013.

[SMC+13] Filippo Sironi, Martina Maggio, Ricardo Cattaneo, Giovanni F. Del Nero, Donatello Sciuto, and Marco D. Santambrogio. ThermOS: System Support for Dynamic Thermal Management of Chip Multi-Processors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 41–50. IEEE Press, 2013.

[Spr02] Brinkley Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, 2002.

[SWV+09] Alvin W Strong, Ernest Y Wu, Rolf-Peter Vollertsen, Jordi Sune, Giuseppe La Rosa, Timothy D Sullivan, and Stewart E Rauch III. *Reliability Wearout Mechanisms in Advanced CMOS Technologies*, volume 12. John Wiley & Sons, 2009.

[Tay12] Michael B. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131 –1136. IEEE, 2012.

[Tou16] Chris Toumey. Less is Moore. *Nature nanotechnology*, 11(1):2–3, 2016.

[TP06] Scott E. Thompson and Srivatsan Parthasarathy. Moore's Law: The Future of Si Microelectronics. *Materials Today*, 9(6):20–25, 2006.

[TWA+10] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. RAMP gold: an FPGA-based architecture simulator for multiprocessors. In *Proceedings of the 47th Design Automation Conference*, pages 463–468. ACM, 2010.

[UKM02] Osman S. Unsal, C. Mani Krishna, and CA Mositz. Cool-Fetch: Compiler-Enabled Power-Aware Fetch Throttling. *Computer Architecture Letters*, 1(1):5–5, 2002.

[VBB13] Augusto Vega, Alper Buyuktosunoglu, and Pradip Bose. SMT-Centric Power-Aware Thread Placement in Chip Multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 167–176. IEEE, 2013.

[VSG+10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 205–218. ACM, 2010.

[WDH+16] Matthew J. Walker, Stephan Diestelhorst, Andreas Hansson, Anup K. Das, Sheng Yang, Bashir M. Al-Hashimi, and Geoff V. Merrett. Accurate and Stable Run-Time Power Modeling for Mobile and Embedded CPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.

[WHB05] Neil Weste, David Harris, and Ayan Banerjee. CMOS VLSI design. *A circuits and systems perspective*, 11:739, 2005.

[WJMH15] Olivier Weber, Emmanuel Josse, J Mazurier, and Michel Haond. Static and dynamic power management in 14nm FDSOI technology. In *IC Design & Technology (ICICDT), 2015 International Conference on*, pages 1–4. IEEE, 2015.

[WJY+07] Wei Wu, Lingling Jin, Jun Yang, Pu Liu, and Sheldon X-D Tan. Efficient Power Modeling and Software Thermal Sensing for Runtime Temperature Monitoring. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):25, 2007.

[WMC+05] Qiang Wu, Margaret Martonosi, Douglas W. Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A Dynamic Compilation Framework for Controlling Microprocessor Energy and

Performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 271–282. IEEE Computer Society, 2005.

[WMC+06] Qiang Wu, Margaret Martonosi, Douglas W. Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. Dynamic-Compiler-Driven Control for Microprocessor Energy and Performance. *IEEE Micro*, 26(1):119–129, 2006.

[WMW09] Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. In *ACM SIGARCH computer architecture news*, volume 37, pages 314–324. ACM, 2009.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, 2009.

[ZWM+14] Runjie Zhang, Ke Wang, Brett H. Meyer, Mircea R. Stan, and Kevin Skadron. Architecture Implications of Pads as a Scarce Resource. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 373–384. IEEE, 2014.

Appendix A

Compiler Driven Cyclic Power Gating

Saving energy while not degrading the performance of an application is an important goal in computer architectural research. Contemporary compute stacks are equipped with mechanisms that allow each layer in the stack to provide means of coordination for energy efficiency and performance improvements [RL16, SKZ08, BGDG+10, PSZ+07]. A cross-layer approach where different layers are equipped with the knowledge of the operation of the other layers enables coordinated decision-making [CHCR11, CMDAN06].

In Chapter 5, Cyclical Power Gating (CPG) was introduced as an alternative for Dynamic Voltage and Frequency Scaling (DVFS). CPG works by quickly power gating a core and waking it up as defined by the duty cycle and the CPG period. The duty cycle defines the ratio of on and off cycles with an example of 80% duty cycle meaning that the core is power gated for 20% of the defined CPG period. The analysis of CPG provided evidence that the CPG scheme has its benefits in terms of energy-efficiency on memory-bound benchmarks. This was mainly due to the power-gating of the core while the memory system was still active bringing in data closer to the core. A compile-time methodology to infer whether a program function is memory-bound or not is presented in this chapter.

A program phase is defined as a categorisable segment of a program with common behaviour. It is possible to categorise a program phase as being either compute-bound, when the processor resources are being utilised at a fairly high level, or memory-bound, when the data required for computation is being moved closer to the core for computation. From the perspective of the core, these two program phases require different measures when energy savings with the least degradataion in performanceis required. Compute-bound phases provide the highest performance at the expense of least energy

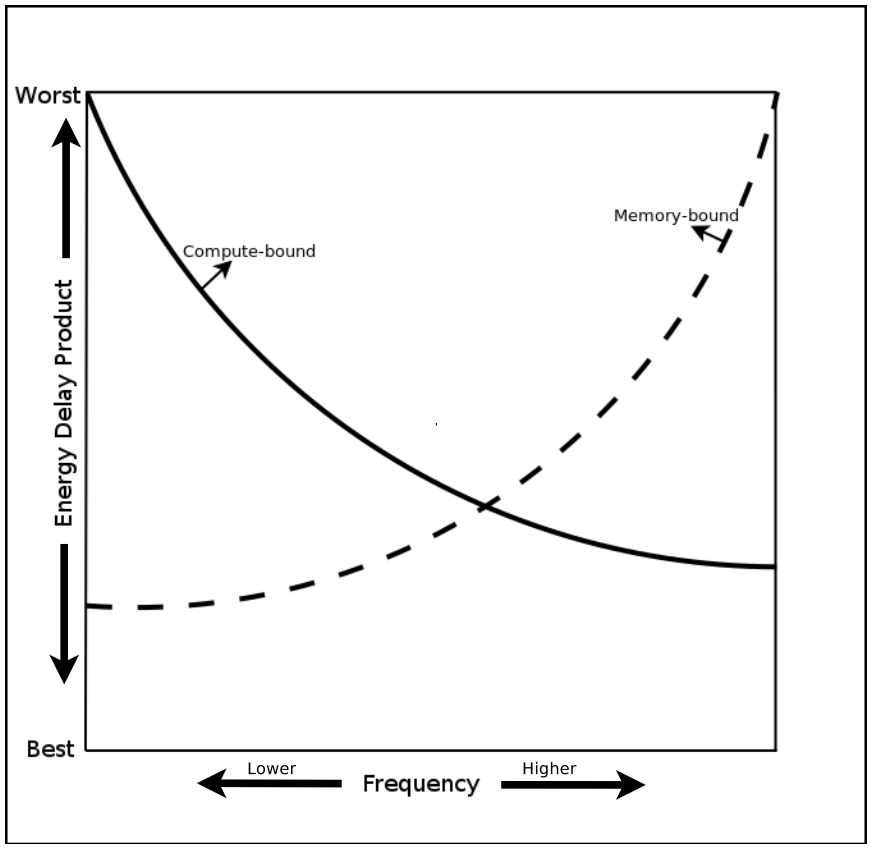


Figure A.1: Energy Delay Product for Memory and Compute Bound Applications

when they *race-to-idle*, meaning that executing without any interruptions at the highest frequency available. The energy efficiency of race-to-idle does not apply to memory-bound phases since most of the work required by the application is being done out of the core. It has been shown in previous work [SKK11], as well as in Chapter 5 and Chapter 4, that executing memory-bound phases at a relatively lower frequency do not have as big an impact on performance as lowering the frequency of compute-bound phases. The decrease in energy consumption outweighs the performance hit in memory-bound phases (Figure A.1). Since applications consist of a combination of compute-bound and memory-bound phases, applying CPG at an application level granularity would not be the most effective approach when energy-efficiency is targeted. In this section a compiler-driven method for lowering the granularity of CPG to function-level will be presented.

Firstly, an analysis of the memory behaviour of selected benchmarks from the SPEC CPU2006 suite will be presented. The candidate functions provided by this analysis will be tagged with an instruction that hints the processor on how to drive

CPG, and compared with a baseline where CPG is not applied. Secondly, a way of automatically doing this using a classifier implemented as a compiler analysis pass will be presented. The results of the compiler-driven approach to applying CPG will be presented, along with a concluding analysis.

A.1 Analysis of Memory Behaviour of CPU2006 benchmarks

Memory behaviour of benchmarks at a granularity of functions needs to be characterised in order to evaluate whether applying CPG on memory-bound functions can improve the metrics. Selected benchmarks from the CPU2006 suite have been evaluated by executing them using the Cachegrind tool on an Intel Xeon E5-2690 CPU. Results of this evaluation will be presented in this section along with a discussion.

Cachegrind is a tool within the Valgrind Dynamic Binary Instrumentation Toolkit [NS07]. It enables instrumentation of code by inserting a layer between the code that is being executed, and hardware making it possible to mimic the behaviour of different memory hierarchy configurations. An instruction cache, first-level data cache and a shared last-level cache can be simulated with parameterized associativity, cacheline size, and cache size. The tool also enables the simulation of branch prediction if desired.

Cachegrind output is composed of cache access behaviour of the instrumented benchmark, and it provides event counts for instructions executed (Ir), instruction cache read misses (I1mr), last-level cache read misses caused by instruction cache (ILmr), data cache reads (Dr), data cache read misses (D1mr), last-level cache read misses caused by data cache (DLmr), data cache writes (Dw), D1 cache write misses (D1mw), and last-level cache write misses caused by data cache (DLmw).

The results provided by Cachegrind are used to calculate data cache initiated last-level cache misses per kilo instructions (LLCMPK), as given in Equation A.1. ILmr is not included in the equation as the problem of picking the appropriate CPG duty-cycle is an open problem, and this might lead to possible performance problems since the fetching of instructions might be delayed.

$$LLCMPK = 1000 \times \frac{(DLmr + DLmw)}{Ir} \tag{A.1}$$

Table A.1 gives the results for the functions that have the highest three LLCMPK value

Table A.1: Function level cache missing behaviour of SPEC benchmarks

Benchmark	Function	LLCMPK	Benchmark	Function	LLCMPK
astar	getregfillnum	0.084	**bwaves**	bi_cgstab_block_	22.327
	isaddtobound	0.018		mat_times_vec_	21.608
	makebound2	0.0083		shell_	16.629
cactusADM	bench_staggeredleapfrog2_	3.114	**gamess**	vclr_	0.002
	N/A	N/A		tftri_	0.003
	N/A	N/A		ddot_	0.002
gcc	ggc_mark_rtx_children	45.104	**gobmk**	hashtable_clear	39.583
	ggc_mark_trees	14.129		do_get_read_result	4.356
	bitmap_element_allocate	3.565		hashtable_partially_clear	4.200
gromacs	mk_mshift	0.0009	**h264ref**	FastPelY_14	0.534
	new_i_nblist	0.0007		UMVPelY_14	0.234
	put_in_list	0.0004		UnifiedOneForthPix	0.086
hmmer	P7Viterbi	3.695	**leslie3d**	setbc_	54.350
	RandomSequence	2.786		update_	25.094
	sre_random	7.223		extrapi_	23.816
mcf	flow_cost	166.465	**milc**	su3mat_copy	76.565
	refresh_neighbour_lists	92.082		sub_four_su3_vecs	68.106
	suspend_impl	67.418		uncompress_anti_hermitian	60.344
namd	calc_self_energy	0.002	**omnetpp**	deliver	78.283
	calc_self_merge_fullelect	0.0005		findGate	62.588
	calc_self	0.0004		get	35.516
perlbench	Perl_sv_upgrade	1.256	**libquantum**	quantum_sigma_x	50.000
	Perl_sv_setsv_flags	1.050		quantum_cnot	33.101
	Perl_re_intuit_start	0.891		quantum_toffoli	27.249
sjeng	QProbeTT	48.515	**xalancbmk**	push	0.661
	ProbeTT	43.022		StylesheetExecutionContext	0.621
	checkECache	36.999		findXObject	0.472

within each analysed benchmark. The outcome of this analysis will provide the basis to enable further experimentation to test whether applying CPG at function granularity can provide improvements in metrics of concern when comparing against a race-to-idle configuration.

The results show that some of the examined benchmarks such as gamess, gromacs do not contain any functions that could be considered memory-bound as the function with the highest LLCMPK in gamess is vclr_ with an LLCMPK of 0.002, and mk_mshift in gromacs with an LLCMPK of 0.0009. Other benchmarks that do not contain functions high in LLCMPK are astar, namd, perlbench, gamess, h264ref, and xalancbmk. Benchmarks with the functions that contain functions that have a high LLCMPK are gcc, mcf, sjeng, bwaves, gobmk, leslie3d, milc, omnetpp, and libquantum.

Since there is no possibility of achieving energy-efficient execution against race-to-idle in workloads that only contain compute-bound functions, the focus will be put on workloads that have memory bound functions with a high LLCMPK.

To test the hypothesis that applying CPG on highly memory-bound functions will provide energy-efficient execution compared to race-to-idle and higher performance

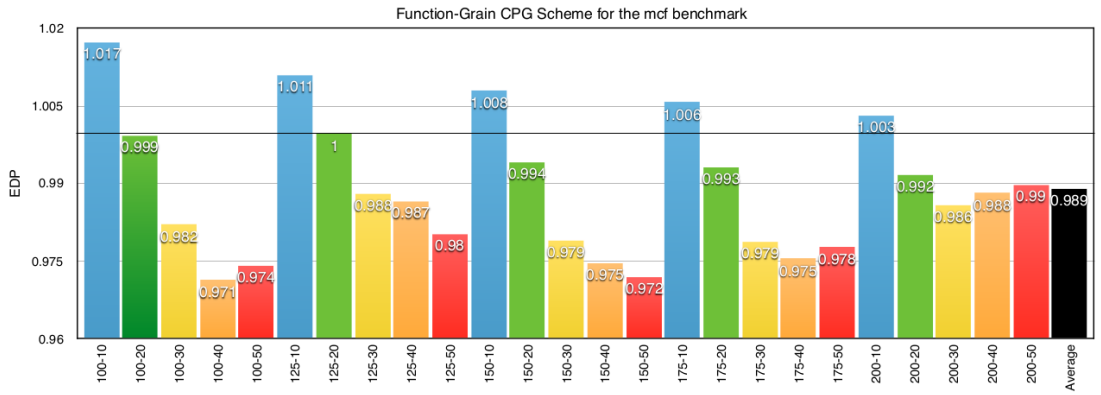


Figure A.2: EDP of the CPG scheme for the *mcf* benchmark.

compared to application granularity CPG, the functions in the mcf benchmark with high values of LLCMPK are picked as candidates. These selected functions have a mean LLCMPK of 56.26730, with the highest LLCMPK function being flow_cost (166.46561) and the lowest LLCMPK function being price_out_impl (6.02613). A pseudo-instruction that provides a software interface to the CPG controller has been implemented in the simulator. This instruction is manually inserted at the entry and exit points of these functions through the modification of the C source file using *asm()* function calls. The performance of the function-grained CPG scheme is 10% worse than race-to-idle, but the energy consumption is 11% less. This results in a 1.1% EDP increase as given in Figure A.2.

A.2 Memory Operations on LLVM IR

Low Level Virtual Machine Project (LLVM) is a highly modular collection of software enabling applications within the domains of compilers, virtual machines, and development tools. An intermediate representation (IR) is a layer of abstraction that eases the transformation of a high level source language into a target language that could be machine code, or another high level language. It represents the language for an abstract machine that has its limits bound by a high level programming language on one end and machine language on the other. On one hand, there is the machine-abstracted domain of high level programming languages, which provide means of clarity and portability, and on the other there is the machine-specific assembly that is the actual binary code which executes on digital logic. Although an IR is not necessary for compiling a high level language into binaries, it enables a more portable and modular compilation framework.

LLVM IR is a strongly-typed instruction set for an abstract machine with a theoretically infinite number of registers, where each instruction represents an operation that affects the control behaviour of the machine. It uses Static Single Assignment (SSA) when setting scalar values to virtual registers. SSA form means that a new register must be used each time a scalar value is assigned, meaning that a variable can only be assigned once.

An LLVM program is a collection of one or more modules, where each module contains functions that are composed of basic blocks. Basic blocks are composed of binary and memory operations. Control between other basic blocks defined in a function is established using terminator instructions, which are synonymous to branch instructions. All LLVM instructions with the exception of terminator instructions produce an output value.

A description of each memory operation will be provided below, before going further with the representation of memory accesses in LLVM IR.

- alloca: This instruction is used to allocate stack storage memory as specified in its source operand. The memory allocated is located at the stack of the function that it was called from.
- load: Loads data of size defined by its type, from the address provided as its source operand.
- store: Stores the value in its first operand into the address pointed to by its second operand.
- fence: This instruction is used to insert a fence, also known as a memory barrier, to enable memory ordering at the point it is executed. It is essentially a way of serialising pending memory operations.
- cmpxchg: Represents a compare-and-swap operation, where a value residing at an address in memory is loaded and compared to the value provided as a source operand. In the case that the two values are equal, the data in the memory is updated with a new value.
- atomicrmw: Provides a representation for an atomic operation to modify the memory.
- getelementptr: Being one of the most crucial instructions for memory operations in an LLVM machine, this instruction generates the address that the other

memory operations use to read from and write to. The address to be generated could be as simple as an offset, or more complicated like being the address for an element of a data structure that is indexed by an array.

```
%1 = alloca i32, i32 4
store i32 0xabababab, i32* %1
%2 = load i64* %1
%3 = add i32 %2, %2
```

Listing A.2: Stack Operations

Modern operating systems divide a process memory address space mainly into two partitions; heap and stack. Stack memory is used to store local variables that are local to a function. On the other hand, heap memory is used to allocate memory that is global. This means that memory allocated within the heap can also be accessed outside function scope. Each function is provided a stack frame as defined by the Application Binary Interface (ABI) of the architecture that the program is being executed on. LLVM IR represents the allocation of memory within the stack using the *alloca* instruction. This enables the definition of local variables, and pointers. Pointers are essentially local variables that contain the address of a memory location. Output register of the alloca instruction can later be written to and read from using *store*, and *load* instructions.

To illustrate how stack allocation is represented using LLVM IR, an example for the addition of a 32-bit stack allocated integer is given in Listing A.2. A register is assigned as the output of the alloca instruction (%1), followed by the store operation of a constant value into the register. Then, the stored value is loaded into a new register (%2). Finally, the value in this register is added with itself (%3).

An example for the addition of a heap allocated single precision floating point register is given in A.3. First step in the example calls malloc with size parameter of 512 bytes(%1). Then, a single precision floating point variable is allocated on the stack through the alloca function (%2). A bitcast is needed to cast the 8-bit integer pointer returned by malloc to a floating point pointer due to the enforcement of strong-typing in LLVM IR (%3). This is followed by a load that gets the pointer to the address that was allocated by malloc (%4). The address for the 16th element of the allocated memory block is loaded into the register %5, and the data in that address is subsequently loaded into register %6. Finally, the loaded value in register %6 is added with itself (%7).

```
%1 = call noalias i8* @malloc(i64 512)
%2 = alloca float*, align 8
%3 = bitcast i8* %1 to float*
store float* %3, float** %2, align 8
%4 = load float** %2, align 8
%5 = getelementptr inbounds float* %4, i64 16
%6 = load float* %5, align 4
%7 = add float %6, %6
```

Listing A.3: Heap Operations

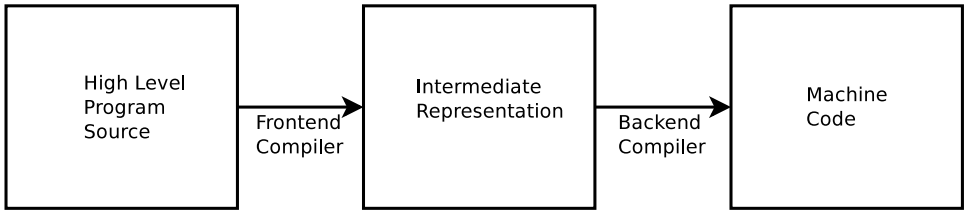
The examples in the previous two paragraphs show how the memory instructions are used in LLVM IR. In the next section, a methodology to provide a binary classification of functions from the SPEC CPU2006 benchmarks will be explained.

A.3 Compile-Time Classification of Memory-Bound Functions

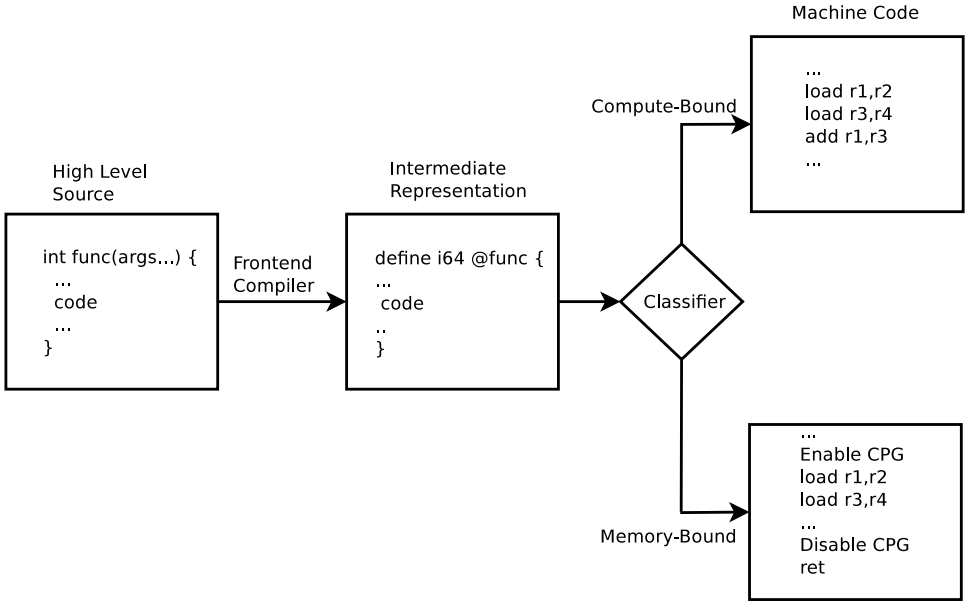
A method to classify functions through features that are obtained from program functions at the LLVM IR using a supervised learning algorithm will be explained in this section. *Supervised learning* is a machine learning technique that tries to classify an unclassified data point by comparing it with previously observed and classified data points.

The transformation of a program function that is written in a high-level source code into machine code in a standard compilation flow is given in Figure A.3a. The high-level source code, usually written by a developer, is input to a frontend compiler which generates an intermediate representation (IR) of the code for a virtual ISA. Then, the intermediate representation (IR) is passed through a number of analysis passes that are used to obtain information which will be used as input for transformation passes that optimise the intermediate representation. Finally, this optimised IR is input to the backend compiler which generates machine-specific code.

The compile-time classification technique proposed in this chapter modifies this standard flow by inserting a classifier that categorises a function as being memory-bound or compute-bound. The proposed classifier is implemented as a function-scope analysis pass. A function-scope analysis pass enables access to obtain all the IR level



(a) Standard Compilation Flow



(b) Compilation Flow with Function Classifier

Figure A.3: Compilation flow

statistics for a program function. Although it is not possible to be sure of the memory-boundedness of a function without having its input data, this methodology may achieve EDP gains by hinting at the CPG controller for a possibly memory-bound function.

A.3.1 k-NN Based Classification using LLVM

An LLVM function analysis pass to classify functions as memory-bound or compute-bound from their LLVM IR features was implemented. The pass uses the k-NN algorithm to classify functions. The k-NN algorithm is a machine learning technique that is used for classification and regression. The algorithm works by mapping a training set that consists of n-dimensional feature vectors onto n-dimensional space, and calculates the distance of each element in a data set to the elements in the training set. The data

Table A.2: A Subset of the Training Set Used for Function Classification

Funct.	**Term.**	**Binary Ops**	**Logical Ops**	**Memory Ops**	**Get Elem. Pointer**	**Alloca**	**Class**
push_pawn	0.22	0	0	0.6	0.09	0.04	comp
sre_random	0.10	0.18	0	0.609	0.04	0.06	mem
DV_push	0.15	0.05	0	0.57	0.17	0.05	comp

instance is assigned a class based on the closest k number of neighbors that belong to the same class.

A training data set was prepared based on the LLCMPK results obtained from the execution of the benchmarks using cachegrind as shown in Table A.1. LLVM IR statistics for these functions were gathered by using a script that analyses the emitted IR code for SPEC CPU2006 benchmarks.Statistics obtained from the function analysis pass are terminator, binary, logical, and memory operators. Terminator operators are the total number of Branch instructions including direct and indirect branches. Logical operators are the total number of bit shifting that have been used. Binary operators are the total number of arithmetic operators for integer and floating point operations. Memory operators consist of load and store instructions. The count of getelementptr instructions is used as a seperate feature, since it is involved in pointer arithmetic and using it as a seperate feature instead of adding it to the memory counts would increase its weight. All the features were preprocessed so that they are represented as a fraction of the total instructions within that function. A subset of the training set with their LLVM statistics can be seen in Table A.2.

To test the implemented classifier *soplex* benchmark from the SPEC CPU2006 benchmark suite was compiled using the modified LLVM infrastructure. The functions tagged as memory-bound were cross-validated against the Cachegrind output. An LLCMPK of 4 was set as a threshold as the condition for classifying a function as memory-bound. The classifier used have correctly classified 70% of the memory-bound functions. The produced binary was executed on the simulator infrastructure to test whether the methodology used resulted in EDP savings. EDP of the execution across different CPG periods and duty cycles is given in Figure A.4. The results show that none of the CPG configurations achieve better EDP than the nominal execution. This can be explained by the false-positive memory-bound benchmarks, that are compute-bound.

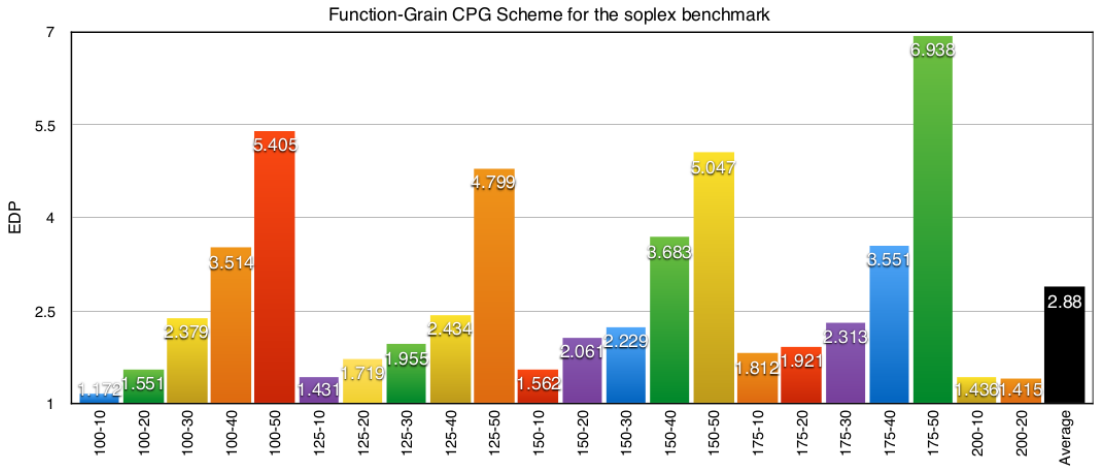


Figure A.4: EDP for *soplex*benchmark

A.4 Conclusion

A compiler-driven methodology to hint at the CPG controller for possibly memory-bound benchmarks was proposed in this chapter. The results indicate that a more accurate classifier is required, since throttling of a compute-bound function by the CPG scheme has detrimental effects on performance and energy consumption.

Appendix B

Specification of GLAM Generated Code

This appendix is intended to provide an exhaustive description for the specification of GLAM generated microbenchmarks. The specifications for each microbenchmark generated for performing the experiments in Chapters 4 and 5 are also provided.

B.1 Components of a GLAM Specification

As discussed in Chapter 4, GLAM uses a hierarchical specification to represent the generated code as shown in Figure B.1. A GLAM generated program, tagged as *experiment*, contains n number of modules where each module consist of m number of functions that contain a directed graph with each node representing a code block that executes an operation that is provided in the specification.

A minimal GLAM generated program contains a single module with a single function, that contains an entry and exit code block. Entry block initialises certain variables that will be needed throughout the execution of the generated function, and the exit block returns the register value as defined in the function prototype. List of all the keywords along with their descriptions and options are provided in Table B.1.

Table B.1: List of keywords used for a GLAM benchmark specification

Keyword	**Description**
experiment	Container for all the modules generated. Specifies aspects common to all modules that are contained.
name	String to be used for specifying an experiment, module, function or a code block.
descr	Description for a given experiment, module, function, or a code block. This is not generated into code, and is only used for increasing the readability of the specification.
repeat	Number of times the execution harness is executed to have statistically significant results.
arch	Target architecture for compiling the execution harness to. Can be any architecture supported by LLVM compiler (llc).
simulate	This boolean value is used to specify whether the harness will wrap code with simulator hooks or read from hardware performance counters.
hwpc	List of hardware performance counters to probe during execution.
module	List of modules to be generated
output	Output file name for the LLVM IR code that the module generates.
function	List of functions to be generated for the module.
prototype	Function prototype definition. ”returns” specify the return type of the function. ”input” specifies input parameters in terms of LLVM types. ”value” specifies the value of each parameter that will be passed onto the function. ”type” specifies whether the parameter is ”data”, ”dataptr”, or ”loopctr”.
block	List of code blocks to be generated for the function. ”distance” specifies instruction dependency distance within the generated instructions of the code block. ”operation” specifies the type of operation that the code block does. Supported operations are integer and floating point arithmetic and memory operations that repeatedly hit, miss, or thrash on cache.
instrcount	Number of instructions within the code block.
dataarg	Index of the input parameter that contains the data for this code block.
loop	Tuple that specifies whether loop post-condition is constant or parameter based.

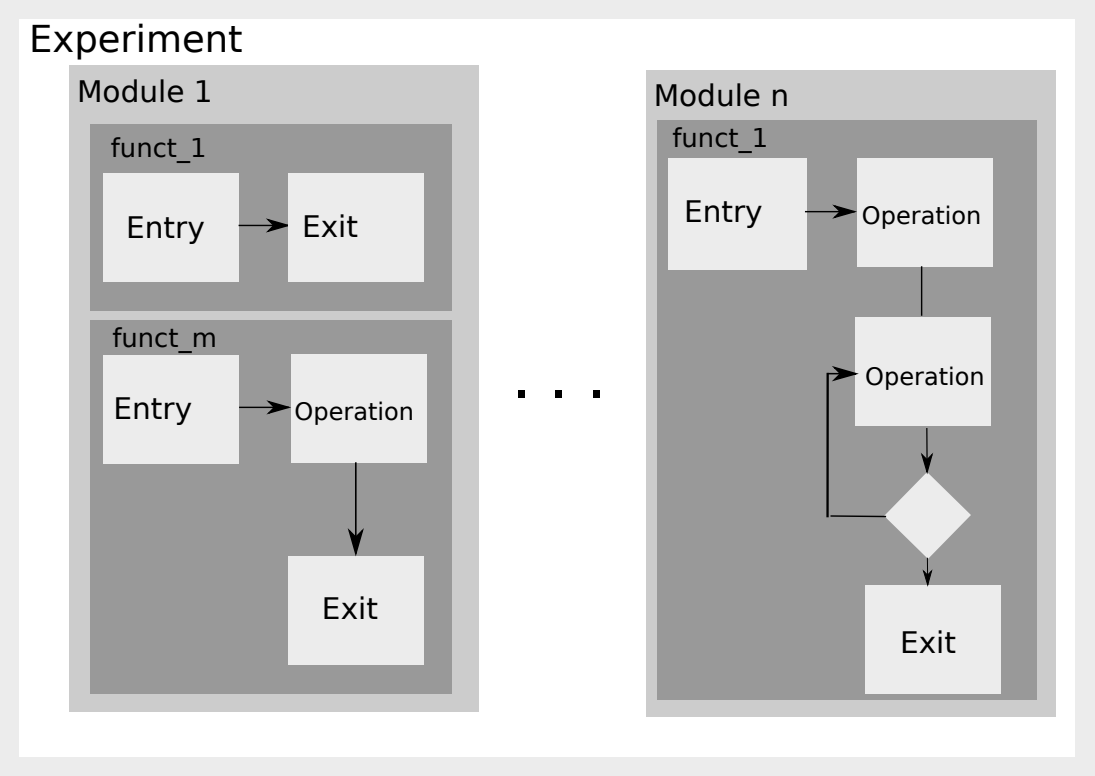


Figure B.1: Graphical representation of the GLAM specification.

Listing B.1: Experiment specification used in 4.2.1

```
{
    "experiment": {
        "name": "alutest",
        "descr": "Tests for the integer alu unit for
            add/sub/mul with dependency of 1:8",
        "repeat" : "10",
        "arch" : ["x86-64", "x86-64", "aarch64" ],
        "simulate" : [true, false, false],
        "hwpc" : ["UOPS_RETIRED", "PAPI_TOT_CYC", "
            IDQ_UOPS_NOT_DELIVERED", "ILD_STALL:IQ_FULL
            ",
                    "rapl:::PACKAGE_ENERGY:PACKAGE0", "
                        rapl:::PP0_ENERGY:PACKAGE1",
```

```
                "rapl:::PP0_ENERGY:PACKAGE0", "rapl
                    :::PACKAGE_ENERGY:PACKAGE1",
                "LLC_REFERENCES", "LLC_MISSES", "
                    BRANCH_INSTRUCTIONS_RETIRED", "
                    MISPREDICTED_BRANCH_RETIRED",
                "MEM_UOPS_RETIRED:ALL_LOADS", "
                    MEM_UOPS_RETIRED:ALL_STORES"],
        "module": [{
            "name": "intalutests",
            "descr": "helps in finding the number of
                alus",
            "output" : "output/intalutests.ll",
            "function": [{
                "name": "iadd1",
                "descr": "iadd1",
                "prototype":{
                    "returns": "int64",
                    "input": ["int64","int64"],
                    "value": ["0", "0"],
                    "type": ["loopctr","data"]
                },
                "block": [
                    {
                        "name": "b0",
                        "descr": "block b0",
                        "distance": "1",
                        "operation": "add",
                        "instrcount": "256",
                        "loop":["0","0"]
                    }]
            },{
                "name": "iadd2",
                "descr": "iadd2",
                "prototype":{
```

```
            "returns": "int64",
            "input": ["int64","int64"],
            "value": ["0", "0"],
            "type": ["loopctr","data"]
        },
        "block": [
            {
                "name": "b0",
                "descr": "block b0",
                "distance": "2",
                "operation": "add",
                "instrcount": "256",
                "loop":["0","0"]
            }]
    },{
        "name": "iadd3",
        "descr": "iadd3",
        "prototype":{
            "returns": "int64",
            "input": ["int64","int64"],
            "value": ["0", "0"],
            "type": ["loopctr","data"]
        },
        "block": [
            {
                "name": "b0",
                "descr": "block b0",
                "distance": "3",
                "operation": "add",
                "instrcount": "256",
                "loop":["0","0"]
            }]
    },{
        "name": "iadd4",
```

```
            "descr": "iadd4",
            "prototype":{
                "returns": "int64",
                "input": ["int64","int64"],
                "value": ["0", "0"],
                "type": ["loopctr","data"]
            },
            "block": [
                {
                    "name": "b0",
                    "descr": "block b0",
                    "distance": "4",
                    "operation": "add",
                    "instrcount": "256",
                    "loop":["0","0"]
                }]
        },{
            "name": "iadd5",
            "descr": "iad5",
            "prototype":{
                "returns": "int64",
                "input": ["int64","int64"],
                "value": ["0", "0"],
                "type": ["loopctr","data"]
            },
            "block": [
                {
                    "name": "b0",
                    "descr": "block b0",
                    "distance": "5",
                    "operation": "add",
                    "instrcount": "256",
                    "loop":["0","0"]
                }]
```

```
},{
    "name": "iadd6",
    "descr": "iadd6",
    "prototype":{
        "returns": "int64",
        "input": ["int64","int64"],
        "value": ["0", "0"],
        "type": ["loopctr","data"]
    },
    "block": [
        {
            "name": "b0",
            "descr": "block b0",
            "distance": "6",
            "operation": "add",
            "instrcount": "256",
            "loop":["0","0"]
        }]
},{
    "name": "iadd7",
    "descr": "iadd7",
    "prototype":{
        "returns": "int64",
        "input": ["int64","int64"],
        "value": ["0", "0"],
        "type": ["loopctr","data"]
    },
    "block": [
        {
            "name": "b0",
            "descr": "block b0",
            "distance": "7",
            "operation": "add",
            "instrcount": "256",
```

```
                "loop":["0","0"]
            }]
        },{
            "name": "iadd8",
            "descr": "iadd8",
            "prototype":{
                "returns": "int64",
                "input": ["int64","int64"],
                "value": ["0", "0"],
                "type": ["loopctr","data"]
            },
            "block": [
                {
                    "name": "b0",
                    "descr": "block b0",
                    "distance": "8",
                    "operation": "add",
                    "instrcount": "256",
                    "loop":["0","0"]
                }]
        },{
            "name": "imul1",
            "descr": "imul1",
            "prototype":{
                "returns": "int64",
                "input": ["int64","int64"],
                "value": ["0", "0"],
                "type": ["loopctr","data"]
            },
            "block": [
                {
                    "name": "b0",
                    "descr": "block b0",
                    "distance": "1",
```

```
                "operation": "mul",
                "instrcount": "256",
                "loop":["0","0"]
            }]
    },{
        "name": "imul2",
        "descr": "imul2",
        "prototype":{
            "returns": "int64",
            "input": ["int64","int64"],
            "value": ["0", "0"],
            "type": ["loopctr","data"]
        },
        "block": [
            {
                "name": "b0",
                "descr": "block b0",
                "distance": "2",
                "operation": "mul",
                "instrcount": "256",
                "loop":["0","0"]
            }]
    },{
        "name": "imul3",
        "descr": "imul3",
        "prototype":{
            "returns": "int64",
            "input": ["int64","int64"],
            "value": ["0", "0"],
            "type": ["loopctr","data"]
        },
        "block": [
            {
                "name": "b0",
```

```
                "descr": "block b0",
                "distance": "3",
                "operation": "mul",
                "instrcount": "256",
                "loop":["0","0"]
            }]
        },{
            "name": "imul4",
            "descr": "imul4",
            "prototype":{
                "returns": "int64",
                "input": ["int64","int64"],
                "value": ["0", "0"],
                "type": ["loopctr","data"]
            },
            "block": [
                {
                "name": "b0",
                "descr": "block b0",
                "distance": "4",
                "operation": "mul",
                "instrcount": "256",
                "loop":["0","0"]
            }]
        },{
            "name": "imul5",
            "descr": "iad5",
            "prototype":{
                "returns": "int64",
                "input": ["int64","int64"],
                "value": ["0", "0"],
                "type": ["loopctr","data"]
            },
            "block": [
```

```
            {
                "name": "b0",
                "descr": "block b0",
                "distance": "5",
                "operation": "mul",
                "instrcount": "256",
                "loop":["0","0"]
            }]
        },{
            "name": "imul6",
            "descr": "imul6",
            "prototype":{
                "returns": "int64",
                "input": ["int64","int64"],
                "value": ["0", "0"],
                "type": ["loopctr","data"]
            },
            "block": [
                {
                    "name": "b0",
                    "descr": "block b0",
                    "distance": "6",
                    "operation": "mul",
                    "instrcount": "256",
                    "loop":["0","0"]
                }]
        },{
            "name": "imul7",
            "descr": "imul7",
            "prototype":{
                "returns": "int64",
                "input": ["int64","int64"],
                "value": ["0", "0"],
                "type": ["loopctr","data"]
```

```
                },
                "block": [
                    {
                        "name": "b0",
                        "descr": "block b0",
                        "distance": "7",
                        "operation": "mul",
                        "instrcount": "256",
                        "loop":["0","0"]
                    }]
            },{
                "name": "imul8",
                "descr": "imul8",
                "prototype":{
                    "returns": "int64",
                    "input": ["int64","int64"],
                    "value": ["0", "0"],
                    "type": ["loopctr","data"]
                },
                "block": [
                    {
                        "name": "b0",
                        "descr": "block b0",
                        "distance": "8",
                        "operation": "mul",
                        "instrcount": "256",
                        "loop":["0","0"]
                    }]
            }]
        }]
    }
}
```

Listing B.2: Experiment specification used in 4.2.2

```
{
    "experiment": {
        "name": "memtest",
        "descr": "cache microbenchmark",
        "repeat" : "100",
        "arch" : ["x86-64", "x86-64", "aarch64" ],
        "archfile" : ["test_input/arch/xeon-e5-2670.
            json", "none", "none"],
        "simulate" : [true, false, true],
        "hwpc" : ["Instructions", "Cycles", "L1_Access
            ",
                  "LLC_Loads", "LLC_Load_Miss"],
        "module": [{
            "name": "llcmiss",
            "descr": "helps in finding the number of
                fpus",
            "output" : "output/memtest.ll",
            "function": [{
                "name": "l3miss",
                "descr": "load1",
                "prototype":{
                    "returns": "float",
                    "input": ["int64","int64","int64
                        ","floatptr"],
                    "value": ["0", "0",<
                        modify_for_datasize>,"0"],
                    "type": ["data","data","loopctr","
                        dataptr"]
                },
                "block": [
                    {
                        "name": "b0",
```

```
                        "descr": "block b0",
                        "distance": "1",
                        "operation": "l3miss",
                        "instrcount": "32",
                        "distance": "1",
                        "dataarg": "3",
                        "loop":["0","2"]
                    }]
            }]
        }]
    }
}
```