ABSTRACT

MAJUMDAR, SUDEEPTO. System Level Dynamic Power Estimation for Temperature-Reliability Co-optimization based on AnyCore. (Under the direction of Dr. W. Rhett Davis).

Accurate power estimation is generally obtained at a much later stage in design phase, after the design has been implemented in a transistor-level netlist. Thus, assessing the power cost of complex microarchitectures is a time and resource intensive task, which impedes temperature-reliability co-optimization efforts significantly. This thesis elaborates an architectural power estimation technique using event signatures, that enables accurate power estimation in early phases of a design implementation.

Characterization of benchmarks in terms of their high-level event signatures, and the correlation between such event signatures and dynamic power consumption were performed under the research presented in this document. 18 register transfer level events were identified to express the computational complexity of any benchmark run using the AnyCore toolset. Linear regression between the event signatures from 6 diverse benchmarks, and their gate-level power consumptions yielded an average prediction error of less than ± 3 %.

System Level Dynamic Power Estimation for Temperature-Reliability Co-optimization based on AnyCore

by
Sudeepto Majumdar

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2017

APPROVED BY:

_____	_____
Dr. Huiyang Zhou	Dr. Eric Rotenberg

_____
Dr. W. Rhett Davis
Committee Chair

DEDICATION

Dedicated To

My Parents

Mahamaya Majumdar & Sebabrata Majumdar

For always supporting and inspiring me to be the best

My Grandfather

(Late) Tarak Brahma Bhattacharya

For making an engineer out of me at a young age

My Sister

Anindita Majumdar Bhattacharya

For always being by my side

BIOGRAPHY

Sudeepto Majumdar was born to Sebabrata Majumdar and Mahamaya Majumdar in the city of Raipur, Chattisgarh, India. He did most of his schooling in Raipur but the last two years in Vishakhapatnam, Andhra Pradesh. Thereafter, he joined the electrical engineering department at the prestigious Indian Institute of Technology Delhi (IIT Delhi) with an All India Rank of 213 in the entrance examinations. During his undergraduate years at IIT Delhi, he worked on many research projects, including the CanSat, in which he represented IIT Delhi and helped secure a worldwide rank of 3. He completed his Bachelor of Technology (B.Tech.) degree in 2013 and joined Intel Technologies, Bangalore, India as a Board Design Engineer. He spent the next two and a half years at Intel, honing his electrical design skills and developing a taste for the corporate world of semiconductor engineering. In order to gain a deeper understanding of computer design, he decided to pursue the Master's program in the computer engineering department at NC State University. After his first semester there, he was selected by Dr. W. Rhett Davis to assist in his research. This research work eventually grew into the thesis topic presented here.

Sudeepto's primary research interests lie in power efficient logic design with a strong emphasis on optimized microarchitecture. During his graduate studies, he interned at the Exascale Computing Group at Intel Federal, Hillsboro. His hobbies include playing video games, watching movies and TV shows, playing badminton, singing and cooking.

ACKNOWLEDGMENTS

First and foremost, I wish to express my gratitude towards my advisor, Dr. W. Rhett Davis. He helped guide me through the uncertainties in my research, and inspired me to find new ways to achieve my objectives. He plucked me out of my ECE 546 class and helped guide me to become a researcher. Without his support and sacrifice, little would have been achieved in this research. I am eternally grateful for having had the opportunity to work alongside him.

I would also like to thank my defense committee members, Dr. Eric Rotenberg and Dr. Huiyang Zhou, for their expert feedback and insights that helped me shape this thesis in a more scientific way. I would also like to thank Dr. Robert Evans for attending my thesis defense and providing his valuable feedback. I am thankful to all my committee members for supporting me in finding a suitable time for the thesis defense, amidst their busy schedules.

This acknowledgement would be incomplete without thanking my family and friends, who kept me motivated and helped me cope with the stress of graduate schoolwork. This work, and graduate school in general would have been a barren experience without them to stand beside me all the way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Motivation

Power consumption is a critical factor in designing hardware, now more than ever before because the over-heating effects of power consumption are getting harder to mitigate through the means of air-cooling. Therefore, simply scaling down device sizes do not guarantee a better result, as smaller devices tend to consume more power. This has led to a shift in research in the semiconductor industry, such that researchers and engineers are trying to come up with clever ways to improve performance without impacting power consumption significantly. Due to the increase in power densities in modern processors, aggressive management of power, temperature and reliability have become prime concerns. In particular, temperature-reliability co-optimization has grown into a major area of research [1] [2] [3].

The most accurate means of measuring power consumption for an unmanufactured design is by using its transistor-level netlist, and gathering the behaviour of each individual transistor and signal by running simulations. In modern devices with billions of nets and transistors, gate-level software tools can estimate power consumption accurately, but require enormous amounts of computing power and time. This makes research in temperature-reliability co-

optimization difficult, because converting a high-level design into a gate-level netlist requires huge amounts of computational capacity and time. Micro-architecture research becomes very challenging, since insights regarding how much a certain architectural implementation will impact power are very hard to achieve. This is important because power consumption is an indicator of thermal behaviour, which in turn, is an indicator of reliability.

Architectural Power Estimation

Architectural Power Estimation is the process of estimating the power consumption that a certain proposed microarchitectural implementation will incur. This is usually done when the design is implemented in a high-level language like C or in a Register-Transfer Language (RTL) like Verilog. The primary advantage of this is the shorter amount of time taken to estimate power. Most research in this area depend on McPAT [4] for power estimation. It has been shown that McPAT power estimates can suffer from significant errors owing to its power models being inaccurate and based on assumptions that do not take into account the core being modelled [5].

Power estimation algorithms from the works of Lizy K. John (UT Austin), Benjamin Lee (Duke University) and Lieven Eeckhout (Ghent University, Belgium) were analysed but this research builds upon the concept called ‘Event-Signatures’ to achieve a novel mechanism to perform Architectural Power Estimation. It is critical, that the power estimation technique be sufficiently accurate and rapid in terms of simulation speeds, so that temperature and reliability can be co-optimized rapidly. This research uses the AnyCore toolset to study,

implement and validate this technique and propose a methodology that can replace McPAT by providing transparency into the power estimation algorithm.

Event Signature Based Power Modelling

We looked at many scientific papers that document various ways of performing architectural power estimation, with varying degrees of accuracy. Some of the major methods described in these papers are:

- Using parametrized power models of common architectural structures [6].
- Transactional level power modelling [7].
- Assigning energy values to every instruction in an ISA [8].
- Assigning power numbers to directly traceable or visible variables in a functional model of a microarchitecture [9].
- Event Signature based power modelling [10].

This research is inspired from 'Event Signature Based Power Modelling', referenced from 'A High-level Microprocessor Power Modelling Technique Based on Event Signatures' by Stralen and Pimentel. Event Signature modelling is an abstract power estimation methodology, that maps an application to abstract instruction sets (AIS). This mapping is a one-time effort. The result of this operation is the *signature* of the application. The power model uses a microarchitecture description file in which the mapping of AIS instructions to usage counts of microprocessor components is described. This description file, along with the signature, provides information on how often a microarchitectural element is accessed, which

directly correlates to switching frequencies. Thus, with a reasonably good model that maps microarchitectural element accesses to power, it is simple to obtain the power estimates for an application.

This approach is an order of magnitude faster (in terms of simulation speed) than ISS based power analysis tools. The accuracy of this approach, though, depends heavily on the accuracy of the underlying power model used.

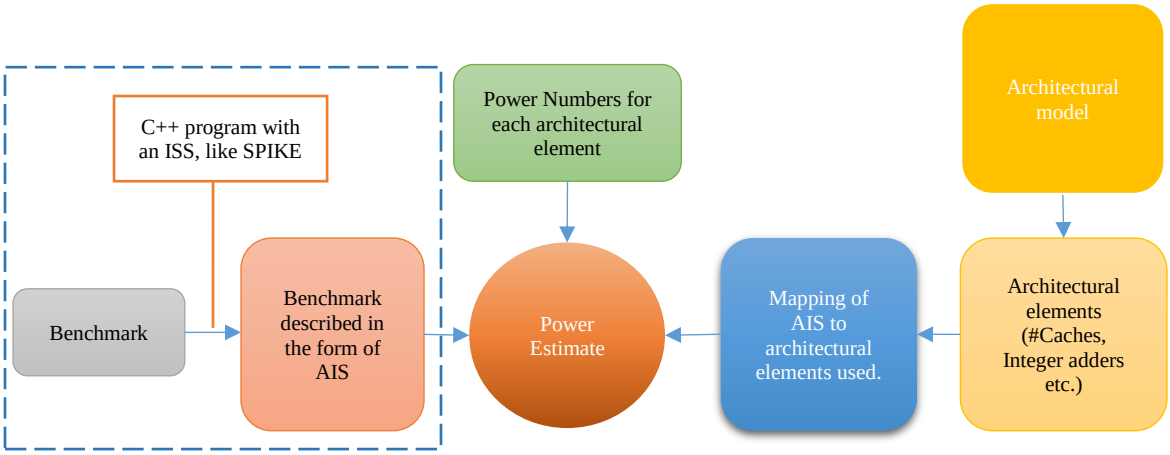

Fig. 1 Event Signature based Power Modeling

In this research, we translated this methodology to the AnyCore toolset and derived event signatures that are indicators of the frequency of resource usages. We devised an AIS like "power event" map using the AnyCore as the base. We explore various aspects of this work in the subsequent chapters. Chapter 2 describes the AnyCore toolset and the efforts spent in modifying it to produce event signatures and power values. Chapter 3 describes event signatures in detail and how power events were identified in the AnyCore pipeline that were considered representative of system power. Chapter 4 talks about the benchmark applications

written for training the power model using linear regression. Chapter 5 details the linear regression framework used for training the power estimation algorithm. Chapter 6 concludes this endeavour with results and analysis of the achieved accuracy, and also talks about shortcomings in the approach.

CHAPTER 2
ANYCORE

The work bases its power model on the AnyCore processor. This chapter describes the AnyCore processor and the methodology for generating the data needed to map microarchitectural element accesses to power consumption.

The AnyCore Toolset [11] [12] [13] provides register-transfer-level (RTL) code that enables computer architects to explore, and fabricate adaptive superscalar cores. Adaptive processors can dynamically resize their execution resources to adapt to the instruction-level parallelism (ILP) of various program phases. AnyCore provides a toolset for architects to research power, performance and area tradeoffs between various microarchitectural schemes.

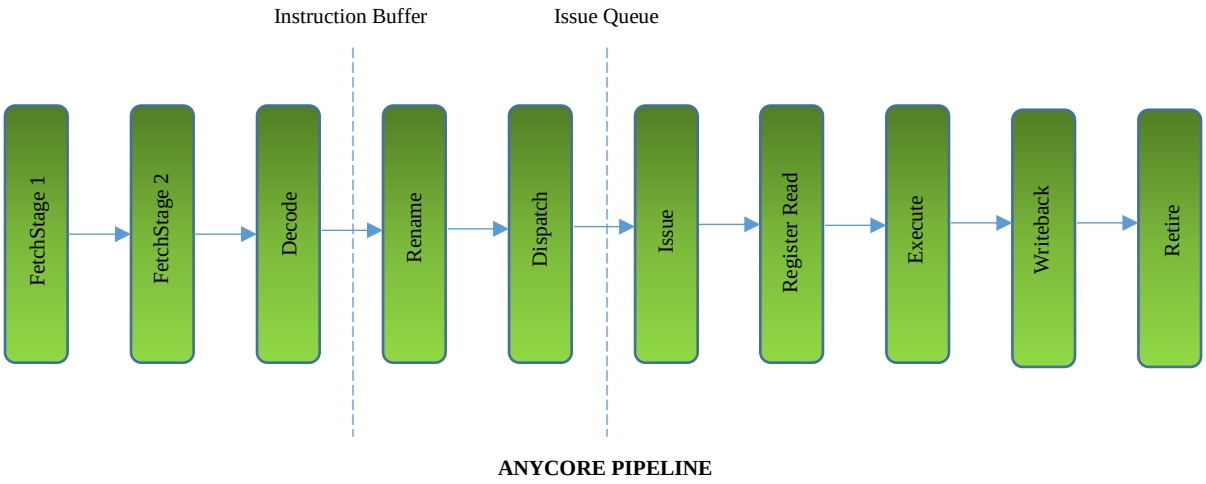

Fig. 2 Pipeline stages in AnyCore

AnyCore consists of a 10-stage superscalar pipeline (figure 2), with three decoupled regions. The first region consists of FetchStage1, FetchStage2 and Decode, and is separated from the second region by the Instruction Buffer. The second region consists of the Rename and Dispatch stages. The second region is decoupled from the third region by the Issue Queue. The third region consists of the Issue, Register-Read, Execute, Writeback and the Retire stages. Over the period of this research, the author of this thesis assisted in debugging some aspects of the AnyCore design, such as the I-Cache and the D-Cache, and identifying other bugs. The AnyCore toolset supports many different configurations for the processor pipeline, but for this research we used the following parametric sizes:

- Fetch Width: 1
- Dispatch Width: 1
- Issue Width: 3
- Commit Width: 1
- ActiveList Size: 96
- Issue Queue Size: 16
- Load Store Queue: 32
- Physical Register File Size: 160
- Free List Size: 96
- RAS size: 16

- BTB Size: 2048

The AnyCore processor supports the RISCV ISA [14], which is an open RISC ISA originally developed at the University of California, Berkeley. Therefore, the riscv64-unknown-elf-gcc compiler was used to generate RISCV binaries for application programs. The flow for running simulations involved running a RISCV functional simulator (called Spike [15]) in parallel to the systemverilog simulation, and cross-checking instructions after every commit. This verification was done using DPI functions calls. The Spike simulator could also be used to skip instructions from the beginning of a program, before transferring control to AnyCore. This feature will be explained in chapter 4. All systemverilog simulations were run using Cadence's NC-Verilog suite.

Synthesis and Simulation of Gate-Netlist

The synthesis of AnyCore RTL code into a gate-level netlist was done using Synopsys's Design Compiler shell environment. Initially the synthesis flow was broken with paths and technology libraries improperly set. This was fixed by changing commands and updating the technology libraries to point to the Nangate 45nm library. The AnyMem [11] portion of the AnyCore synthesis had broken paths too, which had to be fixed. AnyMem is a library generator tool for RAMs and CAMs that leverages FabMem [16] multi-ported RAM and CAM compiler to create Synopsys Liberty libraries for the RAMs and CAMs. It was used to compile RAM structures for the Issue Queue and the Physical Register File.

The synthesized netlist was then used for executing programs using a similar flow as the RTL simulations. Certain netnames that the testbench probed from the AnyCore design had to be changed for the simulation flow to work with the gate netlist, because the process of synthesis modified those netnames. The switching activity information for all internal nodes of AnyCore's gate-level simulation was stored in the form of Value Change Dump (VCD) files. The clock period used for simulations was 10 ns.

Power Extraction

Power extraction was done using the Primetime PX shell of Synopsys. It followed a similar flow as the synthesis, except that it needed the VCD file as an input. The power extraction script reported the power values (Internal, Switching, Leakage, Total) for the entire hierarchy in a power report. The script too had to be modified with proper paths and variables to work seamlessly. In this research, we are primarily interested in modeling the dynamic power aspect of a system. Hence, all our regression efforts henceforth will be focused on the total dynamic power (Internal power + Switching power) and the switching power.

Imperfections in AnyCore

There were certain features in AnyCore that were not implemented ideally, and had to be taken into account specially during power estimation, and for creating new benchmarks. The following are two such areas that were handled differently:

- **BTB**

The BTB RAM is being synthesized to flipflops. This is an imperfect design choice for a RAM as big as the BTB and hence consumes a lot of power (~60 % of the overall system power). This is a methodology caveat of the AnyCore toolset. Having an inaccurate component that is very large compared to others is a problem for regression or statistical learning, since such wide ranges of data can cause the result of classifiers to be governed by these large, inaccurate values heavily.

Owing to this unrealistically high and inaccurate power consumption by the BTB, its power values have been removed from the system level power values for power estimation.

- **FP Instructions**

The Floating-Point unit is not fully implemented in AnyCore and hence is not synthesized. Thus, AnyCore cannot run applications that use floating point instructions. Thus, any FP program had to be compiled with the '*-msoft-float'* flag which implements FP operations as integer routines. Therefore, in this research power modelling was performed using only integer microbenchmarks with diverse behaviors to provide enough variability so that the power prediction regression model does not overfit.

This concludes the description of the AnyCore toolset which acts as the vehicle for implementing and validating the accuracy in the hypothesis that higher level events can

be used to predict the power consumption of a lower level gate netlist. The next chapter elaborates the concept of event signatures and how they were extracted from AnyCore.

CHAPTER 3
EVENT SIGNATURES

An event signature [10] is an abstract execution profile of an application event that describes the computational complexity of the application event (in the case of computational events) or provides information about the data that is communicated (in the case of communication events). With respect to this research, event signatures are indicators of the complexity of an application in terms of power consumed. Just like an assembly code of an application represents the computational complexity of the application, event signatures for an application represent the energy-consumption complexity for the application. These are high level indicators for events that are indicative of major power consumption in a system. With a sufficiently exhaustive collection of event signatures over a period of time, it is possible to predict the average power consumption for the system. In this chapter we discuss the various power events that were identified as good indicators of power consumed in AnyCore.

After analysis of the structure for the AnyCore processor, the following 18 power events were identified for estimating power:

Branch Predictor Lookups

Branch Predictors are extremely crucial to the power and performance metrics of a processor, as their accuracy can affect control-flow hazards which involve pipeline flushes and wastage

of processor cycles. Branch Predictors are RAMs, therefore accessing them also consumes power. Branch Predictors are accessed during fetch and during resolution of a branch/jump instruction. Thus, it is intuitive that the number of Branch Predictor lookups should have a bearing on the power consumption.

Branch Mispredictions

As with branch predictors, Branch Mispredictions are very important to the power and performance of a processor. Branch Mispredictions directly influence pipeline flushes which are major power consuming events. Therefore, the number of Branch Mispredictions was chosen to be a power event.

Branch Target Buffer Accesses

BTBs are used to quickly discern whether a fetched instruction is a branch (or jump) and if it is, what its taken PC target is. BTBs are RAMs and, depending on the fetch width of the processor, can be accessed multiple times each cycle. BTBs also play an important role in determining control-flow accuracy, therefore the number of BTB accesses was chosen to be a power event. Even though the power values for the BTB have been removed from the system power, as explained in chapter 2 the event counters for BTB were retained as they might possess correlation to some other events in the pipeline that will help correlate the power prediction.

BTB Hits

Similar to the reasonings behind using BTB access as a power event, BTB Hits offer insight into the control-flow behavior of an application in the pipeline. Therefore, the number of BTB hits is chosen to be a power event.

Instruction Cache Access

Instruction Caches are one of the most elemental parts of a processor, and are major power consuming entities by virtue of being RAMs that get accessed frequently. The implementation details of the I-Cache and how it is accessed determine the power consumed during instruction fetch, and thus the number of I-Cache accesses is chosen to be a power event.

I-Cache Misses

I-Cache misses incur a huge power and performance cost, because larger and more power hungry structures like main memory and off-chip interconnects come into play. Although main memory and off-chip interconnects are not modelled in detail in AnyCore, the number of I-Cache misses was chosen as a power event. This was done because if this power event has little bearing in determining the system power, then the process of linear regression would automatically assign a negligible linear coefficient to it.

Instruction Decodes

Every valid instruction fetched into the pipeline of a processor gets decoded in the instruction decode stage. This entails identifying the instruction type and the operations required for its execution. Instruction decode is also responsible for setting values and signals that enable the other stages in the pipeline to act in response to the instruction being decoded. This is an important event in the pipeline and plays a role in determining the power consumption of many resources that are attached to it. Therefore, the number of Decodes was chosen to be a power event.

Register Renaming

Every instruction with source or destination registers undergoes register renaming, wherein the logical registers are mapped to physical registers for preventing data-flow hazards. This process of register renaming involves looking up the rename map table, accessing the free list and accessing the active list (Re-order Buffer). This process is followed for every logical register an instruction accesses. Therefore, being a power intensive process, the number of registers renamed was included as a power event.

Instruction Issues

Instructions in the issue queue are issued to the register read stage based on many different factors like instruction age, source registers ready status and wakeup signals from the execute stage. This involves a great deal of arbitration between all the instructions present in the issue

queue. This is a significant contributor to system power as well, and hence the number of Instruction Issues was chosen as a power event.

Load Instructions

Load instructions access the Data Cache and the main memory to fetch data into the pipeline. As with I-Cache accesses, data reads from the D-Cache are sources of immense power consumption. Load instructions also access the load and store queues, and perform arbitrations over their contents. Therefore, the number of loads was chosen as a power event.

Store Instructions

Store instructions store data into the D-Cache and main memory, but access the D-Cache in a similar manner as load instructions. They also access the load and store queues, similar to load instructions. Therefore, the number of store instructions was also chosen to be a power event.

Load Misses

Load instructions that miss in the D-Cache incur heavy penalties in terms of cycles needed to access the main memory and the power consumed for the same. Such events can stall the pipeline for hundreds of cycles. Even though main memory and interconnects are not modelled extensively in AnyCore, this event was included as a power event for the same reason I-Cache misses was included as a power event.

Store Misses

The number of store misses were included as a power event for the same reason as the number of Load Misses.

Store – Load Forwarding

Store – Load forwarding involves forwarding data from a store instruction in the store queue directly to a load instruction in the load queue, instead of accessing the D-Cache for the load. This operation is performed by a load instruction whose address has been computed, by walking the store queue looking for a match in address. When a matching address is found, the data from the corresponding store is forwarded to the load instruction. This is also a major power consuming event as it involves traversing a memory unit. For this reason, the number of store-load forwards was included in the power events.

Writebacks to the Physical Register File

When a register value producing instruction finishes execution, its results are written into the physical register file. The PRF is a considerably big memory unit, and storing data into it is also a major power consuming operation. Also, a writeback into the PRF signifies that an instruction has finished execution. Therefore, being an important and power consuming event, the number of writebacks to the PRF was chosen to be a power event.

Commits/Retires

Committing instructions to architectural state is a major event in the pipeline. It involves accessing the active list, the architectural map table and the free list. All of these events are major power consuming events, and therefore, the number of commits was included in the list of power events for power estimation.

Load Violations

A load violation occurs when an older store in the store queue recognizes that a younger load with the same address in the load queue has progressed with stale data from memory. This involves raising an exception and therefore, flushing the pipeline. Therefore, load violations cause huge power consumptions in the pipeline, and have therefore been chosen as a power event.

Pipeline Recovery

Pipeline recovery is an encapsulation of all the events which cause the pipeline to flush all in-flight instructions and start fetching instructions again. Such events include branch mispredictions, load violations, and all other sources of exceptions and interrupts. By definition, every pipeline recovery consumes a considerable amount of power. Therefore, the number of pipeline recoveries was selected as a power event.

The event signatures for these 18 power events were used to find a correlation with the total system power. As per the definition of event signatures, these 18 events were predicted to be

adequate for accurately estimating the power values for the AnyCore system. Event signatures were collected over 1000 cycle epochs, hence creating datasets where each power event category comprised of the cumulative total over 1000 cycles. The epoch size of 1000 is configurable in the top level systemverilog testbench file. A smaller epoch size of 100 would provide better exposure between the power consumption and the event signatures, but it would increase the time required for the power extraction process using Primetime PX.

The top level testbench files were modified to count the event signatures over an epoch and print them in a text file called the ‘power-cut’ file. The power-cut files contain event signatures obtained over an entire program simulation, where each row corresponds to a single dataset of 1000 cycles and each column corresponds to a power event in that data set. Many of the signals that were used to count the event signatures in RTL simulation got renamed in the gate-netlist, hence the code for collecting even signatures was commented out for gate-netlist simulations.

Correspondingly, the power for each epoch had to be extracted separately. To this end, the power extraction script mentioned in chapter 2 had to be modified to extract and report power for each epoch separately in the power report. These power numbers were extracted from the power report file and appended to the power-cut file’s data, to be used by a python regression script as will be explained in chapter 5.

The next chapter introduces the various applications that were used to collect datasets for power estimation.

CHAPTER 4
BENCHMARKS

This chapter details the various benchmark programs that were used to test and demonstrate the underlying hypothesis of this research. The existing framework for running RTL and gate-level simulations uses the riscv64-unknown-elf-gcc compiler, for compiling programs written in C language into the RISCV binary. Six C language programs were used to produce data for training the regression model, and two other programs were used to validate and measure the accuracy of the predictions.

All programs were kept simple, but diverse in nature. This is because complex programs involving numerous varied operations resulted in long simulation times, and huge value change dump (VCD) files (of the order of 200-700 GBytes). Furthermore, extracting power from such huge VCD files was excruciatingly slow. Diversity in the nature of the programs means that each program was written with focus on specific architectural modules like the data cache, or system-level events like a system call. The simulation framework also involved providing a ‘skip’ number, which signifies the number of instructions the functional simulator would execute before transferring control to AnyCore. This exists primarily to reduce the number of instructions executed by AnyCore, thus providing a mechanism to regulate simulation times. It was observed that arbitrarily chosen numbers would sometimes

cause the simulations to cause errors and fail. Thus, the skip numbers for all programs were chosen by a trial-and-error method.

The details of each program are as follows:

Program 1

Program 1 consists of memory transactions that occur after every 7 iterations of a loop. The idea behind this is to induce sporadic memory access instructions with intermittent phases with relatively low memory accesses. The C code for the program is present below.

```
int main(void)
{
        int  a[100];
        for (int i = 1; i<99; i++)
        {
                if (i%7 == 0)
                {
                        a[i] = i*3;
                        a[i-1] = i*3;
                        a[i-2] = i*3;
                        a[i-3] = i*3;
                        a[i-4] = i*3;
                        a[i-5] = i*3;
                        a[i-6] = i*3;
                }
        }
```

```
        return 0;
}
```

The skip number associated with program 1 was 150000, which implies that execution of the program would be handed over to the RTL or gate-netlist simulation after 150000 instructions. These 150000 instructions would be executed by the spike functional simulator. The primetime environment took approximately 7 hours and 40 minutes for the extraction of power for this program.

Program 2

Program 2 invokes system calls that print to the console display. This involves switching context while the processor executes kernel-level instructions and routines. In AnyCore, the kernel mode routines are handled by a part of the functional simulator called the proxy-kernel. The AnyCore does go through pipeline flushes, before the proxy-kernel executes the syscall routines. The C code for the program can be seen below.

```
int main(void)
{
        printf("Hello world!\n");
        return 0;
}
```

The skip amount for this program was also 150000. The primetime environment took approximately 8 hours for the extraction of power for this program.

Program 3

Program 3 combines memory transactions and arithmetic operations. It initializes an array, then computes the sum of all its elements. Thus, it involves memory stores, followed by memory loads and arithmetic operations. To introduce phases in the simulation when the processor is relatively idle, assembly ‘nop’ instructions were inserted in the program using the ‘asm’ keyword for C language. Using the ‘asm’ keyword makes the compiler insert the assembly instructions specified after the keyword into the compiled binary. The C code for the program can be seen below.

```
int a[20000],b[20000];
int main()
{
        int i;
        int temp;
        int sum = 0;
        int sum1 = 0;
        int sum2 = 0;
        for(i=0;i<200;i++)
        {
                a[i] = i;
        }
      asm("nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop");
```

```
        asm("nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop");
        asm("nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop");
        asm("nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop");
        for(i=0;i<50000;i++){  }
        for(i=0;i<200;)
        {
                temp = a[i];
                i++;
                sum = sum + 3;
                sum1 = sum1 + temp;
                sum2 = sum2 + temp;
         }
        asm("nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop");
        asm("nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop");
        asm("nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop");
        asm("nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop");
        b[0] = sum;
        b[1] = sum1;
        b[3] = sum2;
        return 1;
}
```

The skip amount chosen for this program was 250000. This higher number was chosen because the simulation times for the program was immensely high otherwise, with the total number of cycles in the order of 700,000. The size of VCD files generated from the gate-level simulations ranged between 200 and 300 GBytes. A skip amount of 250000 reduced the

total number of cycles to 82,000. The primetime environment took approximately 22 hours for the extraction of power for this program.

Program 4

Program 4 consists of a simple integer add operation, followed by printing the result to the console. The addition is preceded and succeeded by assembly ‘nop’ operations to induce low intensity phases. The C code for the program can be seen below.

```
int main(void)
{
        asm("nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop");
        int a = 1+ 2;
        asm("nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;nop");
        printf("Result: %d\n",a);
        return 0;
}
```

The skip amount used for this program was 170000. The primetime environment took approximately 5 hours for the extraction of power for this program.

Program 5

Program 5 consisted of branches embedded inside for loops. The branch statements (if..else) were embedded in the for loop to randomize the branch behavior. The branch was predicted taken after every 9 iterations of the loop. Since the branch predictor used in AnyCore has a

prediction depth of 2 bits, this meant that the 8 iterations of not-taken behavior would train the predictor to predict not-taken and therefore mispredict on the 9th iteration. The C code for the program can be seen below.

```
int main(void)
{
        int i,a,b = 0;
        for (i = 0; i<1000; i++)
        {
                if (i % 9 == 0) a++;
                else b++;
        }
        return 0;
}
```

The skip amount for this program was 130000. The primetime environment took approximately 21 hours for the extraction of power for this program.

Program 6

Program 6 consisted of left-shifting integer values. It comprises of two left shift statements. The C code can be seen below.

```
int main(void)
{
        int a = 34 << 2;
```

```
        a = a << 2;
        return 0;
}
```

The skip amount for this program was 150000. The primetime environment took approximately 4 hours for the extraction of power for this program.

Program 7

Program 7 consisted of computing the logarithm for an integer number. This required importing the ‘math.h’ library. The C code for the program can be seen below.

```
#include <math.h>
int main(void)
{
        double a = log(45);
        return 0;
}
```

The skip amount used for this program was 150000. The primetime environment took approximately 21 hours and 30 minutes for the extraction of power for this program.

Program 8

Program 8 included four basic arithmetic operations – addition, subtraction, multiplication and division. The C code for the program can be found below.

```
int main(void)
{
        int a = 2*3 + 6/2 - 1;
        return 0;
}
```

The skip amount for this program was 150000. The primetime environment took approximately 3 hours for the extraction of power for this program.

From the above list of programs, programs 1 – 6 were utilized for training the linear regression model, and programs 7 – 8 were employed for testing the linear coefficients' accuracy in predicting the observed power values. Programs 7 – 8 are referred to as test program 1 and test program 2 respectively from here onwards. The results and detailed discussion of these analyses are presented in chapter 6.

CHAPTER 5
LINEAR REGRESSION FRAMEWORK - PYTHON

Linear regression over the training datasets was performed using python's scikit-learn library. The script to perform linear regression was written in an IPython notebook [17], which is a document with support for code and rich text elements like equations, figures etc. The notebook is comprised of a number of cells. Each cell is a small self-contained code block that can be modified and executed individually. This feature finds its use in research because a large script need not be run from scratch after every modification. Instead, only the parts that are modified can be run while the memory state from previously run cells are retained.

The python script was used to import simulation data in the form of a comma-separated values (CSV) file. Simulation data consisted of the event signatures for each of the 18 power events, and the associated power consumption over a period of 1000 cycles. The power during the initial start-up phase of a system is usually not representative of the power consumption incurred during normal operation of the pipeline because of initial warm-up effects and start-up transients. Therefore, the first 20 datasets (or 20000 cycles) for each program have been removed, to eliminate the warm-up effects on system power. The number

of 1000 cycle datasets obtained from each of the 7 training programs mentioned in chapter 4 can be found in table 1

Table 1 Number of 1000 cycle datasets for each program

Program	# of 1000 cycle datasets
1	50
2	58
3	62
4	46
5	109
6	31
Total	356

Each of these 356 datasets were accompanied by system level power consumption values obtained from Synopsys primetime generated power reports. The event signatures were collated from the individual power-cut files as mentioned in chapter 3. Power values were collated from the top-level power reported in the power report for each program. As mentioned in chapter 3 power was reported for each 1000 cycle period. These power numbers were then put beside their corresponding datasets, in the CSV file.

The python script imported the CSV file using the pandas library [18], which provides data structures and data-manipulation routines, into a dataframe object. Using the dataframe object, the script filtered out the power event signatures along with the power values, into matrices. Then the script performed linear regression on these matrices of data using the linear regression class of the scikit-learn library. The resulting matrix of linear coefficients was saved into a text file.

Testing the accuracy of the computed linear regression coefficients was performed by importing the event signatures and observed power values for the two test programs, mentioned in chapter 4 into a Microsoft excel file. The predicted power values were computed by multiplying the linear regression coefficients and the event signatures, and adding the constant coefficient. The results were then compared against the observed power values. An analysis for this has been presented in the next chapter.

CHAPTER 6
RESULTS

Training the Power Estimation Algorithm

For this research, we have focused on the dynamic power aspect of power modeling. To this end, we have performed regression on the total dynamic power as well as the switching power. Running linear regression on the event signature sets and power for the six training programs (with 356 datasets of 1000 cycles each) yielded the following linear coefficients for each power-event:

Table 2 Linear Regression fitted weights for total dynamic and switching power

Power Event	Linear Regression Fitted Weights	
	Total Dynamic Power	Switching Power
BP_lookups	1.12E-04	-9.320E-06
BP_mispredictions	-2.25E-03	5.145E-04
BTB_accesses	-9.47E-05	-2.718E-05
BTB_hits	2.64E-05	1.081E-05
ICache_accesses	-4.55E-05	6.583E-05
ICache_misses	-3.63E-05	-7.719E-05
Decodes	-3.88E-05	-2.098E-05

Table 2 Continued

Renames	-1.91E-05	-1.078E-05
Issues	1.09E-04	4.384E-06
Loads	-5.87E-05	-1.765E-05
Stores	-1.07E-04	8.294E-06
Load_Misses	1.77E-05	-1.618E-05
Store_Misses	7.88E-05	2.291E-05
Store_Load_fwd	1.15E-03	1.971E-04
Writebacks_to_PRF	2.91E-05	4.492E-05
Commits	-5.16E-05	2.336E-05
Load_Violations	-2.67E-03	1.282E-03
Pipeline_recovery/flush	2.54E-03	-1.245E-04
Constant	0.165888	0.021091

Here total dynamic power is the sum of internal power and the switching power. It should be noted that the constant term for dynamic power – 0.166 is fairly large, and is very close to the average power value – 0.142 W. This is largely because of a near constant bias in the internal power across all datasets. It should also be observed that some coefficients are negative. The reason behind this is that the event signatures manipulate the constant coefficient to produce a prediction for the power consumed, and these manipulations can be positive or negative in nature. The higher the magnitude of a coefficient, the higher is its impact in determining the power consumed.

Using the above coefficients, the predicted and the observed total dynamic power values for each of the 356 datasets have been plotted in figure 3. Figure 4 presents a similar plot using the switching power values.

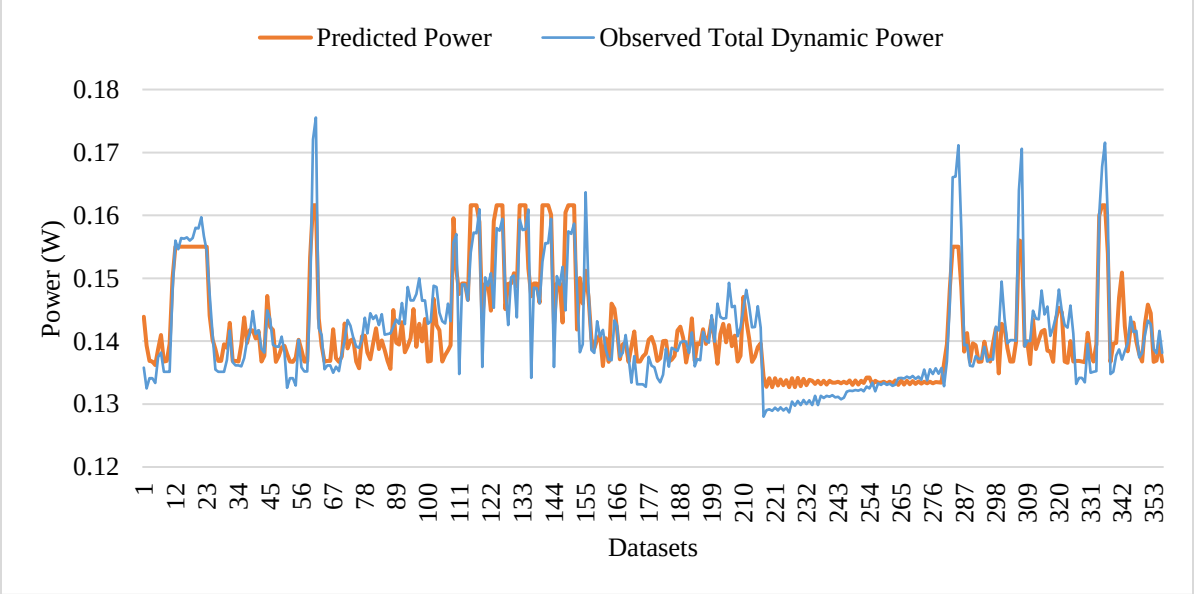

Fig. 3 Observed Total Dynamic Power vs Predicted Power

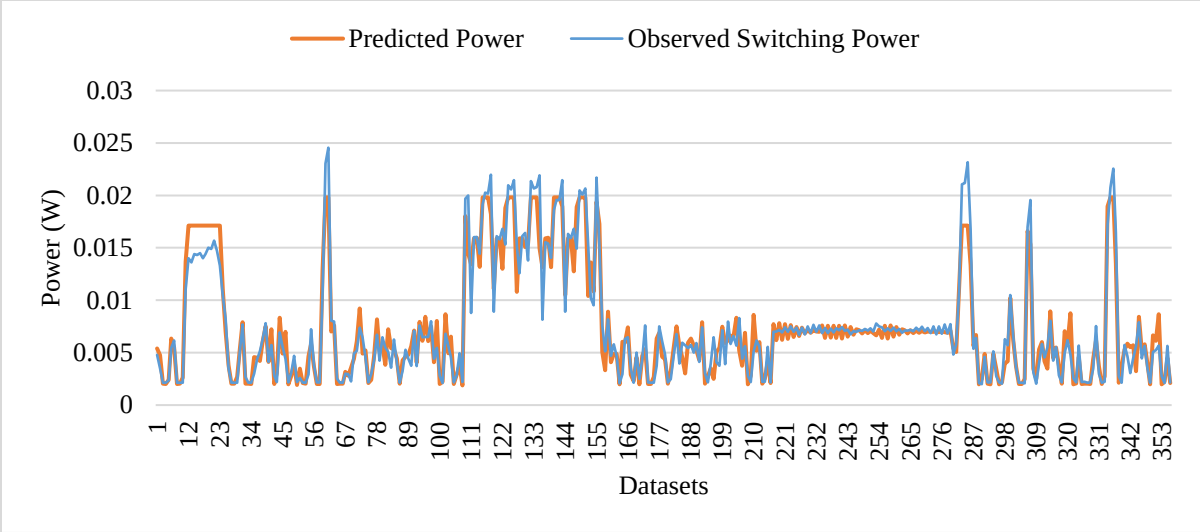

Fig. 4 Observed Switching Power vs Predicted Power

It can be seen that switching power prediction has a much better fit than total dynamic power prediction. Specially, between datasets 70 – 110 and 200 – 280, the observed total dynamic power seems to steadily increase, while the predicted power shows no steady increase. This steady increase was attributable to the internal power. As can be seen in the switching power plot, there is no such discrepancy between the observed and predicted powers. In switching power prediction, the region between datasets 12 – 23 seems to predict the power with error. There seems to be a general agreement where the peaks and valleys occur, but on some occasions the predicted power doesn't match the observed power very accurately. The average error between the predicted values and the observed dynamic power values for the training datasets was computed to be 2.27 %, with the maximum and minimum errors being 10.737 % and 0.00352 %. The error percentages in the case of switching power were significantly higher (average – 15.359 %, maximum – 105.187 %, minimum – 0.04 %)) owing to the fact that the base values against which percentages were computed were very

low. Another statistical metric that was used to measure the quality of prediction is the 'root relative squared error' or RRSE from here onwards. RRSE measures how good a certain prediction is compared to simply taking the mean over the observed values.

$$RRSE = \sqrt{\frac{\Sigma(Prediction - Observed)^2}{\Sigma(Mean - Observed)^2}}$$

A value of 0 implies perfect prediction, and a value of 1 implies that the prediction is as good as simply taking the mean. Values greater than 1 imply that the predictions are progressively worse than just taking the average. In this context, RRSE compares the error obtained when the predicted power values are used, against the error obtained when the overall mean of the observed power values are used. The RRSE for the training dataset is 0.4938 for predicting the dynamic power. On the other hand, the RRSE is 0.2851 for predicting the switching power which is significantly better than the dynamic power RRSE.

Figure 5 contrasts the relationship of each individual power event with the observed total dynamic power values. For most power events as the number of occurrences increases, there is a definite increase in the range of power values. The fact that there exist multiple power values for each value of any power event suggests that the power events correlate with the power consumption but not exclusively. That is, the total power consumption is actually a function of many different power events.

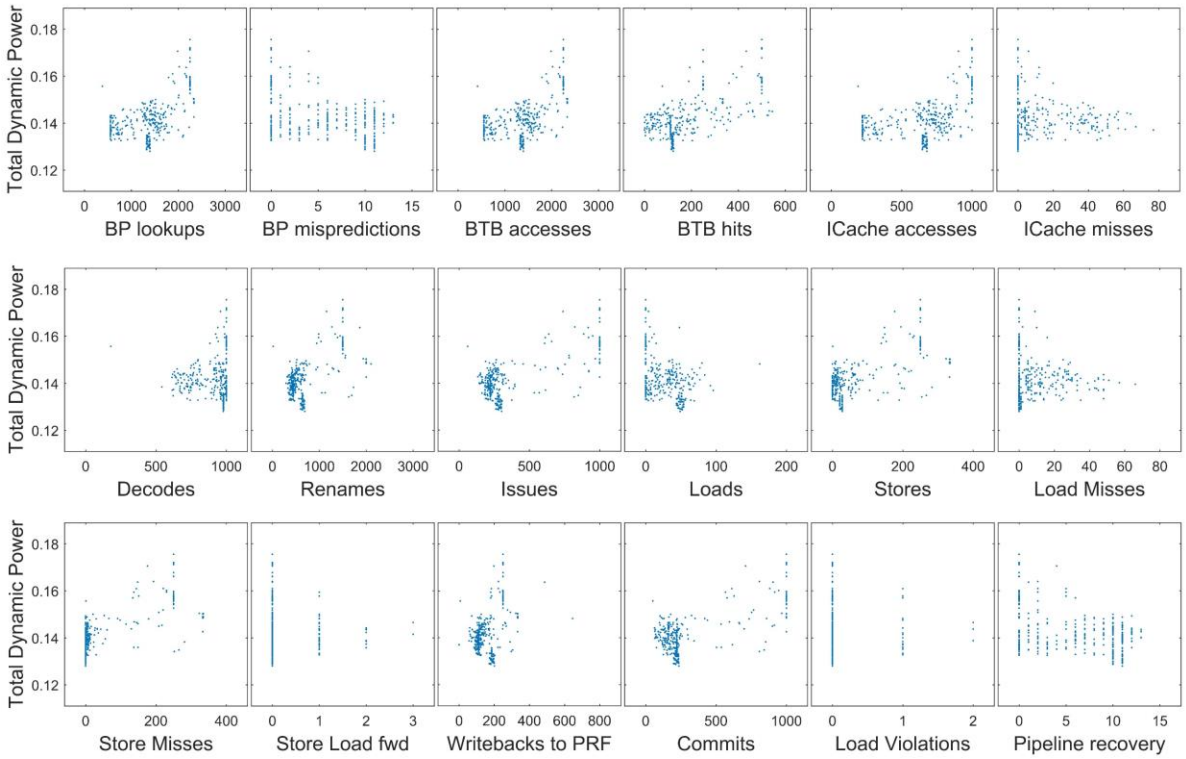

Fig. 5 Correlation patterns between Total Dynamic Power and the power events

It should be noted that in figure 5, some power events (Store-Load fwd, Load Violations etc) seem to have an negative correlation with the power values, such that as the number of these events increases, the total power consumed decreases. The reason behind this is probably that the occurrences of these power events make the pipeline go idle for many cycles, and thus the power consumed by other active elements decreases. Nevertheless, this suggests that these power events do not have a positive correlation with the total power and can probably be omitted for power estimation.

Table 3 presents the correlation p-values for each power event, with the null hypothesis being that there is no relationship between the power events and the observed dynamic and switching power values.

Table 3 Correlation p-values between the observed power values and the power events

Power Events	P-Value	
	Total Dynamic Power	Switching Power
BP_lookups	7.85E-36	1.15E-88
BP_mispredictions	2.85E-14	7.74E-06
BTB_accesses	9.80E-36	8.97E-89
BTB_hits	1.09E-42	4.64E-43
ICache_accesses	2.43E-27	7.16E-69
ICache_misses	0.649044	0.004954176
Decodes	0.77267	0.022762279
Renames	2.10E-41	4.45E-101
Issues	1.20E-59	4.05E-121
Loads	3.25E-12	0.009384934
Stores	1.38E-49	1.37E-96
Load_Misses	0.416755	0.001231693
Store_Misses	1.22E-57	1.24E-98
Store_Load_fwd	0.996337	0.114134962
Writebacks_to_PRF	1.72E-16	2.49E-67

Table 3 Continued

Commits	9.33E-58	4.50E-106
Load_Violations	0.775423	0.609480565
Pipeline_recovery/flush	3.67E-14	3.20E-06

A p-value closer to 1 suggests the hypothesis that the power event does not have a strong relationship with the power values observed. On the other hand, a p-value closer to 0 suggests that the hypothesis is false for the power event and there exists a strong correlation. From the table above, it appears that the power events ICache_misses, Decodes, Load_Misses, Store_Load_fwd and Load_Violations have lower correlation to the observed power values. Therefore, linear regression using the IPython notebook was performed by removing each of these power events individually.

It was observed that on removing any combination of these power events the RRSE for estimating the power consumptions became worse. For example, by removing Load Misses, Store_Load_fwd and Load_Violations the RRSE for total dynamic power consumption became 0.49853, which is slightly worse than what was obtained earlier using all 18 power events (0.4938). With Store_Load_fwd and Load_Violations removed, the RRSE for estimating the switching power consumption became 0.2904 which is also worse than the case when all 18 power events were used for prediction (0.2851). This is presented in table 4.

Table 4 Change in metrics with removal of high p-valued power events

Metric	Total Dynamic Power without Load Misses, Store_Load_fwd and Load_Violations	Switching Power Estimation without Store_Load_fwd and Load_Violations
RRSE	0.4938 to 0.49853	0.2851 to 0.2904
Maximum Absolute Error %	10.74 to 11	105.187 to 120.03
Average Absolute Error %	2.27 to 2.29	15.359 to 15.448

Expectedly the error percentages also increased, with the change much more prominent with the switching power estimation than the total dynamic power estimation. For example, the maximum and average error percentages for total dynamic power changed from 10.74 % and 2.27 % respectively to 11 % and 2.29 % respectively, while for switching power these changed from 105.2 % and 15.4 % respectively to 120.03 % and 15.45 % respectively. The significant change in the maximum error percentage for switching power estimation can again be attributed to the smaller base values used for computing the percentages, while the insignificant change in these percentages for the total dynamic power estimation is because of the high base values used.

On the other hand, when the power events that had the smallest magnitude of linear fitted weights (table 2) were removed, the total dynamic power showed even worse results than the previous example results that were presented in table 4. For switching power, the subsequent

power estimation showed no significant deterioration, and was in fact better than the example in table 4 above. These results can be seen in table 5.

Table 5 Change in metrics with removal of power events with low linear fitted weights.

Metric	Total Dynamic Power without Renames and Load_Misses	Switching Power Estimation without Issues and Stores
RRSE	0.4938 to 0.50316	0.2851 to 0.28671
Maximum Absolute Error %	10.74 to 14.86	105.187 to 109.56
Average Absolute Error %	2.27 to 2.28	15.359 to 15.66

These results imply that not all power events have the same bearing on estimation of power values, and omitting some does not impact the estimation accuracy much. But since better metrics were obtained using all 18 power events, we retained them all for the power estimation of the test programs 1 and 2. It is interesting to note however, that the removal of power events that had high p-values in table 4 made the predictions worse. The reason for this is guessed to be that the 18 power events identified for power prediction are not sufficiently exhaustive and there may be unidentified power events that correlate with certain crucial aspects of power consumption. In the absence of these power events, the less correlated power events that were omitted may be the only means of predicting certain power value trends. Hence, the omission of these power events causes the error values to go up.

Testing the Linear Regression Coefficients

Using a similar approach as used for the training datasets above, the power values were predicted for the test program 1 consisting of 117 datasets. Figures 6 and 7 show the predicted power and the observed power values plotted together. Table 6 shows the computed quality metrics for test program 1.

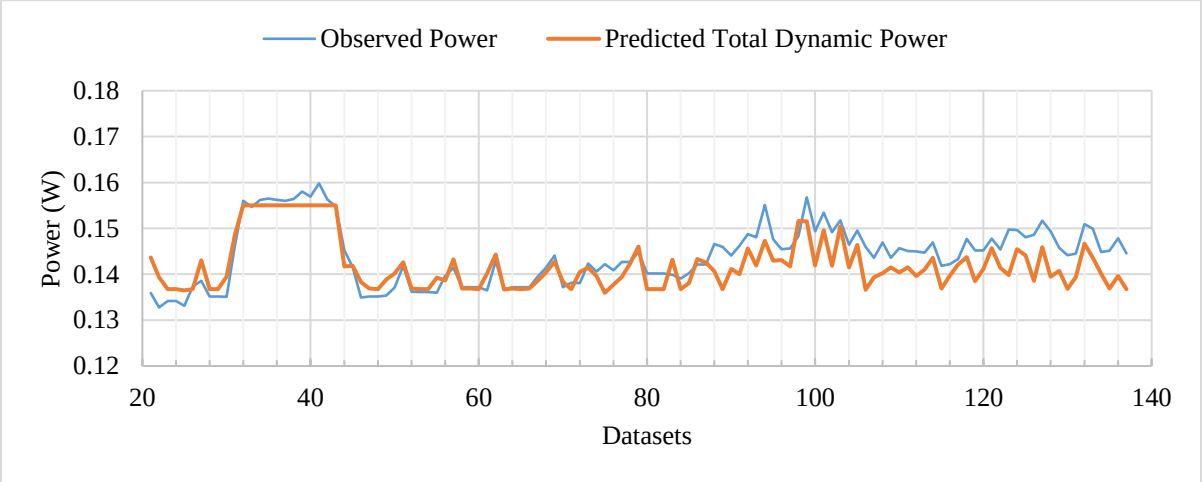

Fig. 6 Observed vs Predicted Total Dynamic Power for Test Program 1.

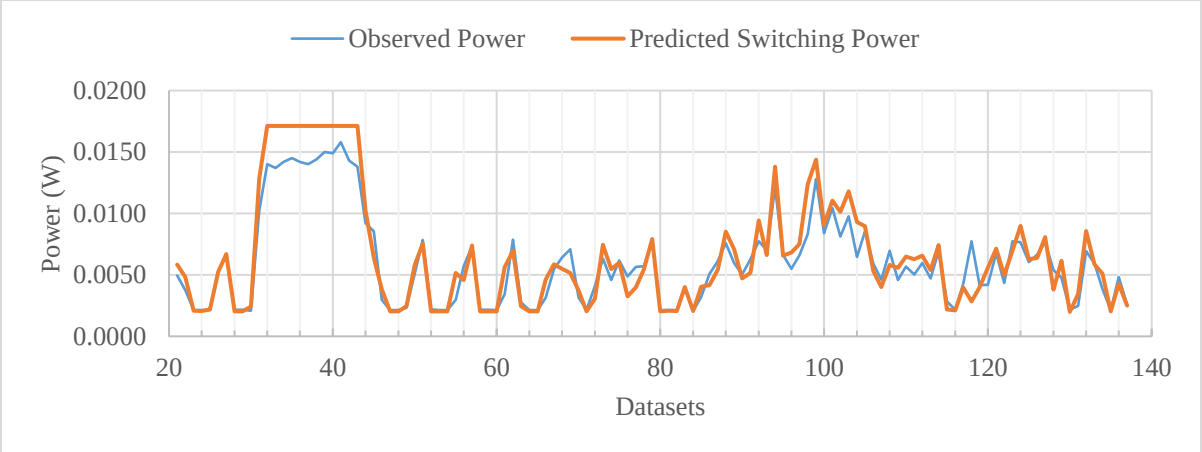

Fig. 7 Observed vs Predicted Switching Power for Test Program 1

Table 6 Power estimation accuracy metrics for test program 1

Metric	Total Dynamic Power Estimation	Switching Power Estimation
RRSE	0.6455	0.3848
Maximum Absolute Error %	6.8036	73.3462
Average Absolute Error %	2.34	15.4327

The observed power values in the region between datasets 104 – 140 seem to deviate from the predicted total dynamic power values. However, the fit for the switching power estimation is remarkably accurate, except for the region between datasets 30 – 44.

Similarly, the power values for test program 2 were predicted using its event signatures. Figures 8 and 9 show the predicted and the observed power values plotted together. Table 7 presents the quality metrics observed for test program 2.

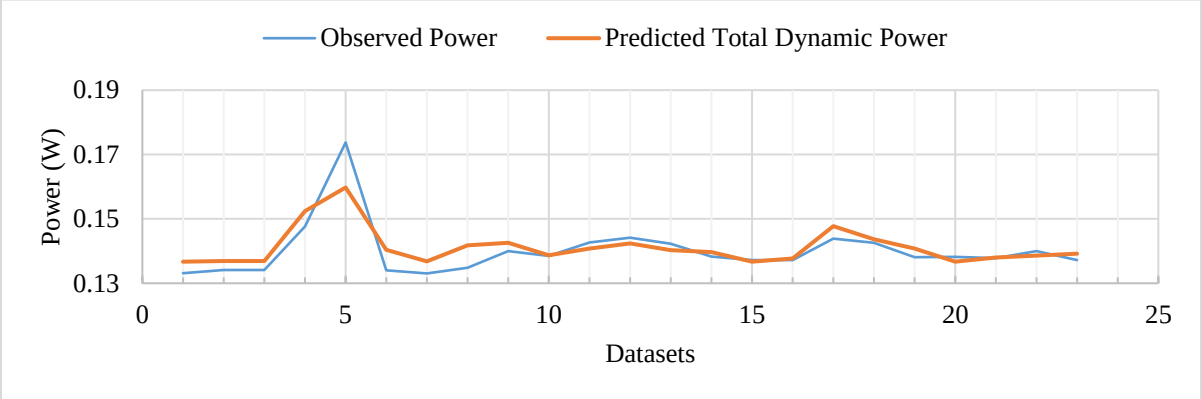

Fig. 8 Observed vs Predicted Total Dynamic Power for test program 2

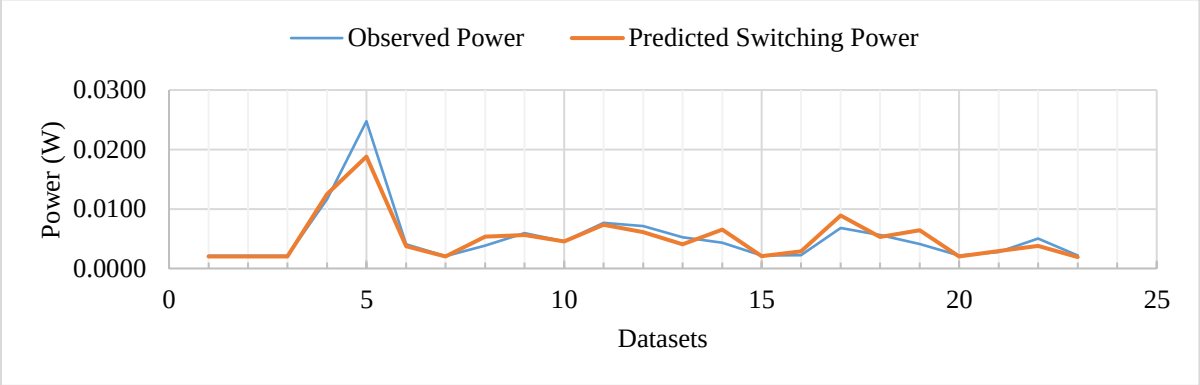

Fig. 9 Observed vs Predicted Switching Power for test program 2

Table 7 Power estimation accuracy metrics for test program 2

Metric	Total Dynamic Power Estimation	Switching Power Estimation
RRSE	0.5133	0.3318
Maximum Absolute Error %	8.0509	55.1011
Average Absolute Error %	2.0589	16.0327

The predicted total dynamic power follows the same trend as the observed power values, but the datasets 3 – 9 incur error in values. The trends for switching power values follow the predicted values accurately, except for the peak at dataset 5, where the predicted power values fall short of the observed power values.

The RRSE metric worsened for all estimations performed for the test programs, compared to the training program dataset estimations. This is because the datasets used for training the model were not completely representative of the nature of the test programs. This means that more number of varied applications should be used for training the model more accurately, as higher number of datasets enable the linear regression tool to learn about more possible combinations of event signatures.

Even though the RRSE metric worsens, the absolute error percentages for the total dynamic power consumed remain fairly low. The magnitude of the maximum error percentages reduced for the test programs, compared to the error percentages incurred in the training datasets. The high error percentages for switching power estimation imply that the magnitude of tolerance to be expected from the regression tool is invariant of the type of power estimation being performed.

Conclusion

It should be noted that the predicted power values follow the same trend as the observed power values, with peaks and valleys coinciding for majority of the datasets. Since these predictions are based on a linear regression model, it supports the hypothesis that there exists a linear correlation between the chosen RTL power events and the gate-level power consumption. This correlation depends on the technology node used, the size of architectural structures used as well as the clock speed used for simulation. The error percentages coupled with the observations from calculating the p-values suggest that we are missing some power events in this algorithm. The absence of these events may be responsible for the stark discrepancy observed in power values in some datasets. Some of the power events that could be explored in this regard are 'Frontend Stalls', 'Backend Stalls', 'Dispatch of Instructions to IQ/ROB', 'Register File Reads' etc, although some of the existing power events may already correlate with these power events. It is worth investigating how much the prediction accuracy changes with the inclusion of these power events.

The fact that the gate level power consumption of a design can be predicted with reasonable accuracy (inferred from low RRSE values) at the RTL level is an extremely useful feature, and enables faster temperature-reliability co-optimization by virtue of not having to depend on synthesis of gate-netlists, simulation of gate-level netlists and power extraction from huge VCD files. This is largely because gate-level syntheses and simulations, and subsequent power estimations are time-intensive tasks, and scale proportionally with design complexity. Being able to obtain information regarding the power consumption, and the trend it will

follow based on microarchitectural changes would be specially beneficial to designers with limited computing resources, or short timelines. Another advantage of this methodology is that it provides transparency in how power is estimated. The power events and their linear fitted coefficients are derived from the core being analyzed, and hence offer greater accuracy. This is in direct contrast to McPAT, which uses power models from one core to predict power for subsequent cores and operating conditions.

REFERENCES

[1] T. Kim, Z. Sun, H. Chen, H. Wang and S. Tan, "Energy and Lifetime Optimizations for Dark Silicon Manycore Microprocessor Considering Both Hard and Soft Errors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[2] M. Haghbayan, A. Miele, A. Rahmani, P. Liljeberg and H. Tenhunen, "Performance/Reliability-Aware Resource Management for Many-Cores in Dark Silicon Era," in *IEEE Trans. on Computers*, 2017.

[3] M. Li, W. Liu, L. Yang, P. Chen and C. Chen, "Chip Temperature Optimization for Dark Silicon Many-Core Systems," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits (TCAD), Early Access Pre-Publication*, 2017.

[4] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. Int. Symp. Microarchitecture*, 2009.

[5] S. Xi, H. Jacobson, P. Bose, G. Wei and D. Brooks, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Burlingame, CA, 2015.

[6] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of 27th International Symposium on Computer Architecture*, Vancouver, BC, Canada, 2000.

[7] A. Varma, E. Debes, I. Kozintsev, P. Klein and B. Jacob, "Accurate and fast system-level power modeling: An XScale-based case study," in *ACM Trans. Embed. Comput. Syst. 7*, -, 2008.

[8] M. Caldari and e. a. , "System-level power analysis methodology applied to the AMBA AHB bus [SoC applications]," in *Design, Automation and Test in Europe Conference and Exhibition*, -, 2003.

[9] N. Bansal, K. Lahiri and A. Raghunathan, "Automatic Power Modeling of Infrastructure IP for System-on-Chip Power Analysis," in *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, Bangalore, 2007.

[10] P. Stralen and A. Pimentel, "A High-level Microprocessor Power Modeling Technique Based on Event Signatures," in *J Sign Process Syst*, -, 2010.

[11] R. Basu Roy Chowdhury, "AnyCore : Design, Fabrication, and Evaluation of Comprehensively Adaptive Superscalar Processors," NCSU, Raleigh, NC, 2016.

[12] R. Basu Roy Chowdhury. [Online]. Available: http://rangeen.org/anycore.

[13] E. Rotenberg. [Online]. Available: http://people.engr.ncsu.edu/ericro/research/.

[14] K. Asanovic and D. A. Patterson, "Instruction Sets Should Be Free: The Case For RISC-V," -, Berkeley, 2014.

[15] "Spike: The RISC-V instruction set simulator," [Online]. Available: https://github.com/riscv/riscv-isa-sim.

[16] T. Shah, "FabMem: A Multiported RAM and CAM Compiler for Superscalar Design Space Exploration," NCSU, Raleigh, 2010.

[17] "What is Jupyter," [Online]. Available: http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html.

[18] "Python Data Analysis Library," [Online]. Available: http://pandas.pydata.org/.

APPENDICES

APPENDIX A

Primetime Scripts

This section describes the script file ‘run_ptpx.tcl’ for extraction of power in epochs of 1000 cycles. The epoch number can be changed by modifying the parameter ‘powercut_count’.

```
set begintime [clock seconds]

set type synthesized

set corner fast

# Define a helpful function for printing out time strings
proc timef {sec} {
  set hrs [expr $sec/3600]
  set sec [expr $sec-($hrs*3600)]
  set min [expr $sec/60]
  set sec [expr $sec-($min*60)]
  return "${hrs}h ${min}m ${sec}s"
}


if {$corner == "slow"} {
    set link_library   [concat "*" $techlib_slow $ADDITIONAL_LIBRARIES]
}
if {$corner == "typical"} {
    set link_library   [concat "*" $techlib_typical $ADDITIONAL_LIBRARIES]
}
if {$corner == "fast"} {
    set link_library   [concat "*" $techlib_fast $ADDITIONAL_LIBRARIES]
}

set VERILOG_NETLIST
"${NETLIST_DIR}/verilog_final_${MODNAME}_${RUN_ID}.v"

set GATE_LEVEL_UPF ${NETLIST_DIR}/${MODNAME}_${RUN_ID}.upf

read_verilog $VERILOG_NETLIST

current_design $MODNAME
```

```
link_design

set_wire_load_model -name "5K_hvratio_1_1"

set_wire_load_mode enclosed

set FILE_EXISTS [file exists $GATE_LEVEL_UPF]
if { $FILE_EXISTS == 1 } {
  echo "Reading UPF ${NETLIST_DIR}/${MODNAME}_${RUN_ID}.upf"

  set upf_create_implicit_supply_sets false

  load_upf     $GATE_LEVEL_UPF
}

create_clock -name $clkname -period $CLK_PER $clkname
set_input_delay $IP_DELAY -clock $clkname [remove_from_collection [all_inputs]
$clkname]
set_output_delay -clock $clkname 0 [all_outputs]
set_driving_cell -lib_cell "$DR_CELL_NAME" -pin "$DR_CELL_PIN"
[remove_from_collection [all_inputs] $clkname]

set_app_var power_enable_analysis TRUE
if { $USE_ACTIVITY_FILE == 1 } {
  set_app_var power_analysis_mode average
} else {
  set_app_var power_analysis_mode average
}
set_app_var power_x_transition_derate_factor 0

set_app_var power_default_toggle_rate 0.2
set_app_var power_default_static_probability 0.5
set_app_var power_default_toggle_rate_reference_clock fastest

set_app_var power_enable_multi_rail_analysis true

check_timing
update_timing

##########################################################################
#       read VCD file
##########################################################################

#run power extraction in chunks of 1000 cycles or 10000 ns.
```

```
set MAX_ITERATION 356
set powercut_count 10000
set i 0
set j [ expr $i + $powercut_count ]
set iter 0

while { $iter < $MAX_ITERATION } {

   echo "==Starting Data==" >>
${REPORT_DIR}/power_ptpx_${MODNAME}_${RUN_ID}.rpt
   echo "$i to $j cycles" >>
${REPORT_DIR}/power_ptpx_${MODNAME}_${RUN_ID}.rpt
   echo "$i to $j cycles"

   if { $iter == [expr $MAX_ITERATION - 1] } {

      read_vcd "${VCD_DIR}/${VCD_FILE}" -strip_path "${VCD_INST}" -rtl -time "$i -
1"
   } else {

      read_vcd "${VCD_DIR}/${VCD_FILE}" -strip_path "${VCD_INST}" -rtl -time "$i
$j"
   }
   ##########################################################################
   #      check/update/report power
   ##########################################################################
   update_power
   check_power
   report_power -hierarchy -nosplit -sort_by name >>
${REPORT_DIR}/power_ptpx_${MODNAME}_${RUN_ID}.rpt
   echo "==End Data==" >>
${REPORT_DIR}/power_ptpx_${MODNAME}_${RUN_ID}.rpt
   set i $j
   set j [ expr $i + $powercut_count ]
   set iter [ expr $iter + 1 ]
}

report_host_usage
exit
```

APPENDIX B

Python Linear Regression Script

This section presents the python script used for performing linear regression using the event signatures and the power values of the training datasets. This script is presented here in the form of a standard python script, but can also be run as an IPython notebook.

```
# In[1]:

import matplotlib.pyplot as plt
import matplotlib.figure as figure
import numpy as np
import pandas as pd
import re
import pylab

#import the event signatures of the training datasets. Thisfile should also contain the
observed power values.
df = pd.read_csv("event_signatures.csv", delimiter=',', header=0, sep='\s*,\s*')
df = df.dropna()
df = df.set_index('bmark_interval')

headers = list(df.columns.values)

print(df.columns.tolist())



# In[23]:

# Create a new dataframe with only the event_signatures required for the regression

#names of the event signatures go here
indf = df[['BP_lookups','BP_mispredictions', 'BTB_accesses', 'BTB_hits', 'ICache_accesses',
'ICache_misses', 'Decodes', 'Renames', 'Issues', 'Loads', 'Stores', 'Load_Misses',
'Store_Misses', 'Store_Load_fwd', 'Writebacks_to_PRF', 'Commits', 'Load_Violations',
'Pipeline_recovery']]


# Import the input dataframe as a numpy matrix
X = indf.as_matrix()
```

```
index = df.index.values

# Import the observed power values as a numpy matrix
#Y = df['Total_power'].as_matrix()
#Y = df['Switching_Power'].as_matrix()
#Y = df['Internal_Power'].as_matrix()
Y = df['Total_Dynamic_Power'].as_matrix()
Y = Y.astype(float)
print("Input matrix dimensions: {:d} Output matrix dimension: {:d} Index dimension:
{:d}".format(len(X),len(Y),len(index)))


# In[24]:

from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from math import sqrt



from sklearn import linear_model
from sklearn.svm import SVR
regr = linear_model.LinearRegression()
#regr = SVR(kernel='rbf', C=0.03, epsilon=0, verbose=True)

# Train the model using the training sets
regr.fit(X, Y)

# The coefficients
print('Coefficients: \n', regr.coef_)
print('Constant: %.10f'% regr.intercept_)

# Explained variance score: 1 is perfect prediction
print('Variance score: %.5f' % regr.score(X, Y))

#compute the RRSE, Mean Absolute Error and the Maximum Absolute Error
rrse = sqrt(((Y -  regr.predict(X))**2).sum() / ((Y -  Y.mean())**2).sum())
percent_err = np.mean(np.absolute((Y - regr.predict(X))*100/Y))
max_err = max(np.absolute((Y - regr.predict(X))*100/Y))
print('RRSE: %.5f' % rrse)
print('Mean Absolute Error: %.5f' % percent_err)
print('Max Error: %.5f' % max_err)
```

```
# Plot outputs
plt.scatter(index, Y,  color='black')
plt.plot(index, Y,  color='orange', linewidth=3, label="observed")
plt.scatter(index, regr.predict(X),  color='black')
plt.plot(index, regr.predict(X), color='blue', linewidth=3, label="predicted")
plt.xlabel("Datasets")
plt.ylabel("Power in Watts")
pylab.rcParams['figure.figsize'] = (50.0, 10.0)
plt.legend()
plt.show()


# In[10]:

# Add the real prediction and misprediction columns to the original dataframe
df['Predicted_Power'] = list(regr.predict(X))

#save the original data along with the predicted powers
df.to_csv("Result_Power.csv")

#save the coefficients obtained from regression into a text file
get_ipython().magic('store regr.coef_ >> coefficients.txt')
```