Energy savings techniques in out-of-order pipeline through value approximation of instructions with data dependencies

Mohd Tariq Azmy
Iowa State University

Energy savings techniques in out-of-order pipeline through value approximation of instructions with data dependencies

by

Mohd Tariq Azmy

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Computing and Networking Systems)

Iowa State University

Ames, Iowa

2018

DEDICATION

To my wife, Aqilah,

my son, Yusuf,

and my parents, Azmy and Zarah,

for being my source of inspiration.

TABLE OF CONTENTS

Page

LIST OF FIGURES v

LIST OF TABLES vi

NOMENCLATURE vii

ACKNOWLEDGMENTS viii

ABSTRACT x

CHAPTER 1 INTRODUCTION 1

CHAPTER 2. BACKGROUND 3
- 2.1 Approximate Computing 3
  - 2.1.1 Region of Approximate Computing 4
    - 2.1.1.1 Approximation in Software 5
    - 2.1.1.2 Approximation in Architecture 5
    - 2.1.2.3 Approximation in Circuit 5
  - 2.1.2 Error Quality of Metrics 6
- 2.2 Value Speculation 6
  - 2.2.1 Value Prediction 6
  - 2.2.2 Value Approximation 7
- 2.3 Out-of-order Pipeline 7

CHAPTER 3. RELATED WORKS 10
- 3.1 Value Prediction Mechanism 10
- 3.2 Dependencies-based Instruction Speculation 10
- 3.3. Eliminating Recovery Mechanism 11

CHAPTER 4 IDENTIFYING INSTRUCTION FOR APPROXIMATION 12
- 4.1 Instruction Selection 12
- 4.2 True Dependencies 13
- 4.3 Benchmark Overview 13
  - 4.3.1 Sobel Operator 13

CHAPTER 5. DESIGN AND IMPLEMENTATION 15
- 5.1 Computer System Simulator 15
- 5.2 Dependent Instructions Grouping 15
- 5.3 Execution of Approximate Instruction 16
- 5.4 Approximation Table 16
- 5.5 Source Operand Lookup in Renaming Unit 18

CHAPTER 6. METHODOLOGY 20

CHAPTER 7. RESULTS AND EVALUATION..........22
7.1 Performance Analysis..........22
7.2 Error Analysis..........23
7.3 Energy Analysis 25
7.3.1 Energy in Renaming Unit 27
7.3.2 Energy in Scheduling Unit..........28
7.3.3. Energy in Execution Unit..........28
7.3.4 Approximation Table Overhead 29
7.4 Discussion 30

CHAPTER 8. CONCLUSION AND FUTURE WORK 32

CHAPTER 9. REFERENCES 34

LIST OF FIGURES

Page

Figure 2.1 The image quality between precise and approximate image (adapted from [11]) 4

Figure 2.2 Physical register based OoO pipeline (adapted from [29]). 8

Figure 4.1 Example of x86 assembly instructions with dependencies. 13

Figure 5.1 Overview of execution process in the modified pipeline. 17

Figure 5.2 Flow chart of how approximate instruction is executed. 18

Figure 7.1 Speedup of each group with different degrees of approximation. 22

Figure 7.2 RMSE of each of the approximate images with respect to the original image. 23

Figure 7.3 Image output for select cases (parentheses denotes degree of approximation) 24

Figure 7.4 Overall energy savings for all cases. 26

Figure 7.5 Total energy savings in renaming unit. 27

Figure 7.6 Total energy savings in scheduling unit. 28

Figure 7.7 Total energy savings in execution unit. 29

Figure 7.8 Error vs. energy plot for all cases. 31

LIST OF TABLES

Page

Table 5.1 List of Instructions, Address and Tags. 16

Table 6.1 Simulator Configuration. 20

Table 7.1 Energy Relevant Microarchitectural Event Counts. 26

Table 7.2 Dynamic Energy Consumption... 27

NOMENCLATURE

ALU	Arithmetic Logic Unit
CPI	Cycle Per Instruction
CPU	Central Processing Unit
DCL	Dependence Check Logic
DSP	Digital Signal Processing
FP	Floating Point
GPU	Graphics Processing Unit
IPC	Instruction Per Cycle
ISA	Instruction Set Architecture
ILP	Instruction Level Parallelism
IQ	Instruction Queue
OoO	Out-of-Order
PC	Program Counter
RAT	Register Alias Table
RAW	Read-After-Write
RMSE	Root-Mean-Square Error
ROB	Reorder Buffer
SRAM	Static Random-Access Memory

ACKNOWLEDGMENTS

First, I would like to praise Almighty God, for giving me strength and ability to complete this research. His blessings and guidance enabled me to meet this challenge.

I would to like to express my appreciation to my advisor, Professor Akhilesh Tyagi, for the continuous advice and supervision he has given throughout this research. Professor Tyagi has always been patient and flexible with all the issues I encountered during the implementation of this research.

I would like to thank my other committee members, Drs. Swamy Ponpandi and Long Que, for their support and advice as well as Professor Henry Duwe III, for providing numerous inputs that were beneficial to this research. I am especially grateful for his willingness to substitute for one of my committee members who was unable to attend my defense presentation.

I am also grateful to several people and friends who have provided additional assistance to my work. I am thankful to the gem5 contributors from the mailing list who responded to all the questions and issues that I had while working with the tool. I would also like to express my gratitude to Mr. Christos Sakalis from Uppsala University, Sweden, for sharing his previous gem5 modification work that I used as the basis in my study.

I would like to extend my appreciation to the University Putra Malaysia and the Malaysian Ministry of Education for funding my Master of Science Program of Study at Iowa State University.

Finally, I am grateful to my family members, especially my wife, Aqilah, and my parents, Azmy and Zarah, for their love as well as moral support. I feel truly blessed by their faith in me to study abroad to enhance my professional career.

ABSTRACT

Approximate computing has emerged as one of the areas studied over the past few years to improve the performance and energy consumption computers. Approximate computing tolerates imprecision during computation, and it produces data values that are close to the actual outputs obtained from exact computation. From software to circuit level, approximate computing techniques have been applied across all computing domains. This study was carried out on microarchitectural level, where dependencies between two instructions are relaxed in the scheduling unit. This research proposed a technique that allows dependent instructions to execute without waiting for values produced by their producer instructions. This process enabled schedulers to skip certain pipeline processes such operand rename lookup, and instruction wake-up in the instruction scheduler queue to provide additional energy savings. The results of this work revealed an average performance acceleration of 1.25x. In addition, the total of energy savings was achieved at 4.6% for approximation cases that produced tolerable error at the output.

CHAPTER1. INTRODUCTION

The introduction of solid-state devices in the past has enabled computing technology to evolve significantly over time. With Dennard Scaling [1], the length of the transistors can be further scaled down which enables the computing device to run at higher frequencies at the same power densities. Together with Moore's Law [2], the number of transistors per unit area doubled every two years. More recently, the introduction of multi-core processors has led to a dramatic performance increase in computing.

Multiple techniques were proposed and developed in parallel to further increase the performance of computers. These techniques were centered on a concept known as Instruction Level Parallelism (ILP), where some computer instructions are executed simultaneously as opposed to implementing it in sequential order. The Out-of-Order (OoO) core [3] execution was developed to take advantage of free resources in the CPU pipeline, where independent instructions can be scheduled and executed in parallel, and the instructions are able to retire in in-order manner through the Reorder Buffer (ROB).

Another significant technique for increasing the ILP is through Branch Prediction scheme [4]. For *branch* and *jump* instructions, it typically takes multiple clock cycles to compute the branch target address. This could result in pipeline stall, where the subsequent dependent instruction needs to wait for current branch computation to be completed. Branch prediction forecasts the branch direction at the early stage of the pipeline. As more branch instructions are correctly predicted, the number of overall cycles per instruction (CPI) of the program will decrease significantly.

As computers have become mobile, much of the current focus has been directed towards reducing energy and power consumption, since high performance computers draw a lot of energy in battery-operated devices. Among other techniques used, approximate computing is one of the approaches implemented to reduce energy consumption by making some of the computing data values as approximated instead of precise. The total high energy consumption in computing is accumulated from all over the microarchitectural pipeline stages in the superscalar processor. Data processing operations in pipeline stages that contribute to huge energy consumptions are considered as good candidates for approximation. From a hardware perspective, pipeline units that undergo approximation will offer lower energy consumption. This research approximated the data value of instructions with dependencies. By relaxing instruction dependencies, the user skipped the operand rename lookup process in renaming stage and also ignored the instruction wake-up process in the scheduling unit. Energy savings were obtained from these processes as well as the need to use a smaller number of functional units.

The remainder of this thesis is organized as follows. Chapter 2 discusses approximate computing in detail as well as value approximation. OoO core pipeline is also explained since it was used as the foundation for this research. Chapter 3 discusses related works pertaining to approximate computing, especially research that was closely related to the current study. Chapter 4 discusses the selection of instruction for approximation as well as approximation of dependency-based instruction, which was further narrowed to the possibility of omitting the register operand renaming and instruction wake-up processes. Chapter 5 describes the design implementation through the modification of the pre-existing open source CPU system simulator program. Chapter 6 describes methodology and 7 discusses the evaluation and result. Finally, Chapter 8 provides the conclusion and provides recommendations for future study.

CHAPTER 2. BACKGROUND

This chapter discusses an overview of approximate computing, as well as the area in computing system that can be exploited for approximation. Then, data value prediction and approximation are reviewed; value approximation serves as the basis of this research. Finally, the OoO pipeline is illustrated so that the possible approximation in the hardware can be examined.

2.1 Approximate Computing

Approximate Computing has emerged as one of the areas that is explored over the past few years. The failure of Dennard Scaling, where the transistor operating voltage could not be further scaled down proportionally to the transistor size, has caused the power densities of microprocessor to escalate. This has led to what is called as *Dark Silicon* problem, where all transistors could not be fully utilized to prevent the chip from exceeding the thermal limit [5, 6]. Due to this constrain, approximate computing has been proposed as a technique to improve the energy efficiency of a computer [7]. The term approximate computing can be thought as a method that does not produce an exact result, but rather produces an imprecise value that is closed to the actual output value. This method can save a lot of resources utilization as well as computing time, which implies gain in performance and energy, but as a tradeoff, user would expect to see loss of quality at the output [8, 9, 10, 11]. Approximate computing can be applied in areas that can tolerate a percentage of loss and errors, and an example of this would be in Digital Signal Processing (DSP) domain. A lot of DSP application such as image and video processing, speech recognition, scientific data computing, etc., require lots of computation and resources because huge amount of data is consumed as inputs.

In approximate computing, instead of producing an exact output, the method would undergo couple of approximating processes such as, but not limited to, voltage over-scaling [12], reducing the bit width precision for floating point data (which would save the functional units in hardware) [13], and reusing the previous computed result [14]. The loss in quality would be anticipated but as long as the result is tolerable within human perception, then the output is good enough. Figure 2.1 shows an example of comparison of image quality between precise (left) and approximate image (right). The approximate image still can easily be identified, although the quality degrades.

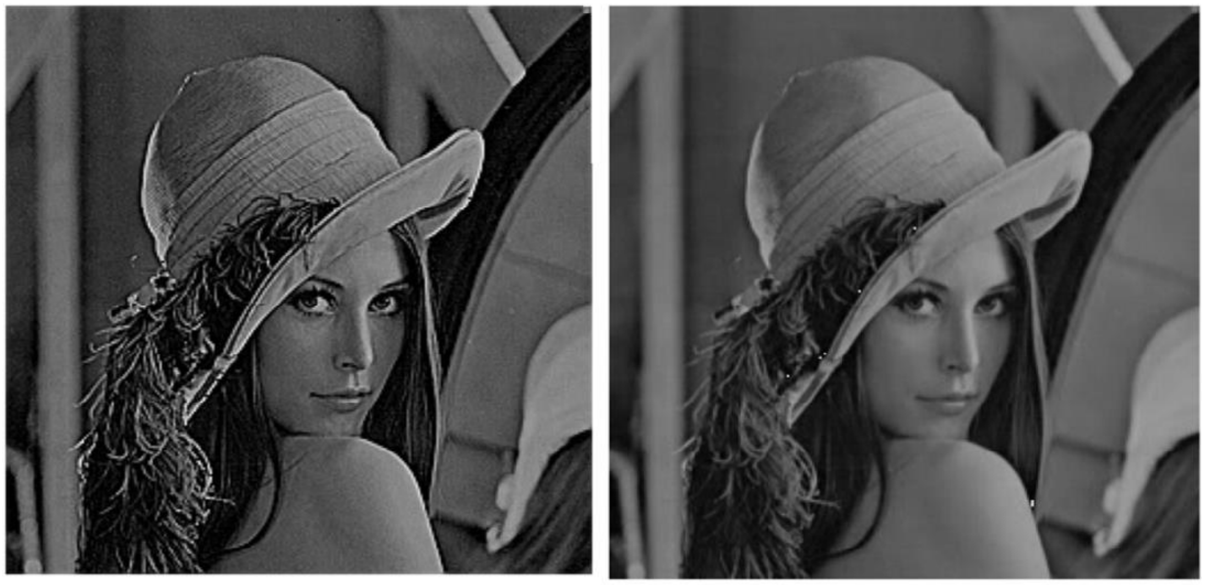


Figure 2.1 *The image quality between precise and approximate image* (adapted from [11])

2.1.1 Region of Approximate Computing

There are various of approximate computing techniques that were introduced, and these are implemented across both software and hardware. According to [8, 15, 16], the region of approximation in computing can be divided into three: software/algorithm, architecture, and circuit. Both architecture and circuit regions fall under hardware approximation techniques.

2.1.1.1 Approximation in Software

Approximation in software is a straight forward technique that require programmers to modify the code or introduce a different algorithm that compute the data in approximate manner. For example, one of approximation techniques in software is *Loop Perforation* [17]: In this approach, some iteration in a loop is skipped, which results in performance enhancement and energy savings.

2.1.1.2 Approximation in Architecture

Approximation in architecture varies from *Instruction Set Architecture* (ISA) level to the microarchitecture level. For example, in ISA extension several instructions are identified at compilation time that are resilient towards approximation. In [20], a region of code in a program can be examined and annotated, and the corresponding instruction are tagged as approximate. On the other hand, for every annotated instruction that is executed during the run time, such as load instruction, the microarchitecture can adapt several approximation techniques and execute them in approximate manner [19].

Data that reside in memory storage can also be tolerated to errors. As indicated in [21], the refresh rate of DRAM is made low to save energy. At the same time, it introduces soft error in the memory cell which leads to alteration of the data value.

2.1.1.3 Approximation in Circuit

For approximation in the circuit level, some techniques include voltage over-scaling in ALU as well as SRAM [12]. Lower voltage in SRAM causes the memory cell to fail and, thus, introduces errors during accessing the data from caches [22], while voltage over-scaling in logic cause timing violations which affect data precision [23]. Logic circuits such as the adder and multiplier can also be modified for approximation. For example, in a full adder circuit the design of the adder is simplified at the circuit level which causes the adder to perform an inaccurate addition function [11, 25].

2.1.2 Error Quality of Metrics

Since approximate computing improves performance and energy at the expense of errors and loss of quality, several quality metrics are used to quantify the error that is introduced. Different applications produce different outputs; therefore, specific quality metrics are used for each application. For DSP application, Signal to Noise Ratio (SNR) of the approximate output are compared with the exact output for evaluation [24, 25]. Root Mean Square Error (RMSE) is another quality metric used in image processing application, and average relative error is used for numerical output [26].

2.2 Value Speculation

Data value speculation is a technique that predicts the data used for computation before the actual value is loaded to secure the computation cycle in the microarchitecture. The current research predicted and approximated the output value of selected instruction to exploit the microarchitecture for additional energy and power savings.

2.2.1 Value Prediction

Research in value prediction was carried out in the past to increase the performance of a computer. The concept of *Value Locality*, where previously seen values are repeatedly appeared in a program, enables the microarchitecture to predict the value of the data [27]. Similar to the branch prediction technique, where the speculation is made towards branch previous history (taken or not taken) and is represented in a single bit, value prediction extends the speculation to the entire data bits (32 or 64 bits).

A typical approach to predict a value is through the incorporation of a value look-up table or buffer in the microarchitecture [14, 33-35]. The computed data are stored in the table and accessed later, when values of an instruction are being speculated. Due to the size of the data, the table is typically large and is generally introduced as an overhead in the pipeline.

Similar to the branch prediction mechanism, in the event of misprediction of a value for an instruction, the pipeline will be squashed and the process will restart by re-executing the instruction precisely (normal execution).

2.2.2 Value Approximation

The concept of value approximation [19, 28] is similar to value prediction wherein locality of the data permits the value to be speculated. However, in value approximation the value does not need to be precise all the time. The exactness of a value can be relaxed and imprecise data can be used to compute the output. Error is expected at the output; however, as long as the effect is minimal and tolerable it should be adequate.

A distinct feature of value approximation is that the recovery step is eliminated during misprediction. A gain in performance and energy can be achieved due to the saving of the clock cycle and energy that are used during rollback/recovery steps.

2.3 Out-of-order Pipeline

An OoO superscalar pipeline core has become mainstream in the commercial CPU due to the ability to dynamically execute instructions in parallel as opposed to single scalar core. Figure 2.2 shows an example of an OoO pipeline that is modeled inside an McPAT tool [29], a power modeling tool for CPU. This model is based on the architecture of Alpha 21264 processor [30] which is a physically-register based OoO core.

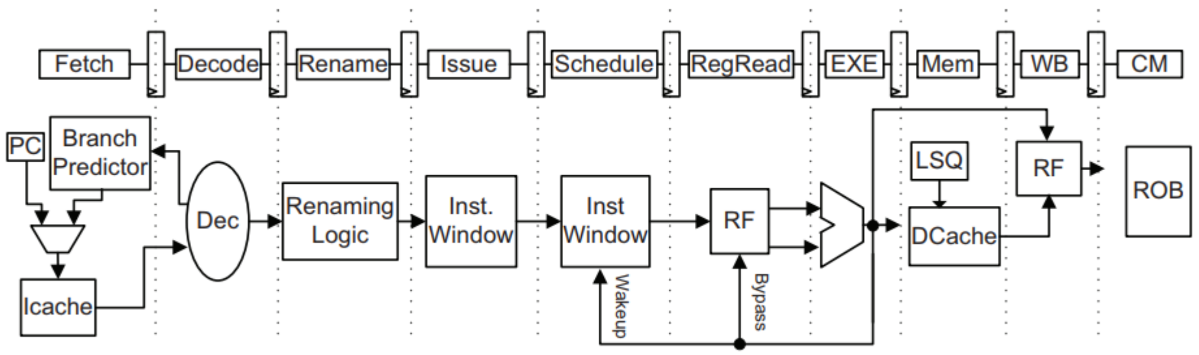


Figure 2.2 *Physical register based OoO pipeline* (adapted from [29])

Due to limited number of architectural registers allocated during compiling time some independent instructions are assigned with the same architectural register, which could lead to false dependencies between those instructions. In the renaming stage, the architectural register is renamed, or mapped into the microarchitecture's physical register to take care of the false dependencies issue. The Register Alias Table (RAT) contains a mapping of the architectural register for the physical register. The Dependence Check Logic (DCL) circuit checks true and false dependencies between instructions, and maps the appropriate physical registers to the destination and source operands in the architectural register.

In the Issue and Schedule stages, the instruction will be dispatched into a buffer known as Instruction Queue (IQ). The IQ holds instructions and schedules any instruction for execution that is ready in an out-of-order manner. For an instruction to be ready its source operands must also be available. Otherwise, for dependent instruction it must wait for another instruction(s) (producer) to yield the result, which will then be written into the register file and finally consumed by a dependent instruction (consumer). This process is known as *Instruction Wakeup/Select*, where the producer wakes up the consumer instruction once the result is ready.

For instructions with Read-After-Write (RAW) dependency, this research proposed to break this dependency by executing the consumer instruction without waiting for the producer instruction. The result value of consumer instruction would be approximated and the instruction, itself, would not have to be issued to the functional unit for execution.

In the rename stage, since the approximate instruction result (destination) is speculated, there is no need to rename the source operand since it is not being used. In the wakeup/select stage, since the consumer does not have to wait for the producer to generate the result, the wakeup process is omitted. Both techniques could lead to additional energy savings.

CHAPTER 3. RELATED WORKS

3.1 Value Prediction Mechanism

Previous works in value prediction are centered on history-based approach. A result value that is computed from an instruction is kept in a prediction table, and this value can be reused constantly by the same instruction that follow in the next cycles. This technique is known as *last value prediction*, as described by Lipasti et al. [14]. In a similar work, Sodani and Sohi [31] presented an *instruction reuse* scheme, in which an instruction can reuse the result produced by a previously-executed instruction based on the condition that both instructions are consuming similar input values (i.e. instructions with different Program Counter (PC) value). Azam et al. [32] implemented a buffer known as *execution cache*; during execution of an instruction, in which registers find and grab the value from the cache and computation in the functional unit is bypassed. The energy spent in cache is less that the energy spent in the functional units which leads to energy savings.

Values stored in the table can also be updated regularly to improve the accuracy in the predicted value. While last value prediction uses the same constant data repeatedly, a certain predictor can also perform some computation to update its stored value. As shown in [33-35], the authors used stride value to predict data based on the difference between two previous values in the history. In two of the same studies [33, 34], a context-based predictor was also used whereby data are updated according to a specific pattern generated from the previous history of the data value.

3.2. Dependencies-based Instruction Speculation

True dependencies limit the ILP of a computer. Couple of works has focused on improving the ILP through speculative dependencies instruction. Calder et al. [34] filtered and selected a group of instructions for value predictions that provided a huge impact on performance, including instruction on the long dependency path. Instruction on a critical path can be retired faster provided that the values consumed by those instructions are predicted correctly.

Selecting instructions with a long latency for speculation has also proven to be beneficial. As described by Alvarez et al. [24], selecting long latency instruction such as floating-point multiplication and division for value approximation, provides a substantial gain in performance and energy savings. In addition, several bits in the operands value are truncated and used as tags in the table. Different instructions that have matching operand tags can use these values; although the final quality of the output is degraded, the error is tolerable. The authors also implemented this scheme for the *in-order* processor.

3.3 Eliminating Recover Mechanism

In a traditional value prediction scheme, in the event of value mispredictions all in-flight instructions will be squashed, and the pipeline will restart from where the instruction is mispredicted. Such a recovery is not needed in value approximation because an inexact value can still be used for computation. San Miguel et. al [19] illustrated the concept of approximating a value for load instruction. In the study, the authors approximated the value when there is a missing cache; instead of spending additional cycles to obtain the value from the main memory, the value was taken directly from the approximation table which might be inaccurate. Approximating the value during cache misses delivers a significant performance gain and reduces energy spent in fetching the value from the main memory. A similar scheme was implemented in a Graphics Processing Unit (GPU) [28], where the main target was to reduce the bottleneck of memory bandwidth. In the same study, Thwaites et al. used last-value prediction for approximating floating-point data and stride for integers.

CHAPTER 4. IDENTIFYING INSTRUCTION FOR APPROXIMATION

In order to execute certain instructions as approximate, the instruction must first be identified and determined prior to running the program on the microarchitecture. It is most important to select instructions for approximation that ensure the program behavior will not be affected.

4.1 Instruction Selection

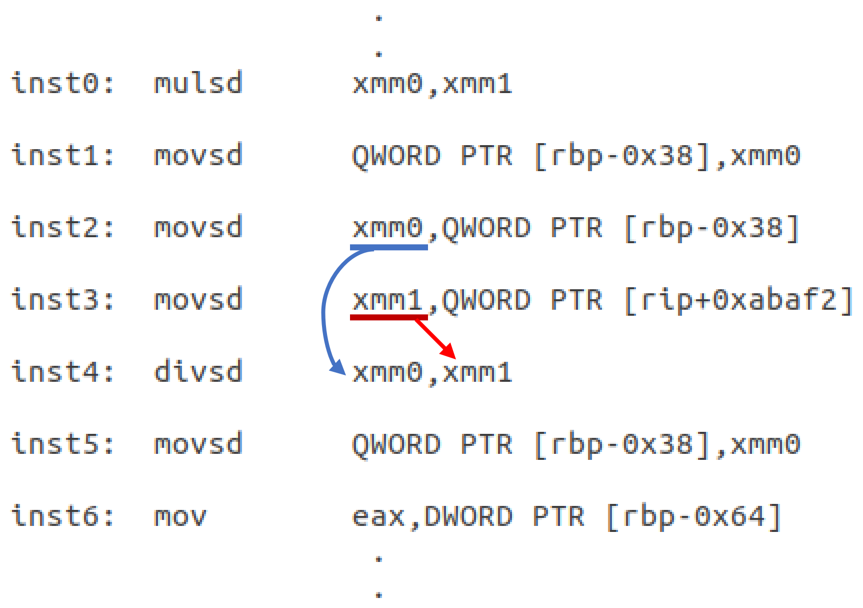
Previous research on approximate computing has made it possible to determine which instruction is suitable for approximation. Through approximation language support frameworks such as [18, 20, 36], programmers can identify the region of approximation in the program and provide annotation to the data in the code which, in turn, is written in high-level programming language. Data can be tagged as precise or approximate in which the framework evaluates the safety of the approximation. To ensure the data are safe-to-approximate, the data must not affect the control flow of a program or involve memory operation, i.e. address of memory [19]. Typically, data that are suitable for approximation are selected from a region that is time intensive as the process is executed multiple times, i.e. expensive loops. [17, 19, 26].

In this study, selection was narrowed to arithmetic instruction, specifically addition, multiplication and division, because the instructions require at least two operands. Similar to previous studies, the purpose was to execute consumer instruction without waiting for its producer to maximize performance gain.

An x86 ISA was utilized for this purpose and *Sobel Operato*r sample code was obtained from ACCEPT framework [18]. The code comes pre-annotated. This work was leveraged to identify the instruction.

4.2 True Dependencies

The foundation of this work was based on true dependencies of instructions, which are also known as RAW dependencies. As discussed in Chapter 2, approximate data from the program that have dependencies were used to speculate consumer instruction without waiting for results from producer instruction(s). Figure 4.1 shows a portion of assembly instruction with dependencies from Sobel operator code generated with *objdump* in Linux. Inst. 2 and Inst. 3 (movsd) are producer instruction while Inst. 4 (divsd) is consumer instruction.



```
                          .
                          .
inst0:  mulsd      xmm0,xmm1
inst1:  movsd      QWORD PTR [rbp-0x38],xmm0
inst2:  movsd      xmm0,QWORD PTR [rbp-0x38]
inst3:  movsd      xmm1,QWORD PTR [rip+0xabaf2]
inst4:  divsd      xmm0,xmm1
inst5:  movsd      QWORD PTR [rbp-0x38],xmm0
inst6:  mov        eax,DWORD PTR [rbp-0x64]
                          .
                          .
```

Figure 4.1 *Example of x86 assembly instructions with dependencies*

4.3 Benchmark Overview

4.3.1 Sobel Operator

Sobel operator [38] is an image processing filtering technique used for edge detection of a greyscale image. Detection is based on the gradient measurement of an image through derivative approximation function. The filter algorithm has two kernels, horizontal and vertical kernels that are used for x-direction and y-direction edge detection, respectively. The image is convoluted with these two kernels and combined to produce the gradient magnitude of the image. Equation 4.1 shows the horizontal and vertical kernel used in Sobel operator algorithm.

$$\mathbf{G_x} = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \qquad \mathbf{G_y} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{4.1}$$

Sobel application is suitable for approximation because the algorithm has loops that keep iterating to compute the convolution, in which the number of iterations depends on the size of the image. The input image is stored as a 2-dimensional array whereby each iteration in the loop of the element of the array is incremented which corresponds to different pixels of the image. The pixel value can be approximated; thus, instead of using an exact pixel, the adjacent pixel can be reused.

Not all instructions in Sobel code are safe for approximation; therefore, the source code was modified in this study to include instructions that are insignificant and use additional instructions to demonstrate the concept. In addition, based on the provided code, filtering was done only for vertical direction; calculation for horizontal axis was not included.

CHAPTER 5. DESIGN AND IMPLEMENTATION

5.1 Computer System Simulator

A Gem5 simulator [37] was used in this study. Gem5 is an open source simulator that is widely used in computer architecture research. Gem5 is also a modular platform that includes several microarchitectures such as an in-order and OoO processor, and can be customized according to the user's need. The OoO model in Gem5 was designed based on Alpha 21264 [30] which implements physical-register based instruction window, known as issue queue (IQ), in the issue stage. The IQ only holds instruction and physical register operand tags and, during the wakeup process, the data value will be accessed from the physical register file and issued to the functional unit for execution. In addition to running the simulator in system-call emulation mode, Gem5 is also capable of running the simulation in full system mode. The Gem5 was configured in this study to run in system-call emulation mode to modify the microarchitecture. Gem5 also supports multiple ISA such as x86, ARM, Alpha, RISC-V, etc.

5.2 Dependent Instructions Grouping

In this research it was assumed that some compilers could generate additional tags and information for approximate instruction. Additional bits are used to tell if an instruction is approximate or precise, as well as a producer and consumer type. It was also assumed that, for related consumer instructions, a compiler can assign a group tag to each instruction that is used for an approximation counter. The bit width of a group tag varies with the number of instruction pairs to be approximated. For example, 3 bits of group tag would enable a total number of 8 approximated instructions or fewer. Table 5.1 shows the additional tags of an instruction. For Apx bit, bit 0 is assigned to exact instruction and 1 is for approximate

Table 5.1 *List of instructions, addresses and tags.*

Instruction PC	Apx bit	Inst. Type bit	N-bits Group
0x401244	1	0	1 (01)
0x401248	1	1	1 (01)
0x40124c	0	-	-
.	.	.	.
.	.	.	.
0x40125d	1	0	2 (10)
0x401253	1	1	2 (10)
0x401257	0	-	-

instruction. For Inst. Type, 0 is assigned to producer and 1 is assigned to consumer instruction. Group bit is a unique identifier for each approximate instruction pair.

5.3 Execution of Approximate Instruction

The execution of an approximated consumer instruction takes place in the execute stage. In a normal operation, once the value of source operands of an instruction is ready, the instruction is set to be ready for execution where it will be issued into the functional unit. For approximated instruction, this process is somehow bypassed; instead of issuing the instruction to functional unit, the destination register value will grab its data from an approximation table. This instruction does not have to wait for its producer and can proceed directly for approximation. The value that is grabbed from the table will also be forwarded to the register file. Figure 5.1 shows an overview of the block diagram, slightly modified from [29], with the addition of the approximation table in the pipeline.

5.4 Approximation Table

Similar to previous research on value approximation, a lookup table was developed to store the value used for approximation. To maintain a low overhead, last value approximation was used, whereby the previously seen value in history was re-used. Only the

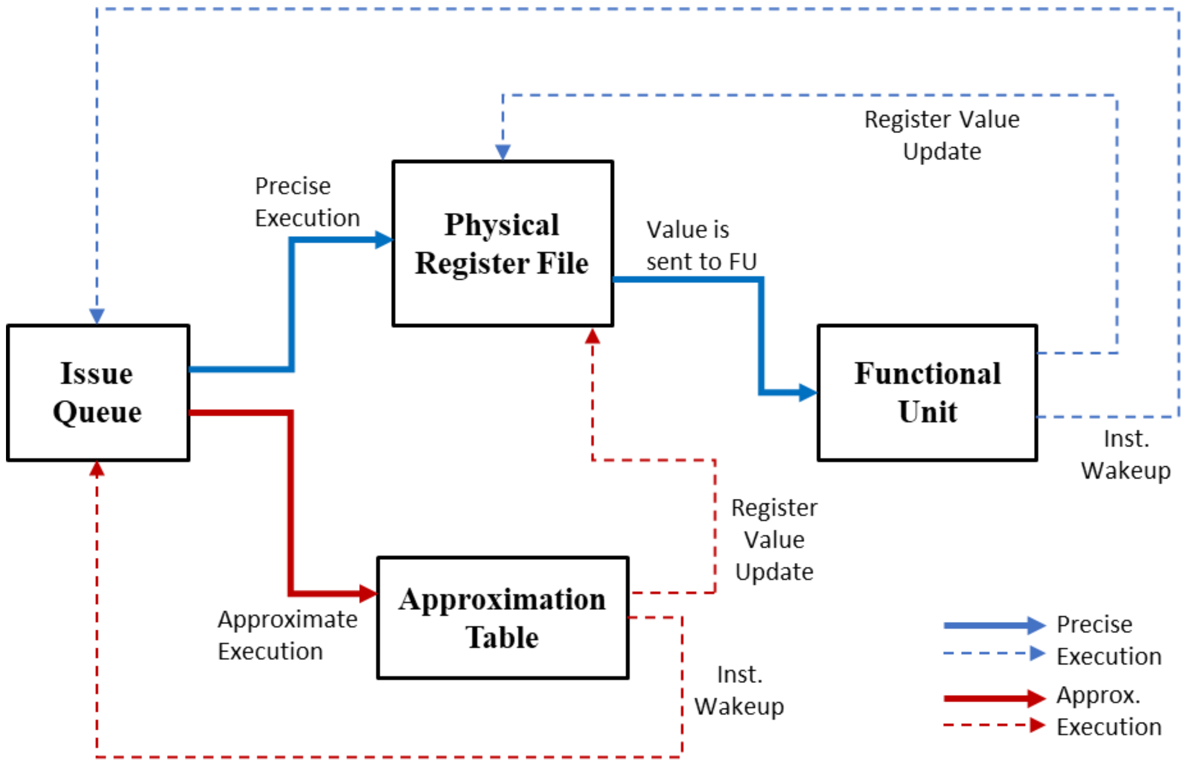


Figure 5.1 *Overview of execution process in the modified pipeline*

destination register value on a consumer instruction is stored in this table, and the instruction's PC/group number can be used as tags, or keys for associative-searching purposes. In addition, there is also a counter field that is allocated for each approximate instruction entry. The counter maximum value indicates degree of approximation, which is the number of times the value in table is used before it is being replaced and updated to a new value. Users set this counter maximum value in advance. For example, if the counter value is set to three, it means the instruction will be approximated for three consecutive times. Once the counter reaches its maximum value, it will be reset to 0. Zero value represents precise execution, and the value computed from precise execution will replace the old value in the table entry. This pattern is fixed until the running program finishes. Figure 5.2 shows the execution flow of executing approximate instructions.

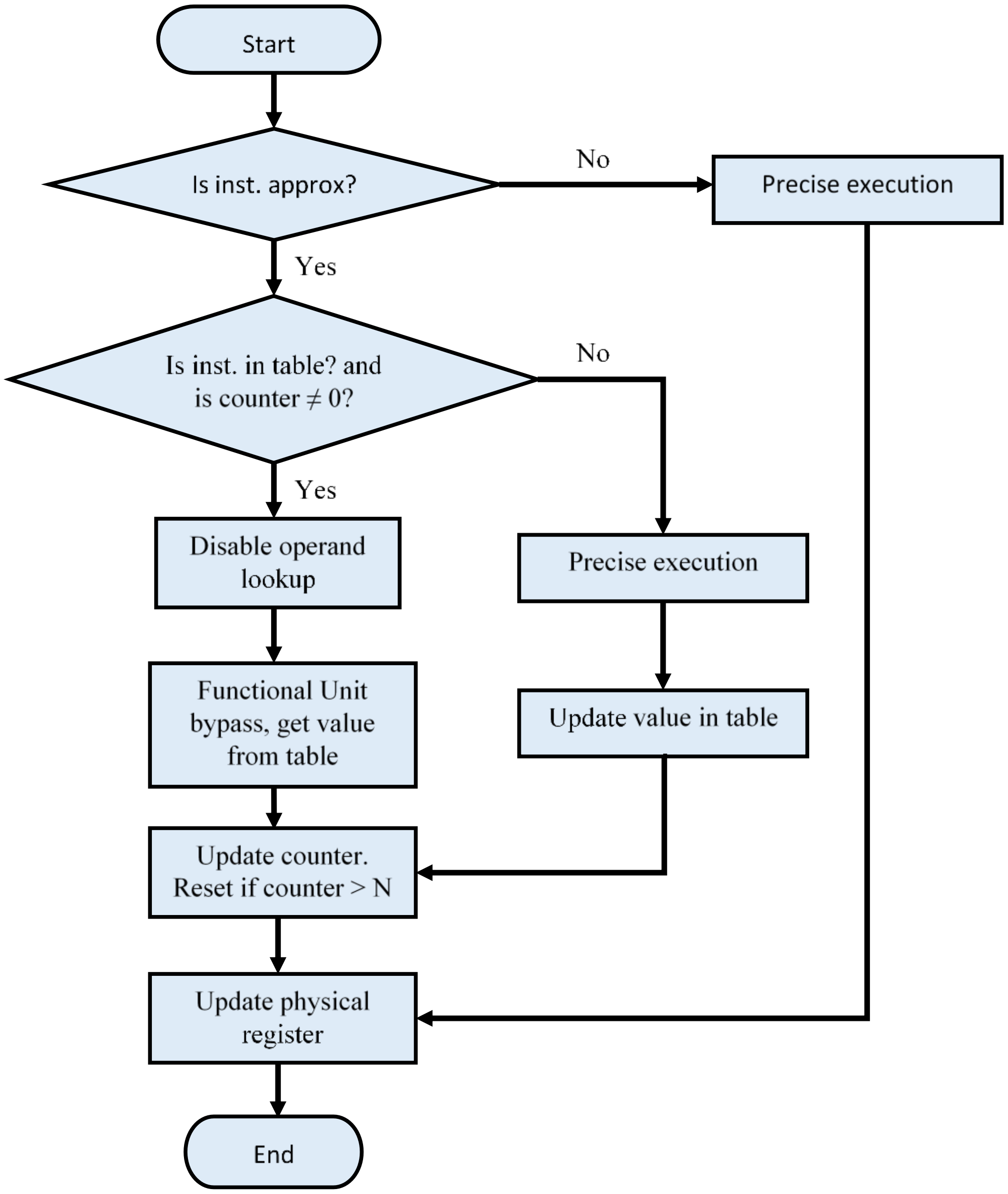


Figure 5.2 *Flow chart of how approximate instruction is executed*

5.5 Source Operand Lookup in Renaming Unit

The source operand lookup process can be skipped for all consumer instructions that are executed approximately. Typically, in a regular lookup process the DCL will perform a check on the RAT and look for the entry that maps the operand architectural register to the physical register. The source operand will be renamed to the physical register index according to that mapping.

Since approximate execution only updates the value of destination register, the value of source operand(s) is not needed to compute an output. In this situation, instead of waiting for the producer to feed the value, any approximate instruction can be set as ready to issue in the pipeline since it has inactive and unnamed source operands.

The renaming unit also needs to check the counter in the approximation table before the operand lookup process can be bypassed. This procedure is needed to keep the renaming process in sync with the execution process. In other words, the lookup process can only be skipped if and only if the instruction is set to be executed in approximate manner.

CHAPTER 6. METHODOLOGY

MATLAB was used to compute and compare the Root-Mean-Square error between the exact and approximate image to evaluate the error from the approximate output. McPAT [29], an open source modeling framework tool that calculates and reports the power and area based on modern processor, was used for energy consumption. McPAT takes its input from Gem5's statistics and configuration reports. These reports contain details about OoO core processor parameters, as well as number of statistics for every important operation that takes place in the pipeline. Some examples of these statistics include total number of cycles, total number of committed instructions, number of branch mispredictions, etc. The number of rename lookups were accessed from the statistics which correspond to the number register operand lookup made in simulation. Performance measurement was accessed by using the CPI value from the statistics. Simulations were run on Gem5's system-call emulation mode, and the settings were configured according to the specifications listed on Table 6.1. Other parameters that are not listed in Table 6.1 were set to a default value provided in Gem5.

Table 6.1 *Simulator Configuration.*

Component	Type/Size
Processor	DerivO3CPU (Out-of-Order), 2 GHz, 4-wide issue
Target ISA	x86
Main Memory	4096 MB
L1 Data Cache	32 KB (2-way set associativity)
L1 Instruction Cache	32 KB (2-way set associativity)
Cache Line Size	64 B

The experiment was divided into three groups which corresponded to the number of instructions that were approximated, i.e. approximate one, two, and three instructions in total. Multiplication was selected for one instruction approximation group. Multiplication and division were selected for the two instructions group, whereas the three instructions group included adding instructions together with multiplication and division.

An approximation pattern was selected for each group, known as degree of approximation. For *Approx-1*, an instruction undergoes approximation with every other precise execution. Thus, the data value in the approximation table is used once before it is updated again through the next precise execution. The process is similar with *Approx-2* and *Approx-3*, whereby the value that belongs to a particular instruction in the table is used twice and three times, respectively, before being updated with new data value. This degree of approximation is set through the counter value in approximation table.

These instructions are not directly dependent on each other; load and store instructions are located between the instructions. The add instruction is integer type, while multiplication and division instruction are floating-point type.

CHAPTER 7. RESULTS AND EVALUATION

7.1. Performance Analysis

The performance of every approximation case was compared to the original simulation. The speedup for every case is shown in Figure 7.1. Group 1 (blue) approximates only one instruction (mult); group 2 (orange) approximates 2 instructions (mult and div); and group 3 (green) approximates 3 instructions (mult, div, and add) as group 3. This group designation is used for reporting the results.

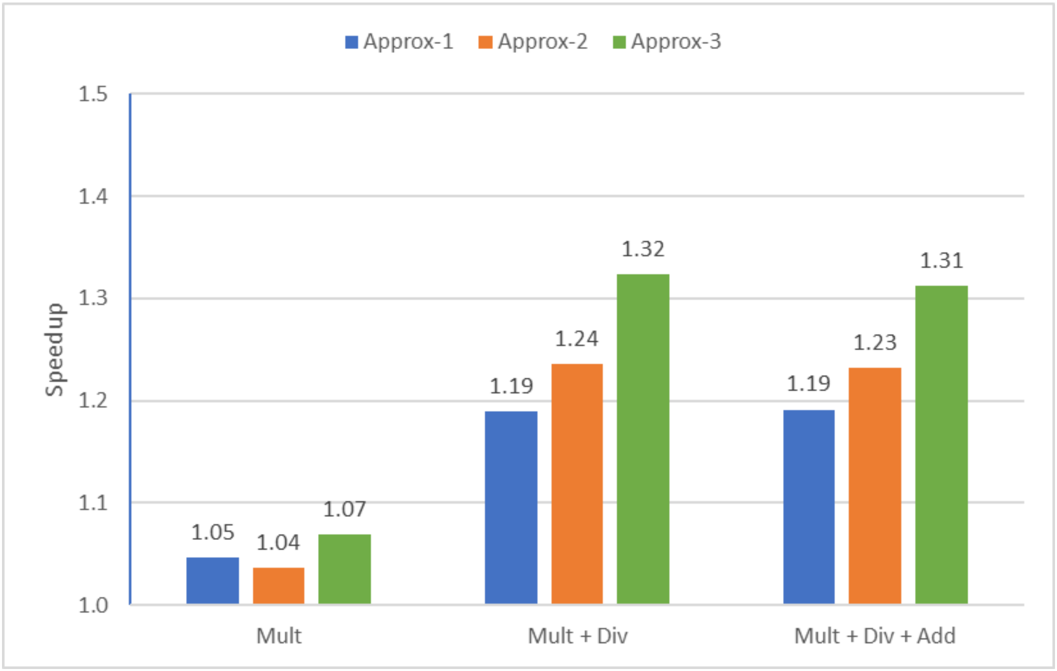


Figure 7.1 *Speedup of each group with different degrees of approximation*

Speedup increases with more number of approximated instructions. In addition, increasing the degree of approximation in the same group slightly improves the performance, although this is not seen in approx-2 case for group 1. In this experiment, the latency of multiplication instruction was set to 4 cycles and the latency of division operation to 20 cycles (non-pipelined). Skipping the computation that takes numerous cycles in the functional unit contributes to the increase of the performance. In the case of group 3, having an additional add instruction does not give any significant boost to the performance. Add instruction typically takes one cycle to compute, and this is equal to the number of cycles that is set when performing an approximation.

7.2 Error Analysis

The Root Mean Square Error (RMSE) metric was used to evaluate the quality of the output image. RMSE evaluates how close a pixel value of an approximate image is to the original image's pixel value, by measuring the difference in the value of pixels between two images. Figure 7.2 shows RMSE of all cases. A lower number indicates better quality.

The output image from original simulation as well as image obtained from select cases are illustrated in Figure 7.3, together with their respective RMSE value. For group 2

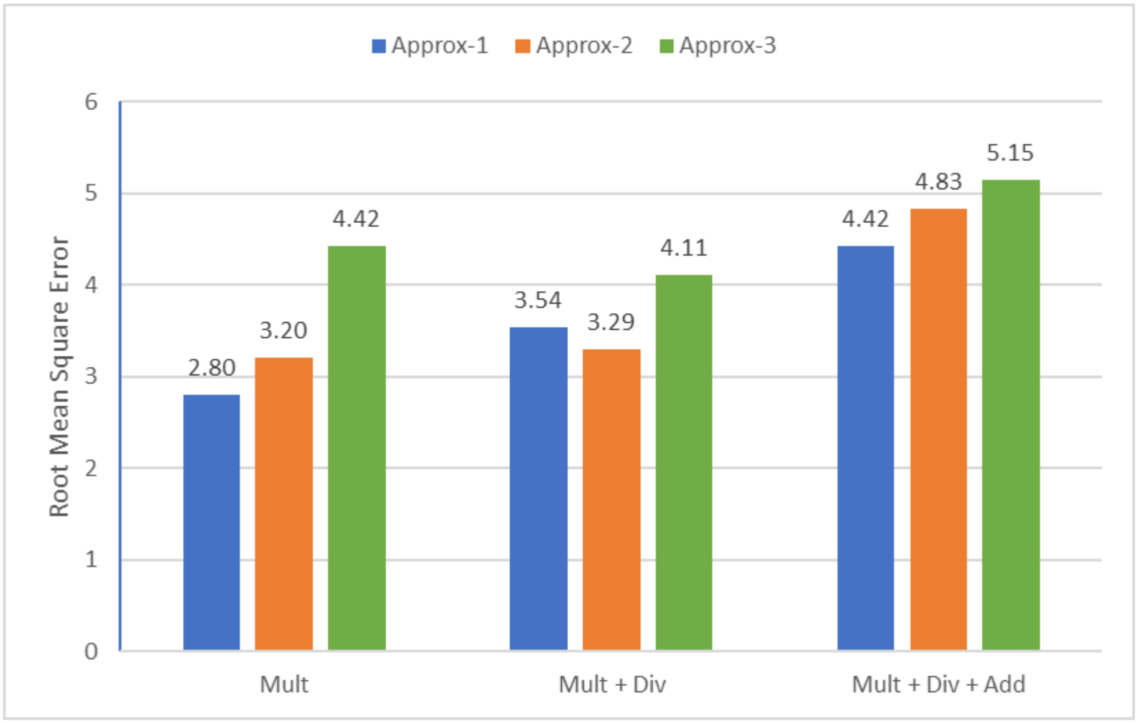
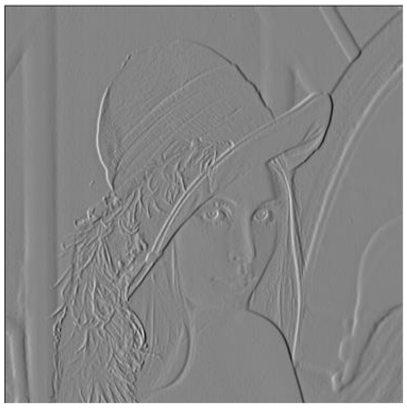
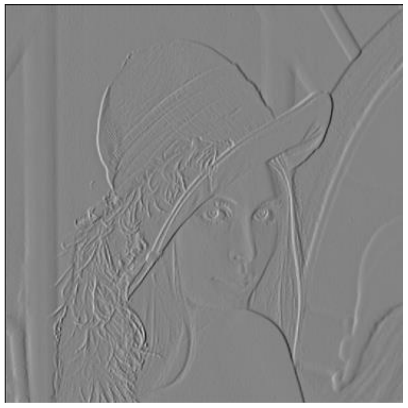


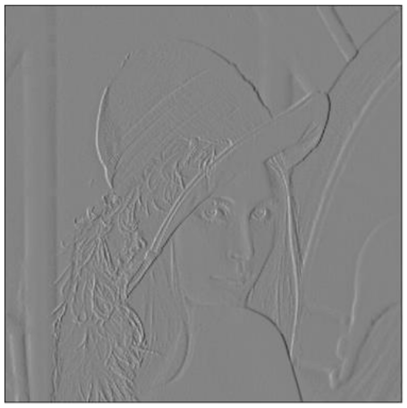
Figure 7.2 *RMSE of each of the approximate images with respect to the original image*



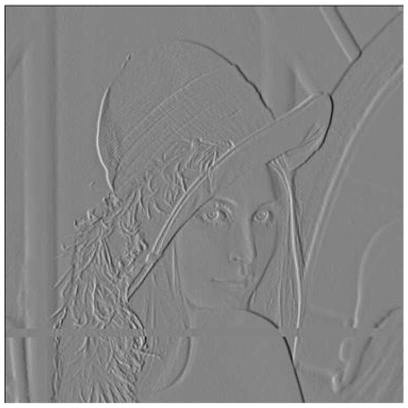
(a) Original, RMSE = 0



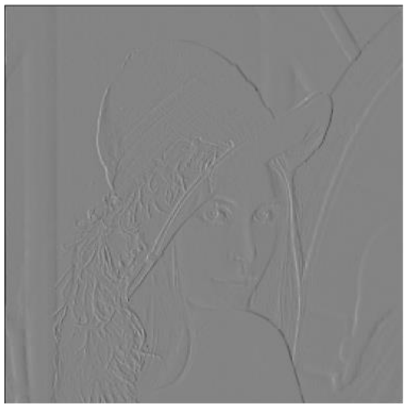
(b) Mult(1), RMSE = 2.89



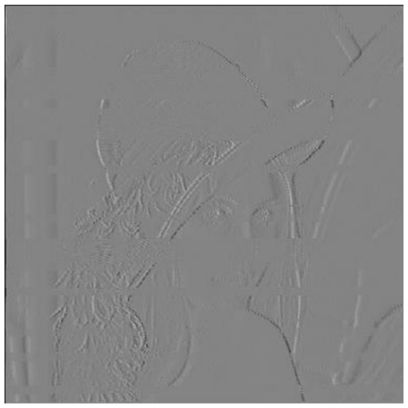
(c) Mult-Div(1), RMSE = 3.54



(d) Mult-Div(2), RMSE = 3.29



(e) Add-Mult-Div(1), RMSE = 4.19



(f) Mult-Div(3), RMSE = 4.83

Figure 7.3 *Image output for select cases* (parentheses denotes degree of approximation)

with approximation degree of 2, there is a slight distortion in the image (Figure 7.2d), even though the RMSE is lower than output from the same group with approximation degree of 1.

The results imply that quality of image is not proportional to the degree of approximation. However, increasing the number of approximated instructions degrades the output quality. For this Sobel application, it was revealed that an approximated output image still looks acceptable with RMSE value of ~3.5, subject to distortion.

7.3. Energy Analysis

The energy consumption was computed based on the static and dynamic power reports generated from McPAT tool. The power measurement is derived based on 90 nm technology node that was set up in McPAT. The energy is obtained by multiplying power with the execution time. In addition to measuring the energy in the whole microarchitecture core, the energy consumption was also examined for certain individual units of interest in the pipeline. The analysis was begun by observing the overall energy consumption in the microarchitecture for all cases of approximation. Based on this approximation technique, energy savings was obtained in most of the cases. Figure 7.4 shows the percentage of energy savings in every case in this experiment.

Due to skipping and omitting some microarchitectural event in the pipeline, the reported number of related events was slightly less. Next, the dynamic power consumption was examined to determine how this method contributes to the energy savings. The readings from the statistics are related to the switching activities in the circuit that occur during the run time. Dynamic power consumption corresponds to *activity factors*, which is one of the parameters used in dynamic power calculation.

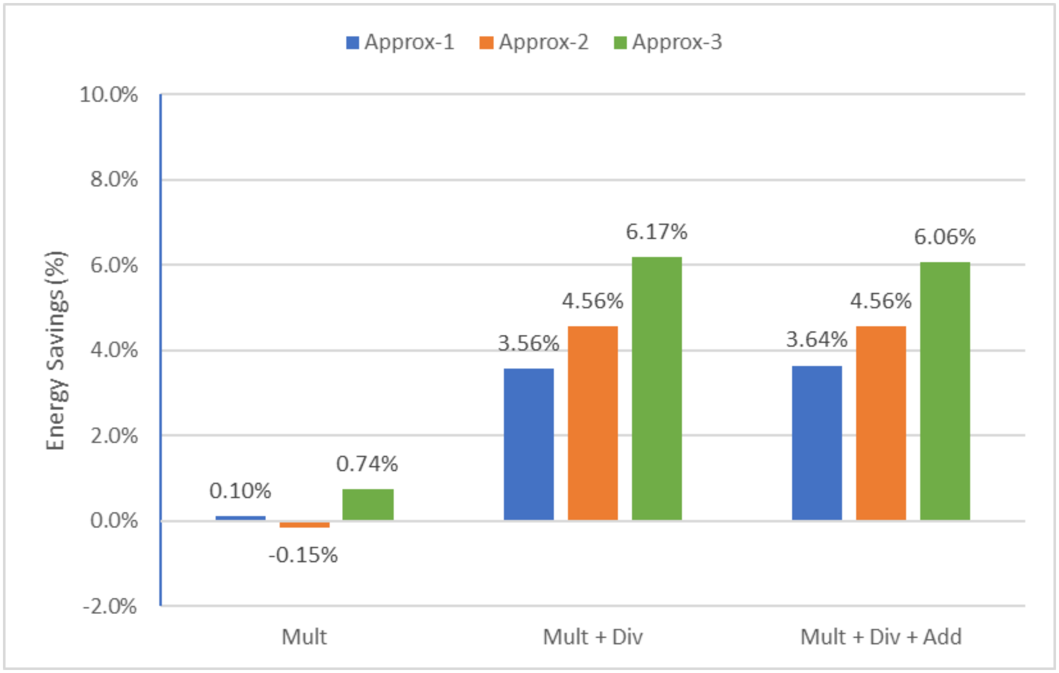


Figure 7.4 *Overall energy savings for all cases*

Several relevant pipeline processes taken from gem5 statistics are presented based on the main purpose of the study to look for energy savings in rename, scheduling, and execution stage. Therefore, the event count is reported for some processes in these stages. The evaluation compares only findings from the original with findings obtained from group 2, approx-2 case (see Table 7.1). The dynamic energy consumption of each unit of interest in comparison is shown in Table 7.2.

Table 7.1 *Energy Relevant Microarchitectural Event Counts.*

Process	**Original**	**Mult-Add (approx-2)**
FP Rename Lookup	73396095	67941158
FP Instruction Wakeup	55672934	50654257
FP ALU Access	55675056	52550295
Approx. Table Read Count	-	3380181
Approx. Table Write Count	-	1301901

Table 7.2 *Dynamic Energy Consumption.*

Unit	Dynamic Energy (mJ)	
	Original	**Mult-Add (approx-2)**
Rename	91.620	91.27
Execution	407.95	393.03
Scheduling	118.65	116.17
Approximation Table	-	12.74
Processor	843.86	831.51

7.3.1 Energy in Renaming Unit

Figure 7.5 shows the total energy savings in the renaming unit for all cases. As predicted, energy savings is higher in group 2 and group 3. Next, an examination of how skipping the operand lookup for approximate instructions affect the energy consumption was made based on the findings in Table 7.2. The savings gained from skipping the operand lookup process for group 2, approx-2 case was only 0.38%.

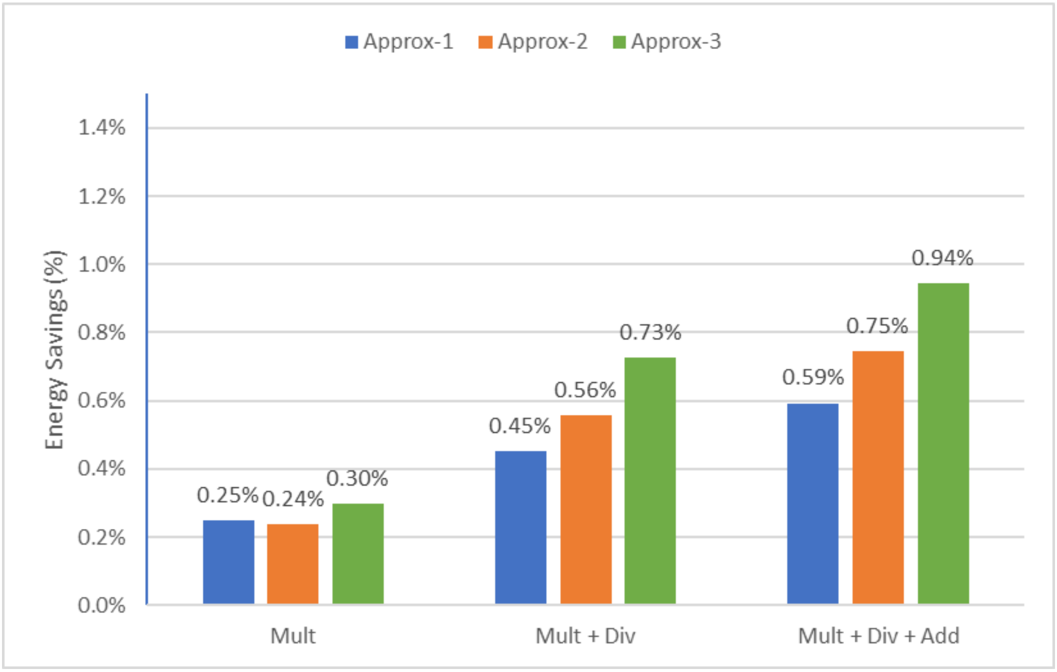


Figure 7.5 *Total energy savings in renaming unit*

7.3.2. Energy in Scheduling Unit

Figure 7.6 illustrates the total energy saving in scheduling unit for all cases. As indicated, the average energy savings in scheduling unit for group 2, approx-2 case is 2.06%. The plot in the figure represents the sum of energy that is obtained from static and dynamic power consumption.

The instruction wakeup process occurs inside the IQ, which is part of scheduling unit in the pipeline. Therefore, the dynamic energy consumption for this unit was examined. Based on data in both Table 7.1 and Table 7.2, a smaller number of FP Instruction Wakeup process has contributed to a dynamic energy savings of 2%, for group 2, approx.-2 case.

7.3.3 Energy in Execution Unit

Figure 7.7 shows the energy savings in execution unit for all cases and based on the percentage, most of the energy savings came from this stage. Approximate instructions were not issued to the functional unit, which result in less number of functional unit used during

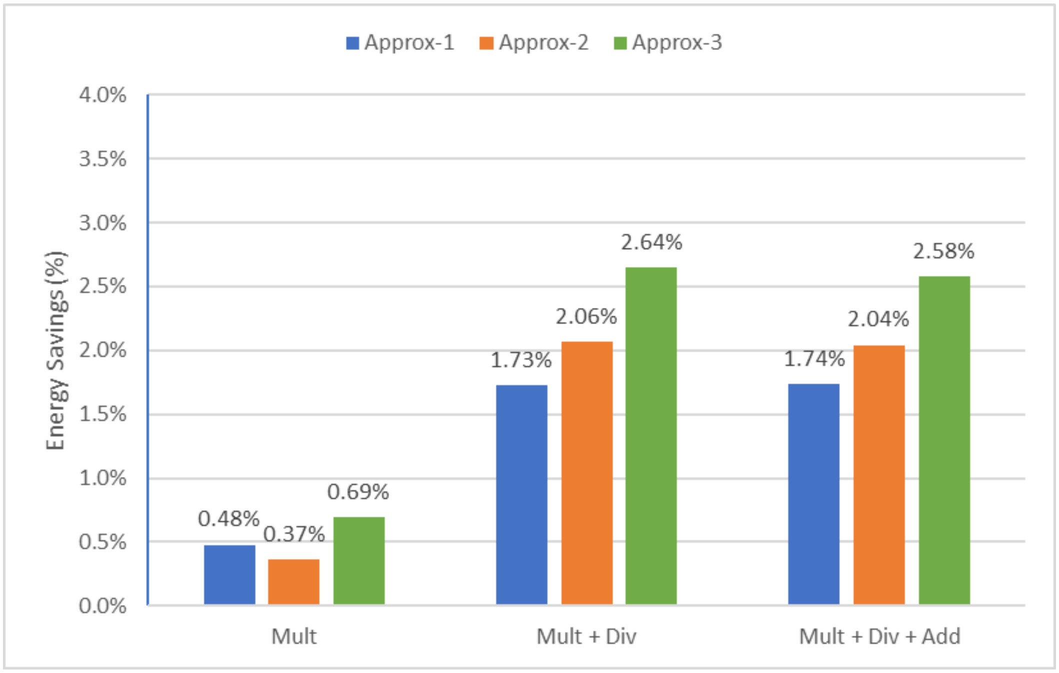


Figure 7.6 *Total energy savings in scheduling unit*

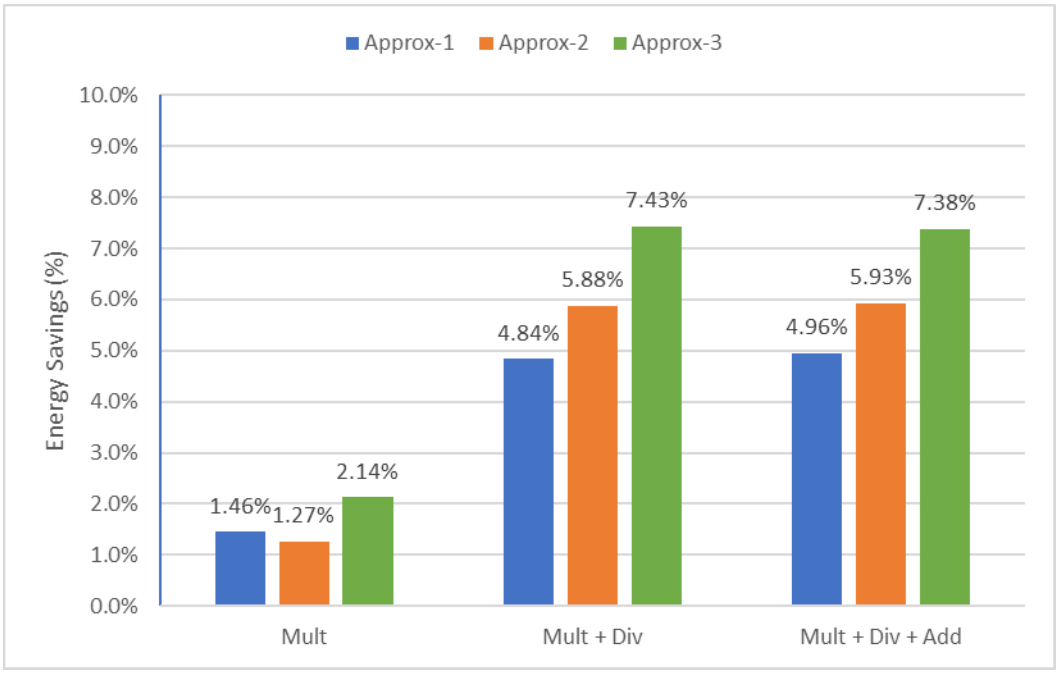


Figure 7.7 *Total energy savings in execution unit*

runtime. The total energy saving from execution unit is around 5.9%, for group 2, approx.-2 case.

7.3.4 Approximation Table Overhead

As described in Section 5.4, a simple approximation table was developed with a goal of not introducing too much overhead to the pipeline. In this experiment, only up to three instructions were selected in total for approximation. Therefore, there was no need for a table with large entries. Thus, there was not a need to use the entire PC bits for tags; group bits are enough for this purpose (see Section 5.2). As shown in Table 7.2, the energy used in the approximation table was low if compared to the amount of energy used in other units. It was also revealed that, for all cases, the energy used (static + dynamic) in the approximation table was in the range of 13.5 – 13.7 mJ. Therefore, the introduction of the approximate table does not contribute to more energy usage, except for the case of group-1, approx.-1, where the total energy spent is slightly higher than the original simulation.

7.4 Discussion

A gain in performance is straight forward; skipping the long latency computation saves lots of cycles. Nevertheless, in this case, because the outcome focused on energy consumption, relaxing the dependency between two dependent instructions enabled tweaking a couple of pipeline processes that could intuitively reduce the energy spent in the whole run.

This research was specifically focused on skipping the operand lookup in the rename stage and omitting the instruction wake up in the scheduler to determine if it would contribute to energy savings. The results revealed that not much energy can be saved from these two methods dynamically. This is because the number of reductions in these two processes is small. The reduction needs be much larger to determine if more savings can be derived from these methods. To achieve this purpose, one would need to include and add more instructions for approximation. Approximating more instructions saves energy; nevertheless, the main argument of approximate computing is: How safe it is to approximate those instructions; and, if they are safe, how many errors will be introduced at the output? Like other approximate computing research, this trade-off analysis needs to be addressed first, as well as how the outcome varies between applications.

For this application, one could determine the instruction group and approximation degree that will provide good resulting values in all aspects. From the error vs. energy plot shown in Figure 7.8, approximating multiplication and division instructions with approximation degree of 1 and 2, can yield a good tradeoff between energy and error.

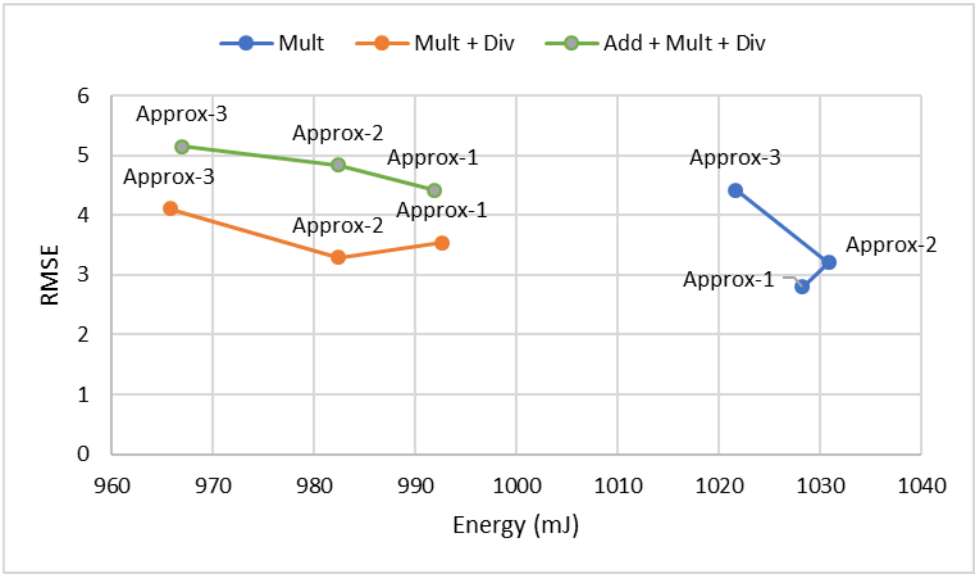


Figure 7.8 *Error vs. energy plot for all cases*

CHAPTER 8. CONCLUSION AND FUTURE WORK

Approximate computing is a widespread technique used to increase the performance of computer system and reduce the energy spent in the pipeline. The technique utilizes less resources as well as less computation time; however, as a tradeoff the user would expect to see a loss of quality in the output. Approximate computing can be applied in areas that can tolerate a percentage of loss and errors. An example of this can be found in the DSP domain. Much application under DSP, such as image and video processing, speech recognition, scientific data computing, etc., require a great deal of computation and resources due to the need for a huge amount of computer data as inputs.

This thesis research introduced another microarchitectural approach in approximate computing. This work demonstrated the effect of these approaches on the performance and energy consumption of a computer. A value approximation technique was implemented by scheduling dependent instructions without waiting for their operand values computed from their producers. In this work, the best result was obtained at 1.25x speedup and total energy savings of 4.6%, with an acceptable error at the final output.

Apart from energy savings that resulted from less usage of functional units, additional energy savings techniques were applied in the pipeline. During approximation of an instruction, the operand renaming lookup process was skipped during register renaming, as well as disabling instruction wake up process in the IQ. An approximation with acceptable image output revealed a minimal dynamic energy saving of approximately 0.4% for operand lookup, and approximately 2% for the instruction wakeup process. The low energy saving is expected because only a very small number of instructions was approximated, which led to very small reduction in related microarchitectural events. Since this study was still in rudimentary stage, future work should include the following considerations:

- An approximation with more numbers of instructions. The number of operand renaming lookup and instruction wakeup processes can be reduced proportionally with higher number of instructions for approximations.
- Experimenting with different benchmarks. One might want to determine if these techniques work across application from different domain. Applying these techniques to different application might also provide more meaningful finding s and results.
- Selecting multiple instructions with longer dependency chain. As described in Section 3.2, speculation of instructions in long data path would enable instructions to commit faster. If possible, the instruction at the beginning of the dependency chain until the one before final consumer can be totally approximated or omitted from execution. This could become another potential research area in the future.

REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous and A. R. LeBlanc, “Design of ion-implanted MOSFET's with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256-268, Oct. 1974.

[2] G. E. Moore, “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.,” *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33-35, Sept. 2006.

[3] R. M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,” *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25-33, Jan. 1967.

[4] J. E. Smith, “A study of branch prediction strategies,” *ISCA*, 1981, pp. 135–148.

[5] M. B. Taylor, “Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse,” *DAC Design Automation Conference 2012*, San Francisco, CA, 2012, pp. 1131-1136.

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

[7] H. Esmaeilzadeh, A. Sampson, M. Ringenburg, L. Ceze, D. Grossman, and D. Burger, “Addressing dark silicon challenges with disciplined approximate computing,” *Dark Silicon Workshop (DaSi),* 2012.

[8] Q. Xu, T. Mytkowicz and N. S. Kim, “Approximate Computing: A Survey,” *IEEE Design & Test*, vol. 33, no. 1, pp. 8-22, Feb. 2016.

[9] S. Mittal, “A survey of techniques for approximate computing,” *ACM Computing Survey.,* vol. 48, no. 4, p. 62, 2016.

[10] T. Moreau *et al*., “A Taxonomy of General Purpose Approximate Computing Techniques,” *IEEE Embedded Systems Letters*, vol. 10, no. 1, pp. 2-5, March 2018.

[11] J. Han and M. Orshansky, “Approximate computing: An emerging paradigm for energy-efficient design,” *2013 18th IEEE European Test Symposium (ETS)*, Avignon, 2013, pp. 1-6.

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Architecture support for disciplined approximate programming,” *ASPLOS*, 2012.

[13] J. Y. F. Tong, D. Nagle and R. A. Rutenbar, “Reducing power by optimizing the necessary precision/range of floating-point arithmetic,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 273-286, June 2000.

[14] M. H. Lipasti, and J. P. Shen, "Exceeding the dataflow limit via value prediction," *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, Paris, France, 1996, pp. 226-237.

[15] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel, "Invited: Cross-layer approximate computing: From logic to architectures," *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Austin, TX, 2016, pp. 1-6.

[16] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Computing approximately, and efficiently," *2015 Design, Automation & Test in Europe Conference & Exhibition*, Grenoble, 2015, pp. 748-751.

[17] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," *ESEC/FSE*, 2011.

[18] A. Sampson, A. Baixo, B., Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A programmer-guided compiler framework for practical approximate computing," U. Washington, Tech. Rep. UW-CSE-15-01-01, 2015.

[19] J. San Miguel, M. Badr, and N. Enright Jerger, "Load value approximation," *MICRO*, 2014.

[20] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," *Peoc. PLDI*, 2011, pp. 164-174.

[21] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving refresh-power in mobile devices through critical data partitioning," *ASPLOS*, 2011.

[22] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *ISCA*, 2002.

[23] V. K. Chippa, S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing: An integrated hardware approach," *2013 Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, 2013, pp. 111-117.

[24] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922-927, July 2005.

[25] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-Power Digital Signal Processing Using Approximate Adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124-137, Jan. 2013.

[26] A. Yazdanbakhsh, D. Mahajan, P. Lotfi-Kamran, and H. Esmaeilzadeh, "AXBENCH: A Multi-Platform Benchmark Suite for Approximate Computing," *IEEE Design and Test*, special issue on Computing in the Dark Silicon Era 2016.

[27] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value locality and loadvalue prediction,” *17th International Conference on Architectural Support for Programming Language and operating Systems*, pp. 138-147, October 1996.

[28] B. Thwaites *et al*., “Rollback-free value prediction with approximate loads,” *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Edmonton, AB, 2014, pp. 493-494.

[29] S. Li, J. Ahn, J. B. Brockman, and N. P. Jouppi, “McPAT 1.0: An Integrated Power, Area, and Timing Modeling Framework for Multicore Architecture,” HP Labs, Tech. Rep. HPL-2009-206.

[30] R. E. Kessler, “The Alpha 21264 Microprocessor,” *IEEE Micro*, vol. 19, no. 2, 1999.

[31] A. Sodani and G. S. Sohi, “Dynamic instruction reuse,” *Proceedings of the 24th International Symposium on Computer Architecture*, pp.194-205, June 1997.

[32] M. Azam, P. Franzon and W. Liu, “Low power data processing by elimination of redundant computations,” *Proceedings of 1997 International Symposium on Low Power Electronics and Design*, Monterey, CA, USA, 1997, pp. 259-264.

[33] Y. Sazeides and J. E. Smith, “The predictability of data values,” *Proceedings of 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, NC, USA, 1997, pp. 248-258.

[34] B. Calder, G. Reinman, and D. M. Tullsen, “Selective value prediction,” *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, Atlanta, GA, USA, 1999, pp. 64-74.

[35] F. Gabbay, and A. Mendelson, “Speculative execution based on value prediction,” Technical Report 1080, Technion-Israel Institute of Technology, EE Dept., Nov 1996.

[36] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, “Flexjava: Language support for safe and modular approximate programming,” *Proc. 23rd Symp. Found. Softw. Eng*., 2015, pp. 745–757.

[37] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The GEM5 Simulator,” *SIGARCH Computer Architecture News*, vol. 39, no. 2, May 2011.

[38] S. Gupta, and S. G. Mazumdar, “Sobel Edge Detection Algorithm,” *International Journal of Computer Science and Management Research*, vol. 2, no. 2, Feb. 2013.