NTNU – Trondheim
Norwegian University of
Science and Technology

Enabling Research on Energy-Efficient System Software Using the SHMAC Infrastructure

Benjamin Bjørnseth

Master of Science in Computer Science
Submission date: February 2015
Supervisor: Lasse Natvig, IDI
Co-supervisor: Asbjørn Djupdal, IDI

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

The SHMAC prototype is an ongoing research project within the Energy-Efficient Computing Systems (EECS) strategic research area at the Faculty of Information Technology, Mathematics and Electrical Engineering (IME), Norwegian University of Science and Technology (NTNU). SHMAC is planned to run in a Field-Programmable Gate Array (FPGA) and be an evaluation platform for research on heterogeneous multi-core systems. Due to the Dark Silicon effect, future computing systems are expected to be power limited. The goal of the SHMAC project is to propose software and hardware solutions for future power-limited heterogeneous system.

In order to use the SHMAC platform for researching energy-efficient software and hardware, it is necessary to be able to evaluate the energy efficiency of a solution and compare it with others. The ability to gather energy efficiency metrics is therefore paramount. However, as the platform is realized on an FPGA it is not necessarily easy or even possible to attain representative data by simply measuring the power- and energy consumption. By representative data we mean numbers that represent the energy efficiency that can be achieved when SHMAC is implemented in silicon as a processor chip.

Additionally, the construction of an operating system multicore port to the SHMAC is desirable for system software and application software research. A single-core port of the research operating system Barrelfish exists, which may be used as a base for implementing a multicore version. This is necessary to research software in a heterogeneous multicore hardware context, which the SHMAC project aspires to do.

Investigate strategies for evaluating the energy efficiency of software and hardware designs on the SHMAC prototype. Based on the investigations, develop or describe a prototype of an energy efficiency measurement framework for SHMAC. Evaluate or discuss its properties such as coverage, accuracy and user-friendliness. Additionally, implement multicore support for the Barrelfish operating system. If time permits, the framework may be used to evaluate the Barrelfish and Linux operating system ports.

Supervisors: Lasse Natvig and Asbjørn Djupdal.

Abstract

The energy efficiency of computer systems is becoming an increasingly important constraint in the design of microprocessors. Energy consumption impacts battery life and electricity bills, while power consumption is important when considering device thermal constraints and cooling costs. These factors have always been important for the embedded, hand-held device and data centre markets. Recently, the breakdown of Dennard scaling has hampered the ability to reduce transistor dimensions while keeping processor power density constant. As high-end processors drive further reduction of transistor dimensions, this breakdown increases the importance of power consumption as a constraint in their design.

Heterogeneous processor architectures have the potential of increasing the energy efficiency of computer systems. To research the design and system software control of such systems, the IME faculty at NTNU launched the SHMAC research project. The project ambition is to explore the heterogeneous multicore architecture design space through customization of a generic architecture, which is instantiated on an FPGA to speed up evaluation. However, the current SHMAC infrastructure lacks a method for estimating the energy consumption of a processor chip implementation of the design it embodies. There is also no multi-core operating system available, which hampers research on system software energy efficiency.

This dissertation enables research on the energy efficiency of system software using the SHMAC infrastructure by filling these two gaps. First, we extend the existing SHMAC-port of the operating system Barrelfish to support running on multiple cores. Second, we complement the SHMAC infrastructure with an energy efficiency estimation framework. The framework includes a method for creating energy consumption models for hardware components for which only an HDL implementation is available. The efficacy of the method is demonstrated through application on the existing SHMAC hardware components. The average estimation error each cycle from all models combined is 1.1 %. A hardware infrastructure which enhances the SHMAC infrastructure to use these models and report online energy consumption estimates is also included in the framework. The infrastructure enables energy sampling periods of approximately 12 milliseconds, does not impact the FPGA execution speed, and has a total FPGA resource overhead of approximately 18 % for the processor core and 104 % for the router.

Sammendrag

Energieffektiviteten til datamaskinsystemer blir en stadig viktigere beskrankning under design av mikroprosessorer. Energiforbruk påvirker tilgjengelig batteritid og strømregninger, mens effektforbruk er viktig med hensyn på enheters varmebegrensninger og nedkjølingskostnader. Disse faktorene har alltid vært viktige for innvevde system, håndholdte enheter og datasentere. I nyere tid har sammenbruddet til Dennard-skalering hindret muligheten til å redusere transistordimensjoner mens effekttettheten holdes konstant. Ettersom prosessorer i ytelsestoppsjiktet presser stadig reduksjon av transistordimensjoner, øker dette sammenbruddet viktigheten av effektforbruk som begrensning i deres design.

Heterogene prosessorarkitekturer har potensialet til å øke energieffektiviteten til datamaskinsystem. For å forske på designet og systemprogramvarekontrollen av slike system, har IME-fakultetet ved NTNU satt i gang et forskningsprosjekt kalt SHMAC. Prosjektets hensikt er å undersøke designrommet til heterogene flerkjernearkitekturer ved å tilpasse en generisk arkitektur, som implementeres på en FPGA for å øke hastigheten på evaluering. Imidlertid mangler den nåværende SHMAC-infrastrukturen en metode for å estimere energyforbruket til en silikonbrikkeimplementasjon av prosessordesignet den representerer. Det finnes heller intet flerkjernekapabelt operativsystem, hvilket hindrer forskning på energieffektiv systemprogramvare.

Denne avhandlingen muliggjør forskning på energieffektiv systemprogramvare via SHMAC-infrastrukturen ved å fylle disse to manglene. Som steg én, utvider vi en eksiterende SHMAC-variant av operativsystemet Barrelfish til å støtte eksekvering på flere prosessorkjerner. Som steg to, utvider vi SHMAC-infrastrukturen med et rammeverk for estimering av energieffektivitet. Rammeverket inkluderer en metodologi for å lage modeller av energiforbruk for hardware-komponenter som kun eksisterer som en HDL-implementasjon. Metodologiens vellykkethet demonstreres ved anvendelse på eksisterende komponenter i SHMAC. Gjennomsnittet av estimeringsfeilprosentene hver sykel fra samtlige modeller kombinert er 1.1 %. Rammeverket inneholder også en hardware-infrastruktur som utvider den eksisterende SHMAC-infrastrukturen til å bruke energiforbruksmodellene og rapportere løpende energiforbruksestimat. Infrastrukturen muliggjør energimåleperioder på cirka 12 millisekund, påvirker ikke FPGA-ens eksekveringshastighet, og medfører en gjennomsnitlig økning i FPGA-ressursbruk på cirka 18 % for prosessorkjernen og 104 % for chipnettverksruteren.

Preface

This report is submitted to the Norwegian University of Science and Technology in partial fulfilment of the requirements for an MSc degree in computer science.

This work has been conducted at the Department of Computer and Information Science, NTNU, part time throughout 2014 while simultaneously beginning on a PhD education. I have as such been part of the CARD research group which offered this master project. Professor Lasse Natvig has been the dissertation supervisor, and Asbjørn Djupdal the co-supervisor.

Acknowledgements

I would like to thank my supervisors Lasse Natvig, Magnus Jahre and Asbjørn Djupdal for their project guidance, technical assistance and dissertation proofreading efforts.

Thanks to Snorre Aunet, Tore Barlindhaug, and Trond Ytterdal at the IET department for their assistance regarding the use and installation of ASIC tools.

Thanks to Bjørn Christian Seime for his diligent and laborious proofreading, yielding invaluable feedback on dissertation content, clarity and style.

Thanks to Stian Hvatum for tipping me about the existence of the program *socat*.

Thanks to Yaman Umuroglu for helping me appreciate the complexity of DDR power consumption.

Thanks to Ragnar Andreassen for reading through the final text, providing important insight into content clarity for an external professional.

Finally, I would also like to thank my wife for her patience as well as her insightful comments on general report structure.

Contents

List of Figures

List of Tables

List of Abbreviations

ALU Arithmetic Logic Unit.

APB Advanced Peripheral Bus.

ASIC Application-Specific Integrated Circuit.

BRAM Block Random-Access Memory (RAM).

BSP Boot-Strap Processor.

CAS Column Address Strobe.

CL Column Address Strobe (CAS) Latency.

CMP Chip Multi-Processor.

CNode Capability Node.

CPSR Current Program Status Register.

CPU Central Processing Unit.

CWL CAS Write Latency.

DDR Double Data Rate.

DRAM Dynamic RAM.

DSE Design Space Exploration.

DSP Digital Signal Processor.

DVFS Dynamic Voltage-Frequency Scaling.

ECC Error-Correcting Code.

EECS Energy-Efficient Computing Systems.

FIFO First-In First-Out.

FPGA Field-Programmable Gate Array.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HDL Hardware-Description Language.

I/O Input/Output.

IME Faculty of Information Technology, Mathematics and Electrical Engineering.

IRQ Interrupt Request.

ISA Instruction Set Architecture.

LMP Local Message Passing.

LUT Look-Up Table.

NAT Network Address Translation.

NIC Network Interface Controller.

NTNU Norwegian University of Science and Technology.

NUMA Non-Uniform Memory Access (UMA).

OS Operating System.

RAM Random-Access Memory.

RT Register Transfer.

RTL Register Transfer Level.

SDRAM Synchronous Dynamic RAM (DRAM).

SPSR Saved Program Status Register.

SRAM Static RAM.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UMA Uniform Memory Access.

UMP User-level Message Passing.

VCD Value Change Dump.

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware-Description Language (HDL).

VHSIC Very High Speed Integrated Circuit.

VPD VCDPlus Dumping.

ZBTRAM Zero Bus Turn-around RAM.

Chapter 1
Introduction

1.1 The SHMAC Project

The energy efficiency of microprocessors is important across several market segments. For embedded systems, reduced energy consumption leads to longer battery lives. For portable devices, reduced power consumption leads to lower heat dissipation. For data centres, reduced energy consumption leads to reduced electricity bills and reduced power consumption leads to reduced cooling costs. As such, energy efficiency is often an important constraint when designing processors.

Recently, the importance of energy efficiency has increased in the design of high-performance processors. Historically, the performance and energy efficiency of computer systems doubled approximately every 18 months [KBSW11]. This exponential growth was sustained by decreasing transistor dimensions while also reducing the power consumption per transistor, using the method known as Dennard scaling [DGnY+74]. This lead to faster transistors, more transistors, and constant power density for a processor chip of a fixed area. However, the ability to scale transistors in this fashion has recently broken down. Although it is still possible to reduce transistor dimensions, the power consumption per transistor can not be sufficiently reduced [DCK07]. We therefore get more transistors for a processor chip of a fixed area, but they can not all be used simultaneously as the power density would increase beyond the limits of cooling technology. This leads to a situation where an exponentially increasing fraction of the transistors on a processor must be disabled, a phenomenon which is called the *Utilization Wall* [VSG+10]. The large fraction of the chip which must be turned off is referred to as *Dark Silicon* [EBSA+12]. Using the increased number of transistors to improve performance therefore requires more energy-efficient processor designs.

There are several alternative approaches to exploiting the Dark Silicon [Tay12]. One interesting approach is building heterogeneous processors. The heterogeneity may come from processing elements representing different power-performance trade-off

points, or from specialized processing elements where generality is traded for efficiency. Heterogeneity can improve energy efficiency within the limits of the Utilization Wall by only enabling the most suitable processing units at any given time. However, such a system poses significant hardware and software research challenges such as how processor cores should be selected, what level of specialization to use, how scheduling should be done across heterogeneous processing, and how the energy consumption of the system is best managed [BC11].

To research energy-efficient computer systems in general, and the challenges arising from Dark Silicon in particular, the IME faculty at NTNU has started the research initiative EECS [EECa]. A subproject of EECS is the SHMAC project [EECb]. The project ambition is to investigate both software and hardware solutions for heterogeneous multicore processor chips by using a generic heterogeneous multicore infrastructure. The infrastructure fixes the interconnect, but leaves the characteristics of each core unspecified. Different heterogeneous configurations may then be created through variation of the core selection. This idea is illustrated in Figure 1.1. The entire architecture is instantiated and run on an FPGA for evaluation purposes, which may speed up the exploration of the large heterogeneous multicore design space compared to using traditional software simulators.

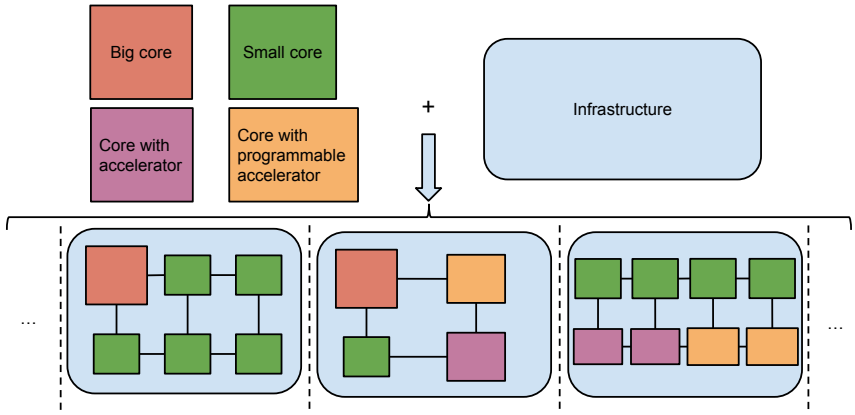

Figure 1.1: The idea behind the SHMAC infrastructure is to easily study different heterogeneous designs by providing an infrastructure and a set of cores. Heterogeneity is attained by including different cores in the infrastructure.

1.2 Energy-Efficient System Software

One of the goals of the SHMAC project is to research how system software may be created to most efficiently exploit the resources available in heterogeneous architectures. The SHMAC infrastructure offers a suitable platform for running experiments,

but operating systems with which to experiment is missing. As such, several projects were started with the goal of porting different operating systems to the SHMAC infrastructure.

At the outset of this dissertation project, I had been part of a project group which developed a single-core SHMAC-port of the research operating system Barrelfish [BS13][BBD+09]. In addition, master dissertation projects for porting Linux [AA14] and Free-RTOS [Wal14] to SHMAC had been initiated. Completing the Barrelfish port by implementing multicore support was as such a natural extension of my fall project, as it would provide the SHMAC project with an operating system alternative with a more innovative architecture with which to experiment.

1.3 Estimating Energy Efficiency

Looking further than the software and hardware required to run experiments, research also requires some method of obtaining the result of an experiment. In particular, research on energy-efficient system software requires a method of evaluating the power consumption of the system. However, at the beginning of this dissertation no method had been devised for obtaining power consumption results.

The heterogeneous processor designs instantiated with the SHMAC infrastructure do not exist as actual processor chips. As such, the processor power consumption can not be measured directly. Since the processor designs are implemented on an FPGA, one possible strategy would be to measure the FPGA power consumption and from this somehow derive the expected Application-Specific Integrated Circuit (ASIC) power consumption. However, this is an unattractive option for the following reasons:

– It is not clear that FPGA power consumption correlates with that of an ASIC implementation of the processor design. One reason to suspect otherwise is that hardware components such as multiplexers have different power characteristics when implemented in FPGAs than in ASICs [WBR11]. Activity in certain components in a processor may therefore in theory lead to high FPGA power consumption and low ASIC power consumption, or vice versa.

  Another possible source of error is that power consumption differences measured between different processor designs may be owed to one design being more suited for FPGA implementation. Such effects may not be exploitable in an ASIC.

  A third concern is that the FPGA implementation embeds the processor design in an FPGA-specific top-level module, whose power consumption would not be

present in an ASIC system. The same goes for power dissipated due to activity on FPGA chip I/O pins.

Finally, FPGAs implementations must pay certain fixed overheads such as needing to maintain their configuration. It may be that this base power consumption would drown the contributions from the design itself, making power consumption differences stemming from design decisions of the researcher indiscernible from measurement noise.

- Even if the previous point was refutable, it would be difficult to determine the nature of the correlation. An earlier study comparing the power consumption of ASIC and FPGA implementations demonstrated that the dynamic power consumption ratio varied between 5.2× and 26× depending on the hardware module and specific FPGA implementation techniques [KR06]. Thus, even trying to calculate the ASIC power consumption by determining the ratio between the FPGA and ASIC implementations would likely be both difficult and inaccurate.

- Finally, there are drawbacks to measuring FPGA power even if it provided a meaningful indicator of ASIC power. Being dependent on FPGA measurements would limit the power-saving architectural features which may be investigated to the subset supported by the FPGA. Specifically, techniques such as power- and clock gating might not be available in the form or extent desired by the computer architecture researcher. It may also be cumbersome to perform the measurements, as physical intervention in the FPGA system is required.

The lack of a clear answer to how energy efficiency may be evaluated motivates a study which investigates the available energy efficiency evaluation alternatives. Developing an energy efficiency estimation framework which enables obtaining power consumption results from experiments would take the SHMAC project ambition a long way from idea to reality.

1.4 Assignment Interpretation

The goal of the assignment is enabling research on energy-efficient system software. To meet this end, the assignment text describes two project goals: extending the existing Barrelfish port with support for running on multiple cores, and investigating energy efficiency estimation for the SHMAC infrastructure. Both goals contribute towards the ultimate goal of research on energy-efficient system software, but the completion of one goal is independent of the completion of the other. Therefore, the goals will be solved independently.

Subtasks related to the completion of each goal are enumerated in the following lists. A label beginning with B indicates a task associated with the completion of the Barrelfish port, and a label beginning with E a task associated with energy efficiency estimation support.

Mandatory Tasks

B1 Extend the existing Barrelfish implementation to provide support for multiple cores.

E1 Investigate how energy efficiency estimation using the SHMAC infrastructure may be supported.

E2 Describe a prototype of a framework which implements the strategy identified in task `E1`.

E3 Evaluate the prototype framework described in `E2` based on the following three metrics:

Coverage How many factors affecting the system power consumption are included? Have any notable hardware components been excluded?

Accuracy How precise are the power consumption estimates? How truthfully do they represent a real implementation of the design under study?

User-friendliness How easy is it to use the prototype to run experiments and gather the energy data?

Optional Tasks

B2 Add support in Barrelfish for new features of the SHMAC platform introduced during the course of this project.

B3 Add support for the Barrelfish console application.

E4 Implement the framework described in `E2`.

E5 Use the framework to evaluate the energy efficiency of the Barrelfish and Linux ports.

Task `E1` is assumed to not entail detailed qualitative analysis based on experiments, as this would require implementing several different alternatives whereas the problem description only optionally requires the implementation of the most promising alternative.

Task E3 involves the evaluation of the accuracy of the framework prototype. Accuracy is here interpreted to mean a combination of trueness and precision, corresponding to the definition in ISO 5725-1. Trueness is a measure of how truthfully an estimator will represent the true values, and precision indicates the estimate variability. These terms are also known in statistics as validity and reliability, respectively. A more elaborate explanation of trueness and precision is given in Appendix A.

The SHMAC infrastructure is expected to be enhanced with binary-incompatible hardware upgrades. Although not strictly mandated by the problem description, task B2 will allow the resulting operating system to be of use on the most recent platform present at the end of the project. Of the optional tasks, this is assigned the highest priority owing both to its importance and its perceived ease.

Task E4 is an important contribution to render the work in E2 and E1 useful. It will also facilitate quantitative evaluation in task E3. Consequently, this task is also assigned high priority. It is, however, not clear how large the workload in E4 will be, nor how successfully it may be executed. It is therefore given a lower priority than task B2, as this task is more likely to yield profitable results.

Task B3 is an extension of the project description. Although the assignment text only mandates the implementation of multicore support for Barrelfish, the intent is to make the operating system complete for research purposes. A working console application allows researchers to start different benchmarks or vary benchmark input without having to recompile and restart Barrelfish. As the task is not of direct relevance of the project description text, and as it only makes Barrelfish more convenient to use, it is given a lower priority than tasks B2 and E4.

Finally, task E5 is perceived as an arbitrary example application of the developed estimation framework. Being dependent on the completion of all the other tasks, possibly with the exception of B2 and B3, and not resulting in any product directly useful for the SHMAC project, this task is assigned the lowest priority of all identified tasks.

1.5 Contributions

The contributions made in this dissertation are listed in Table 1.1. The table also lists which subtasks the contribution at least partially solves, as well as what section in the dissertation the contribution is described.

1.6 Dissertation Organization

The remainder of the dissertation is organized as follows.

Table 1.1: The contributions made in this dissertation.

	Contribution	Subtask	Section
1.	Multicore support in the SHMAC-port of Barrelfish	B1	Section 5.2
2.	Support for ISA upgrades and working data caches in the SHMAC infrastructure	B2	Sections 5.3 and 5.4
3.	Console support in Barrelfish	B3	Section 5.5
4.	A description of modelling methods suitable for different parts of the SHMAC infrastructure, both current and future.	E1 and E2	Sections 2.4 and 3.3
5.	The development of a complete hardware infrastructure, which enhances the SHMAC infrastructure with live energy consumption estimation capabilities.	E2 and E4	Sections 3.2 and 4.2
6.	A quantification of the potential modelling method efficacy through its application to the current SHMAC infrastructure components.	E3 and E4	Sections 4.1 and 6.2
7.	The software and tool infrastructure necessary for the execution of the proposed modelling methods.	E4	Appendix B

Chapter 2: Background presents background information on the SHMAC infrastructure, the Barrelfish operating system, and a survey on energy efficiency estimation methods used in research. The chapter ends with a presentation of an energy efficiency modelling strategy suitable for the SHMAC project.

Chapter 3: Energy Efficiency Estimation Framework Design presents the design of an energy efficiency estimation framework, encompassing both modelling methods based on the presentation in Chapter 2 and enhancements to the SHMAC infrastructure enabling online energy consumption estimates.

Chapter 4: Energy Efficiency Estimation Framework Implementation presents the implementation of the design presented in Chapter 3.

Chapter 5: Barrelfish Implementation discusses the modifications and enhancements which were implemented to complete the port of Barrelfish to the SHMAC infrastructure.

Chapter 6: Evaluation presents tests of correctness of the Barrelfish port and the energy efficiency estimation framework, and analyses of the framework accuracy, user-friendliness, coverage and performance.

Chapter 7: Discussion highlights important attributes or limitations in the current implementations, and discusses the degree of completion of the tasks in Section 1.4.

Chapter 8: Conclusion and Future Work summarizes the contributions made in this dissertation, and concludes on the value of the contributions to the SHMAC project. Finally, it presents possible future extensions and improvements.

Chapter 2
Background

This chapter will present the foundation on which this dissertation is built.

We will first discuss technical details of the SHMAC project relevant for this dissertation. Section 2.1 will describe the SHMAC infrastructure and test environment. Section 2.2 will then present the Barrelfish operating system, as well as the state of the port to the SHMAC platform.

Finally, we will list related work on energy efficiency estimation methods. Section 2.3 contains an overview of state-of-the-art techniques for conducting research on energy efficiency in a variety of situations. These techniques were known to our research group prior to the master dissertation. Section 2.4 will then present a method tailored for estimating the energy consumption of the SHMAC, and studies which demonstrate the successful use of such a method. The design of this method and the survey of similar work were conducted as part of this dissertation to answer the investigation mandated by task `E1` in Section 1.4.

2.1 The SHMAC Infrastructure

The SHMAC infrastructure is a generic, single-Instruction Set Architecture (ISA), tile-based [BEA+08] processor architecture designed to accommodate numerous different heterogeneous multicore designs. The architecture is depicted in Figure 2.1. The tiles are connected in a mesh topology, with the same kind of router included on each tile providing a common routing interface. Heterogeneity may be introduced into the system by varying the kind of tiles included. Some tiles may contain simple, in-order processor cores; others may contain complex, out-of-order cores. Hardware accelerators may also be included in the system, either connected to a processor on a single tile or in tiles of their own. Not all tiles are used for computation: tiles may also provide services such as off-chip or on-chip memory access and host communication. To maintain hardware configuration independence for the software, all general-purpose computation tiles are required to support the same instruction

set. In this aspect, ISA heterogeneity is discarded as a design space dimension in order to maintain the ambition of simple exploration of the remaining dimensions.

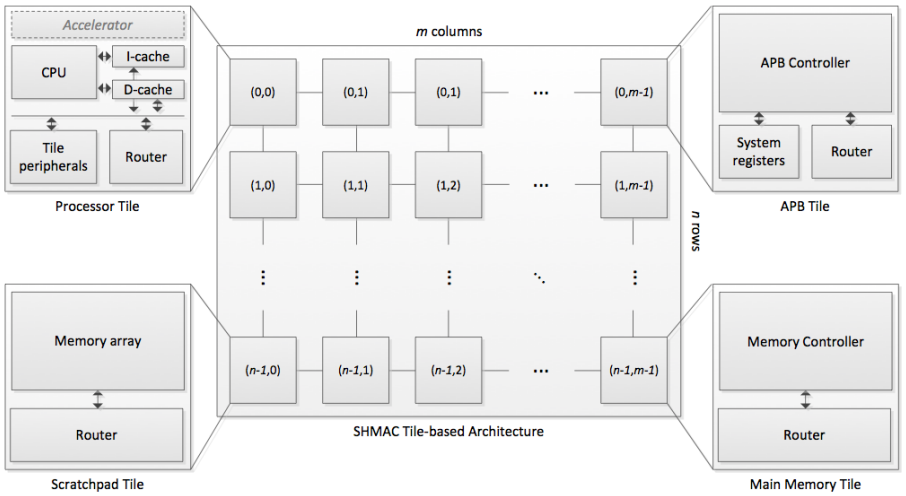

Figure 2.1: The SHMAC processor infrastructure. Source: [EECb].

At the beginning of this dissertation project, there were four kinds of tiles available. The first tile kind communicates with the host system through an Advanced Peripheral Bus (APB), an ARM bus standard used for low-complexity, low-bandwidth interfaces to peripherals [APB]. The second tile kind contains a memory controller, providing access to off-chip memory. The APB tile and the main memory tile must always be included in a SHMAC configuration. The third kind of tile, called the scratchpad tile, implements on-chip memory access by creating a memory array using the Block RAM (BRAM) resources on the FPGA. The BRAM resources on the FPGA are also used for other purposes in the SHMAC, such as caches, but in this dissertation BRAM will refer to memory available through scratchpad tiles unless explicitly stated. The fourth tile kind contains the Amber [Ope13] ARMv2a processor. The core is a 5-stage in-order core with one level of data and instruction write-through cache. Caching may be selectively enabled for different regions of the address space, with a 128 MB granularity. Each processor tile contains this core, coupled with local timer and interrupt controller peripherals.

In ARMv2a, there are four processor modes. The modes are listed in Table 2.1. The IRQ and Fast IRQ modes are entered when the system receives interrupts of either type. In SHMAC, no fast interrupts are used, so the Fast-IRQ mode is unused. Interrupts on the IRQ line can come from either a timer module or serial input from the host. Since all modes use their own, private stack pointer register,

each stack pointer must be initialized to a separate stack when booting the system. The supervisor mode is entered on any other exception, including reset; undefined instruction; prefetch abort; data abort; and address exception. User mode is only entered by manually changing the mode in any of the other modes.

Table 2.1: Processor modes in the ARMv2a ISA. Banked registers refer to which registers have mode-specific versions.

Name	**Banked Registers**
Supervisor	Stack pointer (`r13`), link register (`r14`)
IRQ	`r13`, `r14`
Fast IRQ	`r8`-`r15`
User	None

One peculiarity with the ARMv2a ISA, is the semantics of register `r15`. The role of the register is two-fold: bits 25-2 contain the program counter, whereas bits 31-26 and 1-0 contain control- and status information. Bits 1-0 contain the processor mode; bits 27-26 contain IRQ and FIRQ mask flags, and bits 31-28 contain ALU status flags (overflow, carry, zero and negative-bits).

The memory map of the SHMAC infrastructure is listed in Table 2.2. Most of the 32-bit address space is used for off-chip RAM. The last 128 MB of the address space is used for scratchpad memory, tile registers and system registers. The tile registers contains access to per-tile information, such as the coordinates of the tile, and peripherals, such as the interrupt controller. The system registers give access to global state, which is input and output from the host; a system tick counter; the total number of tiles; and a register called `SYS_READY`. This register is used to control multicore boots. When the system is started, only the first CPU tile will begin running to allow initialization of shared data structures and mutexes guarding them. The first core may start all the other cores by writing 1 to the `SYS_READY` register.

The router uses a parallel implementation with 196-bit wide links. The interconnect has no direct support for either cache coherency or message passing. This means that communication between processor tiles must use uncached memory, or flush the cache before every read to a channel. The scratchpad tiles are designed to accommodate this: mapped to the upper 128 MB of the address space, this part of the memory can be left uncached independently of main memory to support shared, coherent memory with potentially efficient access depending on the distance to the tile. It should, however, be mentioned that at the outset of the project, a bug in the SHMAC implementation prevented data caches from working. As such, no programs were run with cache enabled, and communication between cores was handled simply through uncached access to any memory location.

Table 2.2: The memory map of the SHMAC infrastructure.

Description	Address Range	Size
Off-chip RAM	`0x00000000 - 0xf7ffffff`	3968 MB
Scratchpad tile 0	`0xf8000000 - 0xf8ffffff`	16 MB
Scratchpad tile 1	`0xf9000000 - 0xf9ffffff`	16 MB
Scratchpad tile 2	`0xfa000000 - 0xfaffffff`	16 MB
Scratchpad tile 3	`0xfb000000 - 0xfbffffff`	16 MB
Scratchpad tile 4	`0xfc000000 - 0xfcffffff`	16 MB
Scratchpad tile 5	`0xfd000000 - 0xfdffffff`	16 MB
Scratchpad tile 6	`0xfe000000 - 0xfeffffff`	16 MB
Scratchpad tile 7	`0xff000000 - 0xfffdffff`	15.875 MB
Tile registers	`0xfffe0000 - 0xfffeffff`	64 KB
System register	`0xffff0000 - 0xffffffff`	64 KB

The SHMAC infrastructure is instantiated and run on ARM Versatile platforms [Ver], configured to combine a host controller board and an FPGA board. This SHMAC platform is illustrated in Figure 2.2. The host controller communicates with the SHMAC infrastructure by accessing memory-mapped registers on the APB tile. The registers are used to reset the SHMAC, read and write memory, and implement serial I/O. There are two different Versatile platforms in use, one new Versatile Express platform with a Virtex 7 FPGA and one older RealView platform with a Virtex 5 FPGA. From the point of view of software running on the SHMAC, the two platforms differ primarily in the memory available to the FPGA: the Versatile Express provides 4 GB of DDR3 RAM, while the RealView is limited to 32 MB of ZBTRAM [ZBT14].

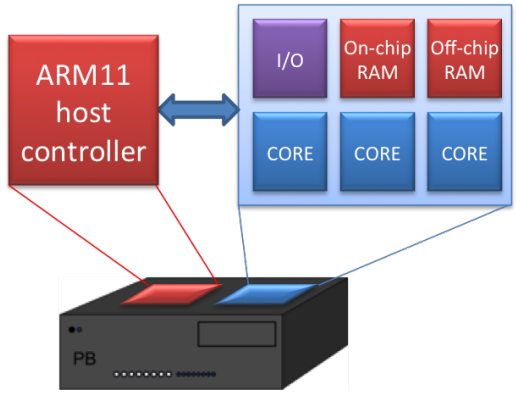

Figure 2.2: The platform with which the SHMAC infrastructure is tested. Source: [BS13].

The SHMAC configuration to build is controlled through a text configuration file. The file contains one letter per tile to include, written on multiple lines to reflect the desired layout into rows and columns. Each letter denotes a specific tile. The available letters and their mapping to tiles are listed in Table 2.3.

Letter	Tile
`V`	APB tile
`Z`	ZBTRAM main memory tile (only used on RealView platforms)
`D`	DDR main memory tile (only used on Versatile Express platforms)
`A`	Amber tile
`R`	Scratchpad tile

Table 2.3: Letter-to-tile mapping in SHMAC configuration file

An example configuration file and the corresponding SHMAC instantiation is illustrated in Figure 2.3. As per Table 2.3, the configuration file denotes a 2×4 SHMAC configuration with an APB tile, a DDR main memory tile, three Amber tiles and three scratchpad tiles.

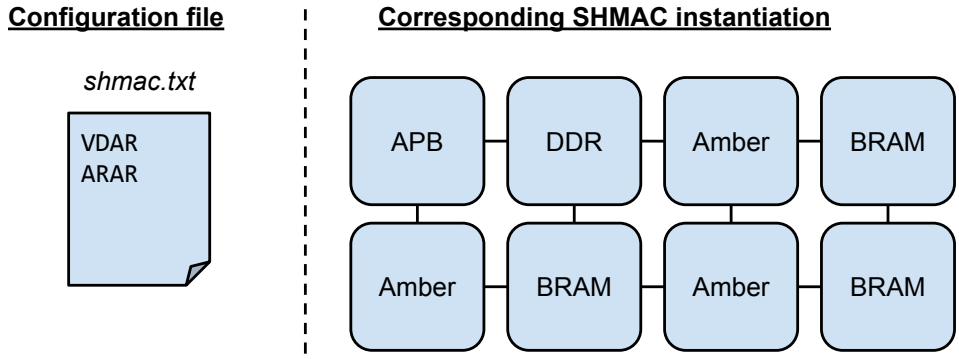

Figure 2.3: An example configuration file and the corresponding SHMAC instance.

Concurrently with this project, several other enhancements to the SHMAC implementation were planned. One relevant potential contribution was extending the Amber core to support the ARMv3 instruction set, and subsequently the ARMv4T instruction set [AA14]. The ARMv3 upgrade changes the ISA in binary-incompatible ways. The reason is that the semantics of the `pc` register is altered, such that it uses all 32 bits for the program counter value. A new register called Current Program Status Register (CPSR) is included to keep the status information. Each mode gets its own CPSR. Another register called Saved Program Status Register (SPSR) is used to back up the value of CPSR when entering a different mode. New instructions

were also included to manage these registers: the `msr` instruction moves a value from a regular register to a status register, and the `mrs` instruction moves values in the opposite direction. In addition to the new status registers, two new CPU modes were included. The first is *abort mode*, entered instead of supervisor mode on prefetch abort and data abort exceptions. The second is *undefined mode*, entered instead of supervisor mode when taking an undefined instruction exception. Each mode has their own versions of CPSR, `r13` and `r14`.

The ARMv4T upgrade is binary compatible with ARMv3, as it only extends the capabilities of the ISA without modifying what was there previously. The main addition from the point-of-view of system software is a new mode called *system mode*. This mode has no banked registers, using the user mode `r13` and `r14`, and like user mode it must be explicitly entered from another privileged mode. Unlike user mode, it has the same privilege level as supervisor mode. The mode is intended to support nested handling of interrupts: if interrupts are handled in a mode which may also be entered through the exception vector, then the link register could be trashed if an interrupt arrives while the interrupt handler is executing a function which has not (yet) backed up the link register [Sys]. ARMv4T also includes new instructions, relevant for compilers and possibly assembly language routines.

In addition to the ISA upgrade plans, there were plans to fix the bug preventing the use of data caches. This would be relevant for the Barrelfish port, as enabling data caches for main memory would require inter-core communication to be conducted through scratchpad memory.

2.2 Barrelfish

2.2.1 Operating System Architecture

Barrelfish is an experimental Operating System (OS) developed by ETH Zürich in collaboration with Microsoft, with the first public release in 2009 [BBD+09]. The idea behind the system is to investigate an operating system architecture which better fits future computer systems, which are characterised by having increasingly diverse system properties such as interconnects and cache structures, an increasing number of increasingly diverse processor core types, and where there may possibly be no shared memory available between processing elements. The architecture of the operating system is called the *multikernel*, in which the system is viewed as a collection of independent, networked, communicating computing devices[1]. The OS architecture is depicted in Figure 2.4. As illustrated, one OS node runs on each core, and contains its own replica of the shared OS state. Communication and coordination amongst the different OS nodes is performed using message passing.

[1] E.g. CPUs, Network Interface Controllers (NICs), and Graphics Processing Units (GPUs)

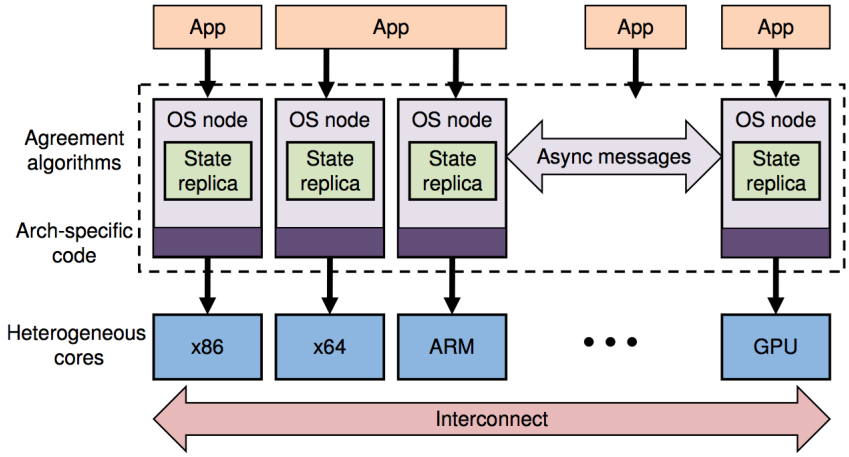

Figure 2.4: The multikernel operating system architecture. Source: [BBD+09]

2.2.2 Barrelfish Implementation Details

System Structure Barrelfish is an implementation of the multikernel architecture. Its structure is depicted in Figure 2.5[2]. Comparing with the general multikernel architecture in Figure 2.4, the main difference is that the OS node of the multikernel is split into a *CPU-driver* and a *monitor*. The CPU-driver manages core-local resources, most notably the CPU time but also core-local peripherals such as timers and interrupt controllers. It is not preemptible, leading to single-threaded kernel code. The CPU driver is the only part of Barrelfish which executes in privileged mode. CPU drivers will typically not communicate directly with each other: this is instead handled by the monitor. The monitor program coordinates the OS-nodes running on different cores, using message passing. For instance, page unmapping operations are implemented using a one-phase commit protocol to ensure no stale data are present in any TLB before the page is reclaimed. The monitor also provides an access point for applications to the OS-node: some operations use the monitor, while some use the CPU driver directly. The default inter-core communication method is User-level Message Passing (UMP), in which a message-passing channel between monitors is allocated from a shared memory resource. Messages are then sent by writing cache-line-sized blocks to this channel, and read by polling the channel for updates.

Above the OS node, applications are run. In Barrelfish, a running application is

[2]Barrelfish ports may also use inter-processor interrupts for low-latency communication directly between CPU drivers, and cache coherency for communication directly in the hardware. However, the user/kernel split and general structure is the same.

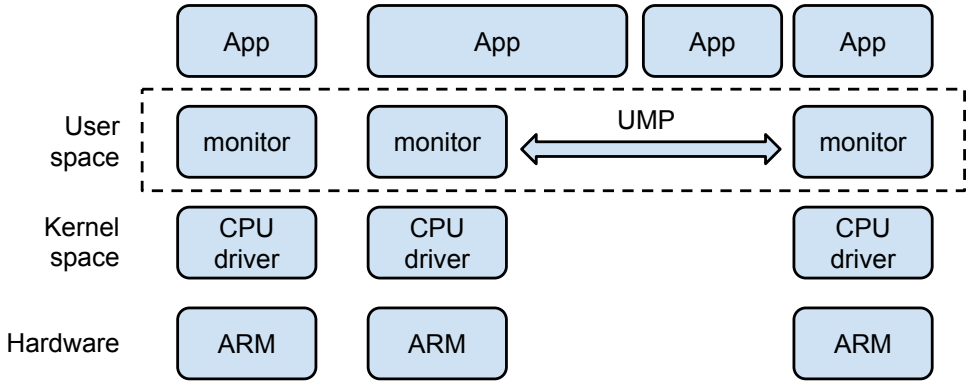

Figure 2.5: The structure of the Barrelfish operating system on ARM. Adapted from [BBD+09].

represented as a set of dispatcher objects. There is one core-local dispatcher for each core the process is running on. The dispatcher is the unit of scheduling in the CPU driver. Each dispatcher then manages its own threads, using a Barrelfish-specific thread library. In theory, threads could be migrated across dispatchers, but this is not currently supported in Barrelfish. The relationship between applications, dispatchers and threads in Barrelfish is illustrated in Figure 2.6.

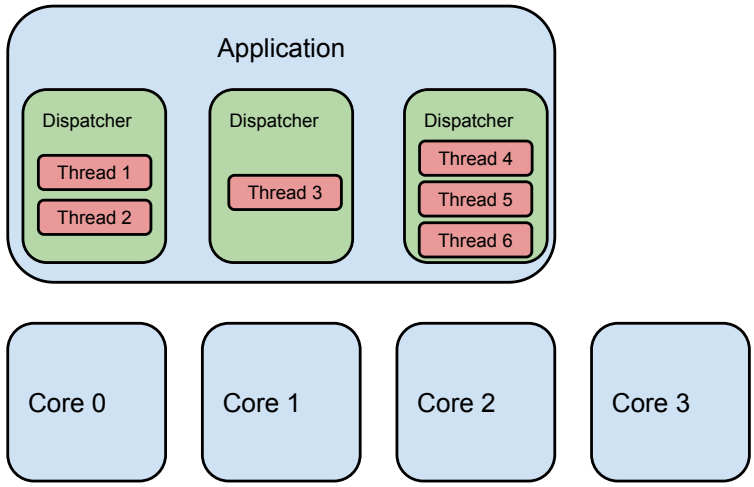

Figure 2.6: The representation of running applications in Barrelfish.

In the x86 and ARM ports, when Barrelfish is booted only a single core should

be running initially. This core is called the Boot-Strap Processor (BSP). When subsequent cores are started these are called application (app)-cores. Similarly, the CPU driver running on the BSP is called a BSP kernel, and a CPU driver running on an app-core is called an app kernel.

Much of the traditional OS functionality, such as memory management, device drivers and file system access, is implemented by applications. In this dissertation, we will refer to applications implementing traditional OS functionality or which are specially treated by Barrelfish as *system applications*. Applications may offer their services to other applications by exporting a set of remote procedures. Such a collection of remotely available procedures is called an *interface*. Note that one application may implement multiple interfaces, each containing one specific set of procedures. Throughout the text, application names will be printed in *italics* and interface names will be printed in `fixed-face`. An interface exported by a running executable will also be referred to as a service. Communication between applications is performed by one application creating a binding to a service, and calling the procedures in its interface. Table 2.4 lists important system applications in Barrelfish; Table 2.5 lists important interfaces, as well as which application exports them. More detailed documentation may be found on the Barrelfish homepage [Bar].

The Capability System As a protection mechanism, Barrelfish uses capabilities similar to what is found in the formally verified microkernel seL4 [KEH+09] and the Capsicum application compartmentalization system in FreeBSD [WALK10]. A capability is a reference to a resource, and a set of operations which may be performed on it. In capability systems, the resources an application is allowed to access is restricted through what capabilities are handed to it. It is important that the capabilities are unforgeable and tamper proof to maintain security [SSF99]. Barrelfish implements the capability system by storing the capabilities available to each dispatcher in a separate capability tree in kernel space. Each node of this tree is called a Capability Node (CNode). The capabilities available to a dispatcher is called the capability space of the dispatcher. In user space, the dispatcher only contains references to capabilities in the kernel-space tree. The structure of the capability tree is statically specified, so the capability references are statically created.

As an example, with a capability to a region of RAM and a capability to a page table, an executable could ask the operating system to map the RAM capability into its virtual memory space. The process is illustrated in Figure 2.7. The dispatcher contains a reference to a level two page table, which is an entry in the level one page table, named `pte_capref`. Additionally, the dispatcher has a reference to a frame, i.e. a page-sized region of free physical RAM, named `frame_capref`. When the map is requested, the kernel will verify that the capability references are valid. It does this by using bits of the capability reference as an index into the current capability

Table 2.4: System applications in Barrelfish.

System application	Description
init	The first application started by the BSP kernel. Responsible for setting up *mem_serv* and monitor.
mem_serv	Manages memory allocation, exported through the `mem` interface.
ramfsd	Creates a RAM file system containing the applications uploaded by the bootloader. Exported through the `trivfs` interface.
skb	The system knowledge base, used for querying architecture information. Also contains a name service, mapping interfaces to applications.
spawnd	Can start programs on its own core by exporting the `spawn` interface. It is also responsible for starting app-cores by sending core boot requests to its local monitor.
startd	Starts a specified set of user programs by invoking procedures in `spawn`.
serial	A serial driver, which manages access to serial interfaces by user applications.
angler	Manages user sessions. Amongst its uses is allocating a serial device to a user session.
fish	The shell used in Barrelfish.

node, starting with the root capability node of the dispatcher. In the figure, the colouring scheme indicates which bits are used as an index in the different capability nodes. For instance, the red-coloured root capability node has eight entries, so the red-coloured first three bits of the capability reference is used to index this capability node. This may also be seen from the colours: the frame root CNode is coloured red, and so are the first three bits of both the `pte_capref` and the `frame_capref`. If the lookups succeed, the kernel has found two valid objects with which to perform the operation, and will therefore complete the mapping request.

There are some limitations with the capability system implementation in Barrelfish. First, capabilities may only refer to regions of RAM with a power-of-two size. Second, although capabilities to memory may be split into smaller capabilities, capabilities to smaller, adjacent memory regions may not be merged into larger capabilities.

Boot Sequence When the BSP kernel is started, it initializes local peripherals and a set of capabilities to peripherals and memory regions. The capabilities are entered into the capability space of *init*, which is subsequently started. *Init* is also

Table 2.5: Some important interfaces in Barrelfish.

Interface name	Description	Exporting executable
`mem`	Memory allocation interface normally used by applications	*mem_serv*
`monitor`	Asynchronous interface between applications and monitor	monitor
`monitor_blocking`	Synchronous interface between applications and monitor	monitor
`intermon`	Procedures used to coordinate different monitors.	monitor
`monitor_mem`	Interface exported by the monitor on the BSP-core, used by monitors on app-cores to request memory allocation.	monitor
`trivfs`	A simple file system interface, with support for opening, reading, and writing files	*ramfsd*
`octopus`	A record service, used among others to implement a name service	*skb*
`spawn`	An interface to spawn, kill, wait for, and monitor applications.	*spawnd*

passed a bootinfo structure as a program argument. This structure contains an array of memory region descriptions. Each array entry contains a base address, the binary logarithm of the size of the region, and a type field indicating what the region contains. The most common region types are *RegionType_Empty*, which denotes an area of free RAM, and *RegionType_Module*, which denotes an area containing an executable loaded by the bootloader.

Init proceeds by spawning *mem_serv* and monitor. The capabilities to free RAM are transferred to *mem_serv*, and the other capabilities are transferred to monitor. The bootinfo structure is further passed along to both *mem_serv* and monitor as a program argument. Finally, *init* sets up a `monitor`-connection for *mem_serv* and a `mem`-connection for monitor, before it terminates.

When *mem_serv* is started, the bootinfo list is used to initialize its allocator. It iterates through the region list in the bootinfo looking for regions of type *RegionType_Empty*. For each such region, *mem_serv* will expect a capability to be present in a designated capability node. This capability, described by the base and size field in the bootinfo memory region entry, can then be added to the allocator.

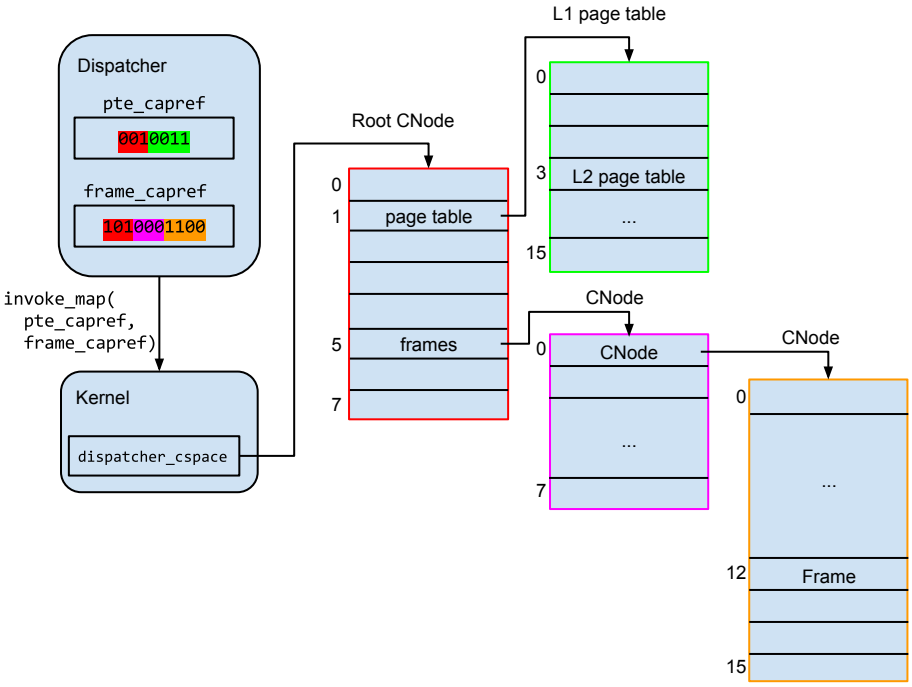

Figure 2.7: An example demonstrating how user-space capability references map to kernel-space capabilities when executing a virtual address space map request.

The monitor first starts a pre-defined set of applications, specifically *ramfsd* and *skb*. Next, it iterates through the bootinfo memory region structure looking for regions of type *RegionType_Module*. Each module which is marked as a boot-time module will then be started. This will typically include the applications *spawnd* and *startd*. Finally, the monitor enters a loop waiting for requests to arrive from applications.

Spawnd is started with an argument stating which cores should be booted. For each such core, *spawnd* sends a boot-request message to its local monitor to start the boot procedure. The initialization on these cores begins with app kernel initialization, followed by starting the monitor. The monitor will spawn a *spawnd* dispatcher on its own core, before announcing to the BSP monitor that its initialization is complete. When receiving these messages, the BSP monitor will set up the communication between the newly booted monitor and any other monitor which has previously started.

Startd will first wait for all *spawnd* dispatchers to be initialized. Next, it starts a set

of applications flagged by the bootloader. For example, to launch the shell application the bootloader would specify that *serial*, *angler* and *fish* should be started.

Figure 2.8 illustrates the started-by relationship between running executables with a tree, where each node represents an executable and each child was started by its parent.

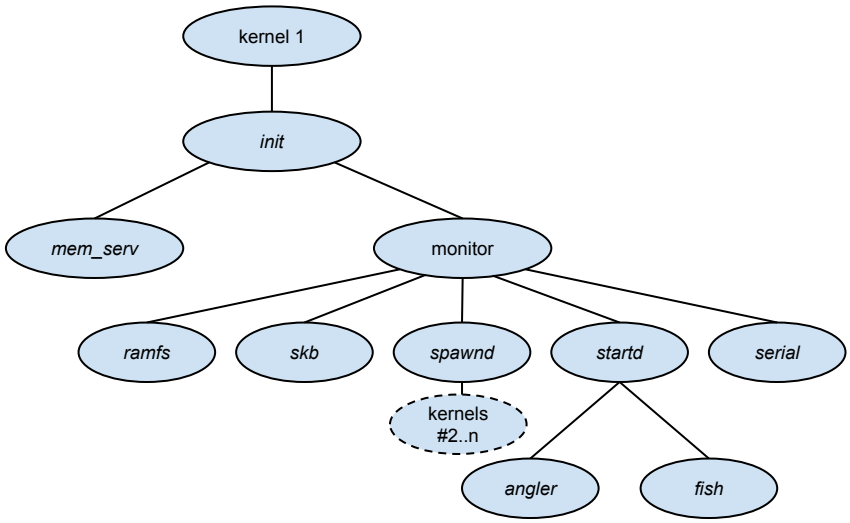

Figure 2.8: A visualization of the spawned-by relationship between executables in Barrelfish. Adapted from [BS13].

Barrelfish Library All executables running on Barrelfish, apart from the CPU driver, will be linked to a special library called *libbarrelfish*. This library implements most of the code specific to Barrelfish, such as capability management, the user-space thread library, and interaction between applications and the system applications. This library also contains the entry point of all executables, which enables initializing the library state before the main function of the executable is called. During initialization, Barrelfish distinguishes between *init-domains* and *non-init-domains*. Init-domains are executables launched before the local OS-node has been fully initialized. The library can therefore not perform any initialization which relies on the presence of the monitor. Non-init-domains are executables launched after the OS-node, which means that a complete initialization can be done using the local monitor.

Memory Allocation For all applications except the CPU driver, memory allocation is initially done using a pre-allocated memory area. The memory area is passed to the executable at a fixed location in its capability space. This enables memory allocation during *libbarrelfish* initialization. Eventually, different executables switch

to different strategies for issuing allocation requests. The differences are summarized in Table 2.6, which divides executables into three categories based on their final allocation request strategies. The kernels, *init* and *mem_serv* are treated specially, and use their own allocators. All other executables request memory allocation through a service. Typically, this is done by setting up a connection to the `mem` service and sending requests there. App-core monitors are an exception, as they instead set up and use a connection to the `monitor_mem` service.

Table 2.6: An overview of how different executables in Barrelfish issue memory allocation requests, once initialization is complete.

Executable	**Strategy**
Kernels, *init* and *mem_serv*	Use their own allocator.
Non-init-domains and BSP monitor	Uses a connection to `mem`.
App-core monitors	Uses a connection to `monitor_mem`.

2.2.3 The Port to SHMAC

The port of Barrelfish to SHMAC was initiated based on the observation that the design goals of Barrelfish were suited for the diverse multicore architectures to be studied in the SHMAC project. Several other research operating systems which are similar to Barrelfish, such as Helios [NHM+09], Corey [BWCC+08] and Factored Operating System (fos) [WA09], could potentially also have been selected. However, Barrelfish was deemed to be the most suited for investigating the research questions in the SHMAC project. Corey and fos focus primarily on many-core systems, while the focus of the SHMAC project is on heterogeneity in many-core systems. Helios also focuses on heterogeneity, but its primary goal is simple application deployment and tuning on heterogeneous platforms rather than creating an adaptive operating system. Another important factor was that Barrelfish is open source, whereas Helios is not.

At the beginning of this dissertation, the SHMAC port supported booting Barrelfish and running a multiprogrammed user workload on a single core [BS13]. A conclusion from the work on this port was that Barrelfish was not suited for a memory-constrained execution environment. Specifically, the RealView platform could not support executing Barrelfish on multiple cores without a rewrite of core components of the OS. This experience, along with that of other projects which encountered memory problems, was the motivation for acquiring the more powerful Versatile Express platform.

The SHMAC port of Barrelfish shares the general characteristics of Barrelfish described in the previous section. One SHMAC-specific utility is the *shmacfish* bootloader generator. *Shmacfish* reads a configuration file, which specifies the available

address space and which Barrelfish executables should be uploaded. Then, it produces a module segment containing all the modules as well as a multiboot-compliant metadata header [FB13]. It also produces a set of OS boot parameters, denoting where in memory the multiboot module specification is situated and the number of modules in it. Finally, it produces a script, which may be run on the SHMAC host to upload the kernel binary to address zero, the kernel boot parameters after the kernel, and the module segment to the end of the address space before starting the SHMAC processor.

2.3 Energy Efficiency Evaluation Strategies

Research on energy efficiency is no novel endeavour; however, the recent recognition of power as a cross-cutting design limitation [DBSDBM13] has spawned an increasing number of research projects on the energy efficiency of computer systems. This section will describe approaches to energy efficiency evaluation taken in different kinds of studies.

2.3.1 Measurements

The most straight-forward solution for evaluating the energy efficiency of a system is to simply measure the power drawn from its power supply. Such measurements can be made by connecting a power meter between the system under examination and the wall outlet; this monitors complete system power, which may be used to compare complete systems against each other [RSR+07]. It may also be possible to derive the processor power by first measuring system power when the processor is halted, and subsequently subtract this from measurements gathered during tests to compare processor power draw specifically [BMS13]. If independent power supplies are present for different components, the power meters may potentially be directly connected there instead for more accurate data [ECX+11].

2.3.2 Performance Counters

Using measurement tools to obtain power consumption is a good strategy for benchmarking system performance. However, it is also useful for system software to be able to gauge the current power consumption in order to implement power-aware system control algorithms. Power measurement tools are not suitable for this purpose, as they will not be able to feed the power data back to the system in a timely manner. Even if this were possible, having a power meter attached for proper system operation would not be a feasible requirement in many situations.

An additional shortcoming of measurement tools, is the granularity with which they can attribute power dissipation to different subsystem components. The granularity

of power meter measurements is limited to the different power supplies, and will typically not be able to provide particularly fine-grained measurements. For instance, it may not be possible to separate the power consumption of different cores on a Chip Multi-Processor (CMP), much less determine which part of the microarchitecture is responsible for the power consumed.

To tackle these issues, researchers have investigated methods for correlating activity in the processor with power consumption. Modern processors often expose a set of registers which count certain hardware events, such as cache misses and pipeline stalls. By correlating these values with measured system power, one can obtain a model which enables estimating the processor power consumption from the performance counters visible to the system software. Singh *et al.* [SBM09] explore this possibility, and demonstrate an implementation of a scheduling algorithm which uses the created model to make sure that the system executes within a set power envelope. Bertran *et al.* [BGM+10] stress the importance of creating power models which are decomposable, i.e. the total power can be attributed precisely to subcomponents, and responsive, i.e. fluctuations in power consumption are accurately captured. They present a generic scheme for creating such models. Bircher and John [BJ07] illustrate how such a scheme may even be used to characterize the power consumption of systems external to the processor, such as graphics cards, hard-drives and network devices.

2.3.3 Architectural Simulators

During research or in an initial phase of a design project, the system under examination might not yet exist. In such cases, measurements can not be used for energy evaluation. As the power consumption of a system is intrinsically related to activity, one may turn to simulators in order to obtain the activity factors.

One common simulator approach is to implement the behaviour of a system architecture in software. In this dissertation, we will refer to such simulators as *architectural simulators*. The software may model the architecture at arbitrary levels of detail, typically ranging between functional equivalence and a truthful representation of the Register Transfer Level (RTL) behaviour of the system. Running the software model under some stimuli will trigger activity events, observable and loggable by the simulator. An example of such a simulator is gem5 [BBB+11], which provides behavioural models for a large number of cores for different ISAs as well as a variety of memory system implementations. The gem5 simulator is a result from merging the older processor core models in the M5 simulator [BDH+06] and the memory system models in the GEMS simulator [MSB+05]. A different, more recent simulator is Sniper [CHE11]. Sniper focuses on scalable simulator performance by using multiple cores, and allows trading simulator speed for accuracy. It currently only supports cores implementing the x86 and x86-64 ISA.

Having obtained activity events, power estimates can be obtained by using software power estimation tools. One such tool is Wattch [BTM00], which uses power models for commonly used hardware structures to estimate processor power consumption given some switching activity. More recently, McPAT [LAS+13] has been developed to succeed Wattch, with more comprehensive power models updated to use the current state of process technology. To estimate the power consumption of on-chip RAM, one can use the tool CACTI [WJ96]; estimates for off-chip RAM power consumption can be obtained using DRAMSim2 [DRA14].

An example of the use of this kind of method can be seen in the work by Govindaraju *et al.* [GHS11]. In the study, a new kind of reconfigurable hardware unit developed for energy efficiency is investigated. The power data is obtained by using the GEMS simulator extended with the new hardware unit to gather activity, and Wattch-based power models integrated in the simulator to calculate power consumption. In a different study, Kumar *et al.* [KFJ+03] combine the simulator SMTSIM and Wattch to investigate the potential energy efficiency improvements achievable by combining different versions of an existing processor core on a single chip. In the study presenting McPat [LAS+13], a Design Space Exploration (DSE) study for investigating the optimal core clustering count considering energy, delay and area is conducted using the M5 simulator for activity data and McPAT for calculating energy consumption.

2.3.4 Circuit Simulators

Instead of developing models of a proposed architecture in software, one may create synthesizable HDL implementations of the architecture. Such models may be synthesized into a gate-level netlist, describing how the hardware implementation would translate into combinations of gates in a given cell library, and potentially also placed and routed to describe precisely the wiring and layout of the gates. As cell libraries specify the power consumption of the different gates, the netlist makes it possible to calculate the power consumption of the entire circuit under the influence of some test stimuli. This is done by employing a circuit simulator such as ModelSim [Men14] or VCS [Syn14e] to gather the switching activity of the different gates, and using a power estimation tool like Synopsys PrimeTime [Syn14d] to combine the activity with per-gate and per-wire power consumption estimates to accurately calculate the power consumption of the entire circuit. With a netlist and layout sufficiently similar to a final silicon implementation, it should in theory be possible to calculate the exact power consumption to within the error margins of the cell library. Considering the errors present also in power meters, the method might rival measurements in accuracy. In contrast to measurements, however, this method is slow due to high computational demands of gate-level simulation where the size and complexity of the design is substantial.

Slow simulation speed notwithstanding, several research projects have used this method to estimate the efficiency of proposed hardware enhancements. A project by Venkatesh *et al.* investigates extending processor cores with energy efficient accelerators called conservation cores, tailor-made for typical system workloads [VSG+10]. They evaluate the efficiency of their idea by generating Register Transfer (RT)-language models of their accelerators, which are subsequently synthesized, placed and routed. To make simulation speeds manageable, they use a software simulator to sample the register state at different intervals, and run the circuit simulator for a limited number of cycles starting from this register state. The power is calculated using Synopsys PrimeTime. To get power estimates for the other parts of the system, they model the processor core power consumption as a constant mW/MHz, and use CACTI for the memories. A similar evaluation strategy is used in the project by Park *et al.* [PPPM12], where a configurable SIMD unit called *Libra* is proposed; and in the project by Gupta *et al.* [GFA+11] which investigates the efficiency of a proposed coarse-grained configurable accelerator unit called BERET.

2.4 Hardware Regression Models

When investigating how energy efficiency estimation using the SHMAC infrastructure may be supported, all the techniques described in Section 2.3 were found to be wanting. The measurement and performance counter techniques described in Sections 2.3.1 and 2.3.2 require measuring power consumption, which is not possible for the SHMAC project as no system exists to be measured. The architectural and circuit simulators described in Sections 2.3.3 and 2.3.4 lift this restriction, but they do not align with the SHMAC project goal of using FPGA prototyping to speed up evaluation.

Ideally, we would like to gather activity data at hardware speeds and estimate the resulting power consumption with the accuracy of measurements without being dependent on the presence of a physical implementation of the design being tested. In order to close in on this ambition, the following observations were made of the capabilities of the SHMAC infrastructure and the known energy efficiency estimation techniques:

1. The SHMAC infrastructure implemented on an FPGA offers RTL activity at hardware speeds.

2. Circuit-level simulators may be used to accurately estimate the power consumption of an RTL-design under some stimuli, as described in Section 2.3.4.

3. With performance counter readings and corresponding power estimates attained from initial benchmark runs, it is possible to create performance-counter-

based power models predicting power consumption in general as described in Section 2.3.2.

4. The SHMAC infrastructure may be extended with any kind of performance counter.

From these observations, we got the key insight that the performance counter method may be used by having circuit-level simulators take the place of measurement tools. Since the performance counter method only requires power measurements during model construction, the dependency on the simulator is limited to design time. Once the model has been created, any performance counters may be implemented on the SHMAC infrastructure and gathered at FPGA execution speed. Presumably, this would fulfil our goals of high accuracy and estimation speed without the need to measure a physical system.

The above reasoning lead us to envision the method illustrated in Figure 2.9 as a potential candidate for enabling energy efficiency estimation with the SHMAC infrastructure. The three leftmost stages implement the circuit-level simulator estimation strategy described in Section 2.3.4. These stages are used to get the power data required to implement the performance counter strategy described in Section 2.3.2, where regression modelling correlates hardware activity with power consumption. The resulting regression model may either be implemented as an expression in hardware, or implemented in software by exposing the selected performance counters.

Further literature studies revealed that the proposed method was considered a sound technique for gathering energy efficiency estimates from a design running on an FPGA. The studies found also highlighted different design choices when using this technique, as well as potential challenges. The studies are summarized in Table 2.7.

Table 2.7: The studies found employing similar techniques to that of Figure 2.9 to evaluate the energy efficiency of HDL designs using FPGA execution.

Authors	**Keyword**	**Summary**
Coburn *et al.* [CRR05]	Power Emulation	Model the RT-level building blocks of a design. High precision, high overhead.
Bhattacharjee *et al.* [BCM08]	Full-CMP	Model the design at an architectural level using event counters. Lower transient precision, low overhead.
Sunwoo *et al.* [SWPC10]	PrEsto	Semi-automate modelling by specifying regression model form in terms of per-cycle signal values. Presumed higher transient precision, medium overhead.

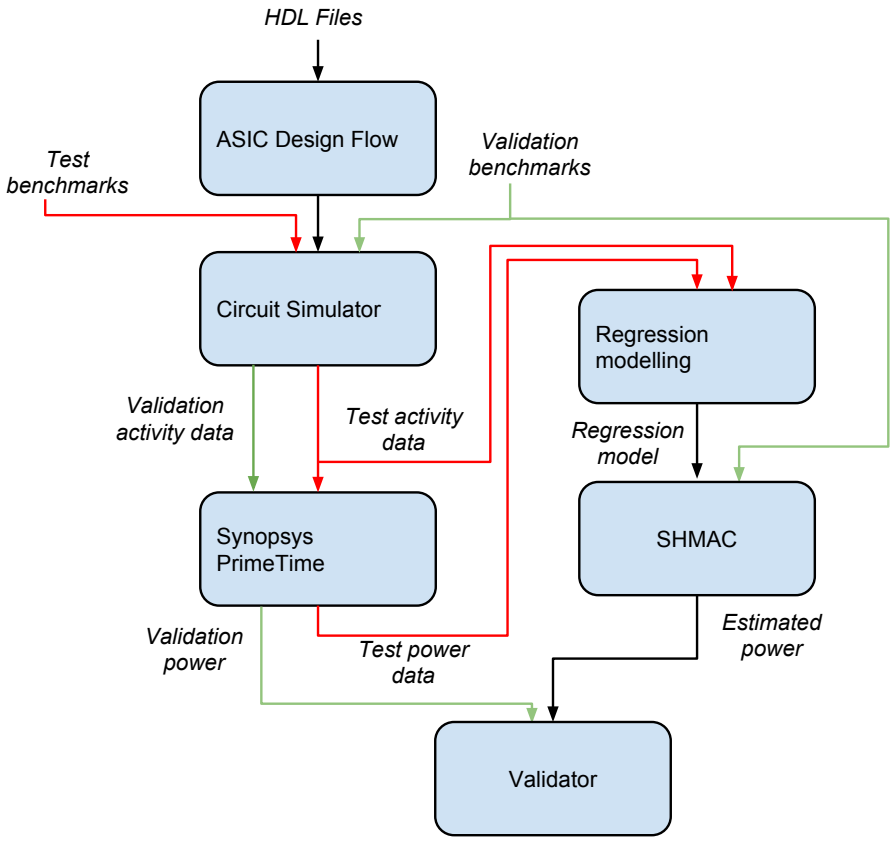

Figure 2.9: Possible method for combining different power estimation techniques to enhance the SHMAC platform with power consumption estimation capabilities.

The oldest study found which considers FPGA-based estimation of the power consumption of HDL designs was conducted by Coburn *et al*, in which they name this technique *power emulation* [CRR05]. Their approach to modelling was to include simple models for components at high granularity. For each RTL building block, such as multiplexers and adders, they associate a fixed power draw from toggle activity in a single bit. Thus, the power consumption of an N-bit component was modelled as $\sum_{i=1}^{N} Coeff_i \cdot Activity(bit_i)$. The hardware overhead per model is small, as it may be implemented with and-gates and an adder. However, naïvely including such a model for every RTL component in the system lead to an average increase in system size by 18.2×. Their work discusses techniques which reduce this resource requirement, leading to an average area overhead of 3.1×.

The high overhead of the technique proposed by Coburn *et al.* limited its applicability

to specific hardware functions with the largest being an MPEG4 decoder unit. The next study found, conducted by Bhattacharjee *et al.* [BCM08], proposed overcoming the area overhead limitations by focusing on power models for higher-level architectural components based on similarly high-level event counters. Their system under test was a 2-core Leon cache-coherent CMP. They first incorporated 36 event counters into this system, counting events such as cache misses and pipeline stalls. The selection of counters was based on intuition and experience with the architecture. They subsequently followed the same flow as depicted in Figure 2.9. The ASIC design flow stage was limited to synthesis, as including placement and routing details allegedly lead to exponential increases of simulation time with modest increases in accuracy. The resulting model was found to be within 10% of the PrimeTime estimates on average. To keep the area overhead to a minimum, they opted for incorporating the resulting regression model in the operating system kernel, thus only requiring the addition of the event counters themselves to the system.

Selecting the event counters before gathering the energy estimation training data might be difficult, and yield suboptimal regression models if the selection is poor. Additionally, as event counters represent hardware behaviour averaged over a period they are not particularly suited for modelling transient and peak-power consumption estimation [BCM08]. The latest study found, conducted by Sunwoo *et al.*, proposes a method named PrEsto in which these points are addressed [SWPC10]. The input to the method is activity data for a selection of hardware component inputs and outputs, as well as corresponding power data for the hardware component being modelled. The remainder of the modelling is automated by performing the regression modelling with a pre-defined formula format, in which terms may be a multiplication of at most two binary signals and the Hamming distance of one bus signal[3]. The large number of terms in the resulting regression model is subsequently sorted based on their significance in the model. The most significant terms are selected to create the final model. Thus, the selection of model terms is automated. Since signal values are used directly, instead of being summarized in event counter values, there is also an increased opportunity for precise prediction of transient power consumption.

The PrEsto method is tailored for use with the researchers' own FAST FPGA accelerated simulator [CSK+07], which implements only parts of the simulator on the FPGA instead of emulating the entire system there. Consequently, one target cycle might take several host cycles. For instance, the PrEsto power models demonstrated in their study take seven host cycles at 133 MHz to complete. Whether the models would be suitable as enhancements for a complete FPGA prototype, or whether they would be too costly either in area or computation time, has not been studied.

[3]The Hamming distance between two binary strings is the number of bits which differ between them. The Hamming distance of a bus is the hamming distances of bus values in successive cycles.

Chapter 3
Energy Efficiency Estimation Framework Design

This chapter will introduce the design of an energy estimation framework which uses the modelling method presented in Section 2.4. The framework is split in two parts:

1. Energy models which calculate an energy consumption estimate based on RTL activity. It is the construction of such models that is explained in Section 2.4.
2. The infrastructure which must be added to the SHMAC infrastructure to support using the energy models to estimate energy consumption at run-time.

Section 3.1 will first list what functionality and attributes we would want in an energy estimation framework. From this list, we will derive our design goals. Section 3.2 will next present the design of an infrastructure which meets the functional goals, assuming the existence of energy models for the hardware components in the SHMAC infrastructure. Finally, Section 3.3 will expand on Section 2.4 and explain what different modelling methods are employed when creating energy models for different hardware components.

3.1 Design Goals

The SHMAC platform is designed to be able to quickly run experiments investigating new hardware or software solutions for different heterogeneous architectures. The ultimate goal of an energy efficiency estimation framework is enhancing this platform with support for somehow analysing the energy efficiency of the hardware or software solution being studied when running the experiments. This section will discuss how such an enhancement should work by presenting both functional goals, describing which capabilities should desirably be present in such a framework, and non-functional goals, pertaining to the desired characteristics and quality of the framework components.

Functional Goals

1. The arguably most important framework functionality is the ability to report the energy efficiency to the host system, for storage and subsequent analysis of experiment results.

2. It is desirable to be able to separate the energy efficiency of individual hardware components. For example, knowing that a significant fraction of the energy consumption originates from the router between cores and scratchpad tiles may motivate the development of an algorithm or system improvement in which more local processing is employed and the communication requirements are reduced.

3. To enable research on software which uses the current energy efficiency as feedback to steer its operation in a more energy-efficient manner, it is desirable that the energy efficiency estimates are not only available to the host system, but also to software running on the SHMAC platform.

Non-functional Goals

1. The reported energy efficiency should be as accurate as possible.

2. The energy efficiency metric should preferably be reported per time, and not merely as an estimate of average efficiency over the entire course of an experiment. Having a per-time breakdown of energy consumption enables more accurate analysis of the hardware or software being studied, and may help identify the main sources of energy consumption in a hardware or software design. Additionally, the transient consumption may be interesting in itself, for instance if the research in question considers thermal constraints in which peak power is a limiting constraint.

3. It is also desirable with as fine-grained a report per time as possible. This goal implies that the speed of operation of the framework should be as high as possible.

4. The framework should preferably have little implementation overhead, so as not to constrain the design space available for investigation with the SHMAC platform. Specifically, the framework should not impede the clock frequency of the SHMAC platform as this would slow down simulation runs. and the resource requirements for the infrastructure should be minimal to allow for the instantiation of as large SHMAC configurations as possible.

5. The granularity with which the energy efficiency is reported should be customizable, to reduce communication and storage demands when only coarse-grained reports are required.

6. The coverage of modules contributing to the reported energy efficiency should be high. The energy consumption of a module should not be ignored, as this would increase the possibility of erroneous conclusions.

7. The framework should be as user-friendly as possible. This matters for both developers of the platform, who benefit from simple model creation and integration to be as simple as possible, and researchers using the platform, who benefit from simple acquisition of energy efficiency results.

3.2 Infrastructure Design

In this section, the design of the infrastructure of an energy efficiency framework for the SHMAC project will be presented. The infrastructure design will assume the existence of energy models relating hardware activity to energy consumption, and demonstrate how such models may be employed to best fulfil the design goals of Section 3.1. The design will also illuminate how the exact character of the energy models impacts the potential quality of the infrastructure, and this insight will yield efficiency guidelines for the subsequent design of the models.

The infrastructure is built to use energy consumption as the energy efficiency metric, instead of for instance power or raw performance counter values, for reasons of user-friendliness and implementation efficiency which will be discussed in Section 3.2.2. Implementing the functional goals listed in Section 3.1 therefore involves transferring energy data to the host, attributing energy consumption to specific modules, and exporting energy consumption data to SHMAC software. To construct a solution to these goals, the infrastructure uses a design of three layers of components: energy monitor modules, energy report units, and a scan chain. The composition of these component layers is depicted in Figure 3.1, giving an overview of the complete infrastructure design. The energy monitor module implements the energy models; the energy report units aggregate data from several models and define the granularity at which energy data is available; and the scan chain transfers data to the host. Section 3.2.1, Section 3.2.2 and Section 3.2.3 will discuss the design of each of the layers, how it relates to the one below it and what constraints it imposes. The design of the interface to host will then be described in Section 3.2.4, and finally Section 3.2.5 describes the SHMAC system software interface.

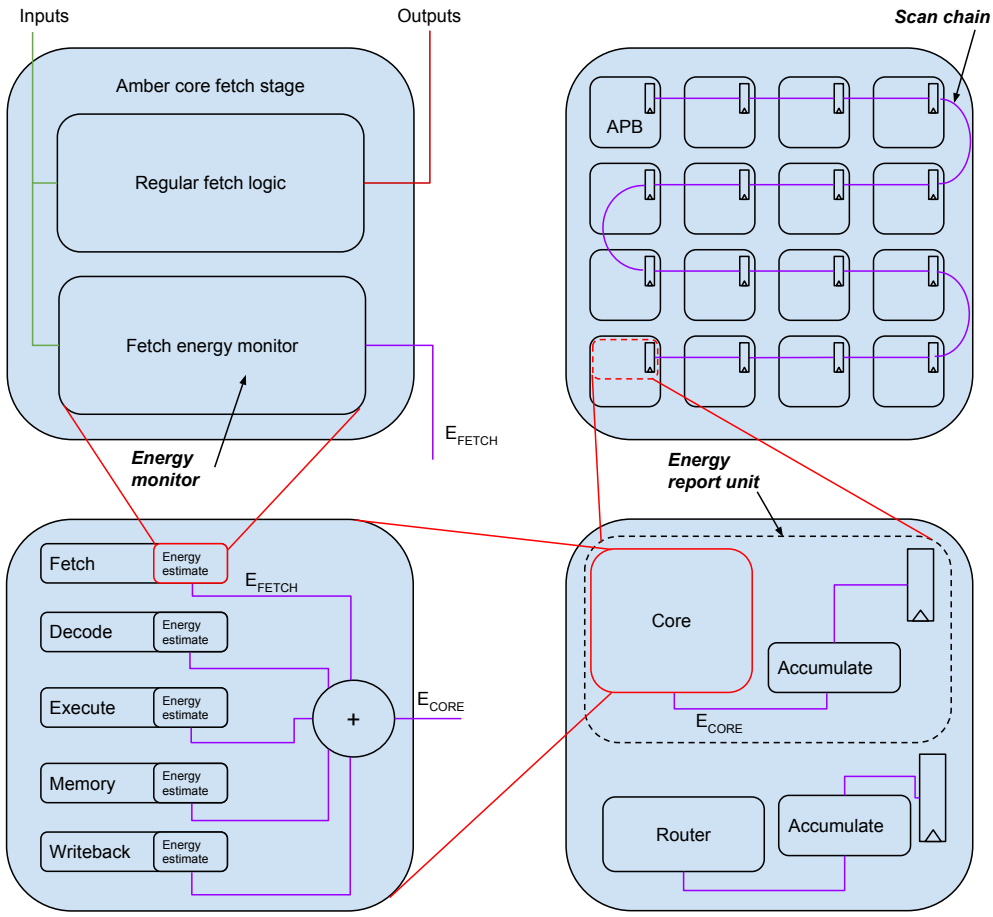

Figure 3.1: An overview of the energy efficiency estimation infrastructure. The main components, namely monitors, report units and the scan chain, are explained in detail in separate sections.

3.2.1 Scan Chain

The scan chain is the primary engine in the transfer of energy data from the SHMAC platform to the host. The chain connects energy report units, whose reported energy values may be shifted one by one into the APB tile for transfer to the host. Using a scan chain instead of point-to-point links between the energy report units and the APB tile leads to lower routing resource overheads on the FPGA, in conformation to the non-functional goals. It is also a more flexible solution, since extra energy report units may seamlessly be inserted anywhere along the chain. The maximum

sample rate is limited by how fast the scan chain can supply values to the APB tile, with optimum value reached if the APB is saturated. Since scan chains typically may provide a new value every cycle, we would expect this to be attainable from an implementation. Since the APB bandwidth is one transfer every two cycles, a scan chain which may transfer one sample per cycle allows saturating the APB and therefore optimally adhering to non-functional design goal 3.

The scan chain is designed to support sampled gathering of energy data, at the discretion of the host. It would not be possible to base the design on the ability to stream out one sample each cycle for several reasons: the APB is not sufficiently fast; using the link to transfer energy data at every chance would hinder other common uses such as terminal I/O; and the APB tile is a slave unit, so data transfer will necessarily be reactive anyway.

Gathering data samples is supported through two mechanisms:

1. A sample signal is used to signal a snapshot of the energy consumption registered so far in a report unit.
2. A shift signal is used to shift energy values stored in the scan chain registers, when a new sample is needed for transfer on the APB.

The APB is 32 bits wide, which makes 32 bits a natural maximum energy sample width for an implementation. However, due to the flexibility of a scan chain it is possible to use a higher bit width simply by using several 32-bit registers in the report unit and connect them all to the chain. This possibility should only be used if it is necessary for sufficient accuracy, since it will have an adverse impact on throughput. A doubled sample width will double the amount of cycles it takes to transfer all samples out to the host, and thereby halving the maximum theoretical sample rate of the infrastructure.

Using the scan chain, the host will be able to attribute samples to sources if it knows two things: the bit width of each sample, and the order of energy report units in the scan chain. Support for dynamically determining this is not part of the current design, thus favouring reduced complexity and overhead over user-friendliness for those interfacing the infrastructure directly.

3.2.2 Energy Report Unit

The energy report unit is used to define the granularity at which energy consumption may be attributed to hardware modules. It is designed to work by summing the contributions from one or more energy monitors, and accumulating the energy

consumed until a sample signal is received from the scan chain. Upon the reception of this signal, the energy accumulated so far is stored in the scan chain register, and the counter is reset to zero. Resetting the values to zero leads to new estimates being reported for each time interval, which fits non-functional goal 2 on the ability to report energy efficiency over time.

The report units may be situated at any location the researcher deems appropriate, thereby fulfilling non-functional design goal 5 on customizable granularity of energy consumption attribution. Since the report units are all connected in a scan chain, a new report unit U can always be inserted between two existing units U1 and U2 by connecting the output of U1 to the input of U and the output of U to the input of U2 at the cost of reduced maximum sample rate because of the extra data transfer requirements.

The design of the energy report unit accommodates using either energy-per-event-based models, as used by Bhattacharjee *et al.* [BCM08], or models calculating energy based on per-cycle signal values, as in the design of PrEsto by Sunwoo *et al.* [SWPC10], in the energy monitors. Supporting both kinds of models gives modelling flexibility, which is good since lack of modelling flexibility may hurt non-functional goals 1, 6 and 7. The PrEsto models are formulae based on transient signal state, and are as such trivially supported by the energy report unit. The energy-per-event models are typically evaluated after longer periods of execution by multiplying the total count of occurrences of one event with the energy per event occurrence. However, the energy-per-event models may also be evaluated every cycle as the sum of energy values for all events which occurred this cycle. Thus, both models may be built to fit the expectations of the energy report unit.

The report unit design is what enforces the infrastructure to work with energy consumption, instead of just exposing event counters which it is up to software to use. Implementing the models in hardware leads to higher overhead than just using event counters, but is still a superior choice for the following reasons:

1. Shipping event counters off the SHMAC to the host system is likely to require a large amount of extra bandwidth. For example, in [BCM08] 36 event counters are used. If we conservatively assume that the average size of the event counters is 3 bits, then transmitting all these counter values would require 108 bits and therefore 4 APB cycles per model. Transmitting 32-bit energy values instead would permit a quadrupled maximum theoretical sampling rate from the infrastructure.

2. It is possible to limit the overhead from implementing energy models by using event-counter based models, since these are simple to implement. This is

demonstrated in Figure 3.2. Instead of a register counting the number of occurrences of each event, an implementation would require one vector and-gate per event and $log_2(n)$ adders. Thus, the modelling flexibility can be traded for implementation simplicity.

3. Finally, it is more practical for both host software and SHMAC software to make use of a completely calculated energy estimate instead of having to implement the formulae relating event counters to energy.

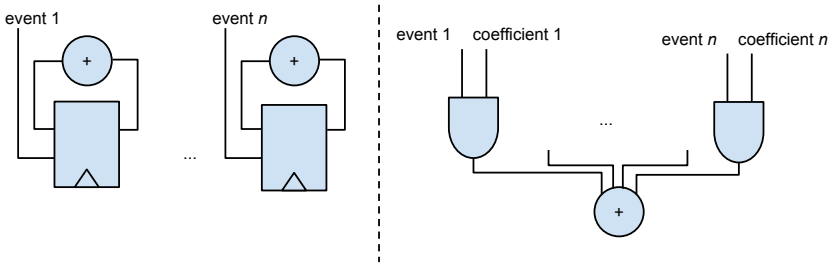

Figure 3.2: Implementing performance counters versus implementing per-cycle energy-per-event energy models.

The energy report unit design also enforces using energy consumption as an energy efficiency metric instead of power. Energy may be accumulated more easily than power, leading to simpler implementations which makes it preferable according to non-functional design goal 4. Power is more difficult to accumulate since it is a rate, and therefore the sum of power numbers have no inherent meaning. The most obvious summary of a sequence of power numbers is its average value, and calculating this requires division. Both divisor and dividend will vary since it is not known how much time will pass between each sample, which mandates the implementation of relatively expensive division circuits conflicting with non-functional design goal 4. Additionally, the latency of performing the division would add a minimum latency before scan chain operation could commence, reducing the maximum sample rate and thereby hurting non-functional design goal 3.

3.2.3 Energy Monitors

The role of the energy monitors is to implement the energy model for the SHMAC hardware module in which it is contained. This model is assumed to be dependent only on input, output and internal signals within the module, and possibly signals derived from these. The infrastructure requires that models calculate energy as an integer value with the necessary resolution determined during modelling. Apart from these restrictions, the monitors are black-boxes in the infrastructure to be determined through modelling. Modelling will also determine the required granularity based

on non-functional goals 1 and 6 on accuracy and coverage, implying where in the infrastructure monitors must be included.

As energy monitors and report units are decoupled, the modelling granularity may be varied independently from the report granularity. It is therefore possible to have high coverage and accuracy from high modelling granularity without incurring significant costs from high report granularity. The left hand side of Figure 3.1 exemplifies this situation. Although the modelling granularity is at the level of pipeline stages, the report granularity may be set to the level of the complete core by feeding a report unit with the sum of contributions from individual pipeline stage monitors.

3.2.4 Host Interface

The host interface to the energy estimation infrastructure extends the existing memory-mapped register interface to the SHMAC platform. Two new registers are included:

1. *SampleEnergy* is a write-only register. Writes will cause a new sample to be taken.
2. *NextSample* is a read-only register. Reads will return the next sample in the scan chain.

The *SampleEnergy* register could be removed if reads to *NextSample* issued a sample request once all sources had been read. However, using two separate signals will require a less complex infrastructure implementation as there is no need to keep track of which samples have been read and optionally asserting the sample signal. Explicit control of the sample signal also offers more flexibility to host software.

3.2.5 SHMAC Software Interface

The interface to the energy estimation infrastructure from SHMAC software is also based on memory-mapped registers, extending the existing interface for access to core peripherals. There is one set of registers per energy monitor whose data is exported to the SHMAC software. The interfaces do not depend on energy report units or the scan chain, to avoid dependence on or conflict with sampling activity by software running on the host. Which set of energy monitors to export to SHMAC software may therefore be decided independently of the host report granularity.

The registers available to software on the SHMAC are only meant to guide run-time system software decisions, and not logging of energy consumption which is better handled through the host interface. Consequently, there are two possible access patterns to the energy estimate registers:

1. The software makes periodic samples, with the intention of extracting the energy consumption since the last sample was taken. Varying the sample period allows the software to balance sample resolution with sampling overhead.

2. The software samples at strategic points, aperiodically spread out in time, with the intention of gauging the current energy or power consumption. What is to be considered current should ideally be specifiable by the software.

Supporting the first use-case of periodic sampling is straight-forward for an implementation, but supporting the second may require complex implementations depending on the exact semantics of the interface. For instance, reporting the energy consumed the past X cycles, with X configurable by the software, requires remembering the past X samples and managing a dynamically varying history size.

The proposed interface design attempts to strike a balance between semantic power and implementation simplicity. The interface consists of five registers per selected monitor, listed in Table 3.1. In addition to these per-monitor registers, a per-tile register reporting the emulated operating frequency of the tile is also included.

Table 3.1: Register interface per energy monitor exported to SHMAC software

Register	**Semantics**
Time Window	Maximum number of cycles in a period. If zero, the period is unlimited.
Current Period Energy	Register to which energy consumption is added each cycle.
Current Period Duration	Set when current period energy is read. Reports the number of cycles from which energy data was gathered when *Current Period Energy* was last read
Previous Period Energy	Set when *Current Period Energy* either overflows, or has accumulated energy for more cycles than *Time Window* allows.
Previous Period Duration	Set when *Previous Period Energy* is written. Reports the number of cycles of energy data accumulated in *Previous Period Energy*

The periodic access pattern is supported by periodically reading the *Current Period Energy* register when *Time Window* is set to zero. The *Current Period Duration* register supports getting precise information on the sample period of the last read value. It is possible to handle one overflow event by checking whether the *Current*

Period Duration is lower than expected, and if so read the *Previous Period Energy* register to get the value at which the energy counter overflowed. Multiple overflows will not be detectable. The strategy of reporting the accumulated value since last read resembles the energy counters available on modern Intel processors [Int14].

The aperiodic sampling access pattern is supported by setting the *Time Window* register to the desired definition of recency, and reading from the *Previous Period Energy* to get the last completed sample for this period. This strategy gives a number which in the average case is outdated by half the *Time Window* resolution. If more accuracy is required, the *Current Period Energy* and *Current Period Duration* may be read first to determine whether the *Previous Period Energy* value is too outdated. The strategy of reporting values for the last time frame resembles the power registers available on AMD processors [AMD13].

3.3 Modelling Methods

Section 3.2 described the infrastructure which enhances the SHMAC platform with energy efficiency estimation capabilities. However, the usefulness of such an infrastructure is no better than the usefulness of the data it emits. This section will present the methods which enable the creation of models relating hardware activity to energy, providing the fundament for a complete energy efficiency estimation framework.

For HDL components which are specially created for the SHMAC infrastructure, the regression modelling method introduced in Section 2.4 provides a structured approach to producing accurate models. This method will be used to design models for the Amber cores and routers in the SHMAC infrastructure, with method details presented in Section 3.3.1. However, for components generic in nature whose HDL-specific implementation is perceived as inconsequential, analytical modelling may be more efficient especially if existing work may be leveraged. Section 3.3.2 presents the modelling of the energy consumption resulting from on-chip RAM access, off-chip RAM access, and the APB tile, and explains why the regression modelling method is either unsuitable or at present unnecessary for these components.

3.3.1 Regression Modelling

This section will flesh out the description of the regression modelling method given in Section 2.4 with details specific to the environment in which the models will exist, i.e. the infrastructure design, and to the design goals for the framework.

Infrastructure Design Implications

1. The infrastructure enforces a hardware implementation of the models.

2. Since the infrastructure is based on reporting energy, the regression model must relate hardware activity to energy. This differs slightly from previous work using this method, where the resulting models have calculated power.

3. A result is expected from the energy monitors every cycle. Thus, the model should be a formula which it is possible and inexpensive to evaluate every cycle, which calculates the energy consumption from just one cycle. Specifically, there is no need to create new event counters as independent variables.

Design Goal Implications

1. As the models must be implemented in hardware, design goal 4 on implementation overhead must be respected. This necessitates making several trade-offs during modelling:

   a) Variable selection, term complexity and the number of terms to use present trade-offs between accuracy and implementation overhead.

   b) The modelling granularity trades off coverage for implementation overhead.

   For our purposes, it is desirable with a trade-off where models have sufficient accuracy and minimal implementation overhead. The modelling step of the method should therefore be executed by first creating a model with sufficient accuracy to establish feasibility, and subsequently optimize for implementation overhead.

2. A necessary assumption which will be made to satisfy the design goal on user-friendliness is that the energy model for a component will be independent of its location in the SHMAC configuration. It would be infeasible to create location-specific models for each possible SHMAC layout, and hardware design space exploration would be severely limited if researchers had to create models for each layout they wanted to experiment with the since the method is both time-consuming and manual. A consequence of assumed location-independence is that the place-and-route step in the ASIC design flow is obviated, leading to simpler and less time consuming modelling. The trade-off is reduced accuracy from missing place-and-route information.

3. To ensure trueness in the energy models, the regression modelling should be done using the HDL of the SHMAC *without* the energy estimation infrastructure added. The infrastructure is merely an addition to enhance the capabilities of the SHMAC as a platform for running experiments, and not a component the system being studied is supposed to include. Therefore, there must be two versions of the SHMAC HDL maintained: one with the infrastructure,

used to generate the FPGA bitfile, and one without it, used when creating the regression models.

3.3.2 Analytical Modelling

On-chip RAM On-chip RAM resources are used in the SHMAC infrastructure for the scratchpad memory and the processor core caches. As these are implemented in the SHMAC RTL code, it would be possible to use regression modelling for these components as well. However, if the SHMAC was realized in silicon it is likely that the RAM blocks used for cache and scratchpad memory would be custom modules selected from a memory library since RAM construction is more low-level than the RTL of an HDL specification. It is therefore uncertain how true a regression model of cache and scratchpad memory energy consumption based on the SHMAC HDL would be to a hypothetical real system.

An alternative for estimating on-chip memory energy consumption, mentioned in Section 2.3.3, is the tool CACTI. CACTI calculates the average energy consumption of a memory module based on parameters such as size, banks, ways, and process technology. The models CACTI use are based on analysis of the low-level structure of the RAM resources. If care is taken to have the CACTI parameters mirror the process technology parameters used for ASIC synthesis, it is likely that the CACTI estimates are more representative of a real instantiation of the architecture. An additional benefit from using CACTI is that it is less time-consuming than regression modelling, making it simpler to experiment with alternative cache organizations. Therefore, on-chip RAM structures will be modelled using CACTI.

Off-chip RAM Off-chip RAM energy consumption is difficult to model accurately, both because of advanced memory controller operation and because the energy consumption of DRAM chips is highly dependent on run-time characteristics such as the access patterns and timing [DRA14]. Creating such a model would require intricate knowledge of the DRAM chip and memory controller in question, and the resulting model would likely be dependent on large amounts of state to properly respect timing and access pattern implications. Thus, constructing a precise hardware implementation of an energy model for off-chip DRAM will incur a high implementation overhead.

Such precise off-chip RAM power models are not likely to be relevant for the majority of research conducted using the SHMAC platform, whose primary focus is the impact and use of heterogeneous computing resources. It is important not to ignore the energy consumption of the off-chip RAM as memory traffic may incur significant energy costs. For instance, evaluating a scheduling algorithm reliant on vast amounts of memory while ignoring memory access energy consumption may lead to misguided

conclusions as to the efficiency of the algorithm. However, having a detailed model for how the energy consumption of the off-chip RAM varies for different access patterns and timings will only be interesting for studies which attempt to exploit this behaviour. This may also fall under the scope of the SHMAC project, for instance having accelerators which attempt to capitalize on application memory request characteristics, but not omnipresently as precise reports of the energy consumption of the cores on the chip does.

Therefore, to limit the scope of this dissertation the off-chip RAM model will be designed using an average-case estimate for the energy consumption of reads and writes. This model will ensure memory activity is not ignored, but it will not report trends apart from access intensity.

Host Communication Communication with the host system is largely uninteresting for the purposes of the SHMAC project, since the cost of terminal I/O is either negligible due to slow user input or suppressible by reducing console output. The only experiments where this might matter are attempts to integrate the SHMAC into a larger system where there would be substantial communication needs between the SHMAC and another component. For instance, if the SHMAC is used as a coprocessor for a regular host processor and the two processors do not share memory, it may be interesting to determine the cost of transferring data back and forth between the systems. However, if this communication mattered it is unlikely that the current low-performance APB connection would be the basis for the experiment. The energy consumption of the APB communication is therefore either uninteresting, or irrelevant for experiments where communication would be interesting. It should also be mentioned that the reported energy consumption would be perturbed by the fact that energy data are transmitted across the APB link to the host, making it difficult to report a true estimate of its energy consumption. Therefore, no attempt will be made in this dissertation to design an accurate model of the energy cost of communication with the host system. If one is desired at some point, the regression modelling method in Section 3.3.1 could be employed. Special care would then have to be taken to ensure trueness of the behaviour of the modelled system, preferably by using separate links between the observer host system and the controlling host system.

Chapter 4
Energy Efficiency Estimation Framework Implementation

This chapter will explain how the methods and designs presented in Chapter 3 are implemented. First, Section 4.1 discusses the implementation of both the regression modelling method and the analytical modelling method. Finally, Section 4.2 discusses the implementation of the infrastructure design, and the utility software which makes use of it.

4.1 Modelling Method Implementation

This section will first present the implementation of the regression modelling method, in the order illustrated in Figure 4.1. Section 4.1 will discuss the work required to execute the ASIC synthesis step for the SHMAC. Section 4.1.2 discusses how the selection of test and validation benchmarks was performed. Then, how the test and validation benchmarks are converted into signal and power data through simulation and power calculation is discussed in Section 4.1.3 and Section 4.1.4, respectively. The final regression modelling step is treated in Section 4.1.5. This section focuses primarily on how a model is created, but also explains how the validation power is predicted[1] and how validation may be performed. The presentation of validation results is deferred to Chapter 6. Finally, Section 4.1.6 discusses the analytical modelling of on-chip and off-chip RAM. A detailed description of the software infrastructure supporting the execution of the modelling method is given in Appendix B, and a walk-through of its use is given in Appendix C.

4.1.1 Synthesis

To synthesize the SHMAC infrastructure, we use the tool Synopsys Design Compiler [Syn14c]. The tool works as illustrated in Figure 4.2. It accepts a set of HDL files, and creates an implementation of the design using logic gates. The set of gates

[1] Prediction for the purpose of validation may be done without executing the benchmarks on a SHMAC platform by using the signal data from simulation, which causes the slight discrepancy between Figure 4.1 and the general method presented in Figure 2.9.

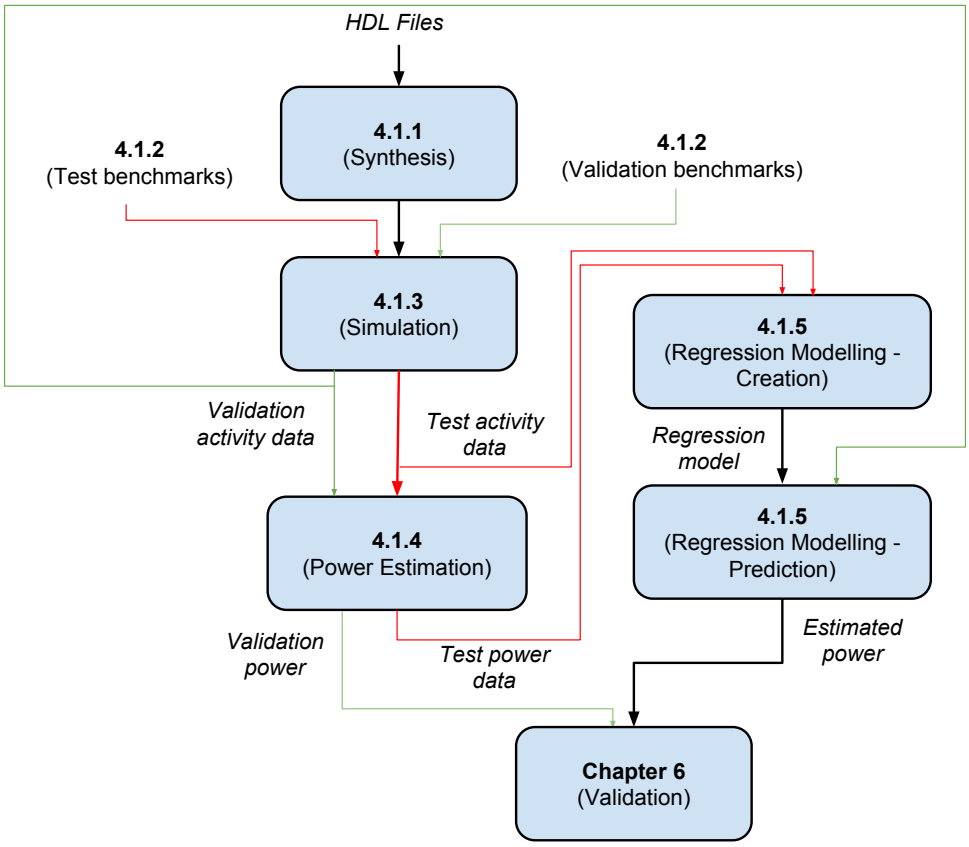

Figure 4.1: The order in which steps in the regression modelling method are discussed.

available is determined by the selected cell library, which is a collection of gates listed in a file which is also fed to the synthesizer. The implementation must also respect a set of design constraints, such as maximum delays, minimum clock frequency, and maximum area, which the user may set. These constraints therefore also act as inputs to the synthesis tool. Implementing the synthesis thus consists of three steps:

1. Select a cell library.

2. Ensure that the HDL is amenable to ASIC synthesis.

3. Determine design constraints.

These steps are described below. In addition to these steps, the Design Compiler tool must be invoked appropriately. The steps for controlling the tool have been written into scripts, automating the synthesis process which simplifies any desired tuning and adjustment of synthesis options by researchers who are not familiar with the synthesis tool. The implementation and use of the automated synthesis infrastructure is described in Appendix B.1.1.

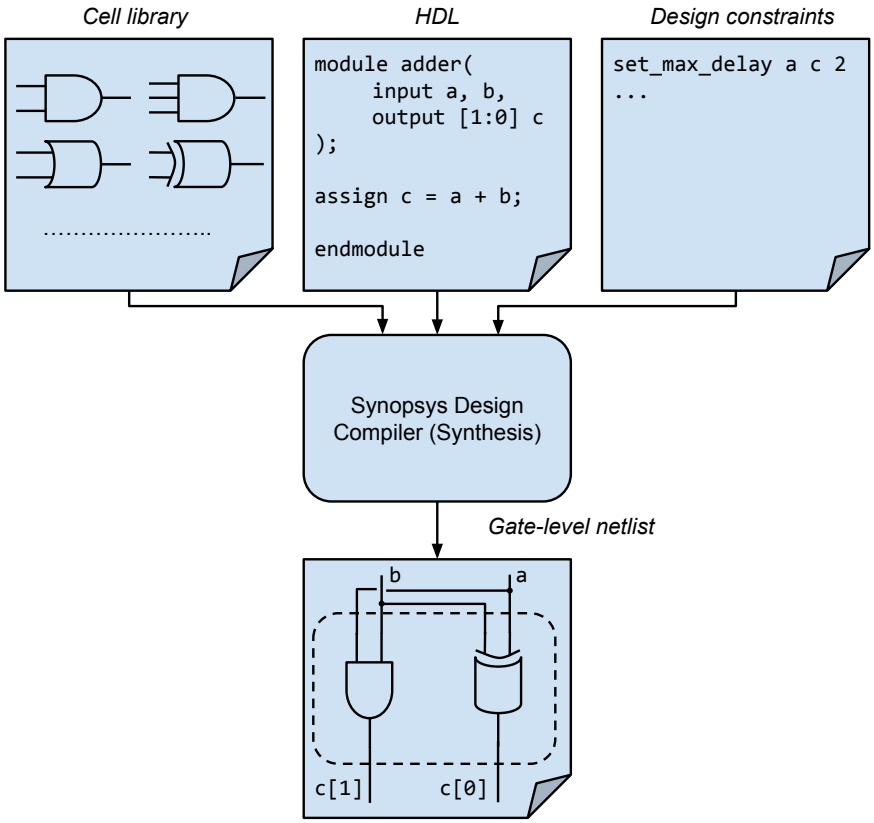

Figure 4.2: An example demonstrating the synthesis process.

Cell Library Selection The selected cell library is a standard 65 nanometre process, with a 1.00 Volt nominal operating voltage. The cell library is selected because of limited availability at IME servers. Still, the cell library is a reasonable target for SHMAC as it strikes a middle-ground between cutting-edge process nodes at 14 nanometre used in high-end processors and older process nodes at 180 nanometre still in use in embedded systems [Sem14].

HDL Modifications Four attributes of the SHMAC HDL required modification to support ASIC synthesis:

1. The multiplier used in the Amber core employs an FPGA-specific component not available during ASIC synthesis.
2. The on-chip RAM resources as described in the SHMAC HDL infer flip-flops for each storage element during ASIC synthesis, resulting in intolerable synthesis times for moderately sized scratchpad memories.
3. The DDR tile uses an FPGA-specific memory controller module.
4. Reset logic was not completely implemented, as read-only registers may be initialized during configuration when loaded on an FPGA.

The first and second issue were both solved by using ASIC synthesis specific components instead, from the Synopsys DesignWare component library [Des14].

The third issue could not be solved this way, since no similar memory controller was available in the ASIC library. Therefore, the DDR tile is not used for ASIC synthesis; instead, the Zero Bus Turn-around RAM (ZBTRAM) tile is used. The choice of off-chip RAM tile is inconsequential, since the off-chip memory tile is not modelled using regression modelling.

The fourth issue was remedied by enhancing the SHMAC with reset logic where this was missing.

The SHMAC infrastructure configuration used during synthesis is a commonly used 2×3 layout with an APB, off-chip RAM and a scratch pad tile on the top row, and three Amber tiles on the bottom row. Synthesizing this takes approximately three hours.

Design Constraints To ensure that timing analysis is performed at all, it is necessary to set some timing constraint on the design. We set a maximum clock period of four nanoseconds, which the synthesis tool is able to comfortably attain with a slack of 0.43 nanoseconds. We do not set any area constraints. It is also possible to set other options, such as chip input and output delays, ambient temperature, and register-level clock gating. No such options are set, either; instead, default values from the synthesis tool and the cell library are accepted. As there is no detailed specification for a physical implementation of the SHMAC infrastructure, there are no real values to mimic, and so default values are acceptable.

4.1.2 Benchmark Selection

The next paragraphs discuss the implementation of the test benchmark and the validation benchmark separately, as they must fulfil different requirements. Once constructed, both are embedded in a benchmark framework which implements the execution environment used during simulation. The details concerning the benchmark framework and its features is provided in Appendix B.2.

Test Benchmark The test benchmark is a custom-made program which executes in several short phases designed to exercise the system in different ways. The phases exhibit the following behaviour:

1. Uncached memory access.
2. Cached memory access.
3. Access to scratchpad memory.
4. Software and timer interrupt management.
5. A sequence of varied ALU operations.
6. A system call followed by the same ALU operations, now in privileged mode.

Internally in the Amber core, these phases cause variation in memory access components; execute stage operation; interrupt management; decoding of instructions; and control flow. All peripherals, namely the interrupt controller and timer module, are also exposed to varied access. Finally, the router faces varied work with access to the scratchpad memory tile, the off-chip RAM tile, and additionally the APB tile because of print statements between the phases.

Validation Benchmark To properly validate the models which are created, the validation benchmark should be more extensive than what is used to create the models. Additionally, it should bear little resemblance to the test benchmark if it is to ensure that models created with the test program works in any scenario. However, designing a good validation benchmark suite is a substantial undertaking. To limit the scope of this dissertation, the Dhrystone benchmark [Wei84] is selected as the primary validation benchmark as a port of it to the SHMAC architecture exists. The benchmark is of suitable length, as it is far longer than the test benchmark but still sufficiently small to run to completion in a circuit simulator. It is also designed to be representative of integer code in systems programming, based on statistical frequency of occurrence of programming language statements. These qualities make the benchmark a suitable first selection as a validation benchmark. Shortcomings

with Dhrystone which should be considered before conducting research with models validated with it are discussed in Section 7.4.1.

4.1.3 Simulation

To run post-synthesis simulation of the SHMAC infrastructure, we use the circuit simulator Synopsys VCS [Syn14e]. As with other circuit simulators, an HDL testbench is used to generate the stimuli which one wishes to run through the design under test. For this purpose, an existing test bench in the SHMAC project is used. The test bench works by loading a program into simulated memory and then starting the first Amber core in the system, which will run the program. This test bench can therefore be used to run the test and validation benchmarks. The test bench was extended to interpret writes to address zero as a termination signal, which the benchmarking framework exploits to stop simulation when the benchmark has completed.

The simulation is configured to operate in zero-delay mode. The goal of simulation is creating models which relate activity visible at the RT-level to power consumption. Gate delays and transition times are not visible at the RT-level, and it therefore makes little sense to use signal data which report state transitions occurring in the middle of clock periods[2] . Using zero-delay mode is also beneficial for both execution time and disk space requirements, since it obviates the need for large per-wire delay files and reduces the required resolution of data files from one picosecond to the target clock period of four nanoseconds.

We use the VCDPlus Dumping (VPD) format to store the signal data, as recommended by the VCS user guide [Syn14b]. This reduces storage needs during synthesis, and the resulting diminished I/O also speeds up simulation. The VPD file is a binary format, so it is converted to the text format Value Change Dump (VCD) before subsequent analysis.

The generation of signal data is controlled through a signal-dump statement in the test bench. Data for all the signals on the tile with the running Amber core are included in the output, which is feasible due to the use of the compact VPD format. Including all signals makes it unnecessary to run the simulation more than once. The signals are limited to one Amber tile, however, since all components which are modelled with regression modelling are included on this tile. Additionally, the assumed location-independent energy consumption of tiles makes it unnecessary to gather activity from other Amber tiles.

[2]Initial modelling work revealed that it is actually harmful to the efficacy of the modelling to take delays into account, since the corresponding power profile exhibits erratic behaviour not possible to capture using an RT-level model.

Using the simulator has also been automated, as described in Appendix B.1.2. Simulating the test benchmark takes approximately five minutes, and yields a VPD-file of 28 MB and a VCD file of 366 MB. Simulating the Dhrystone benchmark takes approximately four hours, and the resulting VPD file is 259 MB. Conversion of the entire VPD file was hindered due to disk quota restrictions on the server with the ASIC tools installed, but the first 2.23 milliseconds of simulation was converted to a VCD file of size 2.7 GB.

4.1.4 Power Analysis

The tool Synopsys PrimeTime PX is used to calculate power consumption from the VPD file. The process is illustrated in Figure 4.3. The tool analyses the VPD file from simulation, and calculates the energy consumed for each gate-level event. The sum of energy is distributed over the selected time resolution, which is one clock cycle in zero-delay mode; thus, PrimeTime calculates the power consumption per clock cycle.

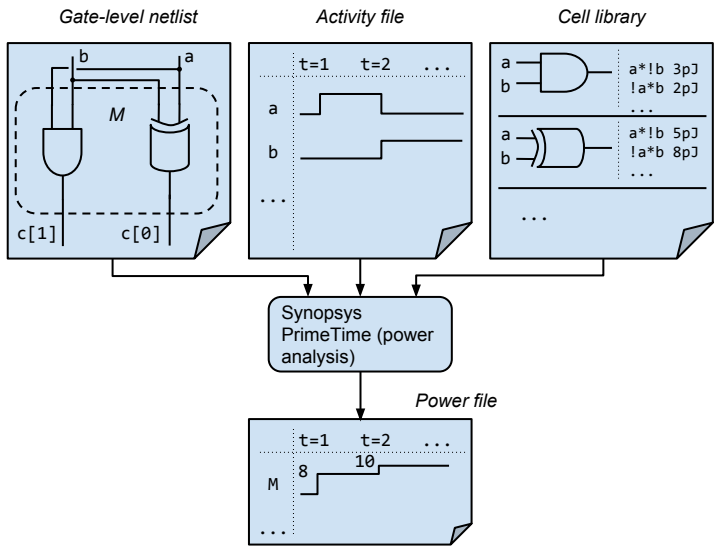

Figure 4.3: A figure demonstrating the power analysis process.

As with simulation, we use a compact binary output format to reduce computation time and storage requirements. The binary format is called FSDB. Before the power values are used in later stages, the data is converted to a text file format called OUT.

PrimeTime offers three options for the granularity of reporting power: one selected module; one selected module and all descendants of this module; or one selected module and all descendants of this module, *except* the leaf modules. The last option

is selected in the interest of time and space: time is saved by not having to rerun the power calculation to get the power consumption of different modules, and space is saved by not including the power consumption of leaf modules which represent individual gates. The use of FSDB as the output format and using the clock period as the time resolution makes it possible to store all non-leaf modules simultaneously in the output file. As with simulation, power data is only gathered from modules on the first Amber tile.

As with the other tools, invocation of PrimeTime PX for power calculation has been automated. Details may be found in Appendix B.1.3. Calculating power for the test benchmark took approximately two hours; calculating power consumption for the validation benchmark VCD file took approximately 60 hours. The FSDB and OUT files for the test benchmark take 3.5 MB and 15 MB, respectively; for the validation benchmark, these files take 41 MB and 239 MB.

4.1.5 Regression Modelling

To conduct the regression modelling, we use the statistical package R [R14]. We will first discuss how input data is managed, before turning to the actual modelling.

Input Data Management The synthesis and power analysis result files contain data for the entire first Amber tile. However, when modelling a given module the modeller will only want some subset of data relating to this particular module. It is undesirable to load unnecessary information, since this will increase the resource consumption of R needlessly. Additionally, the VCD and OUT files must be converted to a format readable by R. Consequently, the information in the output from synthesis and power analysis must be filtered and converted before it may be loaded for modelling purposes.

To meet these ends, utility programs were developed for selecting a subset of signals from a VCD file and a subset of modules from an OUT file. The available signals or modules are presented in a menu, from which a selection can be made. Once completed, the program filters the file for the selected information and writes it to a data frame file which may be loaded in R. A screenshot of the signal selection program is provided in Figure 4.4; the program which filters and converts power data is similar. These utility programs are described in further detail in Appendix B.3.1.

Model Creation The creation of models is supported by a set of R utility functions, which is described in detail in Appendix B.3.2. These utility functions operate on three kinds of data structures: benchmarks, power models, and evaluations. When creating models, the first step is to load a test benchmark object from signal and power data frame files. The next step is to create a power model object by correlating

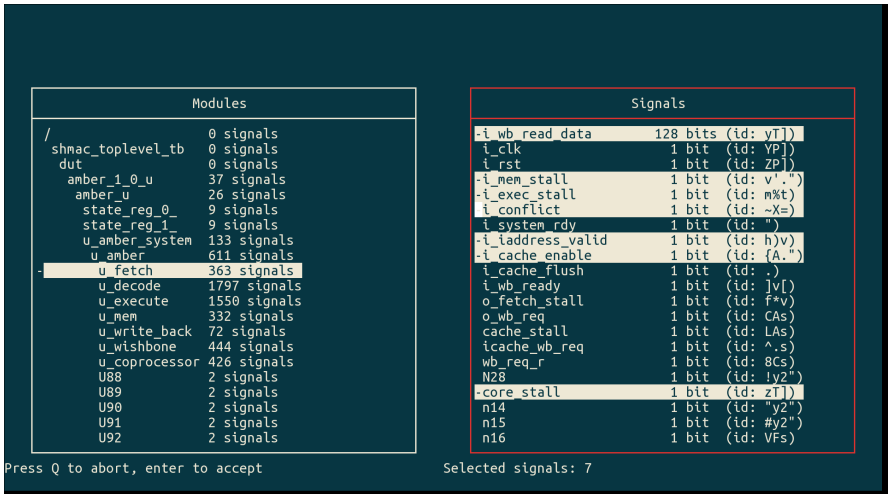

Figure 4.4: A screenshot demonstrating how VCD files may be converted and filtered before modelling.

the benchmark activity and power data. To do this, the modeller first decides on an expression with a sum of terms created from hardware signal data. For a hardware module with inputs a, b and c, this expression could for instance be $a + b \cdot c + a \cdot c$. Then, the R linear regression engine is invoked to calculate coefficients for each term, using the least-squares method. For our example, the result would be an expression $\alpha_1 \cdot a + \alpha_2 \cdot b \cdot c + \alpha_3 \cdot a \cdot c + \beta$. This expression is the power model of the module. Finally, the precision of the power model is evaluated using validation benchmarks objects. If unsatisfactory, the modeller must revise the expression input to the regression modelling engine either by modifying the terms, including more terms, or increasing the modelling granularity.

The end result from modelling is a formula $P = \alpha_1 x_1 + ... + \alpha_n x_n + \beta$ relating terms of signal values to power. The final goal, namely a model for energy, can be attained simply by scaling the regression coefficients by the clock cycle Δt since $E = P\Delta t = \Delta t\alpha_1 x_1 + ... + \Delta t\alpha_n x_n + \Delta t\beta$.

As described in Section 3.3.1, the first goal of modelling is reaching a certain accuracy. Implementation complexity may then be reduced while retaining accuracy. When creating modelling expressions, the model terms and granularity were therefore managed as follows:

Variable Selection The selection of variables used to create terms initially includes all named input, output and internal signals. For most buses, the Hamming

distance of their value is calculated and used as the independent variable. This choice is inspired by the work on PrEsto by Sunwoo *et al.* [SWPC10]. Trying to relate the numerical value of the bus to power consumption makes little sense; there is no intrinsic reason why the most significant bit should be responsible for twice the power consumption of the next-most significant bit. The Hamming distance, on the other hand, reflects the amount of charge movement having occurred from one cycle to the next, which is the origin of dynamic power consumption and therefore makes more sense to use as an independent variable.

Buses which act as multiplexer control signals are treated differently. The reason is that the amount of change in the selection of the multiplexer is not important; rather, it is the fact that the multiplexer control signal has changed that matters, since this results in the propagation of a different value from the multiplexer. Such multiplexer control signals are therefore modelled as binary values, reflecting whether the selection signal has changed since the last cycle.

Finally, bus signals may also be split if the bus gathers unrelated signals. For instance, the bus feeding instructions to the decode stage may be split into terms of the individual fields in the instruction.

Once a sufficiently accurate model has been created, reduction in complexity may be attempted by reducing the number of variables taken into account. This reduction should be based on which variables were the most significant in earlier modelling, while also trying to represent as diverse a range of sources as possible.

Term Complexity The initial model term complexity used is also inspired by [SWPC10]. Initially, all Hamming distance variables may be combined with any binary signal value. Based on the state of the binary control signals, different signal values will be propagated through the circuit and therefore different Hamming distances will be significant. This can be viewed as a form of tree regression [BBDM00]: different regression models are used based on whether a control signal value is true or false.

We do not initially include cross-products of binary signals, as in PrEsto, since this leads to high analysis times when the number of signals is high. Instead, extra cross-products between signal values may be included manually guided by what interactions exist in the HDL if accuracy is insufficient.

The term complexity is not further reduced in an attempt to reduce implementation overhead; it is instead initially low, to limit the time requirements for model creation.

Granularity The initial modelling granularity is set to one model for each pipeline stage of the Amber core; the timer, interrupt and tile register peripherals; and the router. The implementation complexity may be reduced for instance by

creating a model for the entire Amber core, instead of one model per pipeline stage. Reduction of complexity by coarsening the granularity is relatively simple, since the significance of inputs and outputs of each submodule has already been determined in earlier modelling steps. This information can be used to guide the variable selection for the more coarse-grained module.

Due to the topology of the synthesized SHMAC infrastructure configuration, the benchmarks only induce activity in the local port, the north-bound port, and the east-bound port of the router. Regression modelling will therefore not deem the west-bound and south-bound port activity as significant for an energy model. Terms for these ports are therefore manually included, based on the terms for the north-bound and east-bound ports under the assumption that similar activity in off-tile ports of the router induce similar energy consumption.

Model evaluation is done while varying the precision used to represent the coefficients, as this will impact the required resolution and bit width of the energy monitor modules. Ensuring that the model is accurate also with a reduced coefficient precision permits its eventual implementation in hardware as an integer computation, with the required precision determining the necessary energy sample bit width. The final models are listed in Appendix D.1, and their evaluation is presented in Chapter 6.

4.1.6 On-Chip and Off-Chip RAM

We will use CACTI to implement a model for on-chip RAM energy consumption of the form $E_{cycle_i} = E_{static} + read_{cycle_i} E_{read} + write_{cycle_i} E_{write}$, where $read_{cycle_i}$ and $write_{cycle_i}$ are binary variables indicating whether a read or write is active in cycle i, respectively.

Static energy consumption is derived from the CACTI report on total leakage power of a bank. The static power consumption is multiplied by the synthesis target clock period of four nanoseconds to attain a static per-cycle energy consumption estimate, and multiplied by the number of banks to get total energy for the RAM resource.

CACTI reports energy per read access, which may be read directly from the output.

To calculate write energy, one may subtract $1/associativity$ of the energy consumption of bitlines and sense amplifiers from the read energy to attain the write energy. The reason is that writes incur the same energy consumption as reads, except that it is only necessary to access the one way which is written [BSDB05].

We set the CACTI configurations to be as similar as possible to the cache and scratchpad parameters in the SHMAC. The CACTI configurations and resulting read, write and static energy consumption values are listed in Table 4.1.

Table 4.1: The CACTI configuration for scratchpad and cache RAM blocks, and resulting energy consumption.

Parameter	Value
Temperature	300 K
Process technology	65 nm, itrs-hp roadmap predictions (1.1 Volt)
Read-only ports	1
Write-only ports	1
I/O bus width	128
Use ECC	false
Caches	Size: 8 KB
	Ways: 2
	Banks: 2
	Energy consumed per read access: 0.0359628 nJ
	Energy consumed per write access: 0.0336836225
	Static energy consumption per cycle: 0.03355264 nJ
Scratchpad	Size: 128 KB
	Ways: 1
	Banks: 1
	Energy consumed per read access: 0.128824 nJ
	Energy consumed per write access: 0.128824 nJ
	Static energy consumption per cycle: 0.3148172 nJ

Off-Chip RAM To model off-chip DRAM power consumption, we use a spreadsheet from Micron with which you can estimate average power consumption for a DDR3 memory system. The spreadsheet calculates expected power consumption based on selected memory system characteristics. The spreadsheet also uses an estimate of average access behaviour to estimate the expected average power consumption. The calculations are based on DRAM construction as explained in a technical note [DDR14a]. The spreadsheet itself may be downloaded through the home page of Micron [DDR14c].

We configure the spreadsheet to estimate the power consumption of a DDR3 SDRAM with eight DRAM blocks of 4 Gb density with speed grade -103. A screenshot of the results from the spreadsheet is given in Figure 4.5.

The power consumption is split in three main categories: activation power, read-/write power, and background power. For simplicity, we will implement an energy model similar to that used for on-chip RAM: $E_{cycle_i} = E_{static} + read_{cycle_i} E_{read} + write_{cycle_i} E_{write}$. The energy values are calculated by calculating corresponding

ACT	110,9	mW
Total Activate Power	**110,9**	**mW**
RD	69,9	mW
WR	33,0	mW
READ I/O	18,5	mW
Write ODT	53,1	mW
Total RD/WR/Term Power	**174,5**	**mW**
ACT_STBY	39,5	mW
PRE_STBY	15,0	mW
ACT_PDN	7,3	mW
PRE_PDN	2,0	mW
REF	11,1	mW
Total Background Power	**75,0**	**mW**
Total DDR3 SDRAM Power	**360,4**	**mW**
TERM 2nd rank	**0,0**	**mW**

Figure 4.5: A screenshot of the spreadsheet results when estimating average DDR3 SDRAM power consumption.

power values, and multiplying them by the SHMAC clock period to get energy consumed per SHMAC clock cycle. The calculations are tabulated in Table 4.2. The average activation power, based on the default access patterns in the spreadsheet, is merged with the background power to produce the static power. In order to take read and write access intensity into account at run-time, the reported read and write power consumption values are scaled up by the inverse of the average access rates for reads and writes.

Table 4.2: Calculation of energy consumed by off-chip RAM activity.

E_{static}	$= \Delta t_{SHMAC}\, P_{static}$	
	$= \Delta t_{SHMAC}\, (P_{background} + P_{activate})$	
	$= 4\text{ ns} \cdot (75\text{ mW} + 110.9\text{ mW})$	$= 0.7436$ nJ
E_{read}	$= \Delta t_{SHMAC}\, P_{read}$	
	$= \Delta t_{SHMAC}\, \frac{1}{avg\_read\_access\_rate} P_{total\_read\_reported}$	
	$= \Delta t_{SHMAC}\, \frac{1}{avg\_read\_access\_rate} (P_{RD} + P_{READ\_IO})$	
	$= 4\text{ ns} \cdot \frac{1}{0.45}(69.9\text{ mW} + 18.5\text{ mW})$	$= 0.78577$ nJ
E_{write}	$= \Delta t_{SHMAC}\, P_{write}$	
	$= \Delta t_{SHMAC}\, \frac{1}{avg\_write\_access\_rate} P_{total\_write\_reported}$	
	$= \Delta t_{SHMAC}\, \frac{1}{avg\_write\_access\_rate} (P_{WR} + P_{Write\_ODT})$	
	$= 4\text{ ns} \cdot \frac{1}{0.25}(33.0\text{ mW} + 53.1\text{ mW})$	$= 1.3776$ nJ

4.2 Infrastructure Implementation

This section presents the hardware implementation of the energy efficiency estimation infrastructure, and utility software developed for using it.

4.2.1 Monitor Integration

Integrating the models into the infrastructure entails implementing monitors which evaluate the models each cycle. This section will first describe the implementation of the Hamming distance calculation mandated by the regression model terms described in Section 4.1.5. Next, we will describe the complete implementation of the regression models. Finally, we look at how the RAM models were implemented.

Hamming Distance Calculation Calculating the Hamming distance between two buses is done by summing the number of bits which differ between them, where differing bits are determined using exclusive-or. For wide buses, this sum can be expensive. One alternative strategy is to index a lookup table with the exclusive-or result, which costs memory instead of adders. We implement the Hamming distance by combining the two approaches, drawing inspiration from Sklyarov *et al.*[SS13b] and exploiting the fact that FPGAs are constructed from lookup tables. We can use the 6-input Look-Up Tables (LUTs) on the Virtex 7 to divide the exclusive-or result bus into 6-bit segments, and use three LUTs to calculate its Hamming weight[3]. Then, we use an adder tree to calculate the sum of all Hamming weights. The final hamming distance calculation structure is depicted in Figure 4.6.

Hardware Regression Model Implementation The regression model terms consist of multiplications between constants, binary signals, and at most one Hamming distance value. When terms do not include a Hamming distance, the multiplication is implemented at low cost using and-gates with the coefficient and the boolean condition as input. Some multiplications involving Hamming distances may also be optimized by implementing the multiplication using bit-shifts and sums. Otherwise, the multiplications are mapped to FPGA DSPs. The multiplications are not wide, due to limited coefficient resolution and Hamming distances being logarithmic to the bus widths[4], which renders them possible to implement at reasonable cost.

It is not possible to implement the regression models in a single cycle except in rare cases, for the following reasons:

[3]The hamming weight is the sum of bits equal to 1 in a bit string. For a six-bit long input, the maximum result is six, which requires three LUTs to compute.

[4]The largest widest multiplication is between a seven-bit coefficient and a six-bit Hamming distance.

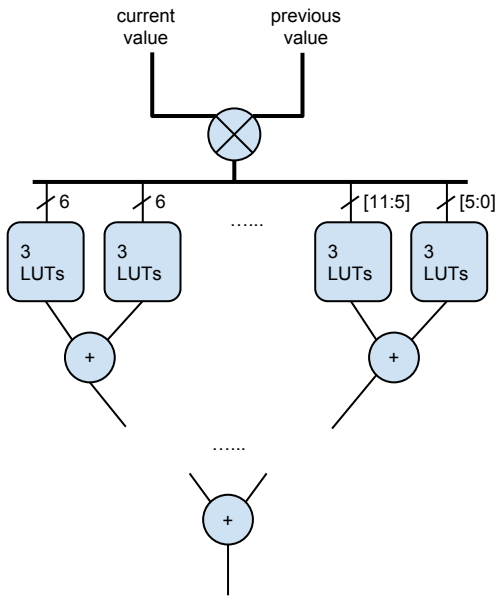

Figure 4.6: The implementation of Hamming distance calculation.

1. The model may use internal signals in the module being modelled, and internal signals may be available only after a certain delay.
2. The Hamming distance calculation delay is significant.
3. The delay through the sum of terms in the model is significant.

These issues are mitigated by using pipelining, which the monitor implementation is amenable to. At most three pipeline stages are used. The first pipeline stage simply stores model inputs, removing delays due to internal signal input. The second pipeline stage calculates all term values, including Hamming distances and multiplications. The final stage calculates the sum of all model terms.

Hardware Analytical Model Implementation The analytical models used for the RAM resources are cheap to implement, as they only consist of sums of conditionally-included coefficients. The models of caches and scratchpad memory are thus trivially implemented. However, there is a slight difficulty in calculating the conditions for including the coefficients in the case of off-chip RAM. The model is based on knowing in which cycles the off-chip DDR is processing requests. However, all DDR requests are immediately forwarded to an FPGA soft-macro memory controller, which obfuscates the relationship between HDL activity and DDR activity. We

therefore estimate the DDR activity based on DDR timing specifications [DDR14b], which state that the latency for read and write operations is determined by the metrics CAS Latency (CL) and CAS Write Latency (CWL). For the DDR module used to create the model, CL is 13.75 nanoseconds and CWL is 10 nanoseconds. As one cycle on the SHMAC is $16\frac{2}{3}$ nanoseconds, we implement the off-chip RAM model by scaling the energy values for reads and writes by the ratio of CL and CWL values to the SHMAC cycle period and counting one cycle of activity per request.

4.2.2 Energy Report Unit

The implementation of the energy report unit is depicted in Figure 4.7. There are no particularly complex elements. There are two separate input control signals: one sample signal, indicating that a new sample should be taken, and a shift signal, indicating that the values should be shifted along the scan chain. The unit also takes as input the current energy sample, and the value from the report unit behind it in the scan chain. The operation of the unit is then to sum the observed energy samples, until a sample is taken. When this happens, the sum is clocked into the scan chain output register, and the sum is reset. When shift operations are requested, the value from the previous unit in the scan chain is clocked into the scan chain output register.

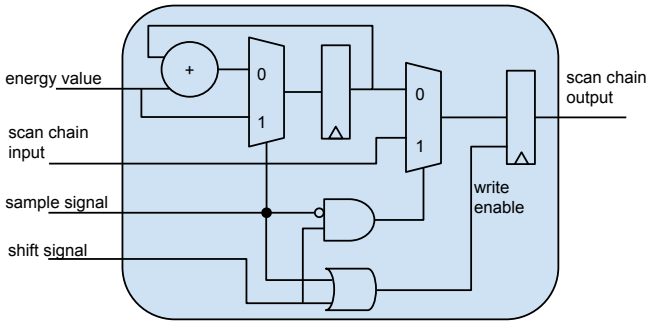

Figure 4.7: The RTL implementation of the energy report unit.

4.2.3 SHMAC Software Interface

The interface from the infrastructure to software running on the SHMAC is implemented by adding an additional peripheral to the existing Wishbone bus [Wis14], as illustrated in Figure 4.8. The peripheral maintains one set of registers described in Section 3.2.5 for each energy monitor which is exposed. Each monitor occupies 32 bytes of address space, allocated sequentially from the new peripheral base. Which monitor counters to access is thus determined by bits 31-5 of the Wishbone address, and which register to access in this monitor is determined by bits 4-2. Including

a new monitor among those available to software may be done by hooking a new register set up to the energy counter peripheral.

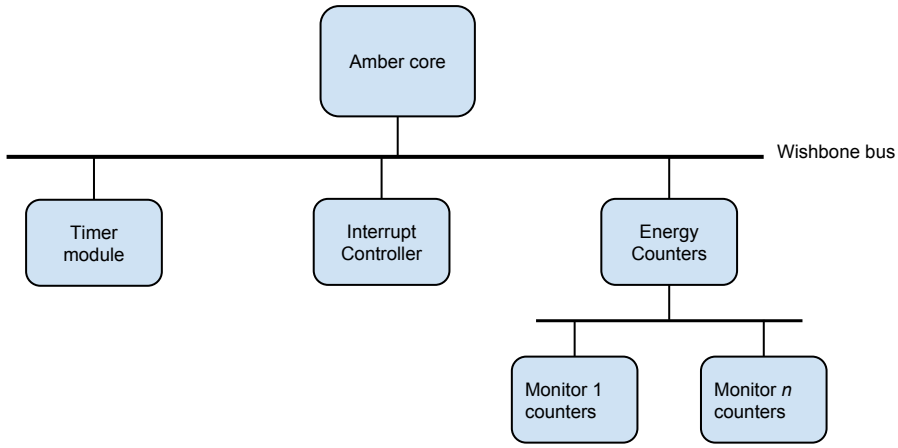

Figure 4.8: The energy counters are integrated into the SHMAC infrastructure by inclusion in the Amber core Wishbone bus.

The implementation of the register set for each monitor is depicted in Figure 4.9. An internal counter register is used to keep track of how many cycles the current energy count represents, and its value is written to the current period duration when the current energy count is read. Since the registers are designed to be read sequentially, we do not need to output the duration before the cycle after the energy is read. Overflow in either the energy sum or the counter value will trigger an update of the previous period duration and energy count. Overflow also resets the internal counter to one, and the current energy count is set to the value of the next energy sample.

4.2.4 Scan Chain

The scan chain is constructed by chaining together the energy report units on each tile, and adding extra signals at each tile boundary to connect the chain across tiles. The sample and shift signals are globally distributed to each report unit, to ensure that the units operate in lock-step. The complexity lies in the implementation of the control unit, which is responsible for triggering the sample and shift signals based on commands received from the host over the APB link. Because the APB uses a different clock than the SHMAC architecture, it is necessary to construct a clock-domain-crossing protocol. The control unit is located on the APB tile, and its implementation is depicted in Figure 4.10.

The control unit is split in three constituents. One part provides the interface to the APB control module and runs in the APB clock domain, and another part

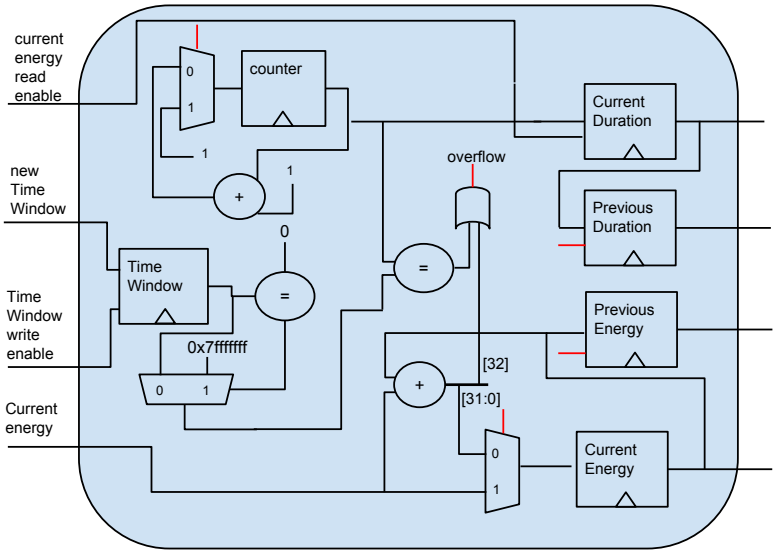

Figure 4.9: The implementation of the register set for one energy monitor.

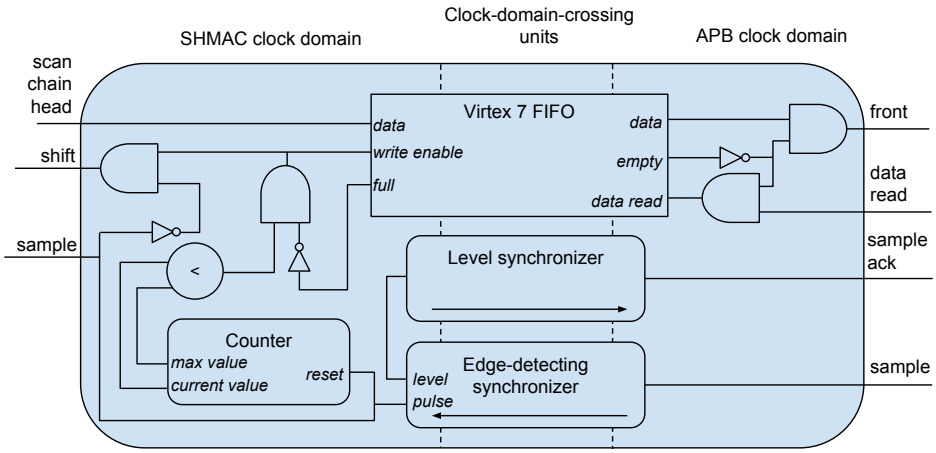

Figure 4.10: The implementation of the scan chain control unit.

interfaces the rest of the SHMAC infrastructure and runs in the SHMAC clock domain. The final part synchronizes the APB interface and the SHMAC interface using an FPGA-specific dual clock domain FIFO for data transfer, and synchronizer circuits for sample signal transfer [Ste03]. The design is simplified by assuming that the APB clock is not faster than the SHMAC clock, which is currently true. It may also be trivially ensured by slowing down the APB clock if necessary.

The scan chain control is primarily situated in the SHMAC clock domain. It is driven by a counter, which keeps track of the number of samples which have been shifted from the scan chain into the FIFO. While below a maximum value, new samples are continuously shifted into the FIFO. Once all samples have been shifted into the FIFO, this shifting is stopped.

The APB clock domain interfaces the APB tile, which may either read a new sample value or request that a new sample is taken as per the design of the host interface in Section 3.2.4. New sample requests are forwarded to the SHMAC clock domain, which cause the counter to be reset and the scan chain sample signal to be pulsed.

4.2.5 Host Interface

Implementing the host interface consisted of adding support for the *SampleEnergy* and *NextSample* registers, described in Section 3.2.4, to both the APB tile and the Linux driver on the host which communicates with the SHMAC.

User-space programs are given access the energy data through a special file, which is exported from the Linux driver as a binary `sysfs` attribute. Reading n bytes from this file causes the driver to read the next n bytes of sample data available from the *NextSample* register on the APB tile. If a read to the file starts at offset zero in the file, the driver first issues a write to the *SampleEnergy* register to request a new sample to be taken. This way, new samples are made available each time the file is re-read.

To facilitate interpretation of the energy sample data, the driver was extended with additional `sysfs` attributes which report the layout of the current SHMAC instance, the distribution of energy report units, and their respective sample sizes. The values exported through these attributes are configured in the driver itself. Exporting the configuration through the driver removes the need to hard-code such values in all software which needs it, so that installing new SHMAC instances in theory will only require updating the configuration in the driver accordingly.

Along with the driver extensions, two user-space tools were developed to give command-line access to the energy data. The first tool is called `shmac_read_energy`, which merely reads one set of energy samples and writes them as text to standard output. The other tool is called `shmac_stream_energy`, and provides more fine-grained control over sampling rates. It accepts a sample period parameter and an output file argument, and operates by copying the contents of the energy samples attribute to the output file once every sample period. Details concerning the user-space utility programs are given in Appendix B.4.1.

4.2.6 Monitoring and Logging Tool

The `shmac_stream_energy` utility makes it possible to log data directly on the SHMAC host. However, the host system uses an SD card with limited storage capacity. For long-running experiments, it is therefore desirable to ship the sampled data directly to a different server instead of requiring to store the data on the SHMAC host first.

To this end, a monitoring and logging tool was developed. The operation of the tool is illustrated in Figure 4.11. First, a UDP socket is set up on the machine running the logging tool. Second, the utility program `socat` [Rie14] is used to set up a link between a pipe file on the SHMAC host and the UDP socket on the developer machine. Third, the `shmac_stream_energy` program is invoked with the pipe file as the output file. In this way, all the data that `shmac_stream_energy` writes to the pipe file is automatically sent as UDP packets to the developer machine, where they may be logged to file and optionally displayed as real-time energy data. Once these three steps have been executed, energy samples are being continually read from the SHMAC instance on the FPGA. The energy consumption of a program may then be analysed by starting it while the sampling framework is running. Further details on the capabilities of the tool is given in Appendix B.4.2.

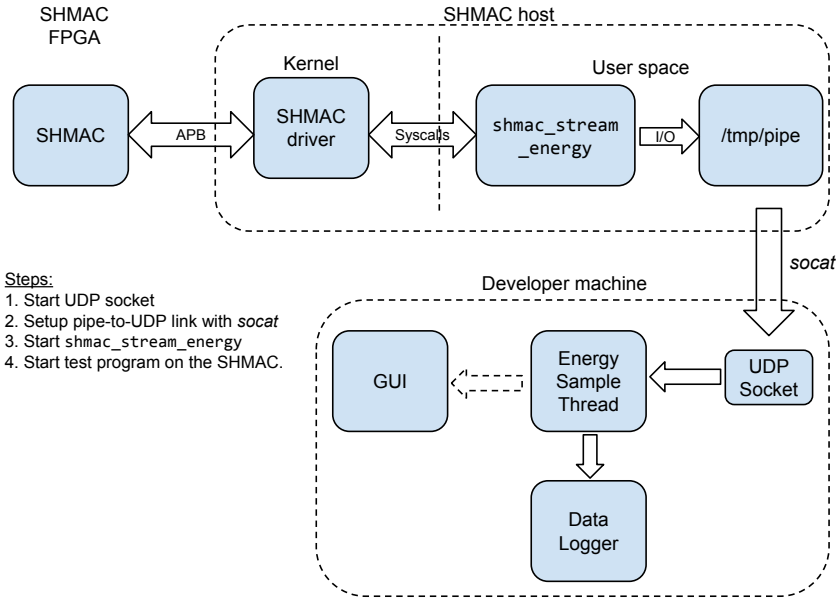

Figure 4.11: The complete flow of energy sample data when using the energy monitoring and logging tool.

Chapter 5
Barrelfish Implementation

5.1 Requirements Specification

In this section, we elaborate on what modifications and enhancements to Barrelfish are necessary to implement each of the tasks identified in Section 1.4:

- Supporting the Barrelfish boot process which starts multiple cores, i.e. task `B1`.
- Maintaining compatibility with new versions of the underlying SHMAC platform, i.e. task `B2`.
- Development of console support in Barrelfish, i.e.task `B3`.

An explicit non-goal of the multicore port is retaining compatibility with the old RealView SHMAC platform. Since a new execution environment with sufficient amounts of RAM has been acquired partly for the purpose of avoiding the effort required to shoehorn Barrelfish into a memory-constrained environment, it would be counter-productive to undertake this endeavour.

5.1.1 Multicore Support Requirements

Launching New Kernels The most obvious requirement for supporting multiple cores in the operating system is the ability to load kernels and start running them on other cores. This requires loading a new kernel image for each core, and allocating resources the kernel requires when booting. Loading the kernels at arbitrary locations at run-time also mandates implementing support for relocatable kernel images, as suggested in the previous work on the Barrelfish port [BS13]. Additionally, the BSP kernel needs to transfer information, such as where allotted boot resources are located, to the app kernels.

Dynamic Interrupt Vector Dispatch A requirement identified in the previous project [BS13] is supporting dynamic dispatch in the interrupt vector, i.e. branching to a core-local implementation of the interrupt handler. Each Barrelfish kernel is supposed to have its own binary image to avoid sharing state with other kernels; however, the interrupt vector is shared by every kernel since the Amber core expects this to be located at address zero. Without virtual memory support, it is not possible to redefine what memory a core retrieves when accessing the interrupt vector entry addresses. Thus, if special measures are not taken then upon execution of a core interrupt routine all cores will execute with the kernel image of core 0, thereby also accessing its local state.

Intercore Communication Finally, in addition to booting kernels on new cores the Barrelfish monitors must be set up to communicate with each other.

5.1.2 SHMAC Compatibility Requirements

Supporting Upgrades to the Instruction Set In the work on porting Linux to SHMAC [AA14], the Amber core was upgraded first from ARMv2a to ARMv3, and subsequently from ARMv3 to ARMv4T. These upgrades were necessary in order to run Linux. The ISA upgrades were also adopted in a master dissertation on creating a high-performance core for SHMAC [AB14]. As such, any future experiments on system software conducted using the SHMAC platform would likely be based on a platform with the upgraded cores. Thus, for the Barrelfish port to stay relevant it was necessary to update the operating system to handle any binary incompatibilities introduced from the ISA upgrades. Since ARMv3 is not backwards compatible with ARMv2a, support for this instruction set would require modifications to Barrelfish. ARMv4T, on the other hand, is binary compatible with ARMv3; supporting ARMv3 would as such be sufficient for Barrelfish to run on the latest SHMAC CPU tiles.

A benefit of upgrading to ARMv4T, however, is that this is the oldest ISA properly supported by default by the GNU toolchains. At the beginning of this dissertation, Barrelfish as well as all other software for SHMAC had to be built with a specially-patched toolchain. This requirement may be dropped if the software supports being built targeting ARMv4T, so ensuring that Barrelfish supports this build target would make development more accessible to others. Additionally, Barrelfish would then also benefit from the new instructions of ARMv4T.

Allocate Shared Memory from Scratchpad Tiles Another improvement which was planned for the SHMAC platform, was to fix the bug mentioned in Section 2.1 which prevented data caches from being used. If this bug was fixed, it would of course be desirable to run Barrelfish using caches. However, since there is no cache coherence either existing or planned for the SHMAC platform, any memory resources shared

between cores must be allocated from an uncached area of memory. Specifically, this would entail allocations from the FPGA BRAM resources available on the scratchpad tiles, since these were created for the purpose of sharing data between cores.

Avoid Architecture Dependencies As the SHMAC project is still in its early stages, there will most likely be other changes apart from those already planned for completion during the timeframe of this dissertation. In an attempt to keep Barrelfish as relevant for the future as possible, an overarching design goal would be to avoid architecture dependencies to as large a degree as possible. For instance, in anticipation of virtual memory the port should ideally make use of the Barrelfish memory system abstractions instead of accessing raw addresses directly. This particular goal is a bigger trade-off than it might seem: the memory system and capability abstractions in Barrelfish are relatively complex, and using them in a port to a completely unprotected SHMAC platform will yield more complex code than necessary. The temptation to bypass these systems should be resisted, however, to increase portability within the SHMAC infrastructure families.

5.1.3 Console Support

Barrelfish may run benchmarks by including them as modules in the *shmacfish* configuration and specifying that they should be started when the boot is completed. To run different benchmarks, it is therefore necessary to edit the configuration, rebuild Barrelfish and restart SHMAC. If support for a console application was included, then a researcher wishing to run different benchmarks or the same benchmark multiple times could simply include every benchmark in the set of uploaded modules and start them interactively as desired. Porting the console application in Barrelfish, *fish*, to SHMAC would enable this scenario. The implementation of *fish* itself is architecture-independent, as it uses the *angler* session management application to connect to the *serial* driver. The *serial* driver, on the other hand, must be extended with support for the SHMAC-specific interface to host serial communication. Since *serial* is a user space driver, it is also necessary to extend the SHMAC kernel to support registering user-space interrupt listeners and deliver interrupts to them.

5.2 Implementing Multicore Support

This sections details how the Barrelfish port was extended with multicore support. Section 5.2.1 explains what steps are taken from the request to boot a new core is received until a new kernel is running on this core. Next, Section 5.2.2 explains how interrupts are forwarded to the correct kernel. Finally, Section 5.2.3 explains how communication between the cores is set up.

5.2.1 Booting New Cores

Boot Request Message Handling

As described in Section 2.2, multicore boot is initiated by *spawnd* who sends one boot request message to its local monitor for each core it has determined to start. The handling of said message in the monitor is architecture-dependent, and is in the SHMAC port implemented as illustrated in Figure 5.1.

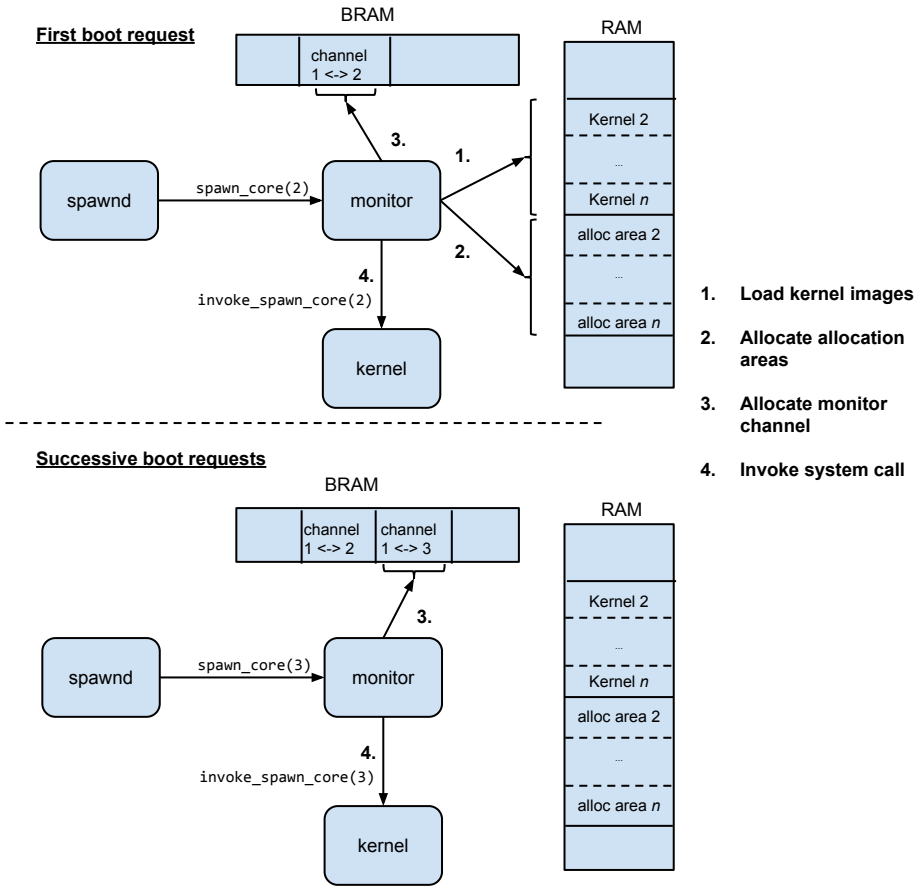

Figure 5.1: Handling the request of the boot of a new core in the monitor. The numbered arrows are steps in the implementation. Note that kernels and allocation areas are only contiguous in memory for illustration purposes; this may not be the case in actuality.

As can be seen, the first boot request is handled specially, with successive boot requests being handled in a different manner. This is due to the discrepancy between the Barrelfish assumptions that cores are booted one at a time, and the actual

capabilities of the underlying SHMAC platform where the only option is to start all the remaining cores simultaneously. Therefore, certain boot tasks must be done for all cores immediately. However, the semantics expected by Barrelfish for the boot-request message are upheld by blocking app kernels in their initialization code until they have been explicitly started. The reason is that the BSP monitor initializes the bindings to other monitors when the request for booting them arrives. If all app kernels are permitted to start after the first message, Barrelfish may fall victim to a race condition where an app-core monitor sends a message to the BSP monitor before its binding is initialized.

The implementation of the first boot request is comprised of the following four steps:

1. Load a New Kernel Image As Barrelfish is a multikernel, each kernel executes using its own image to avoid sharing data. As such, one must be allocated when a new core is started. Therefore, kernel images for all cores are loaded when first receiving a boot request message, to make sure they all can execute their own kernel. To support arbitrary load locations, the build system was enhanced to produce relocatable kernel images. Extra relocation support was then added to the Barrelfish ELF loader.

2. Provide a Memory Allocation Area When starting a new kernel, memory is required for initializing the capability tree, loading the first unprivileged executable, and initializing its capabilities. As such, when the monitor starts a new kernel it allocates a memory region on its behalf which may be used to service the initial allocations. It is possible to know how much memory the app kernel requires for loading the monitor image by consulting its ELF file; it is harder to automatically determine the amount of memory required for initial capabilities, as this is dependent on the behaviour of the app kernel. To minimize any overhead from allocating memory up-front, the amount of memory required by an app kernel for capabilities was first summed up on a test run, and subsequently used to calibrate the amount of memory allotted to the app kernel. The calibration works for all app kernels as they allocate the same resources, but modifications to the app kernel allocations may require recalibration.

3. Allocate Channel Memory Intercore communication in Barrelfish is in general set up by having the core-local monitor forward connection requests on the behalf of application programs to monitors on other cores. The requirements for bootstrapping intercore communication is therefore to initialize the communication between the monitors on different cores manually. When processing the boot request message, a message passing channel memory area is therefore allocated. Its address is sent to the app kernel which may subsequently send this to its monitor. Using

such lazy channel allocation supports using only a subset of the cores on the SHMAC platform without wasting the scarce BRAM resources.

4. Invoke System Call The final thing which is done is invoking a system call, requesting the BSP kernel to start the other cores. The monitor sends the address of the kernel image; the allocated memory area; the location of the monitor ELF file; the size of inter-monitor channels; and the address of the channel between the BSP monitor and the monitor on the core being booted. Having invoked the system call, the work in the monitor is completed.

Handling any successive boot request messages in the monitor only requires executing points 3 and 4.

Core Boot System Call Implementation To ensure that the app kernels are able to run, several parameters must be passed from the BSP kernel. These parameters are explained in Table 5.1. The parameters are stored in shared scratchpad memory. Since the location of the parameters must be determined by convention between the kernels, they are deterministically stored at the beginning of the first scratchpad tile.

Table 5.1: The boot parameters sent to an application kernel by the BSP kernel.

Parameter name	Description
`relocation_distance`	Kernel image location
`monitor_binary`	Location of the monitor ELF file
`monitor_size`	Size of the monitor ELF file
`free_memory`	Free memory for initial allocations
`free_memory_size`	Amount of free memory
`log2_channel_size`	Length of a channel
`ump_frame_base`	Location of channels to BSP monitor
`is_started`	Allow app kernel to proceed
`printf_lock_addr`	Address of the mutex guarding printf

Most of the parameters are supplied from the monitor as parameters to the system call. The two which are not, are `is_started` and `printf_lock_addr`. The `is_started` flag is used to emulate starting only one core at a time. When the BSP kernel handles a boot request system call for a core, it will set its `is_started` flag to `true`. The app kernels can therefore check whether it has logically been started by checking the value of this parameter.

The `printf_lock` guards access to the SHMAC output register, to avoid garbled output when multiple cores wish to print something. Locks must also be located in shared memory, and instead of placing the `printf_lock` at an absolute address it is

allocated after allocating space for boot parameters. Keeping all allocations gathered avoids fragmentation of the shared memory, which is important for capability creation purposes. Since the `printf_lock` is dynamically allocated, its address is sent to app kernels to enable them to initialize their own lock variable.

The implementation of the system call managing core boot is illustrated in Figure 5.2. As with the boot request message handler in the monitor, the system call performs some initialization only the first time it is invoked. Specifically, it sets the parameters it receives from the monitor as well as the `printf_lock_addr` variable for all the app-cores. It also initializes each `is_started` flag to zero. Then, it starts all the app-cores by writing to the system register `SYS_READY`. Apart from these steps, all core boot system calls are handled similarly. First, the address to the `intermon` channel is set: since the monitor allocates `intermon` channels lazily, these are supplied for each system call invoked. Finally, the `is_started` flag of the core which should be started is set to 1, allowing the kernel on this core to progress.

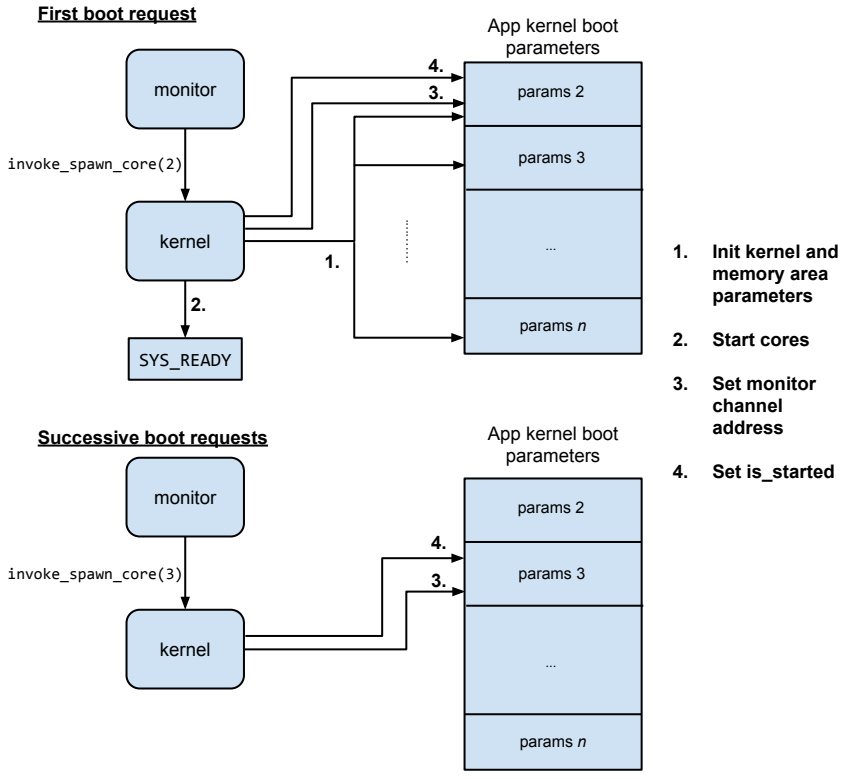

Figure 5.2: Implementation of the system call booting new cores.

Application Kernel Boot Procedure

Once `SYS_READY` is set to 1, the app kernels begin executing their reset handler, located at address zero. In order to start executing in their own kernel, they have to jump the relocation distance. To make it easy for the app kernels to find this parameter, all relocation distances are stored consecutively first in the scratchpad memory instead of being placed with the rest of the boot parameters. This way, app kernels can use only their core ID as index instead of having to multiply it by the required bytes of boot parameter data. This lets the interrupt handler assembly implementations be independent of the number and size of app kernel boot parameters. The complete memory layout describing the BSP kernel's use of the first scratchpad tile is illustrated in Figure 5.3.

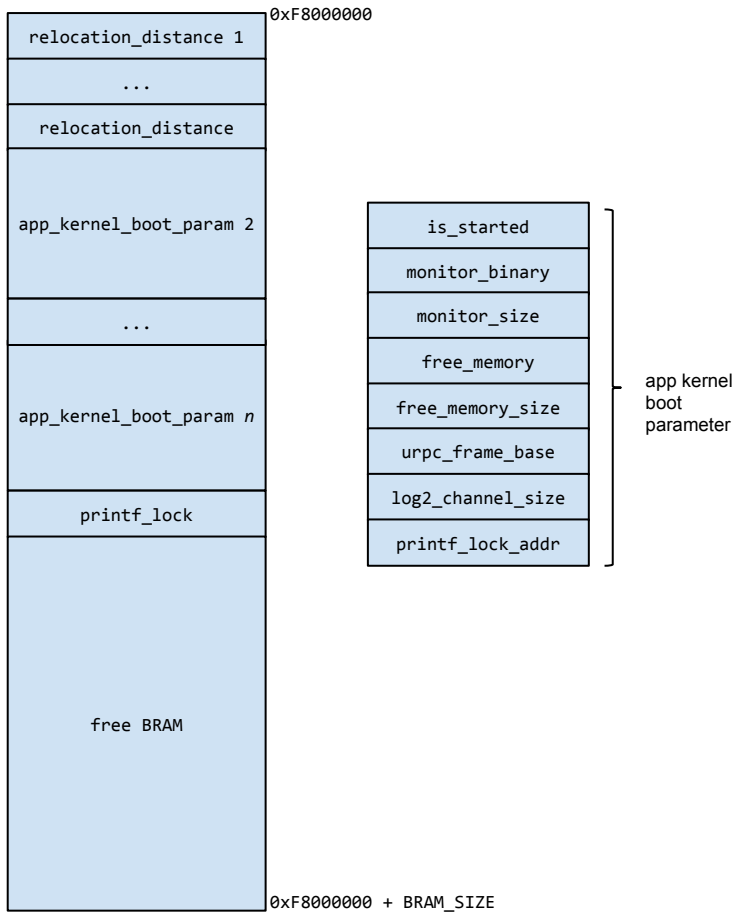

Figure 5.3: The BSP kernel's use of the first scratchpad tile.

Having jumped to its own image, it is important to keep in mind that kernels having an image each imply that the global variables in the app kernel images are not the same as those set in the BSP kernel. Consequently, the app kernels must first initialize such global variables. Specifically, this includes initializing the global variable pointing to the app kernel boot parameters. Using this, the app kernel can loop on the condition of the `is_started` parameter, initialize the printf lock, and initialize its own memory allocator.

Having made all necessary kernel initializations, boot proceeds by loading the monitor, initializing its capabilities, and running it. This is similar to the boot of the BSP processor, with an important difference in the capability initialization: the app kernels have to set up a capability to the channel memory intended to bootstrap communication between the monitor on this core and the BSP monitor. Setting up capabilities to the rest of memory is not required, in contrast to BSP kernel boot code, since this is controlled by *mem_serv* running on the BSP.

5.2.2 Dynamic Interrupt Vector Dispatch

The jump in the reset routine to the local image of an app kernel is just a special case of the implementation of dynamic interrupt vector dispatch. As with the reset routine, all interrupt handlers must jump the relocation distance before executing code dependent on global variables. By including this as a prologue to every interrupt handler, the handlers can be implemented exactly as they were without concern about kernel image location. Since the dynamic dispatch code is generic for all handlers except reset, the functionality can be wrapped in a macro and reused by all handlers. This macro is listed in Listing 5.1.

Listing 5.1: The first code executed in every interrupt handler except reset.

```
1  #define INTERRUPT_HANDLER_PROLOGUE(handler_name)          \
2  handler_name:                                             \
3          push {ip};                                        \
4                                                            \
5          /* Load core number * 4 into ip */                \
6          ldr ip, =TILE_BASE;                               \
7          ldr ip, [ip];                                     \
8          lsl ip, ip, #2;                                   \
9                                                            \
10         /* Index the relocation distances at BRAM0_BASE */ \
11         add ip, ip, #BRAM0_BASE;                          \
12         ldr ip, [ip];                                     \
13         add pc, pc, ip;                                   \
14         nop;                                              \
15 local_##handler_name:                                     \
16         pop {ip}
```

The handler starts by using `ip` as a temporary register, backed up on the interrupt stack in question in case its value is used by the interrupt handler. Next, the index is calculated by loading the core ID into `ip` and multiplying it by 4 since the relocation distance is a four-byte quantity. Next, the index is added with the base to form the final address, which is used to load the relevant relocation distance into `ip`. This distance is added to the current instruction address and stored to the program counter, which will execute the jump into the local interrupt handler code labelled as `local_handler_name`. Since the `pc` register in the Amber core points two instructions ahead of the currently executing instruction, the instruction after the jump will be skipped. It is therefore padded with a `nop` instruction.

The reset vector must be treated differently, both because a stack has not been set up yet so push and pop instructions should not be used, and because the relocation distances have not yet been initialized. The reset vector therefore has a separate, special-case branch checking if it is executing on the BSP processor, and if so it initializes its own relocation distance to zero to make the other interrupt handlers work. The app kernels use the same logic as in the prologue macro.

Having defined this macro, wrapping all the existing interrupt handler labels in it was sufficient to complete the dynamic vector interrupt dispatch support.

5.2.3 Bootstrapping Intercore Communication

Once the app kernels have started their monitors, channels must be set up between monitors on the app kernels to establish point-to-point connections between all cores. When running with caches disabled, this required no special support for SHMAC, and it was possible to reuse the code from the existing ARM ports of Barrelfish. The only exception was that it was necessary to implement SHMAC-specific access to a hardware cycle counter to support the Barrelfish implementation of blocking UMP message reception algorithm, which uses polling for a set number of cycles before yielding the CPU and leaving to the monitor to wake it up in response to the arrival of a message.

5.3 Supporting Upgrades to the Instruction Set

5.3.1 Adding ARMv3 Support

As explained in Section 2.1, the upgrade from ARMv2a to ARMv3 was not a binary compatible one. In order to run Barrelfish on the new processor core, it was therefore necessary to retarget the operating system to the new instruction set. This required the following changes to the implementation:

- All context switch code had to be modified to store and restore the new CPSR register.

- Code manipulating status bits, interrupt mode bits or CPU mode bits had to be modified to use new assembly instructions introduced for this purpose, and adopt to the new format of the status register. This includes stack initialization for different modes, sanity checks in context switch code ensuring proper status bits for user mode, and interrupt management routines.

- As the entire PC now contained a valid address, code which previously masked out the top or bottom bits of the PC now had to use the entire value. Specifically, this affected looking up syscall values using the LR register; printing stack traces by following a chain of LR register values; and checking whether an instruction address was within a critical region at the time of context switch.

- Stacks had to be allocated for the new undefined instruction mode and abort mode. The undefined instruction mode was used by the debugger, and the abort mode was used as a trigger for hardware assertion errors indicating logical faults in the RTL.

- Finally, the build system configuration had to be set up to target ARMv3. This also implied that architecture-dependent code in Barrelfish where the implementation was selected based on the preprocessor macro `__ARM_ARCH_2__` had to instead check for the existence of the macro `__ARM_ARCH_3__`.

5.3.2 Upgrading to ARMv4T

Adding support for targeting ARMv4T was relatively trivial, as it was a binary compatible upgrade. The changes consisted of specifying ARMv4T as the compilation target in the build system configuration file, in addition to altering the conditional compilation clauses introduced in the SHMAC port to be dependent on `__ARM_ARCH_4T__` instead of `__ARM_ARCH_3__`. The *system mode* introduced in ARMv4T could also be considered used. Since the Barrelfish CPU drivers are by design single-threaded and nonpreemptable to simplify the kernel implementation, and since there was currently no need for nested interrupt handling, system mode is currently unused in the SHMAC port.

5.4 Shared Memory Allocation

This section will present the support for shared memory allocation which was added in this dissertation to Barrelfish to support running it on multiple cores on the SHMAC with caches enabled.

5.4.1 Shared Memory Allocator Structure

To simplify allocation of shared memory in applications, the shared memory allocator created in this dissertation was integrated into the existing *mem_serv* application. This section will describe how the allocator represents the available shared memory resources; how it is initialized; and how it operates.

Resource Representation The memory map of the SHMAC supports up to eight different scratchpad tiles. These tiles are represented with their own struct variable in the allocator source code, with minimum and maximum addresses set according to the values specified in the SHMAC memory map. Each variable structures information on the BRAM available on the tile it represents into buckets, where each BRAM region of size 2^i is placed in the bucket with index i. Since capabilities may only refer to regions which has a power-of-two size — the size of each region is actually specified as the binary logarithm of its size rather than its size — this will always be accurate. If there are several regions of the same size, they are stored in a linked list.

The resulting structure is exemplified in Figure 5.4. For simplicity, the figure only includes two tiles. The buckets in the tiles are numbered with their index, indicating the binary logarithm of the size of the regions they point to. Filled circles indicate NULL-pointers. The first tile has 32 KB BRAM available, split into two $2^{14} = 16$ KB regions. The second tile has 64 KB BRAM available, available in a single 2^{16} memory region.

Currently the number of buckets is set to 20, allowing BRAM regions up to size 512 KB as this has been the largest BRAM tile size used so far. This constraint may be easily altered by setting the value of a macro, and Barrelfish will issue an error message and abort the boot if a BRAM region larger than what the implementation allows is passed to it by the kernel. As such, there is no practical drawback with limiting the supported BRAM region size to the practical maximum instead of the theoretical maximum.

Initialization Supporting allocation from BRAM required transferring information about what BRAM resources are available from the BSP kernel to *mem_serv*. Similar to how this information is transmitted for regular RAM, as described in Section 2.2.2, the BSP kernel boot procedure was extended by setting up capabilities to the available BRAM resources, add these capabilities to the capability space of *init*, and add memory region descriptions to the *bootinfo* structure. With the ultimate goal being location-aware allocation, it was necessary to separate the management of the shared memory from the regular memory allocator. To make sure that the BRAM capabilities were not added to the regular memory allocator during initialization, a new region type called *RegionType_UmpRam* was added. Thus, *mem_serv* could

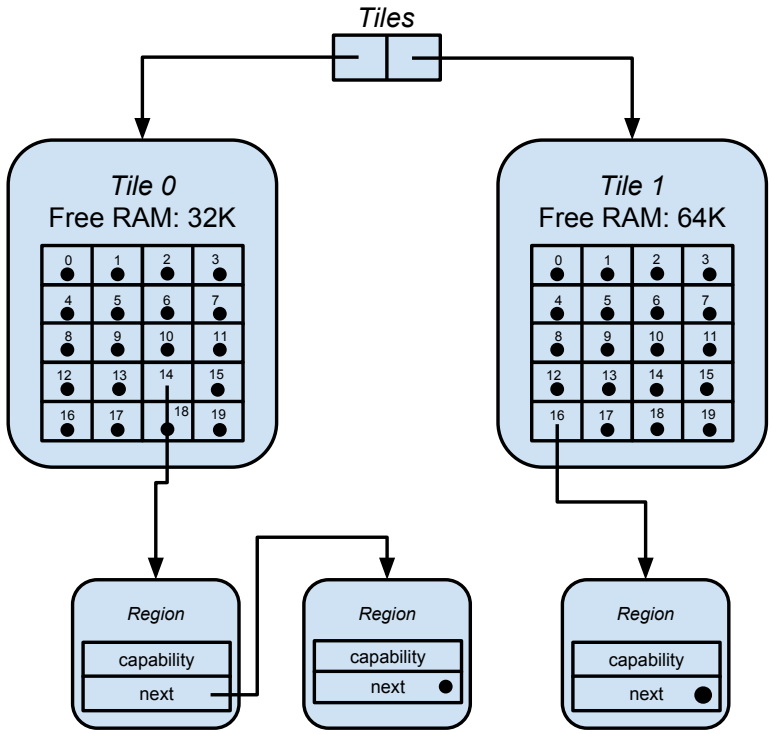

Figure 5.4: The representation of available shared memory in the shared memory allocator.

separate between memory regions of type *RegionType_Empty*, managed by the regular memory allocator, and regions of type *RegionType_UmpRam*, managed by the new BRAM allocator.

When the BSP kernel creates capabilities to the BRAM, it is important to not hand away capabilities to the BRAM used to pass app kernel boot parameters and for containing the printf lock. Special care is therefore taken in the BSP kernel to allocate what is required for these purposes first, and allocate capabilities to the rest of the BRAM afterwards. This is only a concern for the first scratchpad tile; no resources on the other tiles are reserved for any particular purpose, and as such capabilities may be created to their entire address space.

When initializing the shared memory allocator, the *RegionType_UmpRam* regions are enumerated. Each region is added to the appropriate tile structure by checking which address space it falls into, based on the tile minimum and maximum addresses, and entered in the appropriate bin based on its size.

Operation The work of the shared memory allocation algorithm is split in two:

1. First, find the best tile available. When not considering the tile location, the main goal is finding a memory region of the exact size to avoid memory fragmentation. A reasonable strategy is returning the first tile amongst those considered, to keep the memory unfragmented in as many BRAM tiles as possible. Other algorithms may be more desirable if tile locations are considered, as discussed in Section 5.4.3

2. Second, allocate a memory region from this tile. Once the optimal tile has been selected, the only objective for allocation within a tile is finding the region with the best fit. As memory allocation requests send the logarithm of the amount of memory required, this size can be used as an index into the bucket list to check if any region which exactly matches the request is available. If not, progressively higher-index buckets are checked until the first bucket containing a memory region of a sufficient size is encountered. If the region in this bucket is too large, the region is split in two. One of the resulting regions is stored in the lower-numbered bucket. The other is returned if the size matches what was requested; if not, the split procedure just described is repeated until a matching region size is acquired.

If the best tile available does not have sufficient memory, the search is repeated while disregarding the tile just considered. If no tiles have sufficient memory, an error value is returned to the caller.

As an example, consider an allocation request for a memory region of size 2^{15} bytes to the memory allocator instance illustrated in Figure 5.4. With no advanced tile selection algorithm, tile zero will be considered first for providing the BRAM. Since the requested size is 2^{15}, the first bucket considered is the bucket with index 15. Since this is empty, buckets 16 up to 19 are also considered. Since all are empty, the tile is determined to not be able to service the memory allocation request. Note that even though there is sufficient BRAM available in the tile, since it is split in two regions of size 2^{14} no single capability will serve as the return value to the caller. Since Barrelfish does not support merging capabilities, the tile cannot satisfy the request.

Since tile zero had inadequate resources, the next tile to check is found from the remaining tiles—in this case, tile one. Since tile one has a capability to a region of size 2^{16}, the region is split into two regions, and a capability allocated to each. One of the regions is stored into bucket entry 15 of tile one. Since the other region with size 2^{15} matches the allocation request, its capability is returned to the caller.

5.4.2 Bootstrapping Memory Allocation

As explained in Section 2.2, memory allocation in Barrelfish is done by sending messages to the `mem` service hosted on the BSP. For applications to allocate memory in this way, they first need to allocate a message-passing channel to be able to send messages to `mem`. Thus, it is necessary to somehow bootstrap the memory allocation by allocating a channel to `mem` by some other means.

In Barrelfish, initial memory allocation is handled by using a set of pre-allocated frame capabilities stored at a known location in the application's capability tree. Initial memory allocation requests then simply allocates regions sequentially from this set of capabilities, disregarding any specified preference of memory address location.

When applications running on app-cores want to bind to the `mem` service, the message-passing channel must be allocated from shared memory. However, trying to allocate this memory using the standard initial memory allocator in Barrelfish will not work since it disregards any memory area specifications. The same problem holds true for app-core monitors, when trying to set up the initial communication channel to `monitor_mem`.

The issue was solved by exploiting the existing message passing channels between the `intermon` service on the BSP core and the app-core monitors. These are bootstrapped as described in Section 5.2.1, and enable app-cores to communicate with the BSP core running the `mem` service. Initial shared memory allocation requests can then be made by sending requests to the BSP `intermon`, which forwards the requests to `mem`. To support this, the `intermon` interface was extended with an extra method to allocate shared memory.

Figure 5.5 illustrates how this method is used to set up a connection between an app-core monitor and the `monitor_mem` service. The app-core monitor allocates a shared memory channel using the `allocate_shared_memory` method in the BSP `intermon` service. The newly allocated channel is used to set up a connection to `monitor_mem`, which services further allocation requests.

The BSP `intermon` connection is similarly leveraged to allocate the channel between non-monitor applications and `mem`. Since applications do not have direct access to the `intermon` binding, only to their own local monitor, the `monitor` interface was also extended with an `allocate_shared_memory` method which delegates the work across the `intermon` binding. This procedure is illustrated in the first step in Figure 5.6, where the first `allocate_shared_memory` call is between the application and its local app-core monitor.

An additional difference between bootstrapping memory allocation in app-core

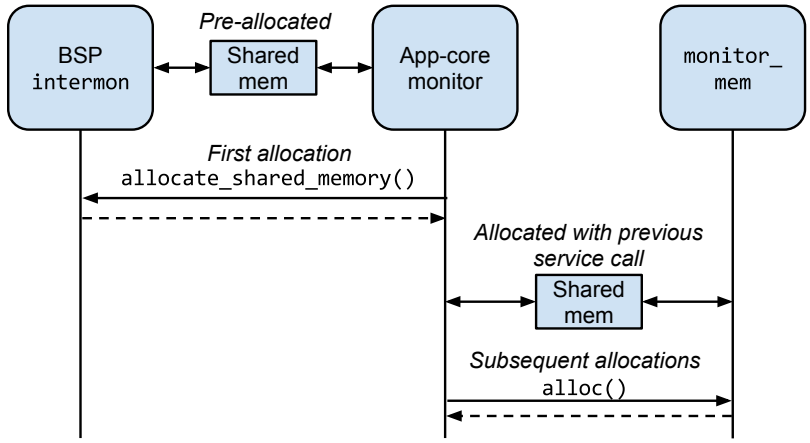

Figure 5.5: To set up the binding to `monitor_mem`, the monitor uses the existing binding to the BSP `intermon` to allocate a message-passing channel in shared memory. Subsequent allocation requests can be sent to `monitor_mem`.

monitors and other app-core applications is how the `allocate_shared_memory` call is used. For regular applications, the `monitor` method is used to allocate a single channel, which is added as a special shared-memory capability in the initial memory allocator of Barrelfish. When the connection to `mem` is subsequently being initialized, the shared memory request to the initial allocator is serviced using the previously allocated channel memory. The reason the allocation is done beforehand is that the connection setup to `mem` is done as part of a message handler. This is illustrated as the second step in Figure 5.6, where the shared memory is requested when handling a response from a message requesting the ID of `mem`. Since the same application may only handle a single message at a time, the message handler cannot request shared memory by sending messages and waiting for a reply. Because the connection to `mem` is set up during the Barrelfish library initialization, however, the limitation is surmountable as it is possible to verify that no other shared memory allocation requests will happen before the connection to `mem` has been completed. Subsequent allocations can therefore be serviced through the connection to `mem`, which is illustrated in step three of Figure 5.6. The bind to `monitor_mem` in the monitor is not executed as part of a message handler, and does therefore not suffer from the same restriction. Thus, initial requests to shared memory in monitor is handled by sending allocation requests to `intermon` when necessary, instead of allocating the required memory beforehand.

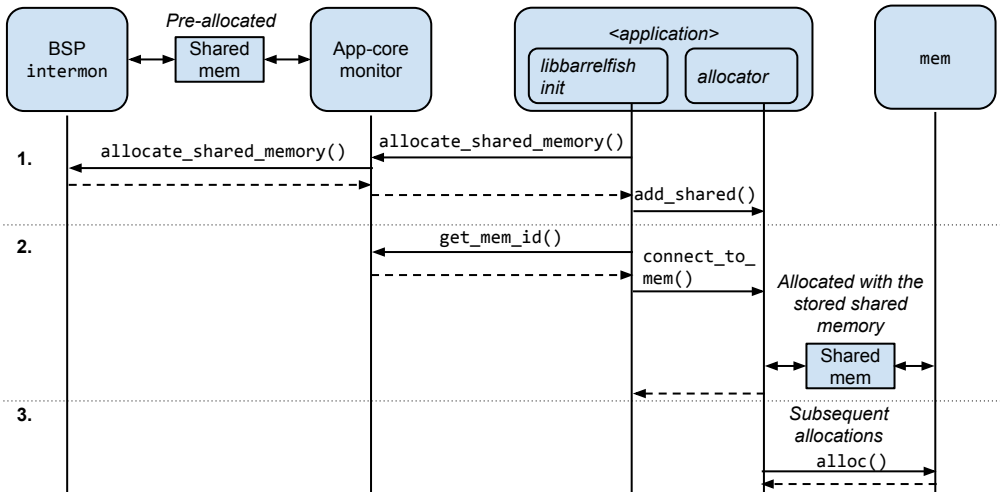

Figure 5.6: Connecting an application to `mem` is done in two steps. First, *libbarrelfish* initialization requests channel memory through its local monitor and stores it with its initial allocator. Second, as part of a message handler the connection to `mem` is initiated using the shared memory stored in the allocator. Subsequent allocations use the connection to `mem`.

5.4.3 Location Awareness Support

A central idea behind the scratchpad tiles is that if the cores use tiles in their proximity, communication may be efficient without special message-passing support. To be able to realize this idea, it is necessary to take core location into account when allocating the shared memory. This required two enhancements to the general shared-memory allocation support

1. Informing the BRAM allocator of which tiles will be using the memory, i.e. enhancing the shared memory allocation requests.

2. Using this information in the BRAM allocator, i.e. enhancing the shared memory allocator itself.

Enhancing Shared Memory Allocation Requests An assumption made when enhancing the requests, is that only two cores will communicate using a shared memory channel. This assumption holds for all known uses of shared memory in Barrelfish, most importantly inter-core communication which always uses dedicated point-to-point channels. Thus, instead of allocating and transmitting an entire array

of core location data, only two parameters were necessary in the allocation interface methods.

Another decision simplifying the allocation requests, is that the request may contain the IDs of the cores involved in the communication instead of their tile locations. A core can not in general determine the location of other cores, so the translation logic is instead gathered in the BRAM allocator. This also makes the interface method simpler, requiring two extra arguments (core IDs) instead of four (two pairs of x-y coordinates).

To facilitate transmitting this information, an extra method called `allocate_shared` was added to the `mem` interface. The method is similar to the existing allocation method, but replaces the minimum and maximum address arguments with two core IDs. The methods introduced in order to bootstrap memory allocation were also expanded to include two core ID parameters.

To make use of the new `mem` method, *libbarrelfish* was extended with a function for allocating frame capabilities from shared memory. Instead of specifying the address range from where you wish to allocate memory, allocating a frame, and resetting it, it is possible to instead call the shared frame allocation function directly. It has the same signature as the normal frame allocation function, with the addition of a parameter specifying with which core the memory is to be shared. Instead of calling the regular RAM allocation function, it calls a new, architecture-specific function called `shared_ram_alloc`. In SHMAC, this allocator checks whether a connection to `mem` has been established. If it has, it uses the shared memory allocation method in the `mem` interface, passing its own core id and the supplied core ID along with the regular allocation parameters as arguments. If the `mem` connection has not been set up, this indicates that the request for a shared memory frame arrived during bootstrapping of memory allocation. The shared RAM allocator then uses the normal RAM allocator with address range limits set to the uncached address range, since the bootstrapping memory allocator previously was enhanced with support for using the shared memory allocation methods in `monitor`. The bootstrapping allocators were also modified to pass the necessary core IDs on to the `monitor` methods.

Finally, all shared-memory allocation sites were modified to use the new allocation function. This requires determining which core will be sharing the memory with the allocator. When allocating shared memory when starting new cores, this is possible since the monitor knows which core it is booting. For generic UMP channel creation, this information is not explicitly available. However, the channel creation routine gets the ID of the service the channel is a connection to as an argument. We exploit the fact that the core ID is contained in the top 24 bits of the service ID to determine which core will use the other end of the channel. Shared memory is also used for bulk

transfer, but no existing means was available for determining the ID of the sharing core when creating bulk transfer areas. The current implementation therefore simply assumes that core zero is the sharing core ID. This holds at least for the Barrelfish system applications used in SHMAC, as bulk transfer is only used when connecting to the `ramfs` service which runs on the BSP.

This library function works well for applications which connect to `mem` to do their memory allocation. However, the monitor uses the connection to the BSP `monitor_mem` to perform allocation. Instead of creating a special-case check in the general library function, the monitor handles shared memory allocation itself by employing the method introduced in the `intermon` interface to bootstrap memory allocation.

Enhancing the Shared Memory Allocator To enable location-aware allocation algorithms, the shared memory allocator had to be supplied with SHMAC layout information. Since there is no method exposed by the hardware for determining this, the bootloader was enhanced to pass this information as a boot parameter to the BSP kernel. The updated boot parameter structure is illustrated in Figure 5.7. The first field added was the amount of memory in each scratchpad tile. Although this size is still hard-coded in the Barrelfish source tree, where the configuration file is located, it arguably makes more sense to hard-code it in the configuration file than in the kernel source code. Next, the SHMAC layout is passed as a NULL-terminated string, using the same format as described in Section 2.1. The parameter to the shmacfish bootloader generator is the path to a configuration file. This may be set either to the file in the SHMAC source tree, or a local configuration file.

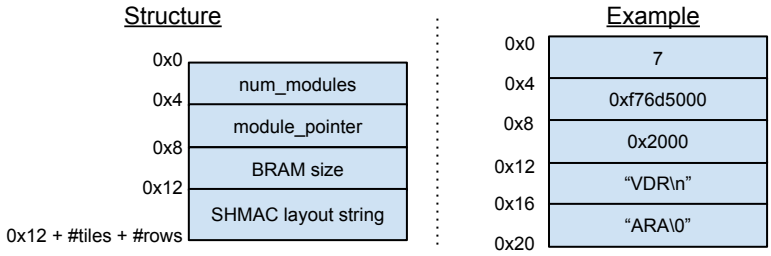

Figure 5.7: The BSP kernel boot parameter structure after adding SHMAC layout specification.

As the shared memory allocator gets information on the available memory regions based on the capabilities it is provided, the kernel does not have to transmit the BRAM size explicitly to the allocator. The layout string, on the other hand, must be explicitly handed to the *mem_serv* system application. The string is therefore first copied into a separate memory frame to which a capability is created. This capability is sent along with the capabilities describing BRAM regions. The shared memory

allocator can then access the string by mapping the memory from this capability into its address space. Using the layout string, the allocator updates the coordinates of each tile in its tile list. It also maintains an array of core tile locations, indexed with the core ID.

When a memory allocation request arrives, the allocator uses a lazy in-place selection sort algorithm to determine which scratchpad tile should be used to service the request. The sort criterion is determined by the ranking function `find_optimal_tile_among(**tiles, core1_loc, core2_loc)`, which returns a pointer to the tile array element in the `tiles` argument deemed to be most suitable. An attempt is then made to allocate memory from this tile. If there is sufficient memory, the allocation request completes. If not, then the best tile is swapped with the tile at the front of the list under consideration. The search then proceeds on the tail of the list, disregarding the first tile.

The algorithm is demonstrated in Figure 5.8. Assume that there are four scratchpad tiles available, and based on some criterion the tiles are ranked as indicated in the figure. Also assume that the two most fitting tiles do not have sufficient memory available to satisfy the request, while the third most fit tile has sufficient memory.

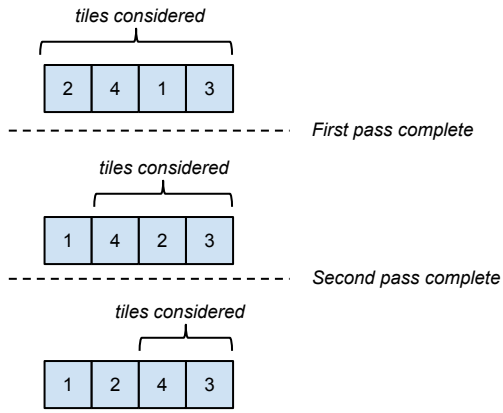

Figure 5.8: An example demonstrating how the lazy selection sort of tiles works.

In the first pass, all tiles are passed to the ranking function for consideration. The ranking function will return a pointer to element number three, as this tile is deemed the most fitting. Since the tile does not have enough memory, it is swapped with the first element in the array, and the first pass completes. Next, all but the first element in the tile array are passed to the ranking function, Again, a pointer to array element number three is returned, and again it is swapped with the front element of the list under consideration as it had no space available. This completes the second pass.

In the third pass, only the two remaining tiles are considered. A pointer to element number four is returned from the ranking function, as this is the best fit among the tiles considered. Since this tile has sufficient memory available, the allocation request completes without swapping its place.

The division of labour in the allocation algorithm makes it simple to experiment with different tile prioritizations, since it is only necessary to implement variants of the ranking function. The following are three algorithm options, of which the first two have been implemented:

1. Always return a pointer to the first tile element. This approximates the best-fit algorithm, since the most segmented scratchpad tile will be considered first as long as no allocated regions are subsequently returned to the allocator. This is because tile zero is the only tile which is not a whole power-of-two size from the start. Since the first element is always returned, there will be no permuting of the tile order in the list, so the most segmented tile will always be considered first. However, location is not considered.

2. Return a pointer to the tile element which is on a shortest path between the tiles through a scratchpad tile to minimize expected latency and energy overhead when routing access requests. In a mesh topology, the distance between tiles is $|tile1_x - tile2_x| + |tile1_y - tile2_y|$, so the best tile will be the tile with a minimal value for the expression $|core1_x - bram\_tile_x| + |core1_y - bram\_tile_y| + |core2_x - bram\_tile_x| + |core2_y - bram\_tile_y|$.

3. Consider the needs of other cores in an attempt to optimize the total system performance. For instance, use the same metric as in algorithm option 2, but in the case of draws select the tile which is the least popular alternative for other cores. Alternatively, it might be beneficial for total system performance to select the next-best tile if the best tile is the only viable alternative for another communicating core pair. No such algorithms have been implemented, but may merit further investigation as research questions when it comes to memory management in grid Non-UMA (NUMA) systems.

5.5 Implementing User-Space Console

The facilities in Barrelfish for getting access to a serial terminal are mostly architecture-agnostic. The architecture-specific enhancements required are two-fold:

1. Adding interrupt registration and delivery facilities in the kernel.

2. Adding a system call for reading the input data. The user space serial driver cannot read the data itself, since reading the data is the way to clear the interrupt in the current SHMAC infrastructure. Additionally, the interrupt notification mechanism in Barrelfish does not currently permit sending data along with the interrupt request.

An illustration of the common interactions between the SHMAC user space serial driver and the kernel is given in Figure 5.9. Four scenarios are illustrated. The first is how the serial driver registers an interrupt listener. Interrupt delivery to user space processes in Barrelfish is done by delivering empty messages to dedicated LMP endpoints. From the perspective of the program trying to register an interrupt handler, the registration is performed through a message to the monitor running on the same core as the application trying to register the interrupt handler. The LMP endpoint is passed to the monitor, as illustrated in the figure. The monitor then requests the kernel to deliver the specific interrupt to the given endpoint. Authorization for making such requests is ensured by passing an IRQ capability, owned by the monitor as illustrated in the figure, with the system call. The syscall is as such invoked with the IRQ capability and the LMP endpoint as parameters, the figure illustrates. When the syscall is invoked, the kernel checks that the IRQ capability is valid and that no other listener is registered. If these checks pass, the listener endpoint is stored in the kernel so that interrupt notifications may be delivered to it. The modifications implemented in this dissertation to support this step amounted to adding the IRQ capability to the initial capability space of the monitor, and implementing the syscall.

The second scenario in Figure 5.9 illustrates how the serial driver is notified of interrupts. In addition to the LMP endpoint, the kernel maintains an input character buffer and a flag denoting whether the listener has been notified of an interrupt. When the kernel receives a serial interrupt, it stores the input character in the buffer. In the figure, an example serial interrupt is received with the input character 'y'. Since the input buffer was empty, the letter is stored as the first character in the buffer. Next, the kernel checks whether the registered listener has been notified of an interrupt previously. If no interrupt has been delivered, which is the case in scenario two in Figure 5.9, a notification is delivered to the registered endpoint and the interrupt notification flag is set to *true*.

Since handling the LMP notification is done in user space after scheduling the serial driver for execution, several interrupts may occur before the user space driver has had a chance to react to the first interrupt. This situation is illustrated in the third scenario in Figure 5.9. In the figure, the kernel gets a serial interrupt with the letter 'o'. This is stored at the end of the input buffer. Since the notification flag is set to *true*, however, no extra notification is delivered to the LMP endpoint.

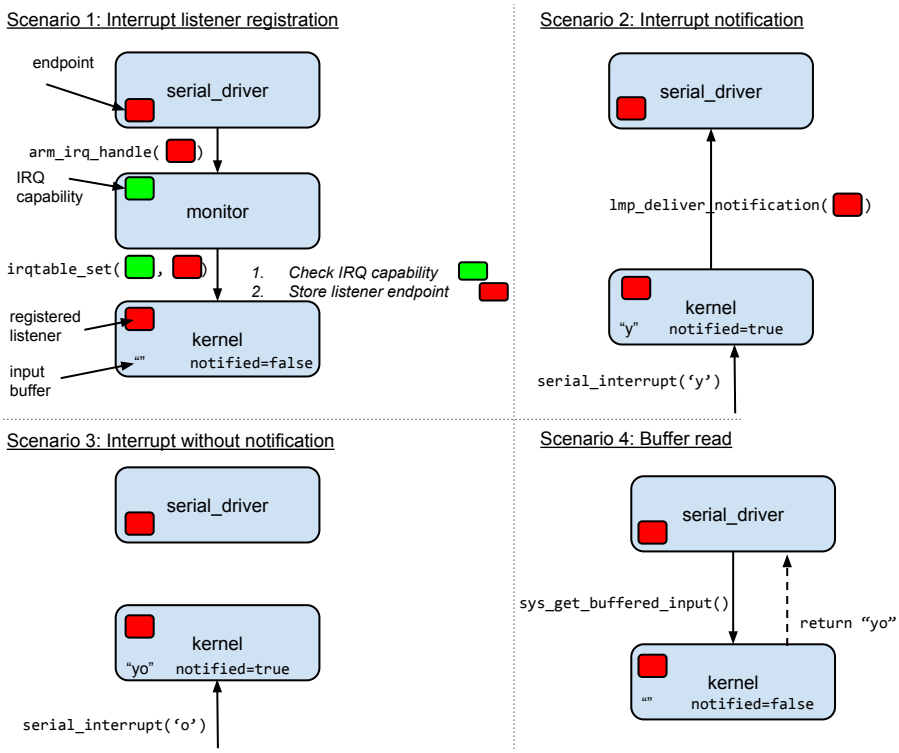

Figure 5.9: An overview of four scenarios in which the SHMAC user space serial driver and the kernel interact with each other.

The fourth scenario illustrates how the serial driver retrieves the input characters using the system call `sys_get_buffered_input`, which was added to the SHMAC kernel in this dissertation. This system call copies the contents of the input buffer to a supplied user-space buffer pointer. In the figure, this causes the input buffer contents "yo" to be returned to the serial driver. Finally, the input buffer and notification interrupt flags are cleared.

Chapter 6
Evaluation

This chapter will present an evaluation of the systems which have been implemented. Section 6.1 will evaluate the functional correctness of the Barrelfish port to SHMAC. Section 6.2 will evaluate important properties of the energy efficiency estimation framework, such as accuracy and performance, as well as functional correctness.

6.1 Barrelfish

Previous work on the Barrelfish port developed a stability test, designed to exercise various portions of Barrelfish including message passing, memory allocation, and context switching [BS13]. The same test program was used in this dissertation to test the stability of the multicore extensions. The test launches three programs, `calc`, `verify`, and `print`, whose communication pattern is shown in Figure 6.1. The `calc` program performs a computation on a randomly sized array, and sends both the array and the result in a message to `verify` which verifies the result by independently performing the same computation. Both programs then send the array and result to `print`, which emits output to the screen. Any errors in verification or other system crashes will cause the program to stop; continued operation indicates stability in the exercised subsystems. Since the test makes heavy use of message passing, launching the three programs on different cores makes the test exercise both the multicore initialization and inter-core message passing developed in this dissertation.

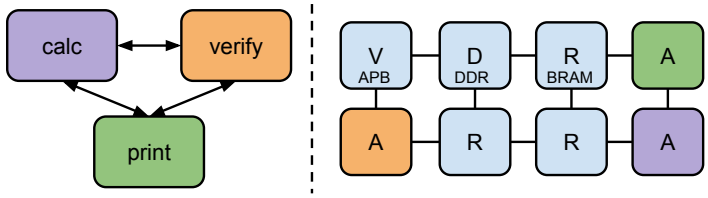

Figure 6.1: The programs included in the stability test of Barrelfish, and their placement on cores on the SHMAC instance which was used during testing.

The test was run on a 2×4 SHMAC configuration with three Amber cores. The distribution of programs to cores is indicated by the colouring scheme in Figure 6.1. The test was first run once without cache support, and therefore not using scratchpad memory, and once with caches on when shared memory support had been implemented.

Without caches and scratchpad memory, we ran the stability test for several days without failure. This indicates that initialization of new cores and the code relating to the ISA upgrades is implemented correctly. When scratchpad memory support was included, the stability test was unfortunately hampered by a bug in the SHMAC platform causing Amber cores to occasionally freeze for unknown reasons on accesses to BRAM[1]. Using the test setup in Figure 6.1, the test program runs for approximately one minute before a freeze occurs. Moving the `calc` program to the same core as `print`, using only two cores, lessens the pressure on BRAM and makes the test run for approximately one hour before freezing. This bug has not yet been fixed, and it is outside the scope of this dissertation to do so. Therefore, the verification of functional correctness of the Barrelfish port is limited to partial completion. The shared memory support is only responsible for the initialization of channels, however. This is extensively tested by the time the test programs start, through the creation of shared-memory channels both between the test programs and between Barrelfish services. Therefore, the tests indicate that the port works as expected.

The user-space console support was separately tested simply by starting the shell program after system boot, and running commands using it. This test was run successfully, verifying the ability to transfer text from the kernel to the registered user-space listener.

6.2 Energy Efficiency Estimation Framework

In this section, we evaluate the quality of the energy efficiency estimation framework as per task `E3`. Where applicable, we will separately evaluate the quality level supported by the generic infrastructure and modelling methods, and the quality of the models which have been implemented. Due to the current volatility of the SHMAC project, the former is of greater importance as it reflects the ability of the framework to adapt to modifications or enhancements to the SHMAC infrastructure. The quality of the models which have been implemented demonstrate to what degree the theoretical potential of the infrastructure may be exploitable.

[1] The fact that there is a hardware fault was ascertained by using the energy efficiency estimation framework to ship out internal signals as debug information. This revealed that the memory stall signals of the frozen Amber core was constantly high, with the address requested being in BRAM.

6.2.1 Accuracy

This section will discuss the accuracy of the models which have been implemented. We will first discuss the trueness and then the precision, as the terms were defined in Section 1.4 and Appendix A.

Trueness Certain aspects of the modelling present a definitive threat to the trueness of the models. The impact of the following points should be considered before drawing conclusions from experiments conducted using the SHMAC platform with the new energy efficiency estimation framework.

- The SHMAC HDL implementation, and the Amber core in particular, is originally tailored for FPGA synthesis [Ope13]. It may therefore be undesirable to consider an ASIC realization of the SHMAC HDL exactly as it is written to be the true system, as there are different microarchitectural trade-offs between FPGA and ASIC designs [WBR11]. Since the models are created using the HDL, any differences between the HDL specification and the true system would cause discrepancies between the model output and the true value, To take one example, register files in ASIC processors may be created using specially-crafted SRAM blocks whereas the Amber register file is implemented with flip-flops and multiplexers. There would likely be significant discrepancies between the model and the true system if the latter was assumed to use SRAM register files, since the register file is responsible for approximately 70 % of the energy consumption of the execute stage in the Amber core as is reflected in its model in Appendix D.1.
- The CACTI analysis of cache and scratchpad memory power consumption is based on process characteristics which are dissimilar to those in the cell library used to synthesize the SHMAC platform. For instance, the CACTI process uses a 1.1 V supply voltage, whereas the target cell library uses a 1.0 V voltage level.
- Place-and-route details are not represented by the model. As physical implementations are placed and routed, there will be differences between the model and the true value. In particular, both the assumption that similar tiles in different locations will have the same energy consumption and that inter-tile router ports have the same energy consumption may be untrue.
- The ASIC synthesis target clock frequency is 250 MHz, but the SHMAC execution frequency is 60 MHz. This discrepancy between target and host time causes untrue representation of interaction between components which are not scaled similarly. For the SHMAC, this concerns off-chip memory requests made by Amber tiles since the off-chip RAM is not slowed. Thus, a memory request

which would require 25 stall cycles on a 250 MHz core would only require 6 stall cycles on a 60 MHz core. As a result, memory requests do not have the same impact on energy consumption and system performance as they would on a true system.

– The fact that the emulator includes an energy estimation framework and the emulator target does not, may adversely affect trueness in two ways:

  1. Terminal I/O operations may be disrupted by host sampling activity. The effect may be avoided by ensuring that terminal I/O is not used during an experiment.
  2. The energy consumption of energy monitors is not included in any estimates. This is appropriate as the monitors are not supposed to be part of the system under study, and will ensure trueness when characterizing energy efficiency under general workloads using the host interface. However, if system software algorithms make use of the energy estimates provided by the SHMAC interface then some way of obtaining such estimates is required in the true system as well. If hardware energy counters similar to those in the infrastructure are required, the cost of implementing these in an ASIC processor should be considered before drawing conclusions as to the efficiency of any proposed algorithm. Alternatively, one may lean on the fact that several studies have demonstrated methods for calculating such estimates based on event counters which are already present in existing processors [SS13a, BGM+10, SBM09]. The impact of the necessity to calculate the energy values during algorithm operation must then be estimated.

The previous points describe definitive discrepancies between the models and an ASIC implementation of a SHMAC instance. In addition, there are aspects of the model construction where arbitrary choices have been made due to the lack of an exact specification of the true system. Although these points do not constitute a threat to the trueness at present, we mention them here since it is still important to be aware of the exact characteristics of the system which has been modelled. These aspects should also be controlled explicitly if the need to faithfully represent a real implementation arises.

– The majority of synthesis parameters, such as load capacitance and ambient temperature, are accepted at default values. The models will as such represent a system which operates in such default conditions.

– The models created are specific to the target cell library. Actual ASIC production would maybe make use of a different cell library.

– By using CACTI to model SRAM resources, the SRAM modules are implicitly assumed to have the same organization and layout as those assumed by CACTI models. Specific SRAM cache and scratchpad modules may be organized differently.

– The off-chip RAM system which is modelled is based on an arbitrarily selected DDR RAM module, and implicitly models a simple memory controller by assuming that its consumption is shadowed by that of the RAM module and not including any estimates for it at all. Particular off-chip RAM systems may use different RAM modules, as well as a memory controller which is sufficiently heavy weight to warrant an energy model of its own.

Precision We evaluate precision quantitatively by comparing model predictions to the values which are calculated by PrimeTime. We will present precision numbers for the model for a complete Amber tile, with five decimal places used as the model coefficient resolution. The characteristics of the model of each submodule are listed in Appendix D.1.

The predicted power consumption, the actual power consumption, and the estimation error in each cycle of the validation benchmark is plotted in Figure 6.2. The plot aggregates estimates over a time window of 500 cycles, as there are too many sample points to visualize clearly otherwise. Figure 6.3 plots a subset of sample points and predictions per cycle, which enables a clearer inspection of individual sample instants. The error in the estimate of total energy consumed is 0.19 %, and the average absolute error each cycle is 1.1 %. Only 0.2 % of per-cycle errors exceed 5 % in absolute value, and 95 % of estimates have an absolute error less than 2 %.

As energy estimates read through both the host and the SHMAC software interface will report energy values summed over a number of cycles, we evaluate the impact of time window on the estimation error. Figure 6.4 plots the average absolute error when considering all time windows of a certain size in the validation benchmark results. Of the time window sizes considered, the highest average error is 1 % when the time window is 500 cycles. This indicates that the longest sequence of consistently erroneous estimates last on the order of a hundred cycles. With higher time window sizes, errors are averaged out, with less than 0.2 % average error when time windows are larger than 5000 cycles.

Figure 6.5 illustrate how the resolution of model coefficients affect the precision of the model. The resolution was varied by rounding off coefficients to their nearest multiple of the x-axis values, which is equivalent to varying the number of digits included in the decimal fraction of the coefficient. Although average absolute cycle-by-cycle error stays relatively low for lower coefficient resolution values, the 99th quantile is doubled going from five decimal places to four, and the fraction of absolute estimation errors

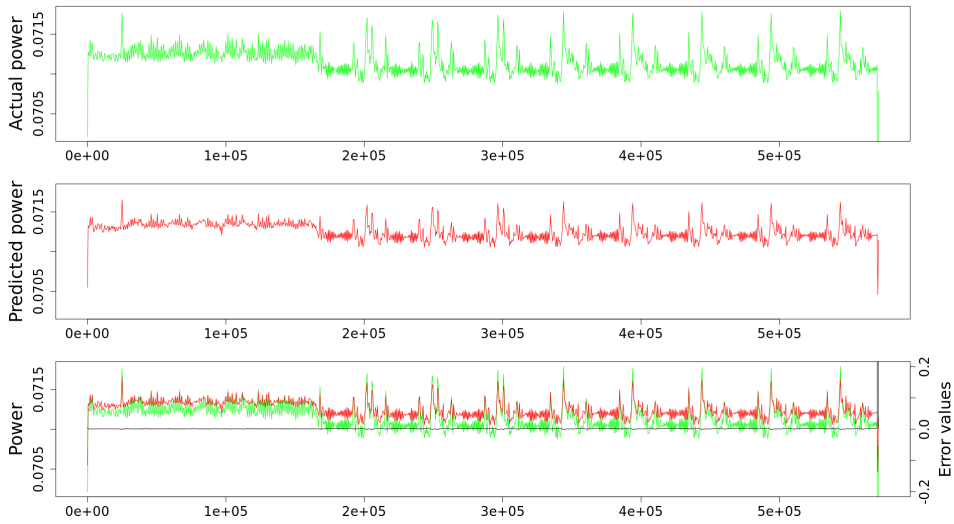

Figure 6.2: The predicted power consumption versus the actual power consumption versus the error per cycle of the validation benchmark. Power values are on the left axis, while error values are on the right.

above 5 % suffers an eleven-fold increase from 0.27 % to 3 %. From five to six decimal places, however, the difference is small.

6.2.2 Coverage

As defined in the introduction, coverage indicates how many system components may influence the reported energy consumption. We will first evaluate how the generic infrastructure supports high coverage, before turning our attention to the coverage of the models which have been implemented.

Infrastructure Coverage Support In theory, any level of coverage desirable is supported by the infrastructure. Monitors may be included in any module in the system, and any input, output and internal signals may be used as input to the module models. Therefore, coverage may always reach 100 % through incremental inclusion of new monitors. However, the attainable coverage may have practical limits due to implementation overhead. The framework infrastructure has been added to all tiles except a newly developed high-performance Amber tile [AB14] and the ZBTRAM tile, and may easily be included on these tiles as well if deemed necessary. The potential for high coverage is therefore also readily exploitable.

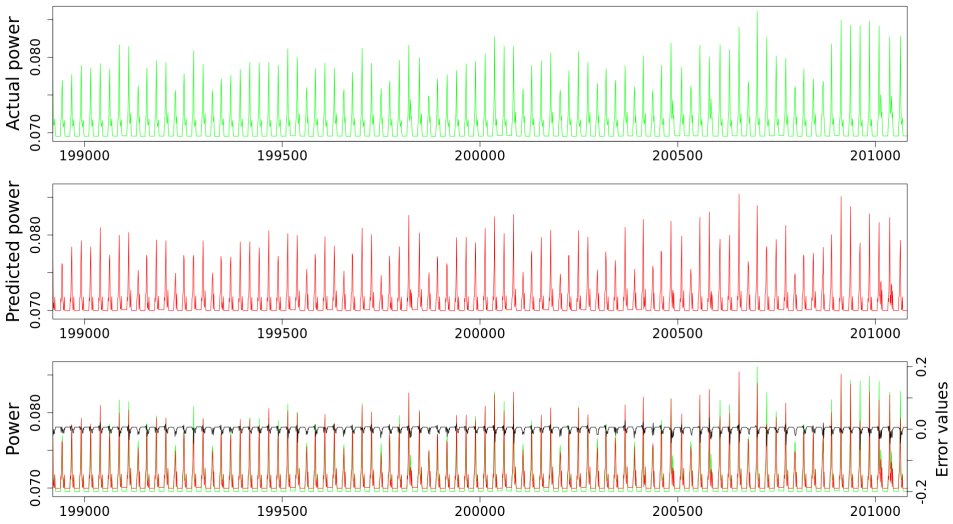

Figure 6.3: A limited set of one thousand sample points, which more clearly demonstrates the precision of the model per cycle.

Figure 6.4: The mean absolute error when considering time windows of varying granularity.

Current Model Coverage Evaluating the coverage of models is primarily interesting as an examination of the quality of the validation benchmark. Coverage by itself is not necessarily indicative of model quality: models with high coverage may still have wrong weights for different signals, and low coverage may be justified if the activity in ignored signals cause low variation in energy consumption and therefore may be represented as a constant value. Model quality is instead evaluated through the validation benchmarks. However, if the validation benchmarks do not contain sufficient activity diversity, models with low coverage may get inappropriately good validation scores. Evaluating the coverage of models may help uncover such issues.

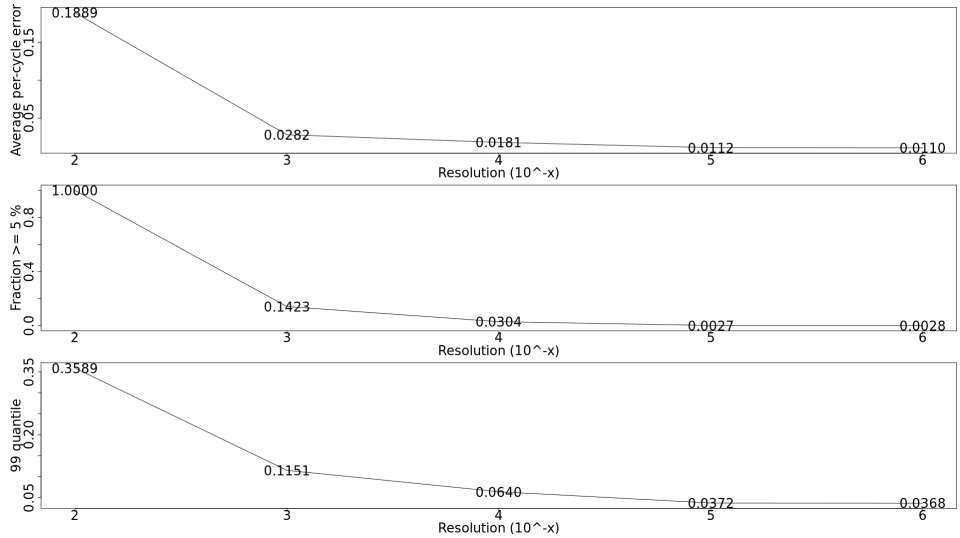

Figure 6.5: The impact the coefficient resolution has on absolute per-cycle error values.

In the current energy models, all HDL modules in the Amber, scratchpad and DDR tile are covered either through a separate monitor or through a monitor in a parent module. For instance, the execute model uses signals from the barrel shifter, the multiply unit, the ALU, and the register bank to ensure that all the components the model encompasses are represented. Thus, there are no modules whose activity is completely unaccounted for.

6.2.3 User Friendliness

We will evaluate the user friendliness for three different roles in which persons may interact with the framework: a user who merely wishes to run experiments on a SHMAC system with the infrastructure and software tools installed; a researcher who wishes to configure the number of energy monitors or energy report units; and a researcher extending the framework with a regression model for a new hardware unit.

Experimentation In order to run experiments using an existing SHMAC installation, the monitoring and logging tool is almost the only tool the researcher has to use. Running an experiment requires no more actions than starting this tool, pressing the 'Connect' and 'Start' buttons, and then logging in to the SHMAC host and launching the experiment application. Disregarding the design of the GUI, the user friendliness is high due to the simplicity of steps required to run an experiment. One shortcoming is that the framework provides no means of announcing the beginning and end of an

experiment. The researcher must therefore manually extract the relevant subset of the logged samples.

Configuration Configuring a new set of energy report units or energy monitors requires manually rewriting the HDL source of SHMAC to include or remove units as desired. This is cumbersome, albeit simple if familiar with the infrastructure. HDL source code modifications also necessitate running FPGA synthesis and installing the new bitfile for modifications to be available. This process is relatively time-consuming, taking approximately three hours. In addition to HDL modifications, the configuration in the driver must be updated with the new information and installed along with the new bitfile. This is simple and fast, but possibly easy to forget. In conclusion, although all required steps are kept simple by design, the time required and the mundaneness of the tasks result in a final evaluation score of low user friendliness for altering the energy efficiency estimation framework configuration.

Extension Extending the framework with models for new components requires ASIC synthesis, benchmark design, simulation and power estimation, regression modelling, and finally hardware implementation. The steps are largely kept as simple as possible. The use of all ASIC tools is automated, as described in Appendix B.1. Benchmark design cannot be trivially automated, but the benchmark framework described in Appendix B.2 reduce the required workload to implementing the actual benchmark program. Having written the C code implementing the benchmark, it is only necessary to run `make benchmarks` to generate signal and power data.

The regression modelling is a difficult step, as the selection of model variables and term construction is a completely manual endeavour. As with benchmark design, the workload is kept minimal by the support software described in Appendix B.3. Notably, the R library which was developed include functions for reading in benchmark data; creating a new model; evaluating a model; and plotting the evaluation in various formats.

Finally, the creation of hardware implementations of models is simple, but tedious. Generic modules for calculating Hamming distance values are available, but as HDLs are verbose, coding a formula may require writing a significant amount of code[2].

Overall, the user friendliness of extending the framework is evaluated as medium. Most tasks are automated, and the manual intervention required is largely kept as focused as possible. However, the manual work is difficult with no support for assisted design space exploration, and the creation of hardware implementations of a model could be better supported.

[2]The longest energy monitor file is 211 lines of Verilog.

6.2.4 Infrastructure Correctness

To test correct infrastructure operation, test benches were created for each hardware unit in the infrastructure. The test benches are similar to software unit tests since they are self-asserting, terminating simulation if an error condition is met and reporting what went wrong. The test benches toggle the control signals of a design under test to test specific operations, and use randomly generated data for any data input. For instance, the test for the energy counter module checks correct operation depending on whether energy read-enable signals or time window write-enable signals are set, while using randomly generated values as the energy sample input. The randomization functions are configured with constant seeds, to ensure reproducibility between different runs of the same test bench. All test benches pass, which indicates that each individual unit works as expected.

The test benches were developed using the Bitvis VHDL utility library [Bit14], which simplifies writing self-asserting test benches in VHDL as common functions for checking values, logging, and generating random values are provided. The test benches were made together with a generic test bench framework, described in Appendix B.5, which may also be useful for the development of tests for other parts of the SHMAC system.

The infrastructure was also verified to work once realised on an FPGA by creating a SHMAC design in which each energy report unit was set to report a different constant value instead of a sum. When reading sample data from this SHMAC system once installed on the FPGA, the same constant values were read out in the expected order. This indicates that the scan chain and the communication between the SHMAC and the host system works as expected.

The correct operation of energy report units and energy monitors was verified by executing a program where only the first SHMAC core is running. The energy consumption estimate for the stalled cores should be a fixed value each cycle. The value attained from the report units on the stalled cores was therefore compared to the known constant value multiplied by the sample period in clock cycles and found to be equivalent as expected.

6.2.5 Performance

In this section, we will evaluate the performance of the energy efficiency estimation infrastructure. We will first study the maximum sample rate we can get from the system, as well as sample period variability. Second, we will evaluate the hardware overhead of including the infrastructure on the system. Finally, we investigate the impact of using UDP in the monitoring and logging tool.

Maximum Sample Rate The infrastructure implementation nearly maximizes the theoretical sample rate. Since the APB is the only connection to the host, saturating the APB bandwidth maximizes sample rate. With the current host interface, only the sample operation uses more than minimal time to complete due to the four-way handshake employed to send the sample request to the SHMAC clock domain. The sample operation therefore has approximately four APB clock cycles and four SHMAC clock cycles of delay, which equals eight APB clock cycles since the clocks currently run at the same frequency. The read sample commands, however, execute as fast as the APB protocol permits. The theoretical minimum sample period is therefore eight clock cycles higher than the optimal value, under the current host connection conditions. Since one APB read request takes 2 cycles, transferring n samples requires $8 + 2n$ cycles.

The previous expression describes the hardware-imposed overhead of energy estimate transfers to the host. As the host interface relies on user space software requesting samples from the driver, however, it is not clear from the design of the hardware alone what sample rates will be supported in practice as it is affected by overhead from driver access in addition to operating system interference such as context switching. The minimum sample period attainable in practice was therefore estimated by creating a program which continuously re-reads the binary energy attribute of the driver. The program was parametrized by the number of samples to request from the driver, and run once for all integers between 1 and 11. The program was run for 10000 iterations each time, to take variability into account. The time taken is calculated from the number of SHMAC cycles spent between each sample, a metric attainable by setting the per-cycle energy of the first report unit to one. This is done because the real-time clock on the SHMAC host runs too slowly, as comparisons to other clocks confirm.

The results are plotted in Figure 6.6. For the range of sample words tested, it is apparent that the number of samples requested has insignificant impact on the sample period. The only exception is a jump between requesting two and three sample words, but there is no evident linear relationship. Thus, the driver infrastructure access appears not to be significant for the sample period. This is also apparent from the sample period values. Fetching one set of samples from the driver and computing the time required takes approximately 12 ms to complete, whereas the infrastructure in theory could support sample periods of $(8 + 2 \cdot 11)\Delta t_{APB} = 30 \cdot 16\frac{2}{3}$ ns $= 500$ ns when requesting 11 sample words. We may therefore conclude that the attainable sample period is completely dominated by software overhead, limiting the granularity at which energy consumption may be inspected.

The limited sample period causes energy report units to overflow before samples may be collected when the sample size is set to 32 bits. Since the sample period is

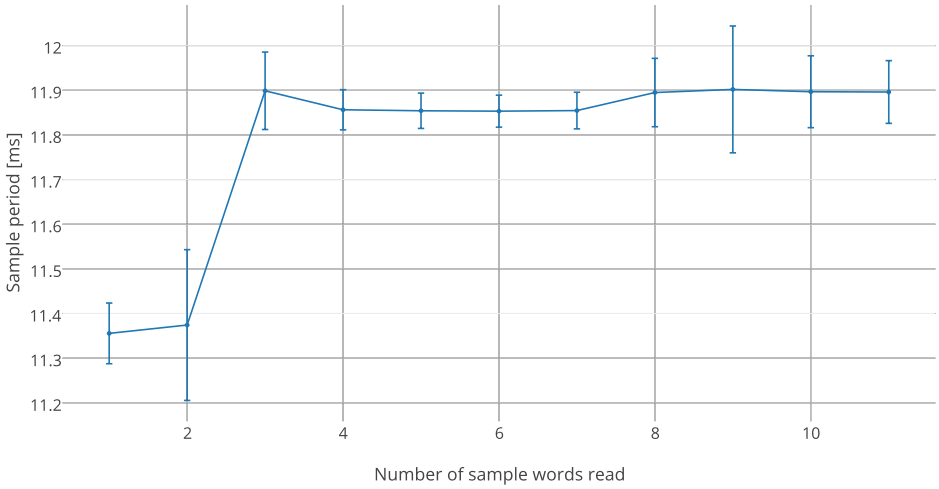

Figure 6.6: The sample period attainable from a user space program requesting a varying number of samples from the driver. Error bars indicate standard deviation.

dominated by software overhead, all sample sizes may be increased to 64 bit with little negative effect on the sample period. This is demonstrated in Figure 6.7. Apart from the difference between collecting two 32-bit and two 64-bit samples, there is no significant increase in maximum sample period going from 32-bit to 64-bit samples.

Sample Period Variability Since the infrastructure reports energy, it is necessary to know the sample period in order to derive the power. Although a specific sample period may be requested from the `shmac_stream_energy` tool, it may not be a reliable estimate of the actual period due to software execution variability. To quantify the sample period variability more thoroughly, we study the distribution of sample periods when continuously gathering 11 sample words. The experiment is run twice: first, SHMAC clock counts are stored temporarily in memory and logged to file when the experiment has ended, and second, the clock counts are logged directly to file after each measurement has been made. The first strategy quantifies the variability which is primarily outside the control of the application, while the second gives results which may be more realistic for applications.

The results are presented in Figure 6.8. Offline logging has a standard deviation of only 2500 cycles with a median of 711500, although certain sample periods spike up to a maximum of 762800. As expected, the offline logging sample period distribution

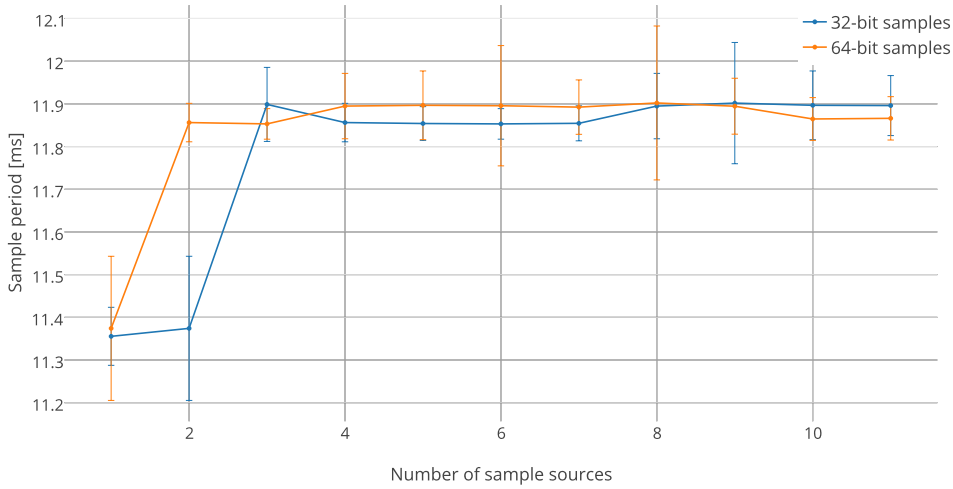

Figure 6.7: A plot demonstrating that the sample period is not significantly affected by going from 32-bit to 64-bit sample sizes.

is lower and more compact than that of online logging. Interestingly, the impact of writing to file while sampling is typically not too severe, resulting in a median value increase of only 0.63 %. However, intermittent writes to file suspect the sample period to large spikes, with a maximum value which is 32× larger the median value.

The variation in sample rate is manageable if a dummy report unit which counts the number of cycles since the last sample is included in the design. However, variability at the current scale may lead to confusing plots of energy consumption trends. This is demonstrated in Figure 6.9, which presents a screenshot of the monitoring and logging tool. The top-left graph plots the clock count each sample, whereas the graphs in the other corners plot the energy consumption of the Amber cores on the system. Each Amber core is set to execute a separate workload, leading to different energy consumptions, but the sample period variability is so considerable that such trends are overshadowed by the difference in number of energy sample values accumulated.

This confusion may be avoided by scaling the energy consumption down by the reported sample period, plotting power instead. Figure 6.10 demonstrates the effect this has on reported trends: the graph for each core now accurately reflects the energy and power profile of the workload it is executing.

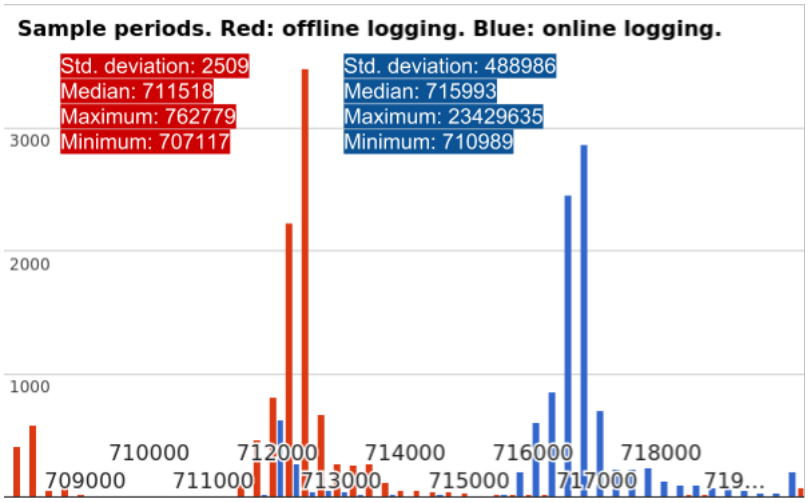

Figure 6.8: The distribution of sample periods. The red bars represent measurements when values are temporarily stored in memory, and the blue bars represent measurements taken when values are logged directly to file.

Infrastructure Implementation Overhead The results in Section 6.2.1 demonstrate that it is possible to use the regression modelling method to create a model with high precision. This section evaluates the cost of implementing a model of such precision in hardware. The models are implemented using five decimal points in the coefficients based on the analysis in Section 6.2.1.

Table 6.1 summarizes the infrastructure cost in each module which is either enhanced or new. The table lists the number of LUTs and flip-flops added to each module, and what fraction this is of the resulting module resource consumption. The final column lists average overhead, which is the increase in resource[3] consumption in each module when including the energy estimation infrastructure. In the Amber core, the average resource overhead is approximately 18 %, whereas the router has an average resource overhead of 104 % indicating that the energy monitor more than doubles its implementation cost. For the complete Amber tile with two energy report units, the overhead of the infrastructure is on average 55 %. Of the resources available on the target FPGA, the tile consumes 1.68 % of the LUTs, 0.52 % of the flip-flops, and 0.37 % of the DSPs.

The cost of each of the regression model monitors is broken down in Figure 6.11. For the router monitor, whose implementation cost is the most substantial, 95 % of the LUT cost is owed to Hamming distance calculation. For the three other non-

[3]LUTs and flip-flops.

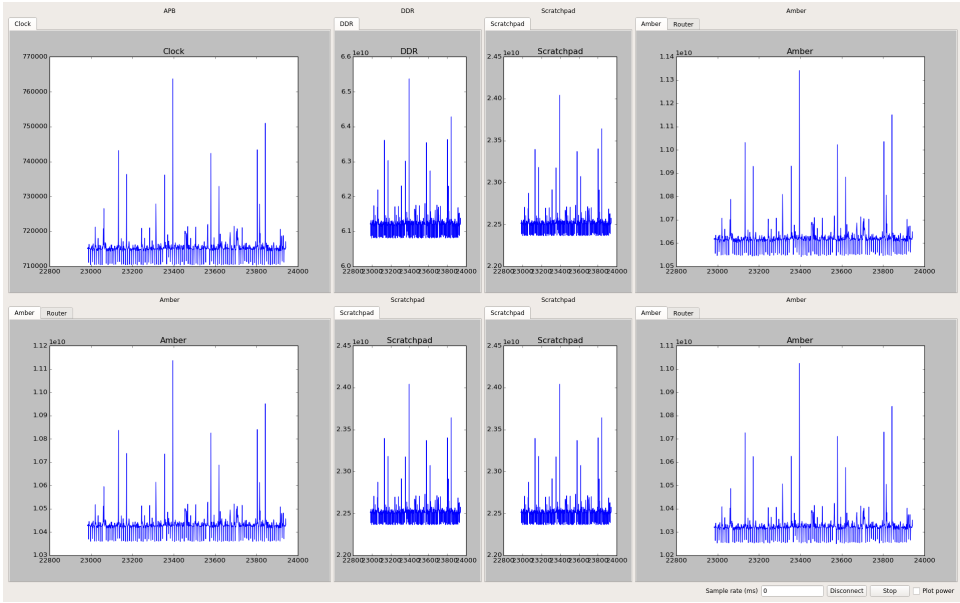

Figure 6.9: A plot of energy consumption when running different tasks on each core. Although the energy consumption differs, the sample period variability dominates the plot trends.

trivial models, namely execute, core and timer, the Hamming distance calculation is responsible for approximately 50 % of the LUT consumption. The remaining fraction is mostly spent on summing all the model terms. The cost of multiplication, included in the *other* category, is overall negligible.

Flip-flop consumption for the router again lies predominantly with Hamming distance calculation. For the execute and core stage, where an extra pipeline stage is included to register input values, the pipeline register cost is the greatest at approximately 50 %.

The possibility of reducing the infrastructure overhead through optimization of the Hamming distance calculation was investigated by implementing the FPGA-specialized technique for calculating the Hamming weight of 36-bit wide buses described by Sklyarov *et al.*[SS13b]. The difference between this technique and the earlier solution is depicted in Figure 6.12. The technique uses a second layer of LUTs to calculate the number of most, middle and least significant bits from the six three-bit outputs from the first layer of LUTs, which reduces the required number of adders. Even this optimization only reduces the LUT consumption by 200 LUTs, which amounts to reducing the overhead in the Amber tile to approximately 53.5 %.

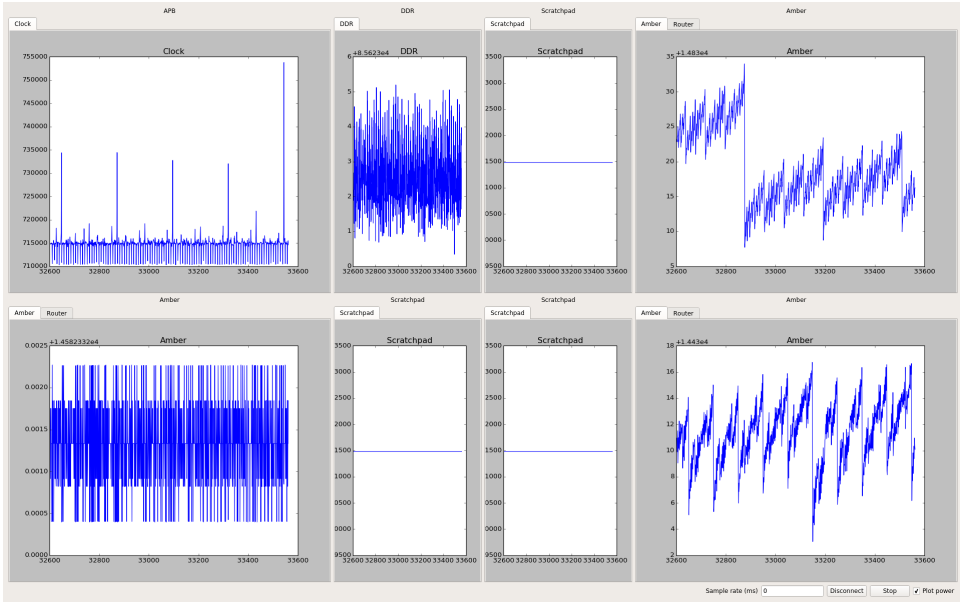

Figure 6.10: A plot of power consumption when running the same program as in Figure 6.9, demonstrating how normalizing reported energy consumption by the sample period may reveal energy efficiency trends in spite of sample period variability.

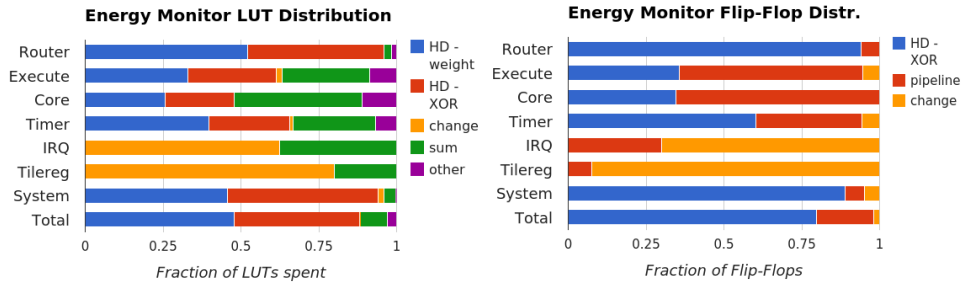

Figure 6.11: The distribution of resource usage in the implementation of regression models.

Monitoring Tool Network Protocol Selection To quantify the effect of using UDP as the network protocol for the monitoring and logging tool presented in Section 4.2.6, we estimate the potential packet loss under near optimal conditions as well as the sample rate difference between a UDP and TCP implementation.

For the packet loss experiment, the developer machine was connected to the NTNU

Table 6.1: Infrastructure implementation resource cost. The percentages in the first two columns indicate the fraction of the resource usage in the unit which is due to the infrastructure.

Unit	**LUTs**	**Flip-flops**	**Average Overhead**
Instruction Cache	1 (00.30 %)	3 (01.10 %)	0.70 %
Execute	563 (09.41 %)	450 (22.95 %)	19.30 %
Data Cache	6 (00.74 %)	12 (03.33 %)	2.08 %
Amber Core	920 (10.69 %)	742 (19.19 %)	17.56 %
Tile Registers	7 (14.29 %)	18 (28.57 %)	27.28 %
Timer Module	615 (59.19 %)	252 (62.22 %)	154.49 %
Interrupt Controller	9 (05.81 %)	21 (11.41 %)	9.42 %
Energy Counters	440 (100.0 %)	417 (100.0 %)	N/A
Amber System	2293 (21.13 %)	1200 (23.43 %)	28.67 %
Router	5614 (65.77 %)	2704 (36.19 %)	104.00 %
Energy Report Unit	65 (100.0 %)	64 (100.0 %)	N/A
Amber Tile	8037 (39.06 %)	4032 (31.64 %)	54.68 %

internal network using a network cable. The experiment was conducted near Christmas, at which point in time the network activity may be assumed to be relatively low. Under these conditions, out of 100318 packets sent from the SHMAC host to the developer machine none were lost. These results indicate that UDP may, at least through environmental control, serve as a sufficiently reliable transport protocol.

To investigate what sample rate benefits are attained by using UDP over TCP, a TCP implementation of the tool was created. The UDP and TCP implementations were run for 20 minutes, and the number of samples sent were counted. The results are tabulated in Table 6.2. Using UDP, a 1.15 % increase in the number of samples sent was observed. The benefit of using UDP is as such relatively inconsiderable.

Table 6.2: The number of samples sent during 20 minutes of monitoring using different transport protocols, and the corresponding average sample period.

	Samples sent during 20 minutes	Average sample period
UDP	98749	729 121 SHMAC cycles
TCP	97631	737 470 SHMAC cycles

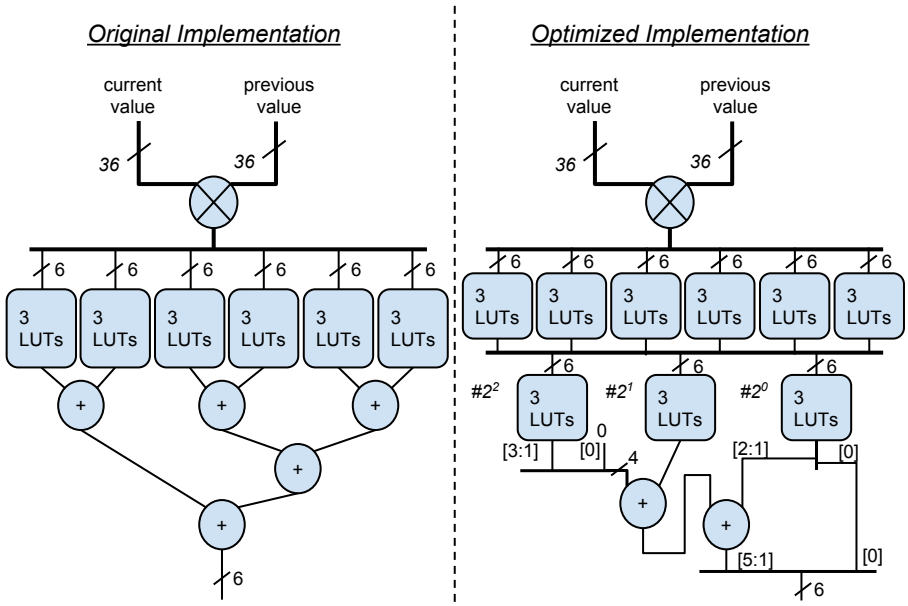

Figure 6.12: The optimized hamming distance calculation implementation from [SS13b], compared with the original Hamming distance calculation implementation, for a 36 bits wide bus.

Chapter 7
Discussion

In this chapter, we discuss important aspects relating to the current implementations. Section 7.1 raises the question of how suitable Barrelfish is as an operating system with which to conduct research in the SHMAC project. Section 7.2 discusses the impact of planned modifications to the SHMAC infrastructure. In Section 7.3 we investigate in what way trueness is important, and the implications on how precisely the true model should be specified. Section 7.4 presents limitations in the present implementation, and proposes strategies for mitigating them. In Section 7.5, we describe a possible approach to automating modelling. Section 7.6 discusses the special considerations required to support common power management features in the energy models. Finally, Section 7.7 describes to what extent the problem description subtasks identified in Section 1.4 have been fulfilled.

7.1 Barrelfish Suitability

There are two questions which should be kept in mind before using Barrelfish as the operating system baseline in system software research.

1. The most obvious question is 'Why Barrelfish', or inversely 'Why not Linux?'. As Linux is an operating system with widespread adoption, experiments based on Linux will be more equivalent to real world conditions than experiments based on the experimental Barrelfish OS.

2. Additionally, one must consider whether Barrelfish may be directly unsuitable for the research in question. In particular, the operating system is heavyweight with a powerful capability and memory management system. This is reflected in the selection of example applications in work conducted with Barrelfish [BBD+09, ZGKR14], where the evaluation of performance is done with server workloads such as database, web server and network subsystem benchmarks. The SHMAC infrastructure realizations, on the other hand, are necessarily

relatively lightweight processors because of the limitations arising from targeting FPGA implementation. As such, it may be more reasonable to evaluate the SHMAC processor designs using embedded system benchmarks. For such benchmarks, Barrelfish may be a questionable choice of an example multikernel.

These points imply that Barrelfish should be used judiciously. In particular:

- Barrelfish should be used when a multikernel is explicitly desirable in the research, for instance to demonstrate the successfulness of a technique in unconventional kernel architectures or to investigate improvements to multikernel implementations in particular.
- It must be possible to argue that the combination of operating system selection, experiment workload and SHMAC infrastructure is coherent so that results are generalizable to a real-world scenario.
- If an experiment involves comparing results attained using two different operating system architectures, the non-multikernel must be similarly targeted at server workloads to ensure fair comparison.
- If two different Barrelfish versions are compared, the researcher should take care that the efficiency of a technique is not owed to inadvertent reductions in Barrelfish complexity.

7.2 Impact of Planned SHMAC Modifications

There are several changes recently planned for the SHMAC infrastructure, which require some extensions to the work described in this dissertation:

- The ISA used in processor cores will be changed from ARMv4T to RISC-V [RIS14]. This change mandates the creation of new energy models and monitors for the new processor cores, and porting the ISA-specific parts of Barrelfish again. This is limited to context switch code and the lowest-level interrupt handling routines, and should as such not require much work. However, whether such work will be worthwhile should be considered in light of the issues raised in Section 7.1.
- The current router uses fully parallel I/O ports, sending entire packets at once. There are plans to modify its implementation to be flit-based, where packets are divided into so-called flits which are transmitted across less wide links. This will mandate a new model and monitor implementation for the router.

Overall, both the Barrelfish and energy estimation infrastructure implementations are relatively resilient to the proposed SHMAC project changes and should therefore retain their value also in the future of the project. The models which have been developed for the current SHMAC components will be largely useless, however, if both the core and the router is replaced. The models made for the DDR tile and the scratchpad tile may still be used.

7.3 The Importance of Trueness

Discussing trueness primarily makes sense when there exists a notion of a true system. When there is no such standard to adhere to, implementation and design choices may be made arbitrarily[1] as long as they are true to *some* realistic system. As described in Section 6.2.1, this dissertation has therefore made arbitrary selections in areas such as clock gating, operating environment, cell library selection, cache organizations and memory system characteristics.

A precise specification of all the characteristics of the true system is not necessarily required to make results attained with the SHMAC infrastructure applicable to real world situations. However, such a specification may simplify ensuring result validity as an explicit specification would make it easier to verify that the SHMAC infrastructure characteristics are not unrealistic. It would also be simpler to spot inconsistencies in the choices made. If it is not clear that the SHMAC infrastructure is similar to a real system, it will be necessary to demonstrate result validity by some other means. As such, trueness even in the details of system characteristics should be considered important.

Therefore, it may be beneficial to create a complete SHMAC infrastructure specification. Note that genericness will not be reduced by specifying the characteristics explicitly, as some value would have to be selected anyway. Such a specification may draw inspiration from characteristics of existing systems. This may be simplified by the planned swap to the RISC-V ISA, since ASIC implementations have been made of an open source core implementing this ISA. If necessary, several specifications could be made for different envisioned realizations of the SHMAC infrastructure. It would be interesting to quantify the impact different choices has on the model results, and thereby also quantifying the importance of trueness in the details.

[1] I.e. driven by convenience and apparent validity rather than some specification.

7.4 Addressing Limitations

7.4.1 Validation Benchmarks

The current validation benchmark selection may be suboptimal. For one, Dhrystone is a synthetic benchmark constructed based on the average frequency of use of different programming language constructs. There is greater value in validating the precise prediction of real programs, which may potentially be used to benchmark the system at some point. In addition, the diversity of Dhrystone is limited since it is based on iterations of the same computation. This attribute may make its power consumption easier to predict than segments from different program phases in real programs, which most likely exhibit more varied behaviour.

To address these limitations, a new set of validation benchmarks should be carefully selected. One possible strategy is selecting programs from existing benchmark suites which are likely benchmark candidates for future SHMAC experiments. The newly created MachSuite [RAS+14] is one possible candidate benchmark suite, as it is designed to benchmark accelerator performance. Other alternatives include MiBench [GRE+01] and EEMBC [EEM14], which have been used to validate similar power modelling work [BCM08][SWPC10].

In addition to selecting the benchmarks, the benchmarking framework described in Appendix B.2 must be extended to support the new benchmark programs. This will likely involve implementing functions from the C standard library, either directly or through the implementation of system calls expected by an existing C standard library implementation. As one example, MachSuite benchmarks read the input data from a file. Supporting this requires extending the test bench to map files into simulated SHMAC memory, and employing this functionality in the run-time.

More realistic benchmark programs most likely have running times which preclude running them through simulation and power estimation tools in their entirety. It is undesirable to mitigate this by using only the first millisecond of each benchmark, as this may be dominated by initialization work. A better alternative is extending the test bench to support fast-forwarding and prematurely terminating benchmark programs. This way, different phases of a benchmark may be included, and diversity preserved. Efficient support for such features also allows including very long-running programs such as Barrelfish among the validation benchmarks.

7.4.2 Modelling

Regression Modelling

Threats to Trueness Section 6.2.1 identifies two threats to trueness related to regression modelling:

1. HDL differences between FPGA and ASIC core implementations.
2. Missing place-and-route information.

The swap to RISC-V may help mitigate the first issue, as there exist both FPGA and ASIC implementations of certain open source RISC-V cores. As long as these implementations do not differ in behaviour such as number of cycles stalled on memory accesses or vastly in what RTL signals exist, the ASIC version may be used for modelling purposes and the FPGA for prototyping.

The second issue is one shared by other work on RTL-based regression models [BCM08, SWPC10]. It would still be beneficial to estimate its impact by attempting to set up and implement the necessary support for the place-and-route step in an ASIC flow, but this requires deeper knowledge of layout issues such as power planes and clock tree synthesis. Additionally, the tool support on IME servers is at present limited[2]. Thus, on this point it may in the interest of project progression be advisable to forsake some trueness certainty for implementation simplicity.

Negative Coefficients The regression modelling is performed using the method of least-squares, which does not restrict coefficients from becoming negative. Energy consumption models will therefore regularly have some negative terms, as may be seen in Appendix D.1, which is natural if the activity represented by the term implies missing activity from some other term. As an example, consider a multiplexer as seen in Figure 7.1 where both a change in the selection and the Hamming distance of the output are used as model variables. A change in Hamming distance alone implies that there has been activity in the multiplexer input driver, whereas a change in both the Hamming distance and the selection variable can only imply activity in the multiplexer itself. This may be modelled by including a large term for the Hamming distance of the output, and a negative term for the case when there also was a change in the selection. This kind of term is present for instance in the model for the execute stage (see Appendix D.1.2).

Although negative coefficients may be reasonable, they appear counter-intuitive as they apparently express conditions under which the system gains energy. There is also no need to support negative coefficients: less activity under certain conditions may be modelled using an extra positive term in the negated condition. Additionally, there is a possibility that the absence of activity indicated by one term is caused by benchmark-specific activity. If the benchmark program never uses the multiply unit and the barrel-shift unit simultaneously, for instance, activity in one may appear to imply no activity in the other. Finally, additions are in general cheaper than

[2] A 2×3 SHMAC design was not loadable in the place-and-route tool currently installed because of a restrictive license.

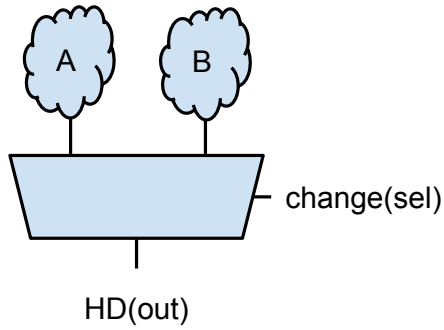

Figure 7.1: A change in the output value with no change in the multiplexer selection implies activity in the multiplexer driver, whereas a change in the selection only implies activity in the multiplexer itself.

subtractions in terms of implementation overhead as there is no need for sign-extension which enables smaller adders.

To circumvent these issues, future models should be considered modelled with coefficients restricted to positive values. This is not a built-in feature of the R linear modelling engine, but it is possible to implement using additional packages [Kei14]. Once encapsulated in support functions in the modelling infrastructure, creating such models should not be any harder than creating unrestricted models.

Coefficient Resolution When implementing the power models in hardware, the floating point coefficients from the R models are converted to a fixed point representation to facilitate hardware implementation. This corresponds to selecting an energy model unit scale and rounding off all coefficients to the nearest multiple of the selected unit. As an example, consider the coefficient 0.005263. If a resolution of 10^{-4} is selected, the coefficient will be represented in hardware as the integer 53 with an implicit model base unit of 10^{-4} Joules. This would cause a representation error of $\frac{53 \cdot 10^{-4} - 0.005263}{0.005263} \approx 0.7$ %, and require $\lceil log_2(53+1) \rceil = 6$ bits.

In this work, the coefficient resolution is set to some number of decimal places as described in Section 6.2.1. This is equivalent to considering energy model unit scales which are inverse powers of ten, which is natural for human modellers. However, this gives a coarse trade-off between precision and hardware implementation overhead as demonstrated in Table 7.1. As increasing the precision with one decimal place increases the coefficient values by a factor of ten, the size required to represent coefficients is increased by three to four bits. Thus, it is not possible to trade one bit of coefficient size for accuracy.

Table 7.1: The maximum numbers of bits required to represent the fractional part of a coefficient at different power-of-ten coefficient resolutions.

Resolution	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Maximum Bits Required	4	7	10	14	17

This limitation may be addressed by using powers of two as the base units instead. The resolution of coefficients may then be varied at one-bit granularities, as increasing or decreasing the unit will correspond to including or excluding single digits from the R binary floating point fractions. This will allow investigation of a larger trade-off space.

Although power-of-two-units guarantee single-bit trade-off granularities, other units may still be preferable as coefficient divisibility significantly impacts how efficiently a coefficient may be represented. This is demonstrated in Table 7.2, which tabulates the efficiency of representing the coefficient 0.1234567 using different powers of two, three and ten as unit scales. Considering powers of two instead of powers of ten does provide extended trade-off opportunities, which is apparent by comparing unit scale ranges $2^{-12} - 2^{-14}$ and $10^{-3} - 10^{-5}$. However, the unit scale $3^{-4} \approx 0.0123456790$ permits significantly higher efficiency since it almost divides the coefficient.

Table 7.2: Representation error when varying coefficient resolution.

	Representing 0.1234567 ($\approx 0.00011111001101_2$)					
	2^{-x}		3^{-x}		10^{-x}	
x	Error	Bits	Error	Bits	Error	Bits
1	100 %	0	100 %	0	19 %	1
2	100 %	0	10 %	1	2.8 %	4
3	1.25 %	1	10 %	2	0.37 %	7
4	1.25 %	2	0.000073 %	4	0.0351 %	10
5	1.25 %	3	0.000073 %	5	0.003 %	14
6	1.25 %	4	0.000073 %	7	0.0002 %	17
7	1.25 %	5	0.000073 %	9	0 %	21
8	1.25 %	6	0.000073 %	10		
9	0.332 %	6	0.000073 %	12		
10	0.332 %	7	0.000073 %	13		
11	0.064 %	8				
12	0.064 %	9				
13	0.0353 %	10				
14	0.014 %	11				

This suggests that future modelling should increase the number of unit scales considered. This will increase the available trade-off space, and allow discovering more efficient representations.

Analytical Modelling

On-chip RAM As mentioned in Section 6.2.1, due to technology differences between CACTI models and the cell library used the on-chip RAM models are only true to an unlikely system which has a different voltage supply for caches than for the processor core. The CACTI estimates are still likely to be more accurate than regression models, as the caches included in the SHMAC HDL consume 96 % of the power of an entire Amber tile when analysed with Synopsys PrimeTime. However, the technology difference is still a probable source of inaccuracy. This discrepancy may be mitigated by re-creating the regression models based on a cell library with a 1.1 Volt operating voltage, which is available on the build server. There may still be other differences between the process technology assumed by CACTI, and the one used in the cell library. Investigating this would clarify any potential accuracy losses.

Off-chip RAM Not including the consumption of a memory controller in the off-chip RAM model limits which systems the model is true to. Additionally, the DDR chip characteristics were arbitrarily selected. If the proposal in Section 7.3 about clearly defining the operating environment of the SHMAC is implemented, it will be necessary to more rigorously ensuring that the off-chip RAM model is true to the selected memory system configuration.

In the spreadsheet which is used to get the DDR power consumption data, the active command power is calculated based on a combination of bank hits in the DDR and the read and write access intensity. Although the first factor is hard to estimate, it may be possible to use the access intensity to scale the contribution of power consumption from activate commands based on the access patterns made at run-time. This may be a simple way in which to make the off-chip RAM model a bit more precise, making bursts of memory reads slightly less expensive.

7.4.3 Sample Granularity

As described in Section 6.2.5, the sample granularity is limited because the current host interface to the energy estimation infrastructure is dependent on periodic software polling. Additionally, the sample period has considerable variance with occasional significant spikes. These issues may be resolved by moving the control of sampling to hardware. We will here outline a strategy for implementing such a change.

First, the host interface would be changed. A sequence diagram demonstrating the difference between the existing and the proposed host interface is depicted in Figure 7.2. The *NextSample* register would remain, but the *SampleEnergy* write-only register would be replaced with a *SamplePeriod* read-write register. The driver would write a value to the *SamplePeriod* register to request the scan chain control unit to collect new samples at every *SamplePeriod* SHMAC clock cycle. The scan chain control unit would then be altered to start sampling not on request from the APB, but once every *SamplePeriod* clock cycles. Samples would still be stored in a cross-clock-domain FIFO.

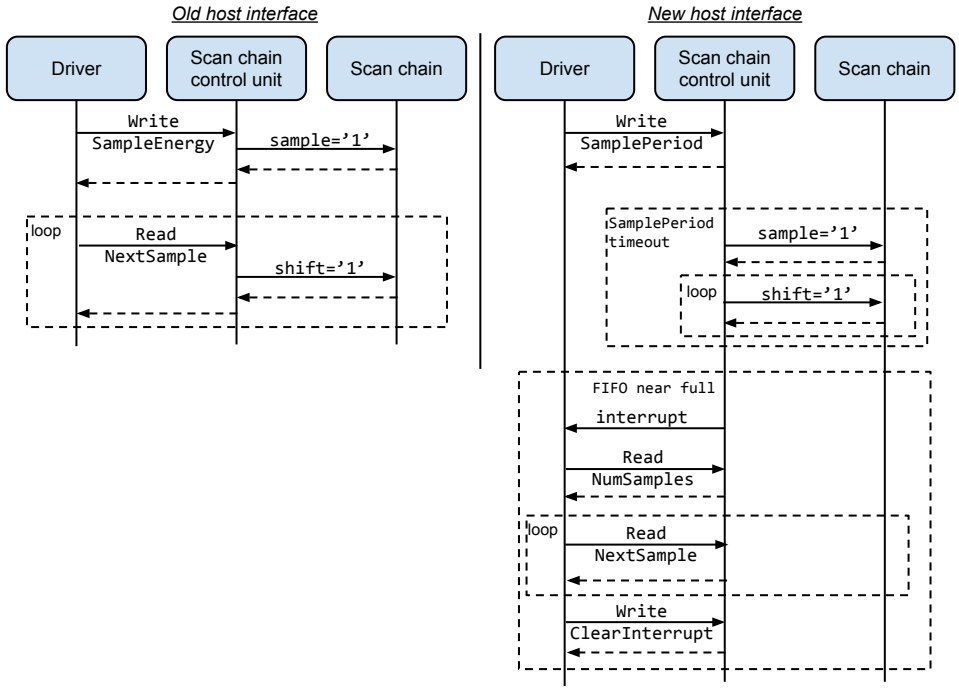

Figure 7.2: A sequence diagram demonstrating how the new interface would differ from the old.

Once the FIFO was nearly full, an interrupt would be sent to the SHMAC host indicating that new samples should be read. The number of samples to read could be determined by a new read-only *NumSamples* register, set to the number of samples in the FIFO at the time of the interrupt. The interrupt could be cleared either by using a separate write-only *ClearInterrupt* register, or automatically by the APB tile when all *NumSamples* had been read. For the scheme to be safe, the *SamplePeriod* value should be restricted to rates at which the driver can read out samples without

constantly handling interrupts. If necessary, lower *SamplePeriod* values may be supported by using several FIFO primitives.

To give access to user-space, the driver could create a new binary `sysfs` attribute sized to some multiple of the number of energy samples included in the scan chain. The contents backing this attribute would be periodically refreshed, once the driver had read new samples. User-space could register a signal handler with the driver, which could be called when the energy sample contents was refreshed. The signal handler should then copy all the data before the driver refreshes the contents.

With these changes, it would be possible to get exact sample periods with no variability. Additionally, it is likely that the achievable sample granularity would be significantly higher as the driver interrupt handler can execute with other interrupts turned off, thus not being subject to context switching. The driver buffering would then make it possible for the user-space application to stream out larger quantities of data, which is faster than reading one set of samples at a time. Determining the exact achievable sample rates would require experimentation. If the improvements are sufficiently large, it will also be possible to keep sample sizes at 32 bits.

As this improvement makes it less critical with timely software readings of energy values, there would be no need to use UDP in the monitoring and logging tool for the 1 % speed increase while risking packet loss under suboptimal network conditions. Instead, a TCP connection could be used to to ensure safe packet delivery.

7.4.4 Implementation Overhead

The implementation overhead of the energy monitors is quite variable. While the overhead for the Amber core is reasonable, the overhead for the timer module and the router is substantial. However, this is foreseen to not be prohibitive for the use of the energy estimation framework for the following reasons:

1. The relative overhead is large because the cost of the units which have been modelled is small. Since the target FPGA is large, one processor tile consumes only 1.68 % of the available LUTs even with the energy estimation infrastructure included. As such, there is enough LUTs for nearly 60 processor tiles. Thus, even a large relative overhead is negligible for the SHMAC infrastructure in its current state.

2. The models have only been rudimentarily optimized with implementation overhead in mind. More careful selection of model variables and minimization of model terms may serve to reduce the overhead. For example, Appendix D.1 shows how the timer model may be improved by removing all Hamming distance terms, reducing the number of terms and improving its accuracy by more careful

modelling. For the router, the Hamming distance overhead may perhaps be reduced by using terms for more specific parts of input packets instead of one term for each 196 bit link.

3. As discussed in Section 7.4.2, more experimentation with coefficient resolution may yield trade-offs with less costly monitors at a reasonable expense in precision.

4. There are unexplored optimization opportunities in the hardware implementation of the monitors. Several implementation techniques may reduce the overhead further:

   – The use of the optimized Hamming distance implementation in [SS13b] may be perfected. For instance, the article describes an optimized implementation for 8-bit buses which we have not used. The optimal combination of Hamming distance implementations may also be investigated for each different bus size. In particular, it would be beneficial to determine whether a 32-bit Hamming distance calculation should be implemented with four 8-bit implementations, one 36-bit implementation, or our original implementation using a full adder tree.
   – More of the monitor logic may potentially be moved before pipeline registers without breaking timing. Since calculation results typically require fewer bits than calculation inputs, this may reduce the number of pipeline registers required.
   – Terms with shared factors may be combined to a single term. For instance, a model such as $C_1 \cdot HD(x) + C_2 \cdot s \cdot HD(x)$ may be implemented as $(C_1 + C_2 \cdot s) \cdot HD(x)$. This reduces the cost of the final term sum. The coefficient sum may be implemented using narrow adders or a multiplexer.

5. The router monitor overhead may be significantly reduced when the plans for a less wide router implementation is realized, as discussed in Section 7.2, since most of the router overhead comes from calculating the Hamming distance of the 196 bit wide input and output links. Since the router monitor accounts for 70 % of the LUT usage of the infrastructure, the ability to reduce its complexity will lead to considerable improvements in overall infrastructure overhead.

7.4.5 Target and Host Clock Frequency Discrepancies

There are two ways to handle the discrepancy between FPGA host and ASIC target clock frequencies:

1. When modelling the energy consumption of the off-chip RAM, ensure that the ratio between the true system off-chip RAM frequency and the SHMAC

off-chip RAM is the same as the ratio between the ASIC target clock frequency and the FPGA host clock frequency. For the current setup, where the ratio between target and host frequencies is $\frac{250\text{ MHz}}{60\text{ MHz}} \approx 4.1667$ and the DDR speed is 800 MHz, this would be equivalent to assuming on off-chip RAM speed of $800\text{ MHz} \cdot 4.16\bar{6} \approx 3333.33$ MHz. Memories with similar clock frequencies do exist [HEX14], although they target niche audiences. The speed is also outside of the range supported of the Micron DDR3 modelling tools, so proper modelling would either require extrapolation or a different source for estimating power numbers. On the plus side, this solution is simple and may be sufficient for experiments where the impact of off-chip memory accesses on performance and energy consumption is unimportant.

2. Add extra delay cycles before issuing responses in the off-chip RAM tile, such that the off-chip memory accesses are slowed down by a factor similar to the target clock frequency. The number of cycles required could be determined by estimating average response latency, and multiplying it by the ratio between target and host clock frequency. This strategy has already been planned for inclusion in the SHMAC infrastructure once the need arises, since the target and host time discrepancy affects the validity of performance results and not just the energy efficiency estimation framework described in this dissertation.

7.4.6 User Friendliness

As described in Section 6.2.3, there are several areas where the user friendliness of the energy efficiency estimation framework could be improved. The following list describes how shortcomings may be addressed, in addition to other miscellaneous possible improvements to the user friendliness:

- Dynamic configuration of the set of active energy report units would make it unnecessary to synthesize a new SHMAC instance for this purpose. Dynamic configuration could be implemented with varying flexibility and overhead. The most flexible but most costly solution would be to permit including any partial sum of energy monitor estimates. This would correspond to optionally including energy report units at any level in the adder tree resulting from an energy monitor setup, illustrated in Figure 7.3. Lower implementation cost and flexibility could be obtained by making only a subset of the partial sums selectable. The configuration could be trivially implemented as a mask register in the scan chain control unit, with one bit per report unit denoting whether the unit should be included or excluded.

- There is at present no support for determining when experiments start and stop. When running an experiment, it is therefore necessary to gather energy

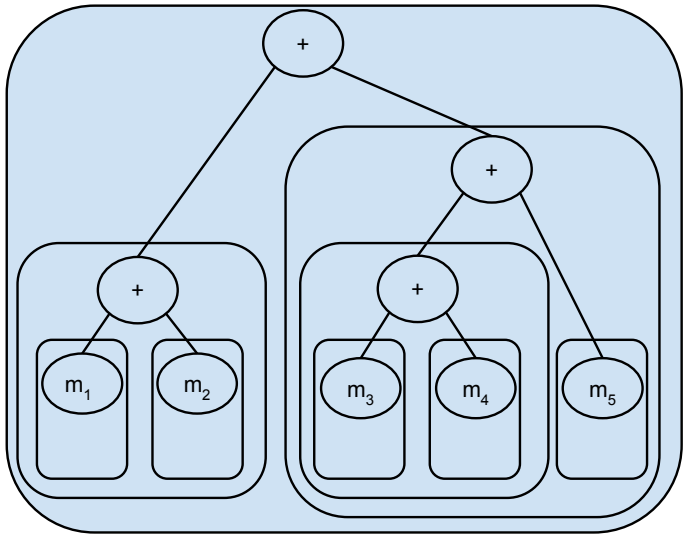

Figure 7.3: An example hierarchy in a SHMAC tile, illustrating how partial sums of energy monitor estimates correspond to the energy consumption of different hardware modules in the hierarchy.

estimates from the reset instant and manually delimit the samples to the desired time interval. Simpler and more precise experiment control may be attained by including a new register in the SHMAC infrastructure, writeable by SHMAC software, denoting whether an experiment is running or not. By inserting writes to this register in the benchmark, it would be possible to control which parts of a benchmark is included in a set of experiment data. This register could work by controlling whether energy estimate sampling is enabled or not. With the current host interface, this could be achieved by by returning energy estimates of value zero if samples are requested when an experiment is not running. When the host interface proposed in Section 7.4.3 is implemented, sampling may instead simply be disabled when no benchmark is running.

– There is no way of determining the units of the values reported by the energy estimation infrastructure. This should be added as a driver attribute, and made use of in the energy monitoring and logging tool.

– As discussed in Section 6.2.3, writing energy monitors is tedious work. This should be possible to automate by generating Verilog implementations from R formula expressions. It would most likely not be possible to determine the required number of pipeline stages, but the possibility of including pipeline stages in the most common locations could be included and used when necessary.

– The configuration of the SHMAC instance which is installed is at present hard-coded in the driver, as described in Section 4.2.5. When installing a new SHMAC instance, it is as such necessary to update and install a new driver with an updated attribute configuration. It may be more user friendly to include the attribute configuration as a set of host-readable registers in the SHMAC infrastructure. This would make it possible to use the same driver for SHMAC instances with the same host interface. A potential downside is that writing the configuration in hardware may be more cumbersome than it is in software.

7.4.7 Barrelfish Correctness

Although the tests of Barrelfish indicate that our extensions work, the limited runtime of the stability test when using scratchpad memory limits its value. Once the SHMAC infrastructure is working, the stability test should be rerun for a longer period to uncover bugs such as race conditions, failure for rare inputs, or failure for rare code interactions such as context switches.

Another option would be to develop unit tests for the parts of Barrelfish developed for the port to SHMAC. Such tests may uncover input-dependent failures, and yield higher confidence of correctness without requiring long-running tests. However, the development of such tests requires developing test stubs for the parts of Barrelfish the code being tested is dependent on. The stubbing required may be substantial if the number of dependencies is large. Unit test development was previously begun and subsequently aborted due to considerable development overhead and limited usefulness. However, a test development framework is ready for use if rigorous testing is prioritized at some later time.

7.4.8 Coverage Analysis

The analysis of coverage has only been done qualitatively. It is possible to instead define coverage quantitatively as the fraction of nets in a design reachable from a subset of nets. A quantitative measure makes it easier to assess the level of coverage of different models. However, a tool which calculates the coverage in this manner from a gate-level netlist was to our knowledge not available. Developing such a tool would require parsing the Verilog netlist, generating a graph describing the connectivity of nets in a module, and traversing the graph from the nets used in the models to count the number of reachable nets.

7.5 Modelling Automation

An interesting extension of this dissertation would be to investigate the possibility of automating the creation of models: given a set of power and signal data from a test

benchmark, can we automatically create a formula which is implementable on an FPGA and relates RTL activity to energy consumption?

The state of the art in automated power modelling is the PrEsto method [SWPC10], which we described in Section 2.4. Automation is achieved by specifying the formula up-front, and using the maximum value of terms as a sorting criterion. Models are then generated by inputting the model term count limit and training data for a set of signals and buses. Since the selection of signals is done manually, the method offers limited automation support. The fixed formula format is also simplistic, as it disregards the structure of the hardware being modelled. Since implementation overhead is a major concern for our purposes, it is desirable to create more specialized models instead of achieving automation through generality.

Based on the modelling experience from this dissertation, we can envision two contributions:

1. A method suitable for creating models used alongside FPGA prototypes. This method should seek to reach a set of accuracy goals, for instance limits on average error, average per-cycle error and maximum error, while minimizing implementation overhead. The method would work as follows:
   - First, model terms are created based on the hardware structures in the module being modelled. For instance, the energy consumption of a multiplexer may be represented as the Hamming distance of the output multiplied with the change in multiplexer selection. Creating terms which more closely resembles the hardware being modelled may yield higher predictive power per term. This was observed when creating a model for a part of the Amber core, where a combination of three boolean signals controlled a multiplexer. Taking the hardware being modelled into account and creating a cross term with all three signals yielded significant benefits to accuracy over restricting the model to the format of PrEsto, with at most two boolean signals in a cross term.
   - Second, terms should be selected for inclusion in the model based on heuristics instead of creating a model with all terms and then selecting those which appear to be most significant. The more terms which are used during regression analysis, the more time-consuming it is. It is therefore preferable to estimate the importance of a term before including it in a model. Example heuristics for estimating term importance may be its coverage of nets not currently covered; the energy consumption of operations it covers; and its impact on implementation overhead.
   - Finally, an optimization should be conducted which searches for a model with minimal implementation overhead within some set accuracy limits,

or vice versa. This search could trade off accuracy and implementation overhead by varying the selection of terms, or by varying the coefficient resolution as described in Section 7.4.2. To keep runtime low, the evaluation of model accuracy should initially be conducted with test benchmark data and heuristics. Evaluation using validation benchmarks should be used in later phases of the search, to ascertain that models which accurately estimates energy for the test benchmarks also are accurate for varied activity.

2. Automated use of the method based on HDL implementations by creating model terms automatically from the HDL. With this contribution, one could in theory completely automate the generation of power models given an HDL implementation and a set of test and validation benchmarks. One possible approach would be to create a new back-end for the open source Icarus Verilog synthesis tool [Wil14]. Another interesting avenue would be extending the open source implementation of the Chisel HDL [BVR+12]. Since Chisel has higher level HDL constructs, such as multiplexers, registers and memories, generation of terms using this tool may be simpler. Since the new RISC-V cores planned for SHMAC are written in Chisel, such an extension may have a significant benefit for the SHMAC project.

7.6 Power Management

Many modern systems include support for clock gating, power gating, and Dynamic Voltage-Frequency Scaling (DVFS). If the system architecture under investigation is supposed to have such power management facilities, it is important that the framework supports emulating the energy consumption characteristics resulting from their use.

Power gating disconnects the power supply to a module, and clock gating stops the clock driving a module. These techniques would most likely be emulated in the FPGA implementation by faking a shut-down and start-up sequence with a suitable delay, and causing module stall when the gating technique was active. The energy monitors could support these techniques by taking two extra binary input signals, denoting whether power or clock gating is active. If power gating is active, the energy consumption is zero: if clock gating is active, the energy consumption should equal the static leakage energy. These extensions are trivially implemented.

Supporting DVFS in the energy estimation infrastructure is more difficult. The most apparent challenge is scaling the output of the power model by the change in voltage. The simplest solution is to use the formula $E = \frac{1}{2}CV^2$ as inspiration, and scale the output by the square of the voltage difference. If the voltage is only altered in steps of

powers of two, scaling the output may be done using a barrel-shifter. A problem with this model is that it is only a valid representation of dynamic energy consumption. The static energy consumption per clock cycle is $E = VI_{leak}T_{clk}$ [KAB+03], so scaling voltage and frequency similarly keeps the static energy consumption per cycle approximately constant[3]. It may be beneficial to create monitor models where the static and dynamic power consumption is separated in order to accurately represent this behaviour [Sno10].

An additional challenge is the varying discrepancies between host and target time due to varying target frequency, the impact of which is discussed in Section 6.2.1. To limit this effect, it is important to introduce stall cycles in the hardware modules to emulate the effect a reduction in frequency has on operation and interaction with other system components.

The effect these techniques would have on energy monitor estimates, assuming a monitor where dynamic and static energy is separated, is summarized in Figure 7.4. The figure demonstrates a sequence of power management operations: first halving frequency and voltage, then enabling clock gating, and finally enabling power gating. Halving the voltage leads to a dynamic energy consumption one quarter of the original. As the monitor models the energy consumption for one target cycle, halving the emulated frequency makes it necessary to only emit an energy consumption estimate from the monitor estimate every second host cycle. When the clock is completely stopped due to clock gating, only the static energy is included. When power gating is active there is neither static nor dynamic energy consumption, so the energy estimate is zero.

Finally, if DVFS support is included it may be desirable to add the current operation frequencies of tiles as host-accessible SHMAC registers. This would make it possible to more accurately correlate power management operation with energy consumption. As the energy monitor estimates are summed by energy report units, it is not possible for the host to estimate the operating frequency from energy estimate waveforms. Note that power may still be calculated directly from the energy estimates without knowing the operation frequency, as the energy estimates from monitors are correctly spread out in time.

7.7 Project Description Fulfilment

This section will discuss the fulfilment of each subtask identified in Section 1.4.

[3]Leakage current is also dependent on supply voltage [KAB+03], but the effect is negligible unless the supply voltage is completely disabled since the leakage current is primarily dependent on technology parameters [BS00].

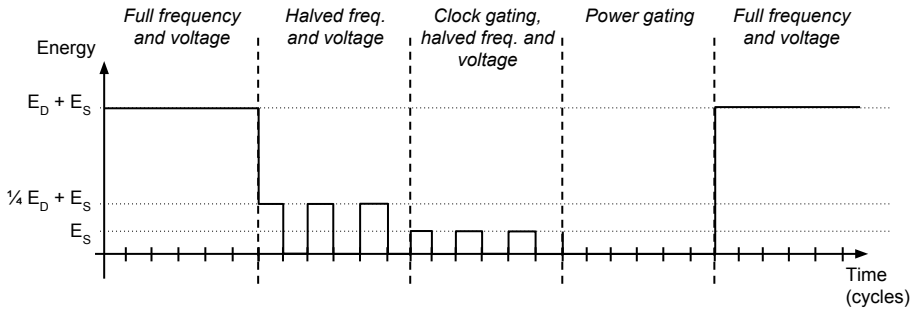

Figure 7.4: The effect of power management on energy monitor estimates. E_D denotes dynamic energy, and E_S denotes static energy. For illustration purposes, the dynamic energy consumption of the module is assumed to be independent of activity. The delays for activating and deactivating the gating techniques are also not taken into account.

Mandatory Tasks

B1: Support multiple cores in Barrelfish Successfully fulfilled, as Barrelfish is able to boot and run communicating programs on multiple cores. The extensions are described in Section 5.2. The stability has been asserted with caches disabled, as described in Section 6.1. The port has not been formally verified, and is as such not guaranteed to be fault-free, but as most extensions relate to initialization the ability to start is sufficient to feel confident in the correctness.

E1: Investigate energy efficiency estimation techniques Successfully executed in Sections 2.3 and 2.4. The investigation began with a small survey on existing estimation techniques in Section 2.3, which were used as inspiration for the technique presented in Section 2.4 tailored to our purposes. Its efficacy was underpinned by existing work, also presented in Section 2.4.

E2: Describe a prototype energy efficiency estimation framework Successfully executed, with a description of a complete framework design given in Section 2.4 and Chapter 3. Shortcomings and potential improvements are discussed in this chapter.

E3: Evaluate accuracy, coverage and user friendliness Successfully executed, in Sections 6.2.1 to 6.2.3. Both trueness and precision is covered in detail in Section 6.2.1, and user friendliness is thoroughly treated in Section 6.2.3. The analysis of coverage in Section 6.2.2 may be improved by using a quantitative measure, as discussed in Section 7.4.8. The evaluation was also extended with

extra evaluation of infrastructure correctness and performance in Sections 6.2.4 and 6.2.5.

Optional Tasks

B2: Support new SHMAC platform features in Barrelfish Successfully executed, as both ISA upgrades and communication support over scratchpad memory is implemented as described in Sections 5.3 and 5.4. The testing described in Section 6.1 indicates correct operation, but stability of operation when using scratchpad memory for inter-core communication has not been verified due to bugs in the SHMAC infrastructure.

B3: Implement console support Successfully executed, as described in Section 5.5. The console is able to obtain serial input data from the Barrelfish kernel, and may be used to start different programs with different input on arbitrary cores.

E4: Implement the prototype framework Successfully executed, with a complete implementation described in Chapter 4 and Appendix B. The implementation exceeds prototype grade as it is sufficiently functional to be used to evaluate the energy efficiency of the current SHMAC infrastructure, with the caveat that the model precision is based on validation benchmarks with room for improvements as discussed in Section 7.4.1. Shortcomings and potential improvements with the implementation are listed in this chapter.

E5: Evaluate Linux and Barrelfish energy efficiency Not executed. Linux does not work on multiple cores and has never been run on the Versatile Express platform, which limits the evaluation opportunities without a considerable investment of work. As the task is unimportant for the SHMAC project, it was not prioritized.

Chapter 8
Conclusion and Future Work

8.1 Conclusion

This dissertation has described how research on energy-efficient system software using the SHMAC infrastructure has been enabled through the implementation of two extensions to SHMAC research project: a multicore-capable operating system port, and support for online energy efficiency estimations.

The existing port of the operating system Barrelfish was extended to run on multiple cores. This entailed adding support for launching operating system kernels on new cores, and managing the allocation of shared memory message passing channels. Additionally, the operating system was made more feature-complete by porting the Barrelfish console application. Tests indicate that the port works as expected, as every extension is used at least once without failure. However, due to bugs in the SHMAC infrastructure it has not been possible to fully assert the stability of the operating system port through prolonged program execution.

The first contribution towards energy efficiency estimation support is a method which enables modelling the power consumption of hardware components which only exist as an HDL description. By using ASIC synthesis tools, it is possible to estimate the power consumption of a design under selected stimuli and thus create regression models such that power consumption can be predicted from RTL activity. We also present an analytical modelling method for on-chip and off-chip RAM, where an alternative modelling strategy is either preferred or necessary.

The second contribution is the application of the methods to the existing processor tile in the SHMAC infrastructure. This contribution is two-fold: first, the software and tool infrastructure which was developed in order to create the regression models facilitates the construction of models for new hardware components; and second, the expected efficacy of the modelling scheme is quantified by comparing model predictions with values from ASIC power estimation tools. The error in the estimate

for average power consumption is 0.19 %, and the average cycle-by-cycle estimation error is 1.1 %.

The final contribution is the development of a hardware infrastructure which enhances the SHMAC infrastructure with support for live energy consumption estimation based on observed RTL activity. The infrastructure exposes the energy estimates to a host system, which may log the energy consumption estimates through periodic sampling. Additionally, the energy estimates are made available to software running on the SHMAC infrastructure for potential employment in system software algorithms. The current infrastructure supports sampling periods of approximately 12 milliseconds. The FPGA resource overhead from implementing the energy models for the core and router in hardware is 18 % and 104 %, respectively.

In conclusion, the SHMAC project has been enhanced with both operating system and energy efficiency estimation support. This constitutes a significant contribution towards the ultimate project goal of research on energy-efficient heterogeneous computer architectures.

8.2 Future Work

Looking forward, there are numerous ways in which this work may be extended or improved. In this section, we list possible future tasks in different categories. The lists are sorted internally by suggested priority, and additional item labels suggest inter-category task importance: high (A), medium (B) or low (C).

Benchmark Improvements

A. Investigate more thorough and representative validation benchmark suites in order to ensure model validity. One possible approach is outlined in Section 7.4.1.

Infrastructure Extensions

A. Implement sample control in hardware to significantly improve the infrastructure performance in terms of possible granularity and variability of energy consumption estimates, as described in Section 7.4.3.

A. Add a mechanism to the SHMAC infrastructure which allows the software running on the SHMAC to enable or disable energy estimation. This makes it possible to precisely control what part of a benchmark is included in experiment data, as discussed in Section 7.4.6.

C. Add clock frequency per tile as well as the unit of energy samples as attributes exposed from the SHMAC driver, in accordance with Section 7.6 and Section 7.4.6.

C. Add a set of host-accessible registers to the SHMAC infrastructure, which report the current SHMAC instance attributes such as number of report units and tile configuration. A more detailed discussion is given in Section 7.4.6.

C. Develop a scheme for dynamically configuring which energy report units should be used, as discussed in Section 7.4.6, which obviates the need to re-synthesize and install a new SHMAC instance for such customization.

Model Extensions

A. Investigate how model quality is affected when positive coefficients are enforced. As discussed in Section 7.4.2, this may yield cheaper and more intuitive models.

A. Experiment with a wider range of resolutions than number of decimal places included when converting model coefficients to integers, as discussed in Section 7.4.2.

A. As there are plans to swap out the processor core currently used in SHMAC with a different one, models should be created for the new processor core once this change is implemented.

B. If support for power management features are added to the SHMAC platform, the energy models should also be extended to support these features. This is discussed in Section 7.6.

B. Create a specification for the operating environment of the SHMAC infrastructure, and tune power modelling parameters to match it as described in Section 7.3. It may also be interesting to quantify the effect different specifications has on the power models.

Tool Extensions

A. Implement features in the benchmark framework which new validation benchmarks require. Possible extensions are described further in Section 7.4.1.

A. Add energy and power units to the plots and logs from the monitoring and logging tool, as discussed in Section 7.4.6.

B. Investigate how modelling may potentially be automated, as discussed in Section 7.5.

B. Implement generation of Verilog implementations from model formulae, as described in Section 7.4.6.

C. Create a script which calculates the coverage of models, which may be used to check the quality of validation benchmarks and possibly to drive heuristics in automated modelling as per Section 7.4.8.

Barrelfish Extensions

B. Port the operating system to the ISA of the new processor core planned for inclusion in SHMAC.

B. Conduct a more thorough series of tests of correctness for Barrelfish, once the SHMAC infrastructure is working.

C. Modify the bootloader to extract information about the SHMAC configuration from the attributes which have been added to the driver, and pass these as boot parameters to the Barrelfish kernel instead of using values maintained in a configuration file in the Barrelfish source tree.

References

[AA14] Håkon Amundsen and Joakim Andersson. Linux for SHMAC. Master's thesis, Norwegian University of Science and Technology (NTNU), June 2014.

[AB14] Anders T. Akre and Sebastian Bøe. Turbo Amber, a high-performance processor tile. Master's thesis, Norwegian University of Science and Technology (NTNU), June 2014.

[AMD13] AMD. *BIOS and Kernel Developer's Guide for AMD Family 15h Models 00-0Fh Processors*, Jan 2013. Running average power value and timer registers described on pages 65, 421, 460.

[APB] AMBA 3 APB Protocol. https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARM_AMBA3_APB.pdf.

[Azo14] Thermal Analysis - Precision, Trueness, Accuracy and Errors. http://www.azom.com/article.aspx?ArticleID=5744, November 2014.

[Bar] Barrelfish Documentation. http://www.barrelfish.org/#documentation.

[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 Comput. Archit. News*, 39(2):1–7, August 2011.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[BBDM00] Alessandro Bogliolo, Luca Benini, and Giovanni De Micheli. Regression-based rtl power modeling. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):337–372, 2000.

[BC11] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[BCM08] A Bhattacharjee, G. Contreras, and M. Martonosi. Full-system chip multiprocessor power evaluations using fpga-based emulation. In *Low Power Electronics and Design (ISLPED), 2008 ACM/IEEE International Symposium on*, pages 335–340, Aug 2008.

[BDH+06] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[BEA+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, Feb 2008.

[BGM+10] Ramon Bertran, Marc Gonzalez, Xavier Martorell, Nacho Navarro, and Eduard Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 147–158. ACM, 2010.

[Bit14] Bitvis utility library. http://bitvis.no/products/bitvis-utility-library/, Dec 2014.

[BJ07] William Lloyd Bircher and Lizy K John. Complete system power estimation: A trickle-down approach based on performance events. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 158–168. IEEE, 2007.

[BMS13] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 1–12. IEEE, 2013.

[BS00] J.A. Butts and G.S. Sohi. A static power model for architects. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 191–201, 2000.

[BS13] Benjamin Bjørnseth and Bjørn C Seime. Porting and Evaluating Barrelfish for SHMAC. Master's Project, December 2013.

[BSDB05] Rajeev Balasubramonian, Viji Srinivasan, Sandhya Dwarkadas, and Alper Buyuktosunoglu. Hot-and-cold: Using criticality in the design of energy-efficient caches. In Babak Falsafi and T.N. VijayKumar, editors, *Power-Aware Computer Systems*, volume 3164 of *Lecture Notes in Computer Science*, pages 180–195. Springer Berlin Heidelberg, 2005.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM.

[BVR+12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[BWCC+08] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[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*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM.

[CRR05] J. Coburn, S. Ravi, and A Raghunathan. Power emulation: a new paradigm for power estimation. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 700–705, June 2005.

[CSK+07] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil Patil, William H Reinhart, Darrel Eric Johnson, and Zheng Xu. The fast methodology for high-speed soc/computer simulation. In *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pages 295–302. IEEE, 2007.

[DBSDBM13] Marc Duranton, David Black-Schaffer, Koen De Bosschere, and Jonas Maebe. The HiPEAC vision for advanced computing in horizon 2020, 2013.

[DCK07] Robert H. Dennard, Jin Cai, and Arvind Kumar. A perspective on today's scaling challenges and possible future directions. *Solid-State Electronics*, 51(4):518 – 525, 2007. Special Issue: Papers selected from the 2006 ULIS Conference.

[DDR14a] Calculating Memory System Power for DDR3. http://www.micron.com/~/media/documents/products/technical%20note/dram/tn41_01ddr3_power.pdf, October 2014.

[DDR14b] Ddr3 sdram device operation. https://www.skhynix.com/inc/pdfDownload.jsp?path=/datasheet/Timing_Device/DDR3_Device_operation_timing_diagram_computing.pdf, Dec 2014.

[DDR14c] System power calculators. http://www.micron.com/support/power-calc, Dec 2014.

[Des14] DesignWare Library. http://www.synopsys.com/dw/buildingblock.php, Dec 2014.

[DGnY+74] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE J. Solid-State Circuits*, page 256, 1974.

[DRA14] DRAMSim2, cycle-accurate modelling of DRAM memory controllers, DRAM modules and buses. http://www.eng.umd.edu/~blj/dramsim/, October 2014.

[EBSA+12] Hadi Esmaeilzadeh, Emily Blem, Renée St Amant, Karthikeyan Sankaralingam, and Doug Burger. Power limitations and dark silicon challenge the future of multicore. *ACM Transactions on Computer Systems (TOCS)*, 30(3):11, 2012.

[ECX+11] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 319–332, New York, NY, USA, 2011. ACM.

[EECa] The EECS research initiative. http://www.ntnu.edu/ime/eecs.

[EECb] The SHMAC Project. http://www.ntnu.edu/ime/eecs/shmac.

[EEM14] EEMBC homepage. http://eembc.org/, Dec 2014.

[FB13] Bryan Ford and Eric Stephan Boleyn. Multiboot Specification version 0.6.96. http://www.gnu.org/software/grub/manual/multiboot/multiboot.html#Boot-information-format, November 2013.

[GFA+11] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 12–23, New York, NY, USA, 2011. ACM.

[GHS11] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 503–514. IEEE, 2011.

[GRE+01] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[HEX14] Corsair to launch 3ghz dominator platinum ddr3 memory. http://hexus.net/tech/news/ram/40197-corsair-launch-3ghz-dominator-platinum-ddr3-memory/, Dec 2014.

[Int14] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Sep 2014. Current energy status register information in volume 3B, chapter 14, on page 35.

[KAB+03] Nam Sung Kim, Todd Austin, D Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.

[KBSW11] J.G. Koomey, S. Berard, M. Sanchez, and H. Wong. Implications of historical trends in the electrical efficiency of computing. *Annals of the History of Computing, IEEE*, 33(3):46–54, March 2011.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[Kei14] James Keirstead. Positive coefficient regression in R. http://www.jameskeirstead.ca/blog/positive-coefficient-regression-in-r/, Dec 2014.

[KFJ+03] R. Kumar, K.I Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92, Dec 2003.

[Kin14] William B. King. R model formulae. http://ww2.coastal.edu/kingw/statistics/R-tutorials/formulae.html, Dec 2014.

[KR06] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM.

[LAS+13] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. The mcpat framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):5, 2013.

[Men14] Mentor Graphics ModelSim. http://www.mentor.com/products/fv/modelsim/, October 2014.

[Mod14] Modelsim compile script. https://www.doulos.com/knowhow/tcltk/examples/modelsim/, Dec 2014.

[MSB+05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[NHM+09] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[Ope13] Opencores.org. Amber ARM-compatible core. http://opencores.org/project,amber, November 2013.

[PPPM12] Yongjun Park, Jason Jong Kyu Park, Hyunchul Park, and Scott Mahlke. Libra: Tailoring SIMD execution using heterogeneous hardware and dynamic configurability. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 84–95, Washington, DC, USA, 2012. IEEE Computer Society.

[R14] The R Project. http://www.r-project.org/, Dec 2014.

[RAS+14] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. *IEEE International Symposium on Workload Characterization*, 2014.

[Rie14] Gerhard Rieger. socat (1) man page. http://www.dest-unreach.org/socat/doc/socat.html, Mar 2014.

[RIS14] The RISC-V instruction set architecture. http://riscv.org/, Dec 2014.

[RSR+07] Suzanne RIVOIRE, Mehul A SHAH, Parthasarathy RANGANATHAN, Christos KOZYRAKIS, and Justin MEZA. Models and metrics to enable energy-efficiency optimizations. *Computer*, 40(12):39–48, 2007.

[SBM09] Karan Singh, Major Bhadauria, and Sally A McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[Sem14] Nordic Semiconductor. Capital markets day presentation. http://www.nordicsemi.com/eng/content/download/60199/1020095/version/1/file/Capital+Markets+Day+September+15+2014+(1).pdf, Sep 2014.

[Sno10] David Snowdon. *Operating System Directed Power Management*. PhD thesis, The University of New South Wales, 2010.

[SS13a] Sriram Sankaran and Ramalingam Sridhar. Energy modeling for mobile devices using performance counters. In *Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on*, pages 441–444. IEEE, 2013.

[SS13b] Valery Sklyarov and Iouliia Skliarova. Digital hamming weight and distance analyzers for binary vectors and matrices. *Int. Journal of Innovative Computing, Information and Control*, 9(12):4825–4849, 2013.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM.

[Ste03] M Stein. Crossing the abyss: asynchronous signals in a synchronous world. EDN design feature, July 2003.

[SWPC10] Dam Sunwoo, G.Y. Wu, N.A Patil, and D. Chiou. PrEsto: An FPGA-accelerated power estimation methodology for complex systems. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 310–317, Aug 2010.

[Syn10] Synopsys. *Design Compiler User Guide*, Jun 2010.

[Syn14a] Synopsys. *PrimeTime PX User Guide*, Jun 2014.

[Syn14b] Synopsys. *VCS User Guide*, May 2014.

[Syn14c] Synopsys Design Compiler. http://www.synopsys.com/tools/implementation/rtlsynthesis/pages/default.aspx, November 2014.

[Syn14d] Synopsys PrimeTime. http://www.synopsys.com/Tools/Implementation/SignOff/Pages/PrimeTime.aspx, October 2014.

[Syn14e] Synopsys VCS. http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx, October 2014.

[Sys] ARM System Mode. http://www.heyrick.co.uk/armwiki/Processor_modes#System_mode.

[Tay12] Michael B. Taylor. Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1131–1136, New York, NY, USA, 2012. ACM.

[Ver] Versatile Express Product Family. http://www.arm.com/products/tools/development-boards/versatile-express/.

[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*, ASPLOS XV, pages 205–218, New York, NY, USA, 2010. ACM.

[WA09] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, April 2009.

[Wal14] Magnus Walstad. Task Based Parallel Programming on the SHMAC Multi-Core Prototype. Master's thesis, Norwegian University of Science and Technology (NTNU), June 2014.

[WALK10] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[WBR11] Henry Wong, Vaughn Betz, and Jonathan Rose. Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 5–14, New York, NY, USA, 2011. ACM.

[Wei84] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[Wil14] Stephen Williams. Icarus verilog. http://iverilog.icarus.com/, Dec 2014.

[Wis14] Wishbone system-on-chip interconnectino architecture for portable ip cores. http://cdn.opencores.org/downloads/wbspec_b4.pdf, Dec 2014.

[WJ96] Steven JE Wilton and Norman P Jouppi. CACTI: An enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, 1996.

[ZBT14] ZBT SRAM Interface. http://www-mtl.mit.edu/Courses/6.111/labkit/ram.shtml, October 2014.

[ZGKR14] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 17–31, Berkeley, CA, USA, 2014. USENIX Association.

Glossary

Amber An open source implementation of a processor implementing the ARMv2 ISA. Both a three-stage and a five-stage pipeline is supported; in SHMAC, the five-stage pipeline is used.

app kernel Operating system kernels in Barrelfish running on other cores than the first.

application In the context of Barrelfish, an application refers to an executable running on top of the OS-nodes.

Barrelfish An experimental operating system developed by ETH Zürich, whose design goal is alleviating the problems of increasing system diversity, core heterogeneity and core counts by modelling the operating system as a network of independent nodes communicating through message passing.

BSP kernel The operating system kernel in Barrelfish running on the BSP.

CPU driver The privileged portion of an OS-node in Barrelfish. Manages core-local resources, such as CPU time and peripherals. A CPU driver is also referred to as a kernel.

Dark Silicon The phenomenon that transistor integration capabilities outpace the transistor power consumption scaling, requiring increasingly dramatic fractions of the chip area to be unused or "dark" at any given time.

ELF An abbreviation for Executable and Linkable Format, which is the object file format most commonly used on UNIX systems. Amongst other things, ELF files specify the memory requirements for loading the file, relocations which must be handled when the file load location in memory is finally determined, symbol tables for finding globally exported names, and the address of the entry point into the binary if the ELF file is executable.

executable An executable is a file with machine instructions, static data and a designated entry point.

interface In Barrelfish, an interface is a set of remotely available procedures which may be exported by an executable.

monitor The user space portion of a Barrelfish OS-node. Manages coordination between different OS-nodes.

OS-node In Barrelfish, an OS-node refers to the operating system part managing a given computing device.

service In this dissertation, in the context of Barrelfish we will refer to a service as an interface being exported by a running executable; an interface is a static concept, whereas a service is an interface made available at run-time. For instance, an application wanting to use an interface will allocate a message passing channel and bind to a service.

SHMAC SHMAC, an abbreviation for Single-ISA Heterogeneous MAny-core Computer, can refer to one of two things: a research project at NTNU, or the processor infrastructure developed as part of said project. In the majority of this dissertation, SHMAC will refer to the latter.

shmacfish A bootloader generator program, written to start Barrelfish on SHMAC.

syscall An abbreviation for system call, which is a function exposed by the operating system to user-space programs through which the operating system may be requested to perform some privileged operation on behalf of the program. The set of system calls exposed by the operating system defines the interface between user-level programs and kernel-level operation.

system application In this dissertation, a system application refers to an application which either implements typical OS functionality such as memory management, or which is otherwise specially recognized by or related to Barrelfish.

system software Software which is responsible for control of the machine resources. How broadly the definition spans may vary, but in this dissertation we will include operating systems; drivers; run-time systems such as threading libraries and garbage collectors; compilers; linkers; loaders; and debuggers.

toolchain A set of programs used to create software. A toolchain will typically include a compiler, assembler and a linker, but may also include debugger, loader, etc.

Appendix

A On Accuracy, Precision and Trueness

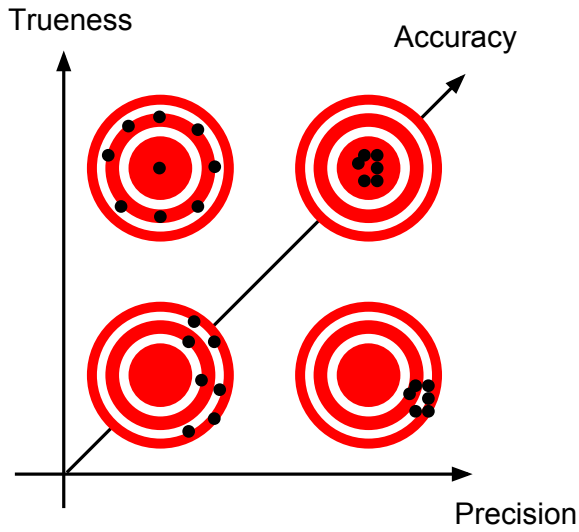

Figure A.1: Figure demonstrating the relationship between accuracy, precision and trueness. Reproduced from [Azo14].

The definition of accuracy, precision and trueness used in this dissertation may be illustrated as in Figure A.1. Along the x-axis precision increases from low to high, and along the y-axis the trueness increases from low to high. These two may be either high or low, independently. The figure illustrates four combinations of precision and trueness: both are low, one is high while the other is low, and both are high. The meaning of the term trueness and precision when talking of estimates of one true value is illustrated using a target. The middle of the target represents the true value. The shots on the targets, drawn as black dots, represent different estimates of the true value. The trueness and precision describe how these estimates are spread

relative to the true value.

The precision of estimates describes the variability of estimates. Low precision means high variability, whereas high precision means low variability. The quality of estimates illustrated in the bottom right corner of the graph, where precision is high but trueness is low, may be describes as “consistent, but consistently wrong”.

The trueness of estimates describes the distance between the mean value of the estimates and the actual true value. When trueness is low, the estimates are consistently skewed from the true value. The mean value of the estimates will therefore have a different value than the actual true value. When trueness is high, there are no such consistent errors. There may still be random estimation errors, leading to low precision, but these will on average cancel each other out. This leads to a mean value of estimates close to the true value.

With accuracy defined to be the combination of trueness and precision, it will increase when both trueness and precision increases. High accuracy requires both high trueness and high precision.

Appendix B
Software Infrastructure User Guides

This chapter presents user guides for all the software infrastructures developed to support the energy efficiency estimation part of this dissertation. The guides give a brief overview of the construction and composition of each tool, what options are available within it, and how it is meant to be used. As always, the authoritative source of information is the source code, and so the sections also refer to where in the SHMAC project source tree the products may be found. All paths will refer to locations relative to the root of the SHMAC source tree unless context makes the relative location clear.

B.1 ASIC Flow

The ASIC flow refers to the use of tools which enable the regression analysis between activity and power necessary to create models of energy consumption. This flow involves three steps: synthesis, simulation, and power consumption calculation. Figure B.1 illustrates the dependence between these steps in terms of inputs and outputs. All the scripts which manage these tasks are located beneath the `asic-synthesis` folder. At NTNU, all the tools are installed on the server `mercury.iet.ntnu.no`.

B.1.1 Synthesis

ASIC synthesis is handled by the tool Synopsys Design Compiler [Syn14c]. This tool takes as input HDL source files, a cell library, a set of constraints, and execution environment specifications. Its output is a gate-level netlist, describing how the HDL design may be implemented using gates from the provided cell library while respecting the set of constraints in the specified execution environment.

The synthesis process is implemented with setup scripts in the `asic-synthesis` folder, synthesis control scripts in the `asic-synthesis/scripts` folder, shell scripts running Design Compiler, and a Makefile target.

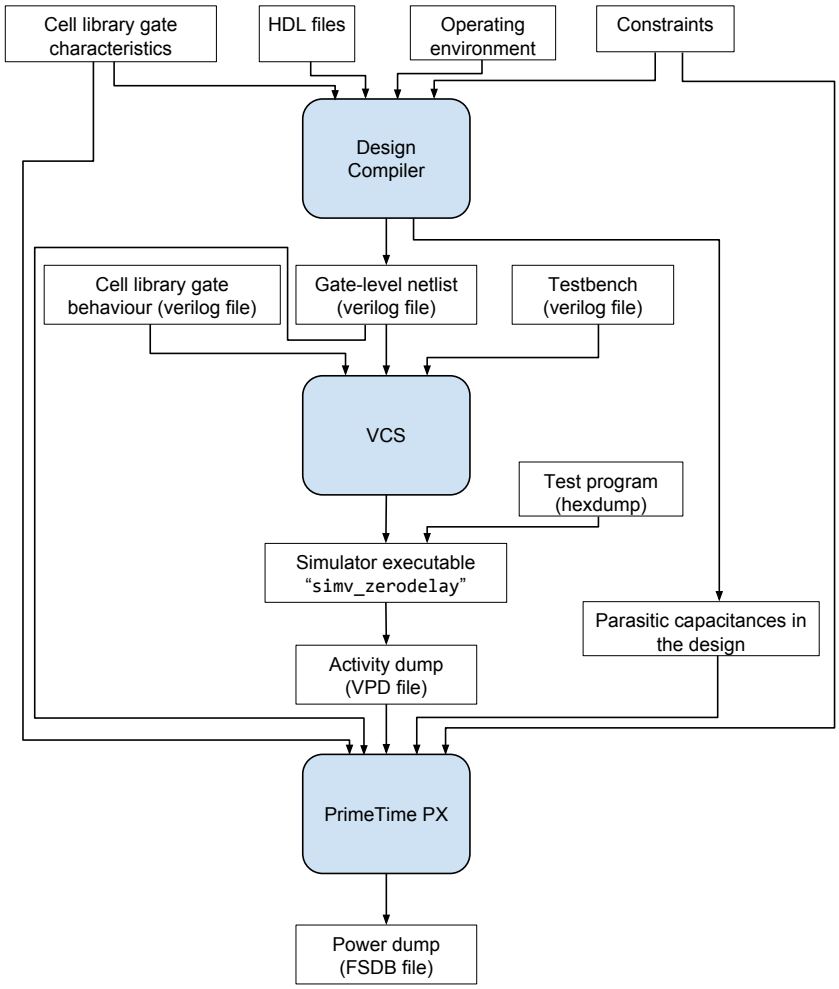

Figure B.1: Dependencies between different tools in the ASIC flow.

Setup Scripts The setup scripts are called `homefolder_.synopsys_dc.setup` and `localfolder_.synopsys_dc.setup`. These scripts initialize synthesis environment variables, such as defining where the cell library file is located and where the SHMAC source folders are. The `homefolder_.synopsys_dc.setup` should be installed in your home directory as `.synopsys_dc.setup` on the server where Design Compiler is running. The `localfolder_.synopsys_dc.setup` file should be modified to reflect your local setup, but its installation is automatically managed by the synthesis shell scripts.

Synthesis Scripts The Design Compiler tool may be programmatically controlled using TCL scripts. The scripts in `asic-synthesis/scripts` read in the HDL source files, elaborate the top-level design, set the constraints and operating environment, and compile it. The entry point is the file called `complete_synthesis.tcl`.

To reduce synthesis time, a partial bottom-up synthesis strategy is employed [Syn10]. The synthesis script first compiles the cache tag and data array structures, the amber core, and the scratchpad memory, setting the `dont_touch` attribute on each design. This causes the compiled design to be reused at all locations, so that recompiling the Amber core for every Amber tile on the system becomes unnecessary.

Which HDL files to read are specified in the `read_hdl.tcl` file in the folder `asic-synthesis/scripts`. If the list of HDL files in the SHMAC project is altered, this file should be updated accordingly, potentially with a version of the change specific to the ASIC design.

Constraints are specified as script files in the `asic-synthesis/scripts/constraints` folder. The only constraints currently used is a target clock period, but other constraints may also be used; see the Design Compiler user guide for details. The default operation environment specification is currently used, so there are no environment definition files. There is, however, a folder called `asic-synthesis/scripts/environment` which should hold any environment definitions created.

The output created are Synopsys design database `.ddc` files for each design which was separately compiled, and a gate-level netlist, constraints file, and parasitic capacitance file for the SHMAC toplevel.

Shell Scripts and Makefile Three shell scripts are responsible for actually executing the design compiler tool. These are located in the `asic-synthesis` folder. When running synthesis, directories called `syn/pass`X are used to store the build results. The X indicates the synthesis run number, which is incremented each time to retain the results from time-consuming synthesis as the default action. The folders should be deleted if the synthesis is outdated. The script `asic-synthesis/new_synthesis_folder.sh` calculates the lowest number Y for which there is no `syn/pass`X folder where $X < Y$, and creates the folder `syn/pass`Y. If no folder exists, the folder `syn/pass1` is created. The name of the newly created folder is sent to the script `asic-synthesis/synthesize.sh`, which copies the `localfolder_.synopsys_dc.setup` file the newly created directory and runs the Design Compiler executable from this location. Design Compiler is passed the file `asic-synthesis/scripts/complete_synthesis.tcl`. Once the synthesis completes, `asic-synthesis/copy_synthesis_results.sh` creates symbolic links to the gate-level netlist, the constraints file, and the parasitic capacitance file which was

produced by the synthesis.

These shell scripts are used in the Makefile target `synthesize`. Thus, running `make synthesize` is sufficient to start a new synthesis job. The target is phony, and will therefore run any time no matter whether other synthesis results exist or not.

B.1.2 Simulation

Simulation is performed using the tool Synopsys VCS [Syn14e]. The tool works in two stages, as indicated in Figure B.1. First, the tool is fed the behavioural description of the gates in the cell library, the gate level netlist from synthesis, and the testbench. When run with this input, the tool produces an executable called `simv_zerodelay`. The executable is suffixed `_zerodelay` because VCS is configured with the `+delay_mode_zero` flag to produce a simulator which does not use delays when calculating signal propagation [Syn14b].

The executable implements the simulation of stimulating the gate-level netlist with the test bench. The testbench used works by reading the contents of a file called `shmac.hex`, which is a text file containing the hex dump of a SHMAC executable, and loading it into the memory of the SHMAC under test. As such, `shmac.hex` is an implicit input for the simulator. Running the executable starts the simulation. The result is a dump of signals in the file `output.vpd`, where the inclusion of signals is as specified in the testbench.

There are two makefile targets which implement the simulation steps. The target `simv_zerodelay` produces the simulator executable, if there has been any change in the netlist and the testbench. The target `output.vpd` produces the output file by running the simulator, if there has been any change to either `simv_zerodelay` or `shmac.hex`. The target `simulate` is a synonym for `output.vpd`, so running `make simulate` will build the simulator and run the simulation if the targets are not up to date.

B.1.3 Power Estimation

Power estimates are calculated using Synopsys PrimeTime PX [Syn14d]. The program works by first reading the gate-level netlist, the cell library, parasitic capacitances and constraint files. Then, it works its way through the activity dump file produced by the simulator, calculating for each event the power consumption resulting from this event based on the resulting activity in gates and the characterized energy responses for these gates. The energy is summed up over the interval of one clock cycle, and then averaged over the clock cycle interval to calculate the power consumption this cycle. PrimeTime PX estimates also include static power consumption [Syn14a].

Which modules energy estimates are collected for is specified in the `modules.tcl` file. The file defines a TCL array of modules in the SHMAC design for which power estimates should be calculated. This file may be modified to reduce the run-time of the power analysis by only including power estimates for relevant parts of the design.

The script `analyze_power.tcl` controls PrimeTime PX, running the power estimation as previously described for each module under consideration. The power estimates also include all children of the modules listed, except leaf nodes in the module hierarchy. This way, the output file does not explode in size with data for all gates. For each module called *module*, the power estimates are stored in a file called *module*`_power.fsdb`. The PrimeTime session is stored in the folder `ptsessions/`*module*.

There is also a Makefile target for invoking PrimeTime PX. The `modules.tcl` file is parsed with an `awk`-command to determine the expected set of output files. If these are not all newer than the analysis dependencies, then PrimeTime is invoked with the `analyze_power.tcl` script. The Makefile target `power` is a convenient alias for the output power files, so running `make power` will run PrimeTime for all modules which do not have up to date analyses.

It is possible to constrain the number of modules for which power is estimated by including the flag `-include top` in the `analyze_power.tcl` file, and listing all the modules for which power should be gathered in `modules.tcl`. This trades off result file size for computation time, since the analysis is rerun for each module. This may be necessary if the amount of activity data which it is desired to analyse results in prohibitively large power estimate files.

B.2 Benchmark Framework

Creating and running benchmarks is the method for creating test and validation data using the simulator and power analysis support described in Appendix B.1.2 and Appendix B.1.3. To facilitate this effort, a generic infrastructure was created which automates and structures the process of obtaining activity and power data from a set of C or assembly benchmark programs. The benchmark infrastructure is located in the folder `asic-synthesis/microbenchmarks`, with its connection to the ASIC flow implemented in the Makefile in the `asic-synthesis` folder.

To run the benchmarks in the VCS simulator, the already existing SHMAC toplevel test bench is used as this is already designed to load programs into the memory of a SHMAC configuration and run it. This is exactly what is needed for running benchmarks. The testbench was also extended with a facility for the running program to signal its termination. If a write to memory address zero is detected, the testbench

will terminate. This allows running benchmark programs from start to end in the simulator.

The `microbenchmarks` folder contents is split in three: a `Makefile` controls the microbenchmark build process, a `lib` folder contains microbenchmark utility code, and a `src` folder contains all the benchmarks.

The benchmark utility code in `microbenchmarks/lib` include the microbenchmark entry point which sets up run-time stacks, and some utility functions such as printing text, registering interrupt handlers, and enabling or disabling caches. The benchmark library run-time jumps to the actual benchmark code by calling the `main` function, which all benchmarks are expected to define. As little configuration as possible is done before `main` is called, since the library is not in charge but only a utility to simplify benchmark development. Each benchmark should still be the master of the state of the system at any point but the very beginning and the very end. Once the `main` function of the benchmark completes, the run-time prints the string "DONE" and ends the benchmark by writing to address zero. Emitting this string right before the end allows gauging by output inspection whether the benchmark terminated successfully or not.

The `microbenchmarks/src` folder is assumed to contain one file per benchmark. Multi-file benchmark must place the files without the `main` function in the `lib` folder. The files may be either assembly files with the extension `.s` or `.S`, or C files with the extension `.c`. Assembly files with the extension `.S` are run through the preprocessor first, so macros are available.

The Makefile in the `microbenchmarks` folder calls the ARM toolchain with prefix specified by the variable `CROSS`. By default, `CROSS` is set to `arm-none-eabi-`. The makefile creates one binary for each benchmark in the folder `microbenchmarks/bin`, which may be run on the SHMAC platform instantiated on an FPGA for fast verification of correct termination. If termination is not almost instant when run on the FPGA, this is a sign that the benchmark is too extensive for running through the simulator. Additionally, the makefile creates one textfile hexdump for each benchmark in the folder `microbenchmarks/hex` in the format expected by the SHMAC testbench.

The `microbenchmarks` folder structure is exploited in the Makefile in the `asic-synthesis` folder. The Makefile contains a phony target called `microbench_activity`, which first runs `make` in the `microbenchmark` directory to create all the hex files. Then, it iterates through all the hex files. For each hex file, it creates a symbolic link to it named `shmac.hex` in the `asic-synthesis` folder, before running `make simulate`. When simulation is done, the resulting `output.vpd` file is copied to `microbenchmarks/results/`*benchmark*`.vpd`. To avoid simulating benchmarks for which activity is already present, there are Makefile targets which explicitly state

the dependence between the output activity files and the input hex files.

In the `asic-synthesis` folder Makefile, there is another target called `microbenchmark_power`. It works similarly to `microbenchmark_activity`, except that it is dependent on the activity files in the `microbenchmarks/results` folder instead of hexfiles and produces power estimate output files in directories called `microbenchmarks/results/`*benchmark*`_power`. The phony target works by iterating through all benchmarks, making a link to the corresponding activity file, running `make power`, and copying the result to the appropriate directory.

To make invocation simpler, there is also a third target `microbench` which is an alias for `microbench_power`. Running `make microbench` will therefore build all unbuilt benchmarks, gather all non-existent activity data, and calculate all missing power data.

B.3 Regression Modelling

This section describes the tools which have been developed to facilitate regression modelling. Appendix B.3.1 will describe how the output files from simulation and power analysis are converted to text files, and how subsets of signals and modules may be selected from these to reduce the amount of data down to what is interesting for the module which one is going to model. Appendix B.3.2 will then describe how the resulting files, containing signals and power values for the module in question, may be analysed, and the workflow which is used when creating models.

B.3.1 Data Scraping

The output from synthesis is in a binary format called VPD, and the output from power analysis is in a binary format called FSDB. Being binary file formats, they are more difficult to reverse-engineer and as documentation is scarce they are not easily processed by custom programs. The first step in the processing of these files is therefore to convert them to a text file equivalent, using the proprietary tools `vpd2vcd` and `fsdb2ns`. The first program may be used to convert the simulation output to a VCD text file, and the second program may be used to convert the output from power analysis to the textual OUT format by passing the flag `-fmt out` to the `fsdb2ns` tool.

The use of these tools has been automated using Makefile targets in the `asic-synthesis` Makefile. The target `vcd` converts all VPD files in the `microbenchmarks/results` folder to VCD files, and the target `out` converts all FSDB files in sub-folders of `microbenchmarks/results` to OUT files. These targets should be invoked after simulation and power analysis of the benchmark programs has been completed.

The next step is selecting a subset of activity and power data from the VCD and OUT files and converting it to a format readable by the statistical processing program R [R14]. This can be done using two programs written for this project: `ConvertVcd`[1] and `OutToDataFrame`. These programs accept a VCD or OUT file, respectively, as well as an output file argument. Once run, the user is prompted for a timescale. This may be used to coarsen the data samples used for statistical modelling. When using zero-delay simulations, specifying the clock period as the time scale is typically sufficiently coarse.

The `ConvertVcd` program also accepts an optional argument of an existing output file, which may be used to specify which signals should be selected. If this argument is passed to `ConvertVcd`, the same signals as those in the file are selected from the VCD file. If not, the user is presented with a menu as depicted in Figure B.2.

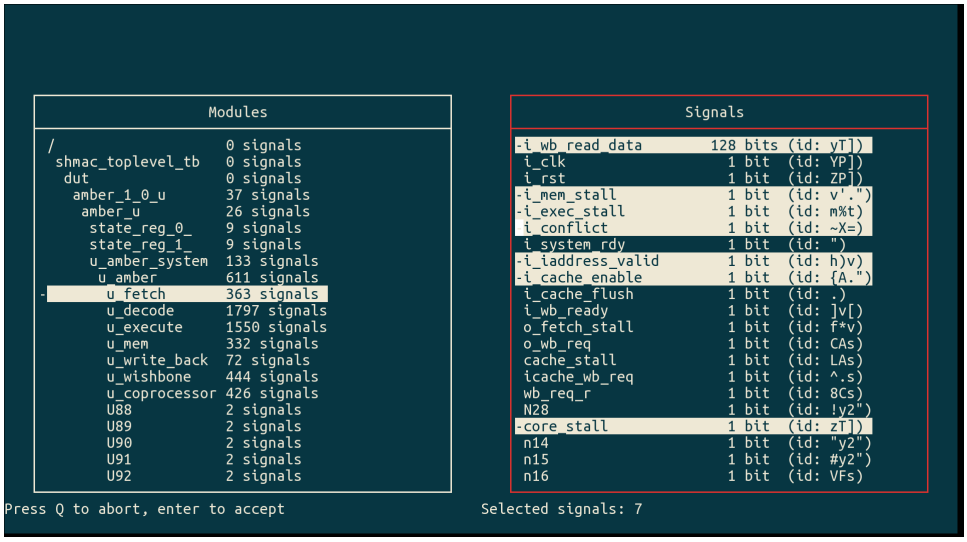

Figure B.2: A screenshot demonstrating the `ConvertVcd` utility program.

The `ConvertVcd` program uses two menus: one for browsing the hardware module hierarchy, and one menu for selecting signals for inclusion in the output file. The menus may be navigated using the arrow keys, and the active menu may be toggled using the tabulator key. Signals which are included are highlighted in the menu. Signal selection may be toggled by navigating to the desired signal menu entry, and hitting the space bar. If the space bar is hit while the module menu is active, all signals in the current module will be added unless they are all already included in

[1] The `ConvertVcd` program will be renamed `VcdToDataFrame` in the future.

which case they will be removed instead. Once selection has been completed, hitting enter will start the conversion. Hitting the Q key will abort program execution.

The `OutToDataFrame` program is similar to `ConvertVcd` in operation. However, since power consumption is reported per module only the left-most menu is used for active selection of modules. The right-most menu is used to list which modules have been selected.

To make it easier to automate the statistical processing, the input file organization has been conventionalized. Data files gathered from the execution of test benchmarks should be stored in a folder named `test-data`, and files gathered from validation benchmarks should be stored in a folder named `validation-data`. These folders should be located where it is desirable to run the modelling. The files for each benchmark should be stored in a separate sub-folder with the name of the benchmark. For instance, data files gathered from the Dhrystone validation benchmark are stored in the folder `validation-data/dhry/`.

Conventions also exist for data file contents. Since power data files are not prohibitively large, power data for all relevant modules should be included in a file called `power.dat`. Signal data should be split into one file per module, named `<module name>_signals.dat`. As an example, if the module *u_execute* is being modelled, power and activity data from the Dhrystone benchmark would be found in `validation-data/dhry/power.dat` and `validation-data/dhry/u_execute_signals.dat` respectively.

The script `regression/convdata.sh` has been developed to simplify following these conventions. This script contains a list of validation and test benchmarks. Additionally, the script makes assumptions about where the OUT and VCD files are located for benchmarks. The list of benchmarks should be updated when new benchmarks are created, and the location benchmark VCD and OUT files must be customized for each user. Having made any necessary customizations, the script should be invoked using one of the following three symbolic links:

benchdata.sh This program accepts a module name and a benchmark name, and runs `ConvertVcd` with input and output file names adhering to the previous conventions. Similarly to `ConvertVcd`, it is also possible to pass the name of an existing output file as a third optional argument which is used to specify the signal selection.

powerdata.sh This program accepts a benchmark name, and invokes `OutToDataFrame` with conventional input and output file names.

validations.sh This program is used to automate the creation of signal data used for validation benchmarks. Since this has to mirror what has been used when modelling with the test benchmarks, the signal data files existing in the `test-data` folder are used to automate the creation of similar validation data. The program thus accepts a list of validation benchmark names, and for each a set of signal files matching those existing in test benchmarks are created in an appropriate sub-folder of `validation-data`. The time scale prompt when running `ConvertVcd` is slightly obstructive for automation, but may be circumvented as follows:

```
while true; do echo <timescale>; done | ./validations.sh <benchmarks>
```

In addition to the programs preparing statistical processing, a program called `OutToMatlab` may be used to generate matlab code which plots the power consumption of selected modules.

B.3.2 Statistical Processing

Initialization Statistical processing starts by running the R program in the directory in which the directories `test-data` and `validation-data` reside. Personally, I use a separate `modelling` directory with symbolic links to `test-data` and `validation-data`. It is recommended to run R from Emacs using the ESS package, which offers an interactive terminal with syntax highlighting, auto-completion, and text editor integration such as debugging.

The statistical processing support is located in two files: `evaluations.R` and `dataconv.R`. It is recommended to create a `.Rprofile` file in the modelling directory which automatically loads the support files as follows:

```
1 source("path/to/evaluations.R")
2 source("path/to/dataconv.R")
```

Data Structures Once R is running and the two support files are loaded, modelling may commence. The R code operates on three different data structures, which the modeller will interface:

1. A *benchmark* is a collection of power and signal data for a selection of hardware components, resulting from running one benchmark through simulation and power analysis. Power data may be included independently of signal data, and vice versa. Power is represented as a list of floating point numbers, whereas signal data contains data for each signal selected with the tools in Appendix B.3.1. An example benchmark instance is depicted in Figure B.3.

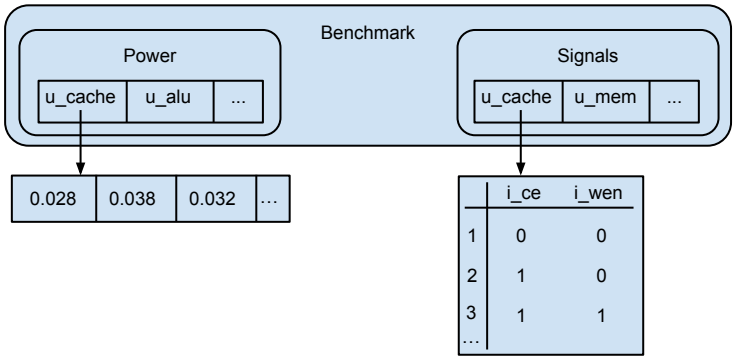

Figure B.3: The benchmark data structure. All the components are fields, accessible by name.

2. A *power model* is obtained through regression analysis of benchmark power and signal data. The data structure is depicted in Figure B.4. The power model targets a specific hardware module, identified by the "target" field, and is most typically composed of models for submodules. Two optional fields, "internal" and "children", are used when modelling the energy consumption of signals included in the target module only. The "children" field is a list of children whose power consumption should be subtracted before running regression modelling, and the "internal" field stores the resulting regression model. A power model may also contain a transform, which is used on benchmarks either when modelling or evaluating a model. For instance, since we model cache power consumption separately we want to transform the benchmark data by subtracting the cache power from all parent modules.

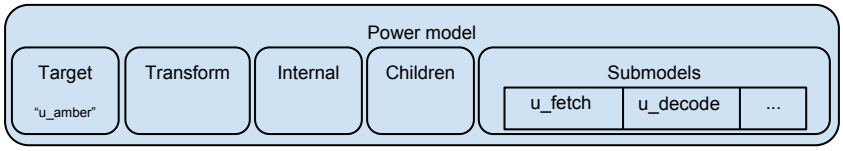

Figure B.4: The power model data structure. All components are fields, accessible by name.

3. An *evaluation* is a collection of statistics denoting how well a power model predicts the power consumption in a benchmark using its signal data. The evaluation objects are created by predicting the power consumption from a benchmark object signal data based on a power model object, and comparing the predictions to the power data in the benchmark object. The fields currently included in evaluation objects are listed in Table B.1.

Table B.1: The fields included in the evaluation data structure.

Field name	Semantics
`predicted.power`	A list of predicted power values per cycle
`actual.power`	A list of the actual power values per cycle
`differences`	`predicted.power - actual.power`
`absolute.differences`	`abs(differences)`
`errors`	A list which gives per-cycle error values. The error is relative to the predicted power if the predicted power was less than the actual power, and relative to the actual power if the predicted power was larger. This makes an error of 100 % correspond to an estimate twice as large as the actual value, and an error of −100 % correspond to an estimate half of the actual value.
`absolute.errors`	`abs(errors)`
`fraction.too.large`	A list of per-cycle error values, always relative to the actual power.
`fraction.too.small`	A list of per-cycle error values, always relative to the predicted power.
`errors.average`	The average value of `errors`.
`absolute.errors.average`	The average value of `absolute.errors`
`mean.fraction.too.large`	The average value of `fraction.too.large`
`mean.fraction.too.small`	The average value of `fraction.too.small`
`most.negative.error`	The minimum value of `errors`
`most.positive.error`	The maximum value of `errors`

Modelling Workflow Overview When creating a model for some module from scratch, the order of operation will typically be the following:

1. Create a benchmark object representing the test benchmark, with signal and power data included as necessary.

2. Create a benchmark object representing the validation benchmark.

3. Create a model object which will represent the module being modelled, with the transform set appropriately.

4. For each submodel in the modelling granularity:

    a) Run regression modelling using the test benchmark and a formula.

b) Evaluate the model quality using the model coefficient of determination (R^2) and the test benchmark (how to do this is detailed further on in this section).

c) If the previous point indicates acceptable model quality, add the necessary signal and power data to the validation benchmark object and evaluate the model quality using the validation benchmark.

d) If quality is deemed insufficient, revise regression formula and retry.

5. Once all submodels are created, a model may optionally be created for signals internal to the target module.

6. Finally, the entire power model should be evaluated using the validation benchmark. The impact of coefficient resolution should be examined. Once a satisfactory model is created, the model is stored to file.

To make it easy to reproduce a modelling session, it may be a good idea to store the commands used in files. I create functions in files `modelling/create.validations.R`, `modelling/create.models.R` and `modelling/create.evaluations.R` which create different validation benchmarks, power models and evaluations respectively. Using ESS-mode and Emacs, these functions can be written to file and fed to the R shell using the "Ctrl+c Ctrl+j" keyboard shortcut. This way, it is not necessary to retrace what commands was executed at the end of a modelling session to write these files.

Utility Functions The modelling workflow is supported by the functions in `evaluations.R` and `dataconv.R`. The available functions are tabulated in Table B.2. The third column of the table indicates at which point in the previous workflow the function is likely to be of use.

For the four categories in Table B.2, the first argument is typically respectively a benchmark object; power model object; evaluation object; or list object. Use argument completion in the R shell or consult the source files for more detailed explanation of function arguments. One special argument worth mentioning here is the `resolution` parameter to the `pm.evaluate` and `pm.write` functions. This optional parameter may be set to an integer, corresponding to the number of decimal places to be included in the coefficients when predicting power consumption. If it is not set, the full coefficient precision is used.

There are other utility functions in the `evaluations.R` file as well. For instance, several transformation functions exist. One example is the `shrink` function, which removes the final element from all benchmark power data since the VCD files contain one less cycle of data than the OUT files. The functions in Table B.2 are, however, the ones most likely to be useful in general.

Table B.2: The functions available in `evaluations.R` and `dataconv.R`. The four groups of functions in the table relate to benchmarks (prefix `bmd.`), power model (prefix `pm.`), evaluation processing, and functions used in regression model terms..

Function name	Effect	Use
`bmd.new.test`	Create a new benchmark by folder name in `test-data`.	1
`bmd.new.validation`	Create a new benchmark by folder name in `validation-data`.	2
`bmd.add`	Add signal and power data for a module.	1, 4c
`bmd.add.power`	Add power data for a module.	1, 4c
`bmd.add.signals`	Add signal data for a module.	1, 4c
`bmd.add.all`	Can be used to add power and/or signal data for several modules.	1
`pm.new`	Create a new power model object.	3
`pm.with.new`	Add a new submodel to a model object.	4a
`pm.with.internal`	Add a new internal model to a model object	5
`pm.write`	Store the power model object to file.	6
`pm.evaluate`	Evaluate the power model quality for a given benchmark object.	4b, 4c, 5, 6
`print.evaluation`	Prints all data in the evaluation.	4b, 4c, 5, 6
`summary.evaluation`	Prints a summary of the aggregate statistics in the evaluation.	4b, 4c, 5, 6
`plot.evaluation`	Create a standard plot of an evaluation.	4b, 4c, 5, 6
`resolution.impact.plot`	Plots the impact of model coefficient resolution given a set of evaluations with different resolution used.	6
`sliding.error`	Calculates the average evaluation error over a given time window size for all time windows of this size.	6
`HD`	Calculates the Hamming distance between subsequent values in a list of signal values, returning a new list. The first Hamming distance value is calculated between zero and the first list element value.	4a
`lbit`	Extract a bit at a given position for a list of signal values, returning a new list.	4a
`lbits`	Extract a range of bits from a list of signal values, returning a new list.	4a
`change`	Calculates changes in a list of signal values, returning a list of boolean values.	4a

Creating a New Regression Model The majority of the creative modelling work is the execution of step 4a, in which a formula using benchmark signals is correlated with the benchmark power data. We therefore illustrate how this step is executed in more detail.

The most frequently used function in this step is `pm.with.new`. To explain it, consider this example invocation:

```
1 core.model <- pm.with.new(core.model, "u_alu",
2                           ~ HD(i_a_in)*change(i_alu_function) + i_carry,
3                           test.bm)
```

There are four input arguments to the function. The first is the power model object, the second is the name of the target submodule, and the fourth argument is the test benchmark object. The third argument is an R expression for a formula. The general syntax for a formula is `<dependent variable> ~ <independent variables>`, which describes a relationship between the dependent variable and the independent variables. The dependent variable will be set by the modelling utility functions, after transforming the benchmark power data with its `transform` function field, and may therefore be skipped. The list of independent variables also describes which variable interactions should be considered. Using a `+` operator between two variables means the two variables should be considered independently, whereas using a `*` means that a cross product should be included as well. An intercept is always included unless the independent variable specification contains the term -1. Thus, our example tries to create a regression formula of the form $\alpha_1 \cdot HD(i\_a\_in) + \alpha_2 \cdot change(i\_alu\_function) + \alpha_3 \cdot HD(i\_a\_in) \cdot change(i\_alu\_function) + \alpha_4 \cdot i\_carry + \beta$. Consult [Kin14] for a more complete overview of the syntax for writing R formulae.

The dependent variables are resolved based on their names. The utility function sets the signal data in the benchmark object to be an extra name space during modelling. In our example invocation, the names of variables in the `test.bm$signals` field is used.

In our example invocation, the dependent variables are largely the result of function calls. R supports this by evaluating the functions before running the regression modelling, as one would expect. It is therefore in these formula specifications the functions in `dataconv.R` are typically used.

For several other examples on how this step may be executed, one may examine the file `modelling/create.models.R`.

B.4 Using the Energy Efficiency Estimation Framework

This section will present the available SHMAC host-side software which enables actually using the energy efficiency estimation framework to perform experiments where the energy efficiency of a hardware or software component is evaluated.

B.4.1 User-space Utilities

The following utility programs, located in directory `software/shmac_tools`, may be executed on the SHMAC host to get energy estimates from the driver:

`shmac_read_energy` prints a new set of energy samples as plain text. The tool is a shell script which prints the contents of the proper driver `sysfs` attribute, and has no arguments or options.

`shmac_stream_energy` continuously reads binary energy data from the proper driver `sysfs` attribute. It has three positional arguments. The first argument is mandatory, and is the name of the output file. The second argument is optional, and specifies the desired sample period in milliseconds. The default value is 5. The sample period argument may be set to zero to read samples in a tight loop with no sleep commands. The third argument is also optional, and specifies the number of sample words to read from the driver. The default value is eight.

B.4.2 Monitoring and Logging Tool

The tools in Appendix B.4.1 are not often used directly. Instead, a separate monitoring and logging tool, runnable on other computers than the SHMAC host, is meant to serve as the front-end for most research purposes. The tool is located in directory `software/shmac_energy_monitor` relative to the SHMAC repository root. The tool is written in python 3, and is started with the command `python3 multicore_plot_test.py`[2]. The tool accepts command line options for specifying the network port to use for incoming energy samples, the IP address of the SHMAC host, the name of the log file, and the option of using Transmission Control Protocol (TCP) as the transport protocol. A summary of command line options may be attained by executing `python3 multicore_plot_test.py -help`.

When the tool starts, it asserts that the computer it is run on is not behind a NAT device. Since the SHMAC host is going to be sending samples across the network to the computer, it needs to be able to set up a connection[3] to the computer running

[2]The application entry point file will likely be renamed in the future.

[3]Support for NAT could be added when using TCP by restructuring the way connections are set up, but there are no plans to implement this at the moment

the monitoring and logging tool. Next, the tool connects to the specified SHMAC host and reads the driver `sysfs` attributes reporting the configuration of the SHMAC infrastructure instance which is currently installed. This way, the tool determines the layout of tiles, the sample sources per tile, and the sample size for each sample source.

Once the tool is initialized, a GUI is presented with one graph area for each tile. An example screenshot is given in Figure B.5. The GUI is split into tiles in the same layout the current SHMAC configuration has. Each tile is used to plot the energy or power consumption of an energy report unit present on that tile. If several units are present, their plots are kept in tabbed windows in each tile. By looking closely at the screenshot, you may see that the upper right, bottom right and bottom left graph tiles contain two tabs. These GUI tiles plot estimates from Amber tiles. The active tab contains the plot of power for the Amber core, and the inactive tab contains the plot of power for the router.

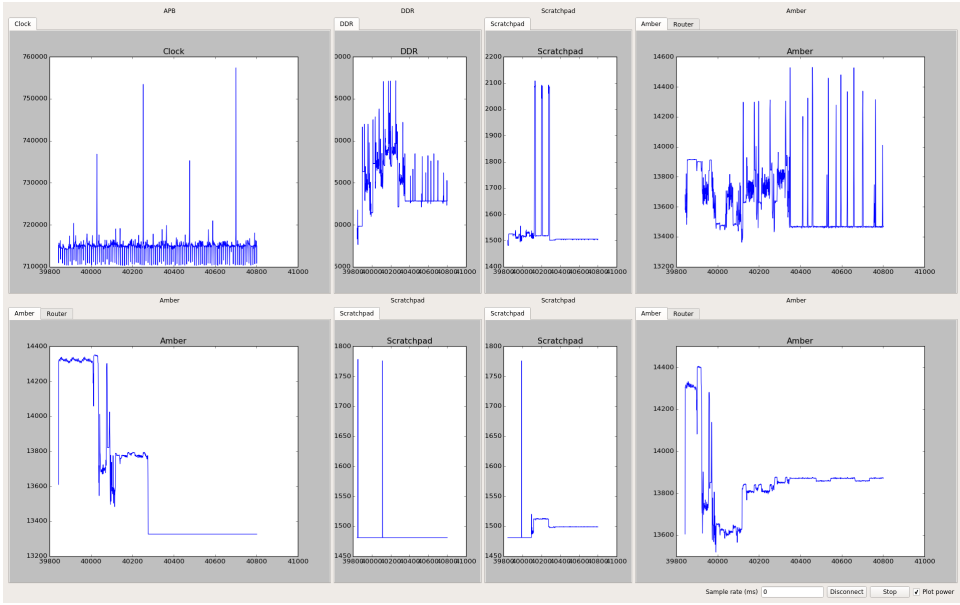

Figure B.5: Screenshot demonstrating the monitoring and logging tool.

The tool is relatively simple to operate, as it contains only two buttons, one check box and one text field. The text field may be used to specify the desired sample rate before starting sampling. The check box controls whether power or energy is plotted by optionally scaling the energy samples with the sample period and clock frequency. The left-most button is used to toggle the connection to the SHMAC host. Pressing

the button while disconnected causes `socat` to be run on the SHMAC host, and a socket of the appropriate type on the specified port to be opened on the researcher's computer. The other button is used to start sampling. Pressing it while sampling is not started causes `shmac_sample_energy` to be started on the SHMAC host, and a new thread is spawned in the GUI program which continuously reads data from the socket connection. New samples are sent to the main GUI thread, which temporarily stores them in an array. In addition, a periodic timer is started which causes an update of the plots every second.

In steps, the tool is used as follows:

1. Launch it with proper command line arguments.
2. Press the left-most button labelled 'Connect'.
3. Configure the sample period. A sample period of 0 ms is the most common choice.
4. Check the 'Plot power' check box (due to sample variability, this is typically desirable).
5. Press the second button labelled 'Start'.
6. The 'Plot power' check box may be toggled during operation if desirable.
7. To finish the sampling, press the second button (which is now labelled 'Stop'). The samples are stored in the file specified on the command line for further analysis.
8. Sampling may be restarted by following the items 5 to 7 again.
9. To disconnect, press the left-most button (which is now labelled 'Disconnect').

B.5 Hardware Testbench Framework

To simplify the testing of hardware modules, a small framework was created which manages the use of the circuit simulator Modelsim [Men14]. The generic simulator control is managed in the files `hardware/tests/setup.do` and `hardware/tests/run_test.do`, located relative to the SHMAC repository root. The first file loads the required libraries, notably the Bitvis Utility test library [Bit14], and reads all the HDL files used in the SHMAC infrastructure. In addition, it sources a file containing useful ModelSim functions courtesy of Doulos [Mod14]. The functions are simply named `r` and `rr`, and cause a recompilation of changed or all HDL files respectively. These functions significantly reduce the turn-around time between updating an HDL file and testing the change in the simulator. The `run_test.do` file accepts an argument

denoting the name of the test bench. This test bench is then loaded, together with the wave form file named the same as the test bench with the suffix `_wave.do` if this file exists.

To make running each specific test easier, separate ModelSim script files called `<test bench name>.do` are created. These files simply first source `setup.do`, and then `run_test.do` with the test bench name as argument. As an example, the test bench `HD_tb` may be run using the file `HD_tb.do` and its wave form is automatically loaded `HD_tb_wave.do`.

Since the per-test-bench script file has a fixed structure, and there is considerable repetition of test bench structure and contents, a script called `gentb.sh` was created. This script accepts the name of the module to be tested as an argument, and generates the files `<module>_tb.do` and `<module>_tb.vhd`. The contents of the ModelSim script file is as previously explained. The VHDL file contains a template test bench, with the overall test bench structure in place but with signal declarations, component declarations and tests missing. A wave form file is not generated: this is created by the tester as needed when running ModelSim.

Appendix C
Complete Modelling Tour

This chapter will present a walk-through of how the software infrastructures presented in Appendix B are used step-by-step to produce a model for some component in the SHMAC platform, and how to integrate it with the infrastructure. This section should be read before developing and embedding new cores, accelerators or other components in the SHMAC platform.

The steps will be presented in the proposed order of execution. More details may be found in the corresponding section in Appendix B. Appendix C.1 will first present a list of prerequisites, which are setup steps which are typically executed only once. Then, Appendix C.2 will list how modelling itself is conducted.

C.1 Prerequisites

- Get access to a server[1] (we will call this the synthesis server) which has the following tools installed:
  - Synopsys DesignCompiler 64-bit edition[2].
  - Synopsys VCS
  - Synopsys PrimeTime PX
  - `fsdb2ns`
  - `vpd2vcd`
- Upload the `hardware` and `asic-synthesis` directories to this server.
- Create a `syn` folder on the same level of the file tree as the `asic-synthesis` folder.
- Install the file `homefolder_.synopsys_dc.setup` as explained in Appendix B.1.1.

[1] I use the `mercury` server at the IET department
[2] The 32-bit edition runs out of memory when synthesizing 2×3 SHMAC designs.

- Modify the file `localfolder_.synopsys_dc.setup` to contain the correct path to your `asic-synthesis` and `hardware` directories.
- Get access to a server with sufficient disk space for containing the large VCD and OUT files[3] (we will call this the storage server).
- Upload the `regression` directory to the storage server.
- Modify the `get_datadir` function in the `regression/convdata.sh` file on the storage server to reflect the location of VCD and OUT files.
- Get access to a server with R version 3.14 installed[4] (we will call this the R server).
- Upload the `regression/{evaluation,dataconv}.R` files to the R server.
- Make a directory on the R server meant for modelling, and create a `.Rprofile` file as explained in Appendix B.3.2.

C.2 Modelling Steps

Synthesis The following steps should operate on the files uploaded to the synthesis server.

1. Modify the list of HDL files in `asic-synthesis/scripts/read_hcl.tcl` according to how you have updated the SHMAC sources.
2. If the addition involves a large module included in several locations in a SHMAC design, it may be desirable to create a new script similar to `asic-synthesis/scripts/compile_a25_core.tcl` which compiles your module alone.
3. If desired, modify synthesis parameters, design constraints and operating environments:
   - Edit the cell library selection in the installed `homefolder_.synopsys_dc.setup` file.
   - Edit design constraints in the appropriate `asic-synthesis/scripts/constraints` file. Consult the DesignCompiler user guide for accepted options [Syn10].

[3] I use the `moog` server at the IDI department, since the disk quota on the synthesis server is small.

[4] I use my own machine, but setting up the environment on a server with large amounts of memory is recommended.

   - Edit the operating environment in the appropriate `asic-synthesis/scripts/environment` file. Consult the DesignCompiler user guide for accepted options [Syn10].
   - Enable clock gating in the synthesis by adding the `-gate_clock` flag to the `compile` command in the compilation scripts in `asic-synthesis/scripts`.

4. Run `make synthesize` in the `asic-synthesis` directory on the synthesis server.

Benchmark Creation The development of benchmarks should be done on your development machine for version control, and be uploaded to appropriate location on the synthesis server when they should be run.

This step may be skipped if your modifications, for instance enhancements to an existing microarchitecture, is sufficiently covered by existing benchmarks.

1. Create either an assembly[5] or a C file in the `asic-synthesis/microbenchmarks/src` folder. The file must define an externally accessible `main` label, which is the benchmark entry point.
2. Write the benchmark. If the benchmark is split into several files, include the files without the entry point in the `asic-synthesis/microbenchmarks/lib` directory. Add include files to the `asic-synthesis/microbenchmarks/include` directory.
3. Specify whether it is a test or validation benchmark in `regression/convdata.sh`.

Simulation and Power Analysis These steps should be run on the synthesis server, in the `asic-synthesis` directory. Ensure that the benchmarks you wish to run have been uploaded there.

1. If you want to get both simulation data and power data from all benchmarks, run `make benchmarks`.
2. If you only want simulation data from all benchmarks, run `make microbench_activity`. To subsequently get power data, run either `make benchmarks` or `make microbench_power`.
3. If you want data from only a selected set of benchmarks, run `make microbenchmarks/results/<benchmark>.vpd` to get simulation data and `make microbenchmarks/results/<benchmark>_power` to get power data.

[5] Use the `.S` extension if you want it to be preprocessed first.

Data Scraping

1. Run `make vcd` and `make out` in the `asic-synthesis` directory on the synthesis server. Copy the resulting VCD and OUT files to the storage server, if necessary.
2. Run `./convdata.sh` for each test benchmark you want to use, with the name of the module being modelled as the second argument.
3. Run `./convdata.sh` for each validation benchmark you want to use, with the name of the module being modelled as the second argument.

Statistical Processing

1. Copy the `regression/{test,validation}-data` directories on the storage server to the modelling directory on the R server.
2. Perform the statistical modelling, as explained in Appendix B.3.2.

Infrastructure Integration

1. Create an HDL file in an appropriate location in the SHMAC source tree which implements the model. Instantiate the `HD` module for Hamming distance calculation, and the `change` module for change calculations.
2. Add the file to the list of HDL files in `hardware/express/run_vivado_2012.4.tcl`, `hardware/sim.do` and `hardware/tests/setup.do`.
3. Use the `./gentb.sh` in the `hardware/tests` directory to create test templates as explained in Appendix B.5, and flesh out the template with tests of your monitor ensuring it works.
4. Instantiate the monitor in the module being modelled. Route the energy output to the appropriate aggregation of monitor estimates, or directly to an energy report unit or energy counter register. Test correct integration with ad-hoc top level testing, or a new test bench if deemed necessary.
5. Synthesize the SHMAC with the monitor added, update the driver if the report unit configuration has been changed, and install the new SHMAC and driver on the SHMAC system.

Appendix D
Energy Models

This chapter presents the energy models which were created for different SHMAC platform components.Appendix D.1 and Appendix D.2 present the regression models and analytical models respectively. All model units are given in nanojoules, and the time to overflow is calculated based on 32-bit sample sizes. The model coefficients are listed with five decimal places.

D.1 Regression Models

This section first presents the metrics which are used to evaluate the precision of the models in Appendix D.1.1, before Appendix D.1.2 presents the models and their precision numbers.

D.1.1 Model Precision Metrics

In expressions in this section, the variables $f_1, ..., f_n$ will denote estimates by the power model, the variables $y_1, ..., y_n$ will denote actual power values as calculated by Synopsys PrimeTime, and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. To get a symmetric graph of error values, we define the error in cycle i as

$$error(i) = \begin{cases} \frac{y_i - f_i}{f_i} & \text{if } f_i < y_i \\ \frac{y_i - f_i}{y_i} & \text{otherwise} \end{cases}$$

Coefficient of Determination The coefficient of determination, denoted R^2, is used to describe the quality of a regression model. It is calculated as $1 - \frac{\sum_{i=1}^{n}(y_i - f_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$. The value ranges from between 0 and 1, with 0 indicating no fit and 1 indicating a perfect fit. The value is typically used when modelling as an indication of when one may expect the model to be of sufficient quality to accurately predict validation benchmarks. The values listed in tables in Appendix D.1.2 are calculated from the

test benchmark power data compared with model predictions, and as such indicate at what R^2 values models of different quality may be expected.

Relative error in average energy The relative error in average energy is the difference between the total estimated and actual energy divided by the total actual energy. Expression: $\frac{\sum_{i=1}^{n} y_i - f_i}{\sum_{i=1}^{n} y_i}$

Average Cycle-by-Cycle Error The average cycle-by-cycle error is the average error seen each cycle. Expression: $\frac{1}{n} \sum_{i=1}^{n} error(i)$

Absolute Average Error The absolute average error is the average of absolute cycle-by-cycle error values. This measure does not average out errors alternating between being too large and too small. Expression: $\frac{1}{n} \sum_{i=1}^{n} |error(i)|$.

D.1.2 Current Models

All graphs in this section plot values summed over a time window of 500 cycles, which is necessary for figure clarity due to the large number of sample points. Models formulae are listed as R formula expressions, described in Appendix B.3.2.

Certain graphs appear to be more erroneous than their statistics say, which is because the resolution of the y-axis is as fine-grained as possible. This allows a more clear inspection of model precision than what is possible if the y-axis scale is set to reflect significant variations.

All models except that of the Amber system have high average precision, but several of the models for the submodules do not accurately capture trends. This is not of great importance for the following reasons:

- Several modules have a power consumption which is completely overshadowed by other modules, so even significant errors in the models for these modules is insignificant.
- The model estimates will likely be gathered at higher time window granularities than 500 cycles, which will make the average power estimate a more valuable metric than per-cycle accuracy.
- The models are most likely being made superfluous by changes to the SHMAC infrastructure, as explained in Section 7.2. Thus, only a modest amount of work has been spent optimizing them.

Amber Wrapper The Amber wrapper model is created from the sum of submodule models, which are those listed in the rest of this section. The aggregated Amber model statistics are listed in this paragraph.

Maximum value	7880
Minimum value	1164
Minimum time before overflow (32 bit)	545046 cycles (9.0841 ms)
Maximum time before overflow (32 bit)	3689835 cycles (61.4972 ms)
R^2	0.884
Relative error in average energy	0.3135 %
Average cycle-by-cycle error:	0.7519 %
Absolute average error:	1.3664 %

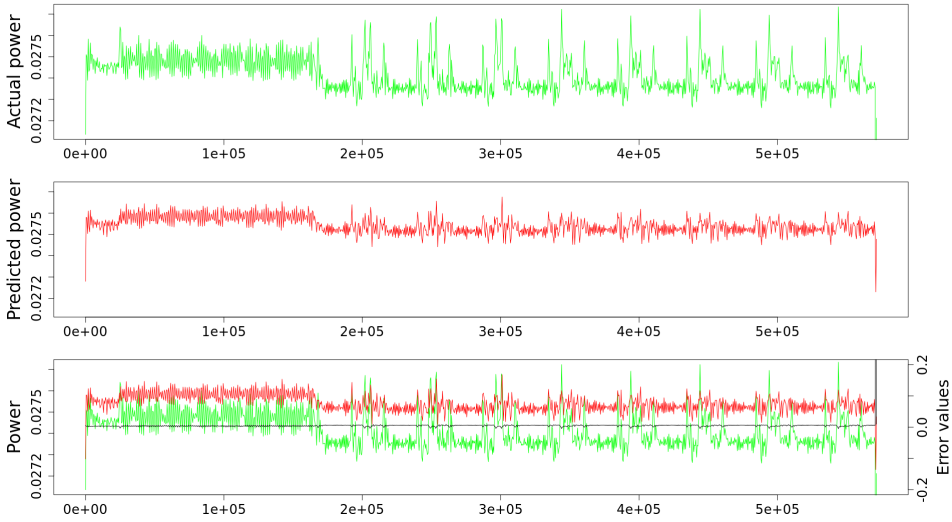

Figure D.1: Amber wrapper model evaluation.

Amber Core Without Execute

$$
\begin{aligned}
E_{core} = \quad & 0.01318 \\
+ \quad & 0.00304 \cdot \text{exec\_stall} \\
- \quad & 0.00209 \cdot \text{core\_stall} \\
+ \quad & 0.00189 \cdot \text{dcache\_wb\_uncached\_req} \\
+ \quad & 0.00184 \cdot \text{dcache\_wb\_cached\_req} \\
- \quad & 0.00126 \cdot \text{dcache\_wb\_cached\_req:icache\_wb\_req} \\
+ \quad & 0.00062 \cdot \text{dcache\_wb\_uncached\_req:icache\_wb\_req} \\
+ \quad & 0.00054 \cdot \text{execute\_iaddress\_valid:HD(execute\_iaddress)} \\
- \quad & 0.00050 \cdot \text{exec\_stall:HD(execute\_iaddress\_nxt)} \\
- \quad & 0.00049 \cdot \text{HD(execute\_iaddress)} \\
- \quad & 0.00029 \cdot \text{exec\_stall:HD(execute\_iaddress)} \\
+ \quad & 0.00019 \cdot \text{core\_stall:HD(execute\_iaddress)} \\
- \quad & 0.00018 \cdot \text{icache\_wb\_req} \\
+ \quad & 0.00016 \cdot \text{core\_stall:HD(execute\_iaddress\_nxt)} \\
+ \quad & 0.00016 \cdot \text{execute\_iaddress\_valid} \\
+ \quad & 0.00015 \cdot \text{HD(execute\_iaddress\_nxt)} \\
+ \quad & 0.00011 \cdot \text{core\_stall:HD(fetch\_instruction)} \\
- \quad & 0.00009 \cdot \text{exec\_stall:HD(fetch\_instruction)} \\
+ \quad & 0.00009 \cdot \text{conflict} \\
- \quad & 0.00005 \cdot \text{conflict:HD(execute\_iaddress\_nxt)} \\
+ \quad & 0.00004 \cdot \text{execute\_iaddress\_valid:HD(fetch\_instruction)} \\
- \quad & 0.00002 \cdot \text{HD(fetch\_instruction)} \\
- \quad & 0.00001 \cdot \text{conflict:HD(fetch\_instruction)} \\
+ \quad & 0.00001 \cdot \text{conflict:HD(execute\_iaddress)}
\end{aligned}
$$

Maximum value	5922
Minimum value	-2370
Minimum time before overflow (32 bit)	725256 cycles (12.088 ms)
Maximum time before overflow (32 bit)	N/A
R^2	0.8301
Relative error in average energy	0.2520 %
Average cycle-by-cycle error:	0.6707 %
Absolute average error:	2.0257 %

Figure D.2: Amber core model evaluation.

Execute Stage

$$
\begin{aligned}
E_{execute} = \quad & 0.01367 \\
+ \quad & 0.00237 \cdot \text{change(i\_alu\_function)} \\
+ \quad & 0.00105 \cdot \text{change(i\_rs\_sel)} \\
- \quad & 0.00082 \cdot \text{i\_core\_stall} \\
+ \quad & 0.00032 \cdot \text{change(i\_rm\_sel)} \\
+ \quad & 0.00031 \cdot \text{HD(reg\_bank\_rm)} \\
- \quad & 0.00021 \cdot \text{HD(reg\_bank\_rm):change(i\_rm\_sel)} \\
+ \quad & 0.00019 \cdot \text{change(i\_rd\_sel)} \\
+ \quad & 0.00014 \cdot \text{HD(barrel\_shift\_out)} \\
+ \quad & 0.00014 \cdot \text{change(i\_rn\_sel)} \\
+ \quad & 0.00008 \cdot \text{HD(reg\_bank\_rs)} \\
+ \quad & 0.00006 \cdot \text{HD(reg\_bank\_rn)} \\
+ \quad & 0.00005 \cdot \text{HD(reg\_bank\_rd)} \\
+ \quad & 0.00003 \cdot \text{HD(reg\_bank\_rs):change(i\_rs\_sel)} \\
+ \quad & 0.00001 \cdot \text{HD(reg\_bank\_rd):change(i\_rd\_sel)} \\
+ \quad & 0.00001 \cdot \text{HD(reg\_bank\_rn):change(i\_rn\_sel)}
\end{aligned}
$$

Maximum value	3982
Minimum value	613
Minimum time before overflow (32 bit)	1078595 cycles (17.977 ms)
Maximum time before overflow (32 bit)	7006471 cycles (116.775 ms)
R^2	0.886
Relative error in average energy	0.757077 %
Average cycle-by-cycle error:	0.9018 %
Absolute average error:	1.7779 %

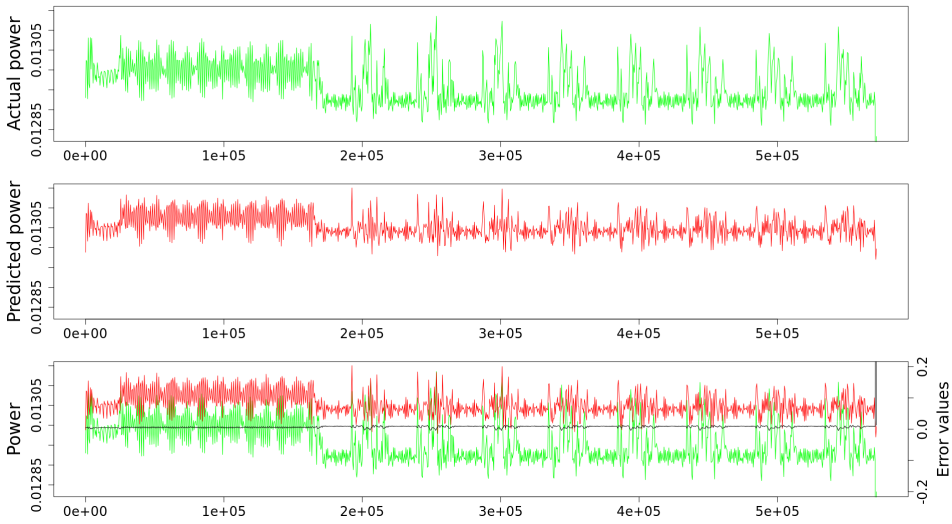

Figure D.3: Execute stage model evaluation.

Timer Module

$$
\begin{aligned}
E_{timer} = \quad & 0.00130 \\
+ \quad & 0.00029 \cdot \text{i\_wb\_stb} \\
+ \quad & 0.00025 \cdot \text{HD(timer0\_load\_reg)} \\
+ \quad & 0.00025 \cdot \text{HD(timer1\_load\_reg)} \\
+ \quad & 0.00025 \cdot \text{HD(timer2\_load\_reg)} \\
- \quad & 0.00015 \cdot \text{HD(timer0\_value\_reg):i\_wb\_stb} \\
- \quad & 0.00015 \cdot \text{HD(timer1\_value\_reg):i\_wb\_stb} \\
- \quad & 0.00015 \cdot \text{HD(timer2\_value\_reg):i\_wb\_stb} \\
+ \quad & 0.00007 \cdot \text{change(lbits(i\_wb\_adr, 0, 11))} \\
+ \quad & 0.00003 \cdot \text{change(lbits(i\_wb\_adr, 2, 3))} \\
- \quad & 0.00002 \cdot \text{HD(i\_wb\_dat):i\_wb\_stb} \\
+ \quad & 0.00002 \cdot \text{HD(timer0\_value\_reg)} \\
+ \quad & 0.00002 \cdot \text{HD(timer1\_value\_reg)} \\
+ \quad & 0.00002 \cdot \text{HD(timer2\_value\_reg)} \\
+ \quad & 0.00001 \cdot \text{HD(timer0\_value\_reg):i\_wb\_we} \\
+ \quad & 0.00001 \cdot \text{HD(timer1\_value\_reg):i\_wb\_we} \\
+ \quad & 0.00001 \cdot \text{HD(timer2\_value\_reg):i\_wb\_we}
\end{aligned}
$$

Maximum Value	1581
Minimum Value	-1033
Minimum time before overflow (32 bit)	2709568 cycles (45.163 ms)
Maximum time before overflow (32 bit)	N/A
R^2	0.7728
Relative error in average energy	0.4852 %
Average cycle-by-cycle error:	0.7253 %
Absolute average error:	0.8358 %

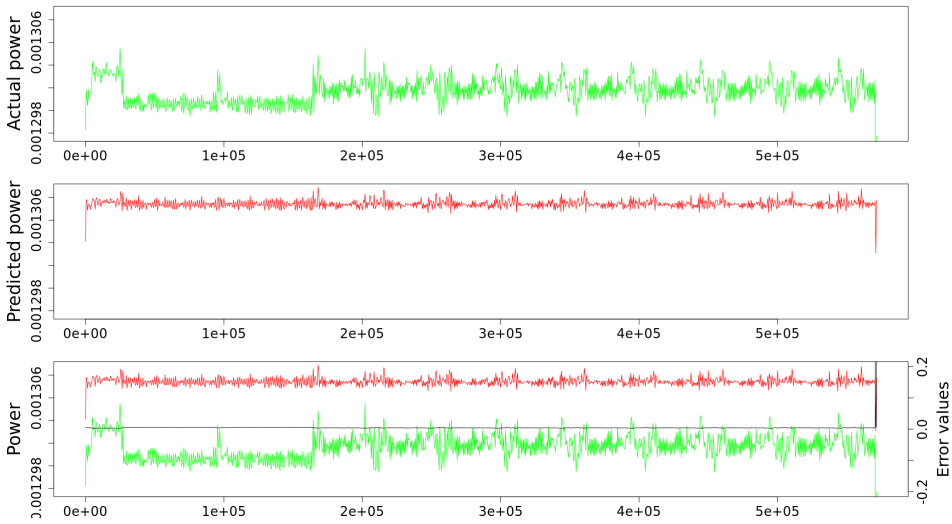

Figure D.4: Timer model evaluation.

Improved Timer Module Model This paragraph showcases how higher modelling effort may improve model accuracy and implementation overhead. Based on the insight from Section 6.2.5, I created a new model which tried to avoid the use of Hamming distances. This reduces the implementation overhead. Interestingly, it was possible to make these changes while also slightly improving accuracy. This demonstration of potential model quality is promising for future work.

$$
\begin{aligned}
E_{timer} = \quad & 0.00130 \\
+ \quad & 0.00012 \cdot \text{change(lbits(i\_wb\_adr, 0, 11))} \\
+ \quad & 0.00006 \cdot \text{lbit(timer0\_ctrl\_reg, 1)} \\
+ \quad & 0.00006 \cdot \text{lbit(timer1\_ctrl\_reg, 1)} \\
+ \quad & 0.00006 \cdot \text{lbit(timer2\_ctrl\_reg, 1)} \\
+ \quad & 0.00005 \cdot \text{i\_wb\_stb:(lbits(i\_wb\_adr, 2, 3) == 1)} \\
- \quad & 0.00004 \cdot \text{change(lbits(i\_wb\_adr, 0, 11)):(lbits(i\_wb\_adr, 2, 3) == 2)} \\
- \quad & 0.00004 \cdot \text{change(lbits(i\_wb\_adr, 0, 11)):(lbits(i\_wb\_adr, 2, 3) == 1)} \\
- \quad & 0.00004 \cdot \text{i\_wb\_stb} \\
+ \quad & 0.00002 \cdot \text{i\_wb\_stb:lbit(timer0\_ctrl\_reg, 1)} \\
+ \quad & 0.00002 \cdot \text{i\_wb\_stb:lbit(timer1\_ctrl\_reg, 1)} \\
+ \quad & 0.00002 \cdot \text{i\_wb\_stb:lbit(timer2\_ctrl\_reg, 1)} \\
- \quad & 0.00001 \cdot \text{change(lbits(i\_wb\_adr, 0, 11)):lbit(timer0\_ctrl\_reg, 1)} \\
- \quad & 0.00001 \cdot \text{change(lbits(i\_wb\_adr, 0, 11)):i\_wb\_stb:lbit(timer1\_ctrl\_reg, 1)} \\
- \quad & 0.00001 \cdot \text{change(lbits(i\_wb\_adr, 0, 11)):i\_wb\_stb:lbit(timer2\_ctrl\_reg, 1)}
\end{aligned}
$$

Maximum Value	171
Minimum Value	115
Minimum time before overflow (32 bit)	25116767 cycles (418.61 ms)
Maximum time before overflow (32 bit)	37347541 cycles (622.46 ms)
R^2	0.914
Relative error in average energy	0.2681 %
Average cycle-by-cycle error:	0.8095 %
Absolute average error:	0.8512 %

Figure D.5: Improved timer model evaluation.

Interrupt Controller Module

$$
\begin{aligned}
E_{IRQ} = \quad & 0.00123 \\
+ \quad & 0.00012 \cdot \text{change(lbits(i\_wb\_adr, 0, 11))} \\
+ \quad & 0.00008 \cdot \text{change(lbits(i\_wb\_adr, 2, 3))} \\
+ \quad & 0.00002 \cdot \text{i\_wb\_stb} \\
+ \quad & 0.00001 \cdot \text{i\_wb\_we}
\end{aligned}
$$

Maximum Value	146
Minimum Value	123
Minimum time before overflow (32 bit)	29417584 cycles (490.29 ms)
Maximum time before overflow (32 bit)	34918433 cycles (581.97 ms)
R^2	0.8611
Relative error in average energy	0.075 %
Average cycle-by-cycle error:	0.2525 %
Absolute average error:	0.8953 %

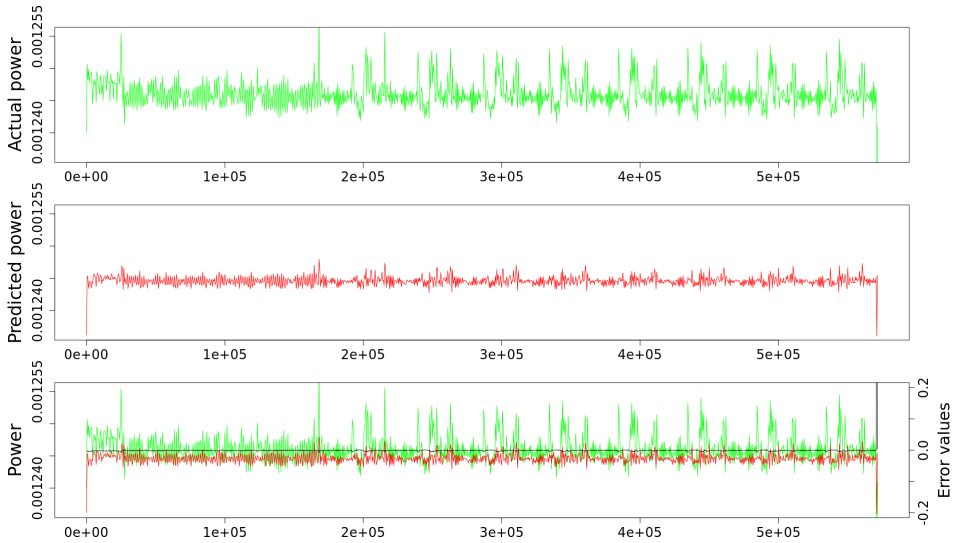

Figure D.6: Interrupt controller model evaluation.

Tile Register Module

$$
\begin{aligned}
E_{TileRegs} = \quad & 0.00047 \\
+ \quad & 0.00005 \cdot \text{i\_wb\_stb} \\
+ \quad & 0.00001 \cdot \text{change(lbits(i\_wb\_adr, 0, 11))}
\end{aligned}
$$

Maximum Value	53
Minimum Value	47
Minimum time before overflow (32 bit)	81037118 cycles (1351 ms)
Maximum time before overflow (32 bit)	91382282 cycles (1523 ms)
R^2	0.6319
Relative error in average energy	0.494 %
Average cycle-by-cycle error:	-0.0674 %
Absolute average error:	1.0386 %

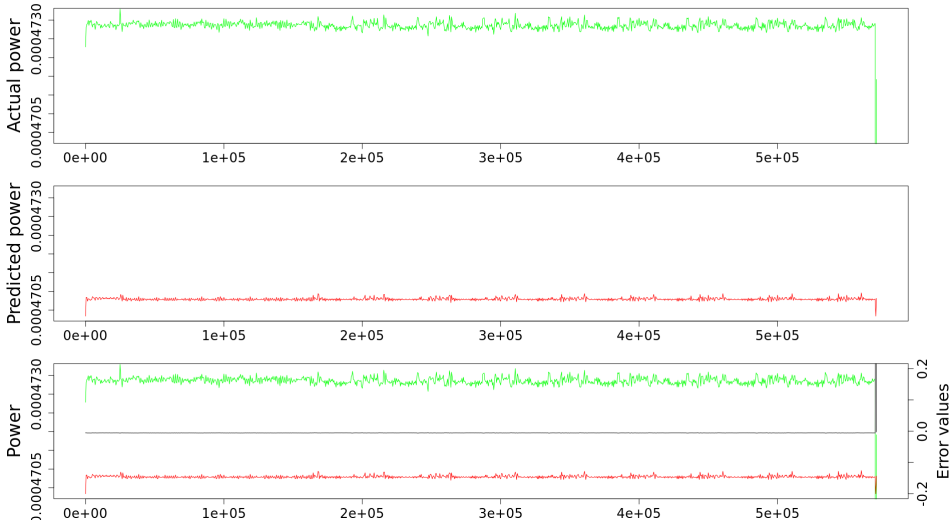

Figure D.7: Tile register model evaluation.

Amber System Module

$$
\begin{aligned}
E_{AmberSystem} = \quad & 6e-05 \\
+ \quad & 0.00037 \cdot \mathrm{HD}(\mathrm{wb\_stb\_tileregs}) \\
+ \quad & 0.00035 \cdot \mathrm{change}(\mathrm{lbits}(\mathrm{o\_wb\_adr},\ 12,\ 15)) \\
+ \quad & 0.00008 \cdot \mathrm{HD}(\mathrm{wb\_ack\_cpu}){:}\mathrm{change}(\mathrm{lbits}(\mathrm{o\_wb\_adr},\ 12,\ 15)) \\
- \quad & 0.00005 \cdot \mathrm{HD}(\mathrm{wb\_ack\_cpu}) \\
+ \quad & 0.00003 \cdot \mathrm{HD}(\mathrm{o\_wb\_stb}) \\
- \quad & 0.00002 \cdot \mathrm{HD}(\mathrm{o\_wb\_stb}){:}\mathrm{change}(\mathrm{lbits}(\mathrm{o\_wb\_adr},\ 12,\ 15)) \\
+ \quad & 0.00001 \cdot \mathrm{HD}(\mathrm{wb\_dat\_r\_cpu})
\end{aligned}
$$

Maximum Value	217
Minimum Value	1
Minimum time before overflow (32 bit)	19792476 cycles (329.87 ms)
Maximum time before overflow (32 bit)	4294967296 cycles (71583 ms)
R^2	0.9401
Relative error in average energy	30.3327 %
Average cycle-by-cycle error:	12.20 %
Absolute average error:	20.1 %

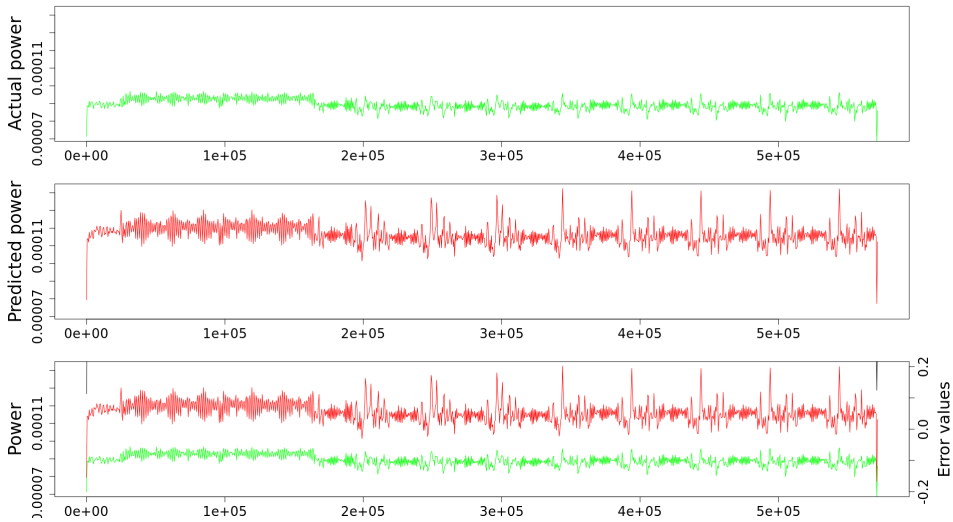

Figure D.8: Amber system model evaluation.

Router Module

$$
\begin{aligned}
E_{router} = \quad & 0.04300 \\
+ \quad & 0.00140 \cdot \text{lbit(req\_out, 0)} \\
+ \quad & 0.00140 \cdot \text{lbit(req\_out, 1)} \\
+ \quad & 0.00140 \cdot \text{lbit(req\_out, 2)} \\
+ \quad & 0.00140 \cdot \text{lbit(req\_out, 3)} \\
+ \quad & 0.00106 \cdot \text{lbit(req\_out, 4)} \\
+ \quad & 0.00062 \cdot \text{lbit(req\_in, 4)} \\
+ \quad & 0.00027 \cdot \text{lbit(req\_in, 3)} \\
+ \quad & 0.00027 \cdot \text{lbit(req\_in, 2)} \\
+ \quad & 0.00027 \cdot \text{lbit(req\_in, 1)} \\
+ \quad & 0.00027 \cdot \text{lbit(req\_in, 0)} \\
- \quad & 0.00003 \cdot \text{HD(lbits(data\_in\_fifo\_head, 0, 195))} \\
- \quad & 0.00003 \cdot \text{HD(lbits(data\_in\_fifo\_head, 196, 391))} \\
- \quad & 0.00003 \cdot \text{HD(lbits(data\_in\_fifo\_head, 392, 587))} \\
- \quad & 0.00003 \cdot \text{HD(lbits(data\_in\_fifo\_head, 588, 783))} \\
+ \quad & 0.00003 \cdot \text{HD(lbits(data\_out\_fifo\_tail, 0, 195))} \\
+ \quad & 0.00003 \cdot \text{HD(lbits(data\_out\_fifo\_tail, 196, 391))} \\
+ \quad & 0.00003 \cdot \text{HD(lbits(data\_out\_fifo\_tail, 392, 587))} \\
+ \quad & 0.00003 \cdot \text{HD(lbits(data\_out\_fifo\_tail, 588, 783))} \\
+ \quad & 0.00003 \cdot \text{HD(lbits(data\_out\_fifo\_tail, 784, 979))} \\
+ \quad & 0.00002 \cdot \text{HD(lbits(data\_in, 784, 979))} \\
+ \quad & 0.00002 \cdot \text{HD(lbits(data\_in\_fifo\_head, 784, 979))} \\
- \quad & 0.00001 \cdot \text{HD(lbits(data\_in, 0, 195))} \\
- \quad & 0.00001 \cdot \text{HD(lbits(data\_in, 196, 391))} \\
- \quad & 0.00001 \cdot \text{HD(lbits(data\_in, 392, 587))} \\
- \quad & 0.00001 \cdot \text{HD(lbits(data\_in, 588, 783))}
\end{aligned}
$$

Maximum value	7880
Minimum value	1164
Minimum time before overflow (32 bit)	545046 cycles (9.0841 ms)
Maximum time before overflow (32 bit)	3689835 cycles (61.4972 ms)
R^2	0.884
Relative error in average energy	0.3135 %
Average cycle-by-cycle error:	0.7519 %
Absolute average error:	1.3664 %

Figure D.9: Router model evaluation.

D.2 Analytical Models

Cache Model

$$E_{cache} = 0.03355 + 0.03596 \cdot read + 0.03368 \cdot write$$

Maximum Value	6951
Minimum Value	3355
Minimum time before overflow (32 bit)	617892 cycles (10.298 ms)
Maximum time before overflow (32 bit)	1280169 cycles (21.336 ms)

Scratchpad Memory Model

$$E_{scratchpad} = 0.31481 + 0.12882 \cdot read + 0.12882 \cdot write$$

Maximum Value	44363
Minimum Value	31481
Minimum time before overflow (32 bit)	96814 cycles (1.614 ms)
Maximum time before overflow (32 bit)	136430 cycles (2.274 ms)

Off-Chip RAM Model

$$E_{DDR} = 0.74360 + 0.67527 \cdot read + 0.81035 \cdot write$$

Maximum Value	155395
Minimum Value	74360
Minimum time before overflow (32 bit)	27639 cycles (0.461 ms)
Maximum time before overflow (32 bit)	57759 cycles (0.963 ms)