8-2017

A Study on the Impact of Instruction Set Architectures on Processor's Performance

Ayaz Akram
Western Michigan University, aakahlown@gmail.com

A Study on the Impact of Instruction Set Architectures on Processor's Performance

by

Ayaz Akram

A thesis submitted to the Graduate College
in partial fulfillment of the requirements
for the degree ofMaster of Science in Emgineering
Electrical and Computer Engineering
Western Michigan University
August 2017

Thesis Commitee:

Dr. Lina Sawalha, Chair
Dr. Janos Grantner
Dr. Steven Carr

A Study on the Impact of Instruction Set Architectures on Processor's Performance

Ayaz Akram, M.S.E.

Western Michigan University, 2017

The recent advances in different instruction set architectures (ISAs) and the way those ISAs are implemented have revived the debate on the role of ISAs in overall performance of a processor. Many people in the computer architecture community believe that with current compiler and microarchitecture advances, the choice of ISA does not remain a decisive matter anymore. On the other hand, some researchers believe that this is not the case and they claim that ISAs can still play a significant role in the overall performance of a computer system. Novel heterogeneous architectures exploiting the diversity of different ISAs have been already introduced. This thesis evaluates applications' behavior compiled for different RISC (Reduced Instruction Set Computers) and CISC (Complex Instruction Set Computers) ISAs using various microarchitectures. We correlated performance differences of same applications across ISAs to certain ISA features. This work shows that ISAs can affect the overall performance of applications differently based on their inherent characteristics.

Acknowledgments

I would like to thank my advisor, Dr. Lina Sawalha for her help, support and guidance to conduct this research. She has been a continuous source of motivation during the period of my Masters' studies. I have learned a lot from her and will always stay grateful to her. I am also thankful to other thesis committee members, Dr. Janos Grantner and Dr. Steven Carr, for serving on my thesis committee. I also acknowledge the help of Brandon Arrendondo and Tyler Bayne to perform this study.

It would not have been possible for me to accomplish anything without the support and love of my parents. I wish to show my gratitude to them for all their sacrifices and help to achieve my goals. I am also thankful to my siblings for their encouragment and support.

Finally, I would like to thank all of my friends at WMU. Discussing various technical and non-technical issues with them has provided me with new insights and exposure to other fields. I wish to say thanks to them for their help.

Ayaz Akram

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Instruction set architecture (ISA) serves as an abstraction layer between hardware and software layers of a computer system. An ISA is considered to be analogous to human language [1]; it is the language of a processor. An ISA defines the instructions available to the programmer. Other components of an ISA usually include registers, addressing modes, data-types, memory management, devices and exception handling, power management and multi-threading support [2]. The actual implementation of an ISA on hardware is known as Microarchitecture. A single ISA can be implemented in different ways, and thus can have various microarchitectures. Most modern ISAs can be classified into two classes: RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer). This thesis explores the impact of an ISA on application's performance using a specific microarchitecture. We chose to study and compare three different ISAs: 64-bit ARM (ARM-v8), x86-64 and Alpha. This comparative analysis is done using six different microarchitectures for selected ISAs.

1.1 Evolution of RISC and CISC ISAs

In the earlier days of computers, physical wires were used to specify the operation of a CPU. Maurice Wilkes proposed the idea of "microprogramming" in 1951 [3]. He proposed that the CPU could have a general design by specifying it's control unit in a program store referred as microcode. This microcode was changeable without any hardware changes. Moreover, it specified the instruction set available to the programmer of a particular CPU [4]. This

was a promising idea to reduce the development costs and provide more flexibilty to computer designers. IBM's System/360 family of computers was the first commercial family of computers having CPU with control units in microcode (firmware) [4]. Individual models of the System/360 series were compatible due to System/360 firmware but had differences in performance and pricing [4, 5]. Commercial success of System/360 established firmware as the future direction of computer architecture. The System/360 family of machines was a commercial success but lagged behind in performance compared to the fastest computers at that time [4]. The firmware layer that intervened betweem hardware and program was responsible for diminishing speed in comparison to hard wired CPUs [4]. Later, John Cocke proposed the idea of "Reduced Instruction Set Computer" (RISC) to make IBM firmware better in terms of performance [6]. Cocke, while analyzing the compiled code for IBM's System/370 machines, noticed that machine compilers do not always select the best sequence of instructions to perform a certain task. He observed that the main reason for compiler's inability to come up with the best sequence of instructions was that the instruction set available to the compilers was too "rich" [4]. Cocke proposed the idea of reducing the available instructions to "a set of primitives carefully chosen to exploit the fastest component of the storage hierarchy and provide instructions that can be generated easily by compilers" [4, 7]. Less number of choices for compilers could lead to generation of more efficient code. After the introduction of RISC, non-RISC architectures got the name of CISC (Complex Instruction Set Computer) [4]. This eventually led to an unending debate that which one of the two classes of instruction sets is better. Table 1.1 shows the distinguishing features that characterized the CISC and RISC ISAs.

Processor technology has been in continuous evolution and has changed significantly since the introduction of RISC ISAs. The microarchitectures (the implementation of any ISA on hardware) are continuously being optimized. As a result, the lines between RISC and CISC ISAs are blurring. The two types of ISAs have adopted various features of each other. CISC architectures like x86 started to decode complex instructions into simpler RISC like instructions called microoperations (μ-ops), to make pipelining more feasible. As Moore's law [8] continues to hold [9], more transistors can fit in a single chip, giving RISC architectures the opportunity to incorporate more complicated CISC like instructions. Some people view the

present time as “post-RISC” era, implying that the current architectures are neither fundamentally CISC nor RISC [10].

Table 1.1: Features of CISC and RISC ISAs

CISC	**RISC**
Complex instructions	Simple/reduced instructions
Emphasis on hardware	Emphasis on software
Can incorporate load and store in other instructions	Load and store are independent instructions
Smaller static code size	Larger static code size
Variable length instructions	Fixed length instructions
Higher number of addressing modes	Limited number of addressing modes
Complex encoding of instructions	Simple encoding of instructions
Usually have specialized instrucions	Avoid having specialized instructions
Limited number of general purpose registers	Large number of general purpose registers
Examples: x86, VAX, Z80	Examples: ARM, Alpha, MIPS, SPARC, PowerPC

1.2 Overview of Instruction Sets Under Analysis

This section provides an overview of the evolution and features of the ISAs chosen for this study: x86-64, ARMv8 and Alpha.

1.2.1 x86-64

x86-64 instruction set is a 64-bit version of the x86 instruction set (CISC ISA), which is largely used in desktop and server applications. x86 has been used by many processor vendors like Intel, AMD, Cyrix and VIA. x86 traces back its origin to Intel's 8086 CPU [11, 12]. Intel 8086 microprocessor was a 16-bit extenstion of Intel 8080 microprocessor and was followed by 80186 and 80286. 80386 (32-bit version) was introduced in 1985 [12] and was followed by 80486 processor. Originally, the 64-bit version of x86 ISA was introduced by AMD [13], but it is currently used by both AMD and Intel.

There have been many additions and extensions to the ISA over time with backward compatibility. x86 had grown a lot in terms of number of supported instructions. 8086 had a support of around 400 instructions, while the number had increased to around 1300 instructions by the time Intel Haswell was introduced [14]. As of today, majority of personal computers are based on x86. According to PassMarks quarterly market share report, Intel and AMD's combined share of CPU market was close to 100% [15] in first quarter of 2017. x86 also holds a huge share in server and high performance computing domain. Approximately 90% of supercomputers in TOP500 supercomputers list [16] were based on x86-64 in 2015 [17].

Following are some of the features of x86-64:

- Variable length complex instructions ranging from 1 byte to 15 bytes in size.
- Higher code density than its RISC counterparts resulting in lower static code size.
- Decoding of x86 instructions into simpler operations at run time called microoperations (μ-ops).
- Single Instruction Multiple Data (SIMD) support through SSE/AVX extensions [18, 19].

- Sixteen 64-bit registers for integer operations and sixteen 128-bit registers for floating-point and SIMD operations.
- Support of absolute memory addressing, sub-register addressing and register-to-register spills, which normally leads to lower register pressure [20].
- Use of implicit operands for various instructions.

The decoding of x86 instructions into μ-ops provides higher opportunity for instruction-level parallelism (ILP) and has become necessary due to high ILP demands of modern, deep pipelined microarchitectures. An example of use of implicit operands in some x86 instructions is a multiply instruction (MUL). In MUL instruction, the destination operand is an implied operand located in register AL, AX or EAX. The use of implicit operands results into extra dependencies in some cases, results into negative impact on available parallelism [21]. Most of the x86 instructions are two-operand instructions which over-write one of the source registers with the result of the operation. If certain register values need to stay alive across instructions they should be first copied to other registers [22]. These copying operations create unnecessary dependencies in the code. A brief guide to main x86-64 instructions can be found in [23].

1.2.2 ARMv8

The 64-bit variant of ARM architecture (ARMv8) targets low power server market, along with embedded systems. Original ARM architecture was a 32-bit RISC architecture having 16 registers [24]. Later on, a condensed ISA extension named Thumb was added to ARM (ARMv4T). The latest evolved 32-bit version of ARM was ARMv7. Introduced in 2011, ARMv8 is a redesigned ISA when compared to ARMv7. Several features of ARMv7 like predicated instructions, load-multiple and store multiple instructions were removed. ARMv4 had support for 300 instructions [14]. The number had grown to more than a thousand in ARMv8 [22]. Overall ARMv8 is complex and has many instruction formats.

ARM is a major contributor in embedded low power systems. For example, ARM's market share by volume was 90% in mobile application processors (smartphones, tablets), 30% in embeded intelligence (microcontrollers, smartcards) and 45% in other mobile chips (modems,

sensors, GPS) in 2016 [25]. Even though ARM server chips are in market for more than five years now, they have not been able to make any significant server market share so far [26].

Below are some of the features of ARMv8:

- Fixed-size 32-bit long instructions.
- Support of eight different addressing modes, but still a load/store architecture, i.e. instruction operands cannot be values residing in memory.
- Thirty one 64-bit general purpose registers.
- No support for compact Thumb instruction encoding.
- SIMD support through NEON extensions [27].
- Addition of an integer division instruction unlike ARMv7.

NEON is mandatory for ARMv8, and software floating-point ABI (Application Binary Interface) is not provided. ARMv8 is compact but does not compete well in code size with ISAs that have variable-length instructions [22]. ARMv8 instructions can boost performance by 15% to 20% in comparison to ARMv7 instructions [28]. A short guide to important ARMv8 instructions can be found in [29].

1.2.3 Alpha

Designed by Digital Equipment Corporation (DEC) in the early 1990s, Alpha is another 64-bit RISC ISA. Alpha was designed for high-performance systems, was simple to implement because of it's simple features and was the first 64-bit ISA [30]. Many of the least attractive features of commercial RISC ISAs like branch delay slots were ommitted to achieve simplicity [22]. The first implementation of Alpha was Alpha 21064 (EV4) and was followed by Alpha 21164 (EV5), Alpha 21264 (EV6) and Alpha 21364 (EV7). Alpha chips were able to run at much higher clock speeds compared to other chips of their time [30]. Alpha was highly optimized for in-order cores and contained some features that could hurt modern out-of-order microarchitectures [22].

Some of the features of Alpha are given below:

- Fixed-size 32-bit long instructions.
- Support of six different instruction formats.
- Support of thirty two integer and thirty two floating point 64-bit registers.
- No support for any compressed ISA extension.
- Imprecise floating-point trap model.
- Support of SIMD operations through an extension named Motion Video Instructions (MVI) [31].
- Larger code size because of it's extreme RISC approach [30].

Alpha defines that exception flags and default values if needed should be provided by software routines. It requires the insertion of trap barrier instructions after most of the floating-point arithmetic instructions. MVI is composed of simple instructions that operate on integer data types. Development of Alpha was discontinued in favor of Intel's Itanium [32], and Alpha ISA has died out now. Last implementation of Alpha was developed in 2004. A brief overview of important Alpha instructions can be found in [33].

1.3 Thesis Contributions

The main contributions of this thesis are as follows:

- Analysis and comparison of different ISAs.
- Analysis of applications' performance across different CISC and RISC ISAs for fixed microarchitectures.
- Analysis of various microarchitecture dependent and independent statistics across ISAs with the same microarchitectures.

- Correlation of performance differences across ISAs to applications' assembly code for each ISA.
- A survey of different ISA related studies.

The rest of this thesis is organized as follows: Chapter 2 surveys various ISA related studies. Chapter 3 discusses the methodology adopted to perform this ISA comparative study. Chapter 4 discusses the results and provides an analysis. Finally, Chapter 5 concludes the thesis.

Chapter 2

Related Work

Most of the ISA studies are old and do not include state-of-the art developments in ISAs and their implementations [34, 35, 36, 37]. Some other studies either focus only on one ISA [14, 38] or target only particular ISA features [39]. Two recent studies by Venkat and Tullsen [20] and Blem et al. [40], have conflicting claims regarding the role of ISA in the performance of a processor. Venkat and Tullsen [20] suggest that ISAs can affect performance significantly based on their features. On the other hand, Blem et al. [40] conclude that microarchitecture is the main reason for performance differences across different platforms and ISA effects are indistinguishable. Our work focuses on analysing different ISAs and checking the validity of such claims by adopting a different methodology than what they used.

In 1990's Bhandarkar and Clark studied different implementations of MIPS, VAX, x86 and Alpha ISAs [34, 35]. They concluded that RISC processors have a performance edge over CISC processors. CISC ISAs required more aggressive microarchitecture optimizations to overcome the performance bottleneck. Isen et al. [36] compared the performance of Power5+ to that of Intel Woodcrest, and they concluded that both architectures matched in performance. They indicated that with the aggressive microarchitectural techniques CISC ISAs can have similar performance as RISC ISAs.

A recent comparative study of ISAs performed by Blem et al. [41] provides a detailed analysis of x86 and ARM ISAs. The authors claimed that the different ISAs are optimized for different performance gains and that none of the ISAs is fundamentally more energy efficient

than the other. Blem et al. concluded that the primary reason for performance differences is microarchitecture. They used diverse hardware platforms for their study and tried to remove non-ISA effects in the study by using the same compilers and by normalizing the effect of any differences across microarchitectures. Different mobile, desktop and server applications were used to perform this study. They also found that, overall, x86 implementations consume more power than ARM implementations. However, choice of power or performance optimized core design impacts core power use more than the ISA. Later, they included MIPS ISA and some other hardware test platforms in their study [40] to conclude that their previous findings still hold true. Although, the authors have considered the microarchitecture differences across different test platforms to study ISA impact on performance and power consumption, it is hard to accurately isolate microarchitecture and ISA effects across different test platforms. On the other hand, our work makes use of a simulator to make sure that the microarchitectures used across different ISAs are same and only ISA effects could be studied.

Weaver and Mckee [39, 42] studied the effect of ISAs on code density. Their study included more than 20 ISAs (including x86_64, ARM64 and Alpha). They also found that code density is mostly affected by: number of registers, instruction length, hardware divisors, the existence of a zero register, number of operands, etc. Their work shows x86 to be one of the most dense ISAs. Lozano and Ito [43] added few 8-bit instructions to ARM 16-bit Thumb ISA to further increase its code density. They used innovative techniques like compression of immediate values in instructions and achieved 30% reduction in code size alongwith 10% reduction in processor's power consumption.

Duran and Rico used graph theory techniques to quantify the impact of ISAs on superscalar processing [44]. Rico et al. also studied the impact of x86 specifically on superscalar processing [21]. They quantitatively analyzed three sources of limitations on the maximum achievable parallelism for x86 processors: implicit operands, memory address computations and condition codes. Lopes et al. [14] analyzed x86 instruction set and proposed a way to remove repetitive/unnecessary instructions, to make space for new instructions to be added to the ISA, while still supporting legacy code. The motivation behind this work is that x86 ISA, which started with somewhere around 400 total instructions has approximately 1300 in-

structions now. This is true for other ISAs like ARM and PowerPC. The proposed recycling mechanism allowed to encode new instructions with less bits and reduced complexity of x86. They showed that 40% of x86 instructions could be emulated with less than 5% overhead. This improved the area and power consumption of the instruction decoder significantly. Ye et al. [37] characterized the performance of different x86-64 applications and compared them with 32-bit x86 applications. They showed that for integer benchmarks, 64-bit binaries perform better than 32-bit by an average amount of 7%. But this is not true for all benchmarks, as some perform slower in a 64-bit mode. They showed that the memory-intensive benchmarks, which use long and pointer data types extensively, suffer from performance degradation in a 64-bit mode.

Some of the previous studies examined different ISA extensions such as compact ISA extensions, SIMD extensions and cryptographic extensions [45, 46, 47]. Lopes et al. [45] evaluated different compact ISA extensions including Thumb2 and MicroMIPS. They also proposed SPARC-16, a 16-bit extension to SPARC processor. SPARC-16 is shown to have better compression ratio in comparison to other compact extensions. It can achieve compression ratio as low as 67% reducing cache miss rate to 9% [45]. Lee [47] provides an overview of various multimedia extensions of different ISAs for general purpose processors used to accelerate media processing (e.g. MAX, MMX, VIS). Similarly, Slingerland and Smith surveyed existing multimedia instruction sets and examined the mapping of their functionality to a set of computationally important kernels [48]. Bartolini et al. [46] analyzed various existing instruction set extensions for cryptographic applications. They reviewed the associated benefits and limitations of such extensions. Jundt et al. [49] studied ARMv8 based XGene and Intel's Sandy Bridge processors to analyze the most important architectural features that affect power and performance of high performance computing applications. Their results indicated that the CPU frontend and branch predictor affected performance the most for X-Gene (ARMv8 based) processor. On the other hand, frontend and cache had the biggest impact on Intel's Sandy Bridge (x86 based) processor. Mayank et al. [50] studied and analyzed NVIDIA Graphics Processor Unit (GPU) instruction set architectures. Mayank et al. showed that the Fermi ISA is a big improvement over Tesla ISA and has more versatile memory access modes, more complicated

ALU operations and control flow management with higher efficiency. Fermi ISA reduces the dynamic instruction count by 22.6% on average and results in 15.4% performance improvement compared to Tesla [50].

Ing and Despain [51] researched instruction sets designed for application specific needs that had a tighter integration with the underlying hardware. They outlined automatic instruction set generation for these application specific designs. This technique, named Automatic Synthesis of Instruction Set Architectures (ASIA), was able to outperform manually designed instruction sets. However, it had some limitations, for instance the need of hardware resources specifications by the designer.

DeVuyst et al. [52] developed a mechanism to migrate execution of a program to cores of different ISAs with a minimum cost. Execution migration is a hard problem as it requires transfering memory image, architecture specific program state and creating registers state. To keep the memory image consistent across all ISAs, the proposed strategy ensured consistency in data section, code section and stack for all ISAs. Migration was only allowed at specific points of equivalence, which were essentially function calls in program's binary. To achieve instantaneous migration at non-equivalence points, a binary translator was proposed and optimized mainly for migration use. They concluded that the total overhead due to migrations was less than 5% even if migrations happened at every timer interrupt. Based on this execution migration methodology, Venkat and Tullsen [20] developed a heterogeneous ISA CMP (chip multiprocessor) by exploiting the diversity of three ISAs: Thumb, x86-64 and Alpha. Their chosen heterogeneous architecture results in 21% increased performance compared to the best single-ISA heterogeneous architecture, in addition to reduced energy and energy delay product. The authors exploited the energy efficiency of ARM's Thumb ISA, the high performance of x86-64 and the simplicity of Alpha to achieve better performance and energy efficiency compared to a single-ISA heterogeneous CMP.

Barbalace et al. [53] proposed a complete software stack (including OS and compiler) to run programs on heterogeneous-ISA asymmetric multicore processors . Their compiler framework calculates the cost of migration offline (also considers information of hardware platform) and marks certain points in program binaries based on this cost, before generating multi-ISA

binaries. Morever, min-cut algorithm is used to consider program affinity for different ISAs. At marked points, the proposed framework selects the most efficient island (island is composed of multiple cores of same ISA) at run-time and migrates thread to that island. They performed some experiments using Xeon and Xeon Phi and showed performance improvement over using OpenCL or only one of Xeon or Xeon phi.

Celio et al. [54] compared RISC-V [55], a new research ISA, with ARMv7, ARMv8, IA-32 and x86-64 ISAs using SPEC-INT2006 benchmarks. They found that RISC-V (RV64G) instruction count is within 2% of the μ-ops on x86-64. The compressed version of RISC-V (RV64GC) is found to be the densest ISA out of the studied ones. Moreover, they found that effective instruction count of RISC-V can be reduced by 5.4% on average by fusing instructions (macro-ops) at run time. The authors claim that a single ISA can be acceptable for both low and high-end implementations using microarchitectural optimizations such as macro-op fusion.

Steve Terpe researched the historical "RISC vs CISC" debate in [4]. The author collected viewpoints of computer scientists Robert Garner, Peter Capek and Paul McJones on this topic. The findings suggest that the Moore's law ended the RISC vs CISC controversy. ISA being RISC or CISC or a combination of both does not matter. Instead, other technology developments (like caching, pipelining, register renaming) play more significant role towards determining the overall performance of a system. Jakob [56] argues that the ISA still matters for performance. He specifically studied AArch64 (ARM-v8) and x86-64 cases to prove his point. Cortex-A57 and A53 when run with AArch64 code can achieve 10% performance improvement over AArch32 due to less register spills and more optimized instruction set in case of AArch64. Similarly, performance improvement of 5% to 10% was observed by the move from x86-32 to x86-64 due to better register allocation and overall clearner instruction set. Jon Stokes asserts that current architectures embody a variety of design approaches, and that in this post-RISC era it is not sensible to keep the RISC and CISC division intact [57]. Instead, current platforms should be evaluated on their own merits.

Our initial work [58] with x86-64, ARMv8 and Alpha ISAs indicated that these ISAs affect performance differently depending on the microarchitecture used.

Chapter 3

Methodology

This chapter discusses the requirements for this study and the methodology adopted to fulfill those requirements.

3.1 Basic Requirements

To explore the impact of ISAs on performance, it is necessary to keep all non-ISA factors constant across ISAs for a particular run. The main requirements to perform this study were:

1. To keep the same ISA-independent microarchitectural features across all ISAs for all runs.
2. To examine a diverse set of microarchitectural configurations that are close to real implementations of the studied ISAs.
3. To keep the same compilation infrastructure when compiling benchmarks for all ISAs.
4. To study the same phases of execution across all ISAs.

3.2 Simulation Environment

We used *gem5* [59] simulator for all of our experiments. This ensured the behavior analysis of different ISAs on the same microarchitectures. *gem5*'s implementation isolates the simulated

hardware (microarchitecture) from ISAs [60], making it a good fit for our study. Another reason for choosing *gem5* is its ability to support many ISAs (including Alpha, ARM and x86-64) [61]. It is also capable of simulating a wide range of microarchitectures and configurations. Some of the modifications that were applied to the simulator to use it for this study are briefly discussed next.

3.2.1 Modifications in the Simulator

We added code to gem5 to allow it to output different microarchitectural statistics like branch mispredictions, cache misses and execution cycles for fixed instruction intervals (50,000 instructions). We also added code to calculate and output different microarchitecture independent statistics like register dependency distance (the number of instructions between the instruction that writes a register and the instruction that reads the same register later), degree of use of registers (number of instructions that consume the value of a register, once the value is written) [62].

Moreover, we observed that x86 microoperations (μ-ops) to instructions ratio resulted from *gem5* was very high. We modified the source code of the simulator to make this decoding more realistic. We relied on the information available in [63] and code of other simulators like ZSim [64] and Sniper [65] to change instruction to μ-ops decoding of various instructions. We then compared the modified gem5's μ-ops to instruction ratio with that of the real hardware to make sure that the new ratio is within 5% of that of the real hardware and what was previously found in [66, 67]. It was also observed that in case of in-order cores, called *MinorCPU* in gem5, the implemented branch predictor was not working correctly for x86. On debugging further we figured out that x86 control instructions were not triggering a call to branch prediction unit. We modified the fetch stage of *MinorCPU*, to make sure that x86 control instructions were able to use simulated branch predictor. Other changes related to specific x86 microarchitecture optimizations were added to the simulator and will be discussed in Chapter 4.

3.3 Target Microarchitectures

This work uses a diverse set of microarchitectures for this study, including three OoO (out-of-order) and three IO (in-order) cores. In-order (IO) cores execute instructions in the program order as provided by the compiler. Out-of-order (OoO) cores execute instructions out of their program order and perform dynamic scheduling of instructions based on ready instruction operands [68]. Independent instructions do not need to wait for long latency old instructions and this results in better utilization of free pipeline resources like functioanl units. The out-of-order execution of instructions is performed at the expense of complicated hardware structures like reservation stations, register renmaing stages and reorder buffers.

The simulated cores are based on Intel Haswell (OoO) [69, 70], Intel Atom (IO) [71, 70], ARM Cortex A15 (OoO) [72], ARM Cortex A8 (IO) [73], Alpha 21264 (OoO) [31] and Alpha 21164 (IO) [74]. Table 3.1 shows the detailed configurations of the selected microarchitectures. This work relies on various sources [75, 76, 77, 63, 78, 79, 80, 41, 81, 82, 83] to find the configurations of these microarchitectures. For ARMv8, *gem5* uses four destination registers (each of 32 bits) to simulate a single FP/SIMD physical register. Thus, for ARMv8 the number of physical floating point registers configured in *gem5* should be four times the actual number of physical floating point registers to be simulated as shown in Table 3.1.

Table 3.1: Target Configurations

Parameter	**Haswell**	**A15**	**Alpha21264**	**Atom**	**A8**	**Alpha21164**
Pipeline	OoO	OoO	OoO	IO	IO	IO
Core clock	3.4 GHz	2 GHz	1.2 GHz	1.6 GHz	800 MHz	500 MHz
Front end width	6 μ-ops	3 μ-ops	4 μ-ops	3 μ-ops	2 μ-ops	4 μ-ops
Back end width	8 μ-ops	7 μ-ops	4 μ-ops	3 μ-ops	2 μ-ops	4 μ-ops
Instruction queue	60 entries	48 entries	40 entries	32 entries	32 entries	32 entries
Reorder buffer	192 entries	60 entries	80 entries	N/A	N/A	N/A
Number of stages	19	15	7	13	13	7
Load/Store Queue	72/42 entries	16/16 entries	32/32 entries	5 entries	12 entries	5 entries
Physical INT/FP Registers	168/168	90/64	80/72	N/A	N/A	N/A
Cache line size	64	64	64	64	32	32
L1D-$ size	32KB	32KB	64 KB	24KB	32KB	8KB
L1D-$ associativity	8 way	2 way	2 way	6 way	4 way	1 way
Note: N/A: Not Available/Applicable, OoO:Out-Of-Order, IO: In-Order						

Parameter	Haswell	A15	Alpha21264	Atom	A8	Alpha21164
L1D-$ latency	4 cycles	4 cycles	3 cycles	3 cycles	2 cycles	3 cycles
L1I-$ size	32KB	32KB	64 KB	32KB	32KB	8KB
L1I-$ associativity	8 way	2 way	2 way	8 way	4 way	1 way
L1I-$ latency	4 cycles	2 cycles	3 cycles	2 cycles	2 cycles	2 cycles
L2-$ size	256KB	2MB	2MB	512KB	256KB	96KB
L2-$ associativity	8 way	16 way	16 way	8 way	8 way	3 way
L2-$ latency	12 cycles	20 cycles	12 cycles	12 cycles	6 cycles	10 cycles
L3-$ size	8MB	N/A	N/A	N/A	N/A	4MB
L3-$ associativity	16 way	N/A	N/A	N/A	N/A	1 way
L3-$ latency	36 cycles	N/A	N/A	N/A	N/A	10 cycles
DRAM latency	57 ns	81 ns	60 ns	85 ns	65 ns	253 ns
DRAM bandwidth	25.4 GB/s	25.4 GB/s	25.4 GB/s	25.4 GB/s	25.4 GB/s	1.6 GB/s
DRAM clock	2GHz	933MHz	933MHz	600MHz	166MHz	166MHz
ITLB entries	128	128	128	16	16	48
DTLB entries	1088	544	128	80	80	64
Note: N/A: Not Available/Applicable, OoO:Out-Of-Order, IO: In-Order						

Parameter	**Haswell**	**A15**	**Alpha21264**	**Atom**	**A8**	**Alpha21164**
Read/Write ports	2/1	1/1	1/1	1/1	1/1	1/1
Branch Predictor (Global / Local Table) sizes)	Tournament (4096/4096)	Tournament (4096/1024)	Tournament (4096/1024)	Tournament (4096/1024)	Tournament (512/512)	2-Bit Counter (2048 entries)
Branch target buffer	4096 entries	2048 entries	2048 entries	128 entries	512 entries	512 entries
Return address stack	16 entries	48 entries	32 entries	8 entries	8 entries	12 entries
Note: N/A: Not Available/Applicable, OoO:Out-Of-Order, IO: In-Order						

3.4 Benchmarks

We used C/C++ benchmarks of SPEC-CPU2006 benchmarks suite [84] and embedded benchmarks from MiBench [85] suite in this study. The benchmarks were compiled for each ISA using gnu gcc version 4.8.5 for x86 and ARM, and 4.3.5 for Alpha, instead of using any vendor-specific compiler. We built the gcc cross compilers using crosstools-ng [86] version 1.22. The same versions of libraries were used to build all of the cross-compilers. While SPEC-CPU2006 benchmarks do not contain SIMD code, the autovectorization feature of gcc can result in SIMD instructions in the compiled binary. We did not disable the autovectorization feature of the compiler. All benchmarks are compiled with -O3 optimization flag.

3.5 Mapping Simpoints Across ISAs

Embedded benchmarks were run completely for this study. On the other hand, because it is infeasible to simulate entire SPEC-CPU2006 benchmarks, five statically relevant simpoint intervals (each interval of 500 million x86 instructions approximately) [87] were chosen for these benchmarks. Using multiple intervals also allowed us to study the varying behavior of an ISA for different intervals of the program. To make sure that we were comparing different ISAs for same intervals of benchmarks, we made use of the pseudo-instruction [88] support in *gem5* simulator.

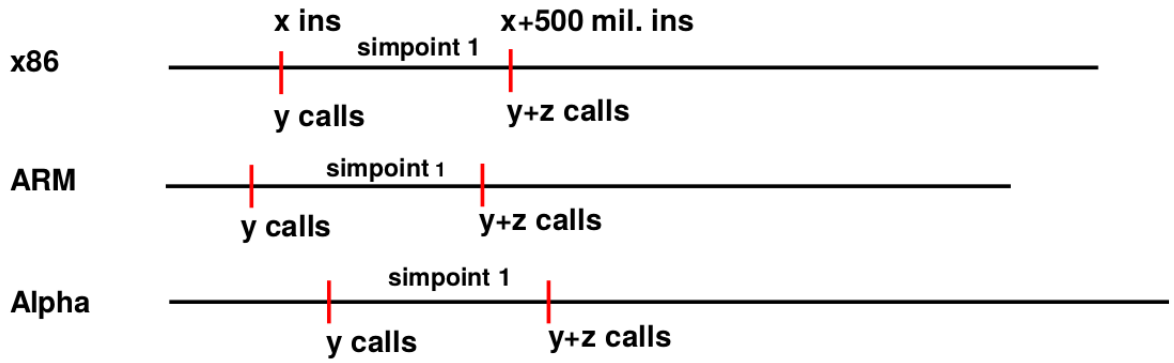

Figure 3.1: Mapping simpoints across ISAs

Next is a summary of the methodolgy for studying the same simpoint intervals across all ISAs:

1. We used simpoint tool with x86 binaries and found five simpoint intervals, each of 500 million x86 instructions. Since, total number of dynamic instructions for all ISAs can be different for a particular benchmark as shown in Figure 3.1, instruction counts cannot be used to map same simpoints across ISAs.

2. We profiled SPEC-CPU2006 benchmarks using gprof [89] tool to identify critical functions. We chose four functions for each program based on gprof output: two where the program spent the most of the time and the other two which were called the highest number of times during the execution of the program. To map x86 intervals to other ISAs, we marked critical functions of the benchmarks.

3. We inserted *gem5* pseudo instructions in all four chosen functions for each benchmark and ran the marked binaries to calculate the total number of marked-function calls at the starting and ending point for each interval for x86 binaries.

4. This call count was then used to map similar simpoint intervals on other ISAs as shown in Figure 3.1. Marking critical functions provides a better opportunity for accurate mapping as there is a higher chance that one of these functions will be executed close to the simpoint boundary.

5. We dumped different microarchitectural performance statistics such as cache misses, branch mispredictions, blocks of different stages, etc. and some microarchitecture independent statistics like register dependency distance and instruction mixes for each simpoint for each ISA over time.

6. Finally, we compared the differences in performance for various ISAs across these phases of execution.

3.6 Definitions of Studied Metrics

This section defines all of the microarchitecture dependent and independent metrics that were considered for this study. Following are the microarchitecture dependent metrics:

Cycle Counts: The number of cycles that each benchmark compiled for a particular ISA takes to execute on a particular microarchitecture. This is the primary metric used to compare performance across ISAs on the same microarchitecture.

Branch Mispredictions: *Branch mispredictions* refer to the number of times a branch predictor predicts a control instruction incorrectly. This metric helps in understanding the extra cycles that a pipeline might be spending because of following an incorrect path of execution.

L1 Instruction Cache Misses: L1 Instruction cache misses refer to the number of times an instruction block is not found in the instruction and cache has to be brought from lower levels of cache.

L1 Data Cache Misses: The number of times a cache block is not found in L1 data cache when an instruction tries to read/write from/to that particular block of memory are known as L1 data cache misses.

Last Level Cache (LLC) Misses: These misses indicate the events when data or instruction cache blocks are not found in last level cache (L2 or L3) and need to be brought from the main memory.

L1 Data Cache Accesses: This metric represents total accesses made to L1 data cache. The total number of data cache accesses can vary across microarchitectures depending on branch mispredictions and memory order violations.

Decode Blocks: This metric represents the number of times the decode stage is blocked because one of the structures in the following pipeline stages is full.

Rename Blocks: *Rename Blocks* refer to number of events when rename stage has to be blocked because either the reorder buffer and/or physical rename register file are full.

IEW Blocks: This metric represents total number of times issue, execute or write-back stages are blocked.

IQ Full: *IQ Full* referes to the total events when the instruction queue is full and new instructions cannot be issued.

LSQ Full: This metric represents the total number of events when load or store queue does not have any empty spots and new memory operations cannot be issued.

ROB Full: *ROB Full* refers to total number of times when the reorder buffer is full and the rename stage has to block because of that.

Rename Register Full: This metric shows the number of events when physical rename register file does not have vacant slots and rename stage has to be blocked.

Mem-Order Violations: *Mem-Order Violations* refer to number of times memory order is violated due to out-of-order scheduling of memory operations and pipeline needs to re-execute violated instructions.

Next is a description of microarchitecture independent metrics that we considered in this study:

Instruction Counts: These are the total number of committed instructions (macro-ops) each benchmark takes to execute.

Micro-op (μ-op) Counts: This metric counts the number of committed micro-ops (μ-ops) each benchmark takes to execute, as many complicated instructions in modern ISAs are decoded into simple operations at run-time called μ-ops.

Types of μ-ops: We considered certain types of micro-ops as well that each benchmark takes to execute. Types were divided into four categories: load operations, store operations, branch operations and others that include all operations that do not fall in the previous three categories.

Average Register Dependency Distance: Register dependency distance refers to the number of instructions between an instruction that writes a particular register and the instruction that reads the value of that register later. This metric calculates the average value of register dependency distance for each benchmark. Higher register dependency distance refers to higher available instruction level parallelism in the code that can be exploited by out-of-order pipelines to improve performance.

Average Degree of Use of Registers: Degree of use of registers refers to the number of instructions that read the value of a register once it is written. Higher degree of use of registers could mean a higher number of dependencies in the code.

Chapter 4

Results and Analysis

We considered various microarchitecture dependent and independent statistics to observe the differences across ISAs. This chapter shows the average performance of ISAs on different microarchitectures and discusses few examples of the observed differences across ISAs.

4.1 Cycle Counts and μ-architecture-Independent Statistics

Figures 4.1 and 4.2 show the cycle counts for ARM and Alpha ISAs (relative to x86) for out-of-order and in-order cores respectively. In case of SPEC-CPU2006 benchmarks, the cycle counts are cumulative cycles for all studied simpoint intervals. On average, ARM takes the least number of cycles for all types of benchmarks on out-of-order cores. As the figures show, the ISAs behave differently for different benchmarks and microarchitectures. For example, *gobmk* always takes a lower number of cycles on Alpha than x86 because x86 suffers from higher branch mispredictions. However, *perlbench* always takes lower cycles on x86 than Alpha as Alpha suffers from higher register pressure in this case. In case of *hmmer*, ARM takes a larger number of cycles than x86 on Haswell-like core, but it takes lower cycles than x86 on other out-of-order cores. In case of in-order cores, the difference in cycles taken by x86 and ARM is reduced significantly for this benchmark and on average as well. There are many cases, where the behavior of ISAs changes over different phases of execution of the same benchmark (some of them will be mentioned later in this chapter).

Figures 4.3 to 4.7 show various microarchitecture independent metrics. Figure 4.3 shows the dynamic instruction counts for Alpha and ARM ISAs relative to x86 instructions. Figure

4.4 shows the dynamic microoperation (μ-op) counts for Alpha and ARM ISAs normalized to x86 μ-op counts for all benchmarks. As the figure shows, the final number of dynamic μ-ops is ISA dependent and x86 has the most number of μ-ops in most of the cases. Figure 4.5 shows the number of different types of μ-ops for all ISAs relative to x86. The total number of μ-ops of a particular type depends on the ISA as well. Figure 4.6 shows the average values of register dependency distance (the number of instructions between the instruction that writes a register and the instruction that reads the same register). Figure 4.7 shows the average values for degree of use of registers (number of instructions that consume a value of a register, once the value is written). x86 has the highest degree of use of registers, which often blocks instruction queue in out-of-order cores.

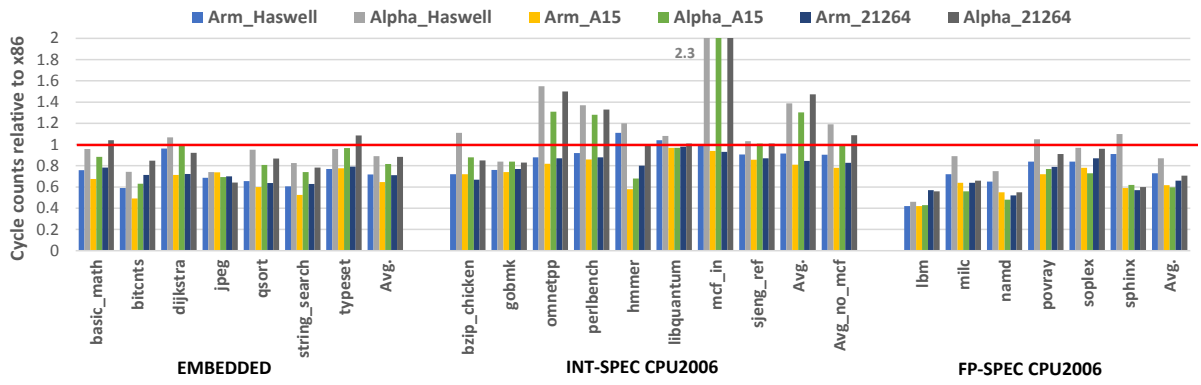

Figure 4.1: Cycle counts relative to x86 for OoO Cores

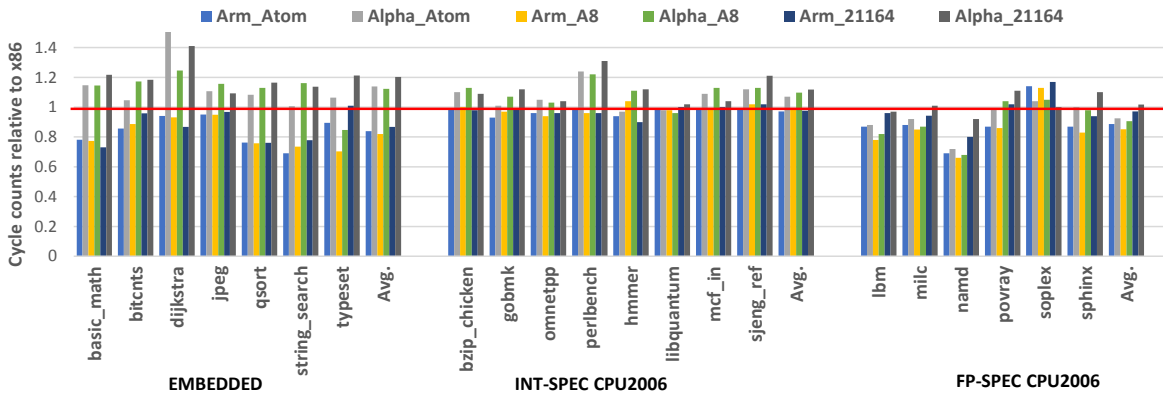

Figure 4.2: Cycle counts relative to x86 for IO Cores

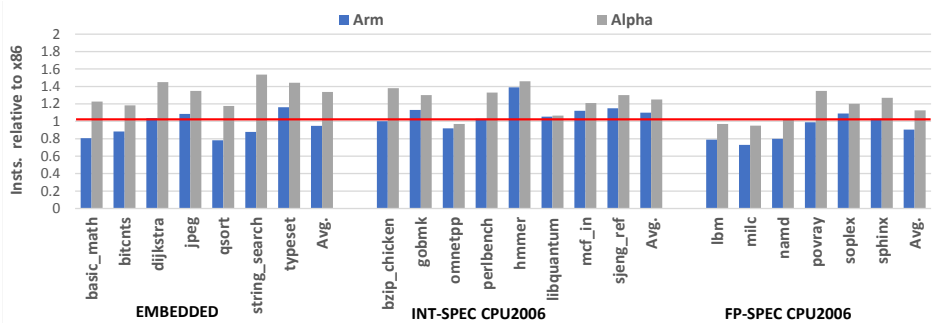

Figure 4.3: Instruction counts relative to x86

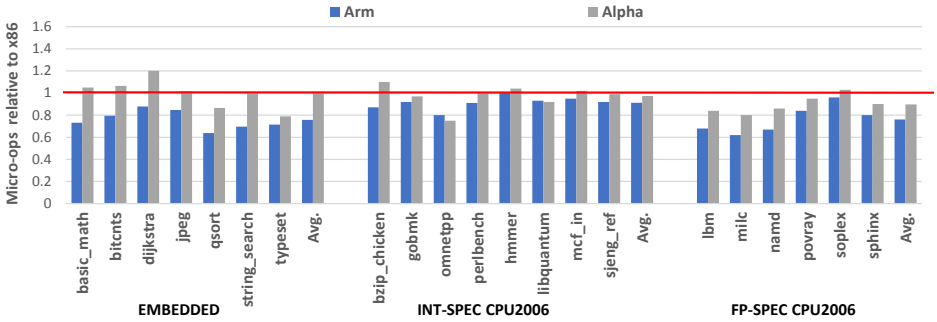

Figure 4.4: μ-op counts relative to x86

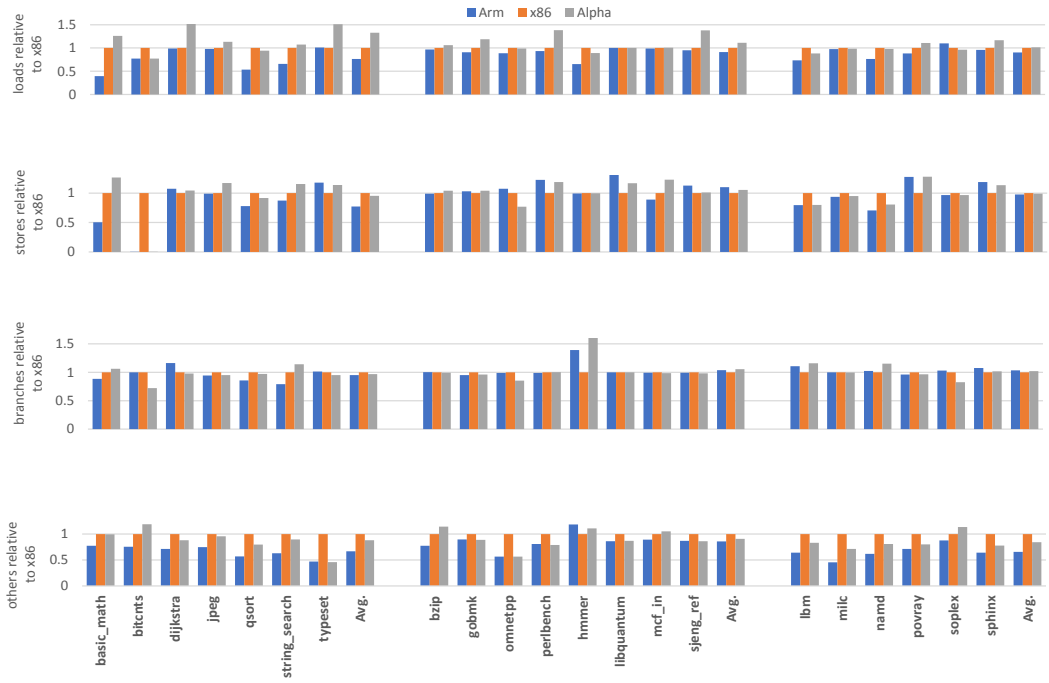

Figure 4.5: Types of μ-ops relative to x86

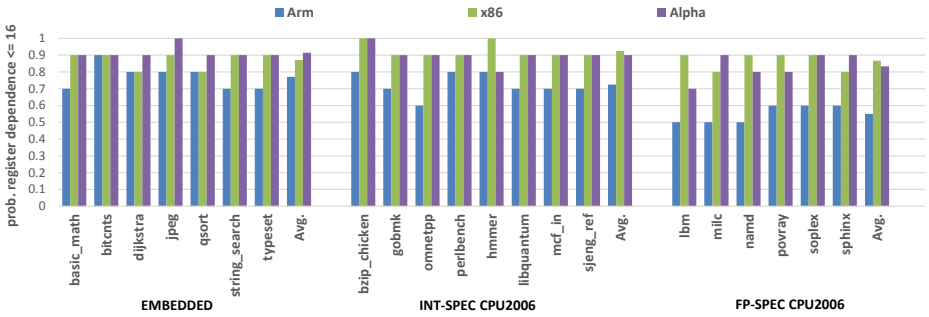

Figure 4.6: Probability of register dependency distance (<=16) for each ISA

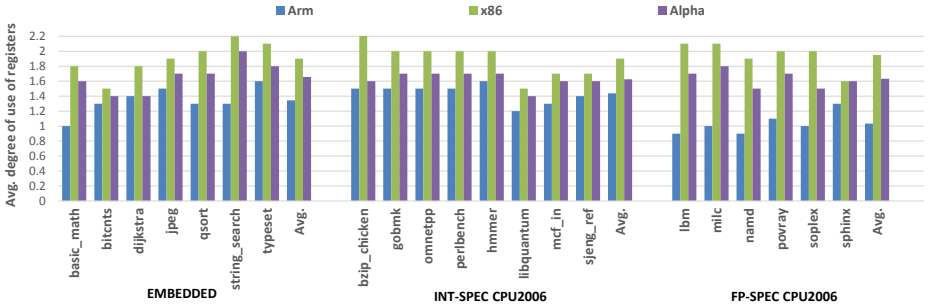

Figure 4.7: Average degree of use of registers for each ISA

4.2 Individual Benchmark Analysis

This section analyzes all studied benchmarks and shows performance differences across ISAs.

MiBench Embedded Benchmarks

basic_math:

This benchmark performs simple mathematical calculations that are usually suitable for embedded processors. It is important to note that we modified the code of this benchmark to replace '*long double*' variables with '*double*' data type in one of the functions *SolveCubic*. Listing 4.1 shows declared '*long double*' variables in this function. When operations are performed on these variables it emulates floating point operations in case of Alpha and ARM although hardware floating point is switched on in both cases during compilation. This is

because Alpha and ARM use more than 64 bits for long double data types. This leads to an increased number of instructions for Alpha and ARM (more than 5 times) as compared to x86, which results in a higher number of cycles on Alpha and ARM. After modifying the benchmark, software emulation is not used and this change also does not cause any functional inaccuracy for the benchmark on Alpha and ARM. Alpha takes the highest number of dynamic μ-ops to execute the modified benchmark.

Listing 4.1: Example from *SolveCubic* function

```
void SolveCubic(double  a,
                double  b,
                double  c,
                double  d,
                int     *solutions,
                double  *x)
{
    long double  a1 = b/a, a2 = c/a, a3 = d/a;
    long double  Q  = (a1*a1 - 3.0*a2)/9.0;
    long double  R  = (2.0*a1*a1*a1 - 9.0*a1*a2
                              + 27.0*a3)/54.0;
    double       R2_Q3 = R*R - Q*Q*Q;
```

bitcnts:

bitcnts is an embedded benchmark that counts the number of bits in an array of integers using different methods. x86 takes the most number of cycles to execute this benchmark on out-of-order cores, but on in-order cores Alpha takes the most number of cycles. Alpha takes the highest number of μ-ops to execute this benchmark as well. Another interesting observation is the behavior of this benchmark over time. Figure 4.8 shows the number of cycles taken to execute each interval of 50 000 instructions for all ISAs on Haswell-like core. As shown in the figure, for the first phase (almost until 18000 intervals or 900 million instructions), x86 takes almost the same number of cycles as taken by the other ISAs for each interval. However, for all following phases, x86 takes a higher number of cycles compared to the other ISAs. Listing 4.2 shows an example of assembly code that leads to reduced performance on aggressive out-of-order cores for x86 compared to the other ISAs. As shown in Figure 4.9, this behavior is not observed in in-order cores like Atom.

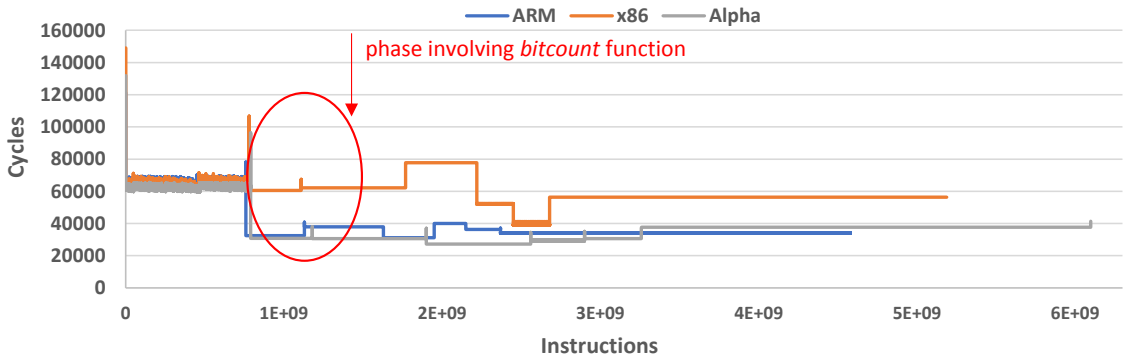

Figure 4.8: Cycles over windows of 50k insts. for *bitcnts* on Haswell core

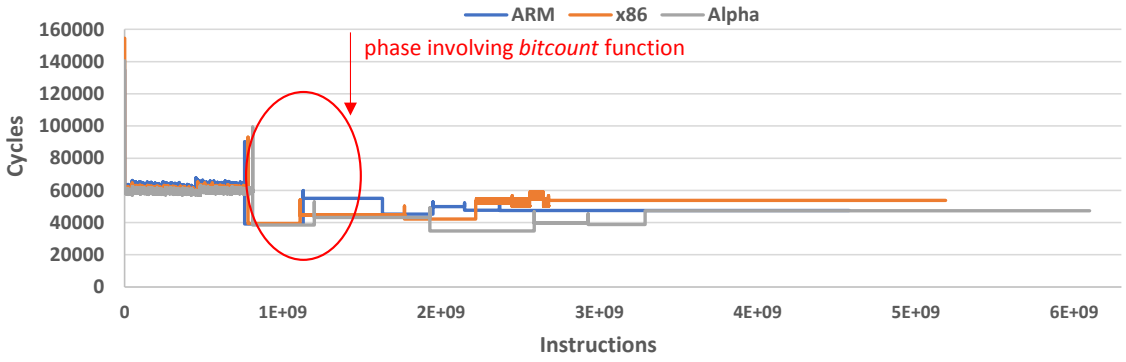

Figure 4.9: Cycles over windows of 50k insts. for *bitcnts* on Atom core

The code snippet in Listing 4.2 is taken from a function *bitcount* that executes repeatedly inside a loop. The execution of this function starts approximately after 900 million instructions; the execution phase involving this function is circled in Figure 4.8. As can be seen in Listing 4.2, there are more dependent operations in case of x86. This piece of code calculates the final value of a variable *i*. One source of more dependent operations is the first *mov* operation in x86, which copies the initial value of variable 'i' from reg %rdx to %rax. This copying is needed to perform two different 'and' operations with variable *i*, as the value of the register (%rdx) containing *i* will be modified after first 'and' operation due to nature of x86 ISA. Since, there are more dependent operations on *i* in x86, this results into congestion in instruction queue on out-of-order cores as our results indicate. It should also be noted that the C/C++ line of code in Listing 4.2 is one of the 5 similar lines of code each of which uses a previous value of *i* to calulate the new one.

Listing 4.2: Example from *bitcount* function

x86:

```
i = ((i & 0xFF00FF00L) >> 8)+(i & 0x00FF00FFL);
mov  %rdx,%rax      // 'i' in rdx, dependent operation
and  $0xff00ff,%edx    // dependent operation
and  $0xff00ff00,%eax  // dependent operation
sar  $0x8,%rax  // dependent operation
add  %rdx,%rax  // dependent operation
```

ARM:

```
i = ((i & 0xFF00FF00L) >> 8)+(i & 0x00FF00FFL);
movk x0, #0xff00, lsl #16   // independent operation
movk x3, #0xff, lsl #16     // independent operation
and  x3, x1, x3  // 'i' in x1, dependent operation
and  x0, x1, x0  // dependent operation
add  x0, x3, x0, lsr #8    // dependent operation
```

Alpha:

```
i = ((i & 0xFF00FF00L) >> 8)+(i & 0x00FF00FFL);
zapnot  t1,0xa,v0  // 'i' in t1, dependent operation
zapnot  t1,0x5,t1  // dependent operation
sra     v0,0x8,v0  // dependent operation
addq    v0,t1,v0   // dependent operation
```

Listing 4.3: Example from *ntbl_cnt* function

```
int cnt = bits[(int)(x & 0x0000000FL)];
if (0L != (x >>= 4))
```

x86:

```
mov %rdi,%rax  // 'x' in rdi, 3 dependent operations
sar $0x4,%rdi
and $0xf,%eax
```

ARM:

```
and x2, x0, #0xf  // 'x' in x0, 2 dependent operations
asr x1, x0, #4
```

Alpha:

```
sra a0,0x4,t2  // 'x' in a0, 2 dependent operations
and a0,0xf,a0
```

Another similar sort of example is shown in Listing 4.3. This code is taken from function

ntbl_cnt. The assembly instructions shown in the Listing 4.3 are responsible for performing an 'and' operation with 'x' to use the result as an index into *bits* array and shift the value of 'x' by 4 bits to the right to compare the result with 0 in the shown 'if' condition. Again, for x86 there is an extra *mov* operation needed to copy value of x from register %rdi to another register %rax.

dijkstra:

This benchmark uses *dijkstra's* algorithm to calculate the shortest path between every pair of nodes in a large graph. For *dijkstra* benchmark, Alpha suffers from high register pressure and has a greater number of load operations compared to the other ISAs as shown in Figure 4.5. This leads to lower performance on most of the microarchitectures for Alpha. One possible reason for higher register pressure on Alpha in some cases (including *dijkstra*) is less flexible addressing modes and instruction formats as compared to other ISAs.

jpeg:

jpeg is a common image compression and decompression benchmark. x86 takes the most number of cycles in all OoO cores, while Alpha takes the most in all IO cores. One interesting thing to note, for this benchmark is that for one phase of exectuion (700-900th window, where each window is 50 000 instructions), the number of memory reads and μops/instruction ratio for x86 is increased significantly.

qsort:

qsort sorts a large array of strings using *quicksort* algorithm. We modified *qsort* to remove all printings to focus on only sorting related code. x86 takes the most number of cycles on OoO cores, while Alpha takes the most number of cycles on IO cores, mainly due to higher μ-op and instruction counts for Alpha. Figure 4.10 shows that there are two major phases of execution for this benchmark. As pointed in the figure, all ISAs exhibit similar performance

for the first phase. However, in the second phase, x86 takes the highest number of cycles for each interval of instructions. In this phase, x86 exhibits higher number of instruction queue full (IQ Full) events compared to other ISAs for each interval of instructions. Moreover, this behavior is not observed in IO cores like Atom as shown in Figure 4.11. Although there are few occasional spikes but largely the number of cycles taken by all ISAs for each interval overlap.

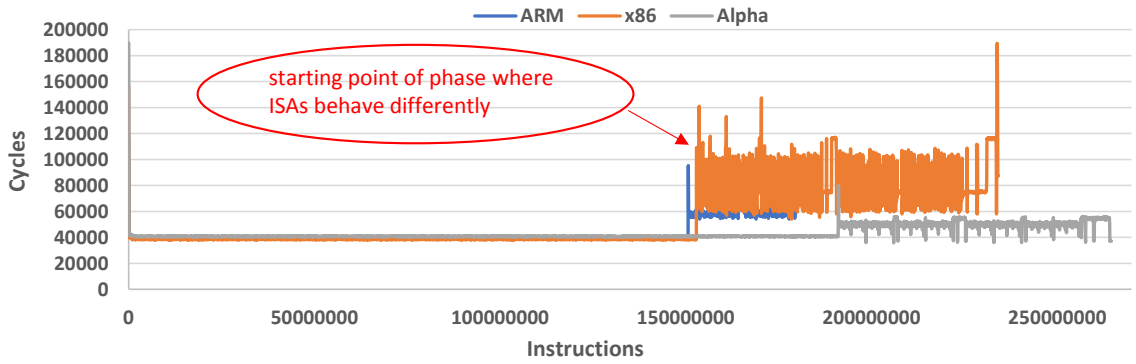

Figure 4.10: Cycles over windows of 50k insts. for *qsort* on Haswell core

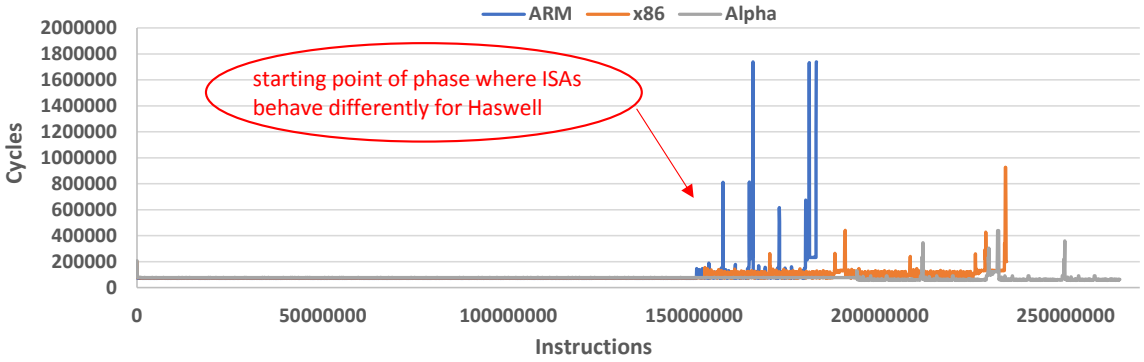

Figure 4.11: Cycles over windows of 50k insts. for *qsort* on Atom core

string_search:

This benchmark searches for given words in phrases using a comparison algorithm. x86 takes the most number of cycles to execute this benchmark, but the total number of instructions for this benchmark is pretty low (less than 5 million x86 instructions).

SPEC-CPU2006 INTEGER BENCHMARKS

bzip2:

Listing 4.4: Example from *mainGtU* function

Used Index Variables:

UInt32 i1, UInt32 i2 //32-bit unsigned

C Code:

```
c1 = block[i1]; c2 = block[i2];
if (c1 != c2) return (c1 > c2);
```

x86:

```
lea     0x1(%r12),%eax
lea     0x1(%rbp),%edx
movzbl  (%rbx,%rax,1),%eax
cmp     %al,(%rbx,%rdx,1)
jne     402987 <mainGtU+0x37>
```

ARM:

```
add     w6, w19, #0x1
add     w0, w1, #0x1
ldrb    w6, [x2,x6]
ldrb    w0, [x2,x0]
cmp     w6, w0
b.ne    4029a0 <mainGtU+0x50>
```

Alpha:

```
zapnot  a0,0xf,t0     // extra inst. to clear upper 32 bits
zapnot  a1,0xf,t1     // extra inst. to clear upper 32 bits
addq    s2,t0,t0
addq    s2,t1,t1
ldbu    t3,0(t0)
ldbu    t2,0(t1)
cmpeq   t3,t2,t0
cmpult  t2,t3,v0
beq     t0,120001b28 <mainGtU+0x78>
```

bzip2 is a SPEC-CPU2006 integer benchmark, which performs compression and decompression of an input file [90]. While Alpha takes the most number of cycles on Haswell-like microarchitecture and less on the other out-of-order cores on average, this behavior is not true

for all simpoint intervals. For example, in one of the simpoint intervals (interval 3) Alpha always performs worse on all OoO cores. There are two important functions in this benchmark; *mainGtU* and *mainSort*. They both use unsigned 32-bit integers to access arrays. In case of Alpha, extra instructions are required to clear the upper 32 bits of the index variables as pointed out in Listing 4.4. This interval, in which Alpha performs the worst, uses these functions frequently. ARMv8 and x86-64 have addressing modes that can only read/write parts of a 64-bit register. So they do not need any clear operations. The same behavior was also observed by Celio et al. in [54]. Another interesting thing to observe is the behavior of x86 within the simpoint interval number 2. As shown in Figure 4.12, in the circled phase x86 takes a higher number of cycles compared to other ISAs. The observed reasons are higher number of μ-ops per instruction and memory accesses for this phase for x86. Similar behavior is observed in simpoint interval 5 as well.

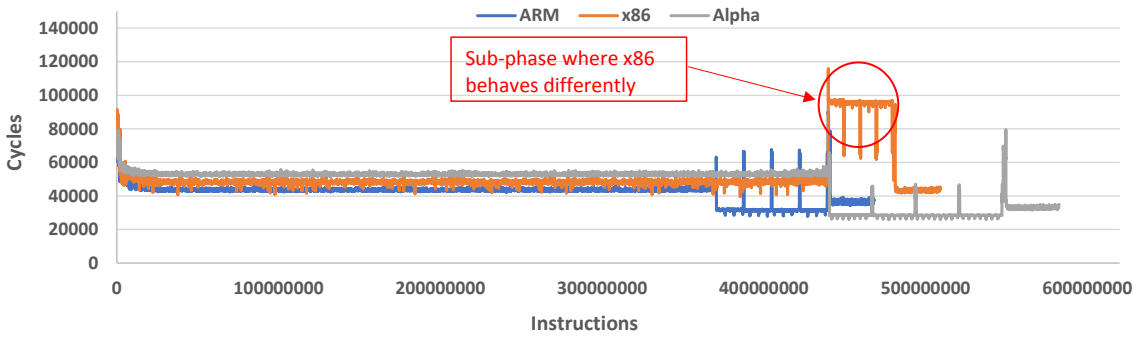

Figure 4.12: Cycles over windows of 50k insts. for *bzip2* simpoint 2 on Haswell core

gobmk:

This program analyzes different moves on a Go board. In case of *gobmk*, x86 suffers from high number of branch mispredictions in all out-of-order cores and thus takes a higher number of cycles in comparison to the other ISAs. One of the possible reasons for higher number of branch mispredictions is higher number of branch operations for x86 in comparison to other ISAs, as shown in Figure 5. It should also be noted that the fraction of branch operations out of the total operations is very high for this benchmark (more than 20%). Interestingly Alpha and x86 take the same number of cycles on one of the 5 studied simpoints, even in case of out-of-order cores. On this particular simpoint Alpha takes almost 12% more dynamic μ-ops

than x86.

omnetpp:

This benchmark simulates a large Ethernet network. Although it is an integer benchmark, it includes many floating-point operations. Alpha takes the most number of cycles for all simpoints. Alpha suffers from much higher number of memory order violations in case of out of order cores, resulting into more dcache accesses. One interesting thing to observe is the behavior of Alpha during one of the 5 simpoint inervals. As shown in Figure 4.13, there are clearly two phases of execution within this simpoint. Alpha performs similar to other ISAs on Phase 2, but takes a higher number of cycles on Phase 1 because of the increased number of memory accesses in case of Alpha compared to the other ISAs.

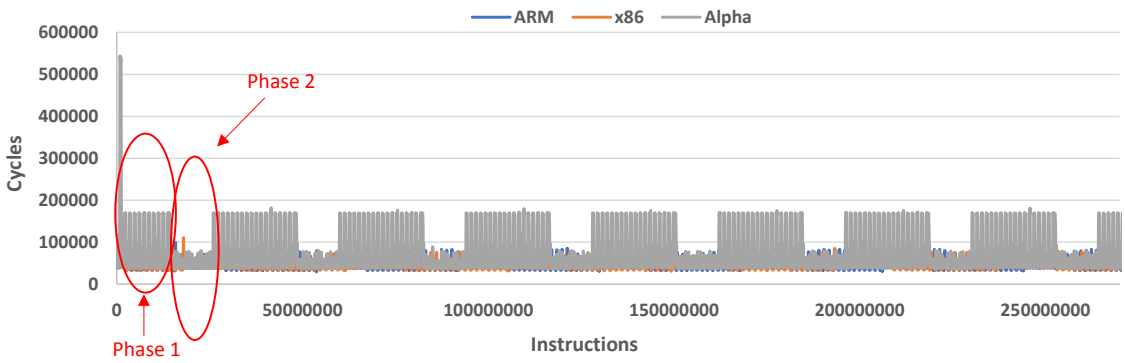

Figure 4.13: Cycles over windows of 50k insts. for *omnetpp* simpoint 2 on Haswell core

perlbench:

This is a benchmark for Perl language, where most of the OS-specific features are removed. Like *dijkstra*, *perlbench* is another benchmark where Alpha suffers from high register pressure. *perlbench* has a significant number of stack operations in the generated code for all ISAs. However, Alpha has significantly larger number of stack operations compared to the other ISAs. For example, in one of the most used functions of this benchmark *S_regmatch*, Alpha has approximately 39% more stack operations than the other ISAs. This results into lower performance of Alpha on all simpoints compared to the other ISAs.

hmmer:

This program benchmarks searching operations in a gene sequence database. In case of *hmmer*, x86 takes the minimum number of cycles on Haswell-like core, but fails to do so on other out-of-order cores. This benchmark has a high number of ARM and Alpha μ-ops compared to x86. An important function in the benchmark *P7Viterbi* (where program spends most of the time) contains many 'if' statements around store operations. An example of such statements with corresponding assembly instructions for all ISAs is shown in Listing 4.5.

Listing 4.5: Example from *P7Viterbi* function

C Code:

```
if ((sc = ip[k-1] + tpim[k-1]) > mc[k])
    mc[k] = sc;
```

x86:

```
mov     (%r8,%rax,4),%r15d    //complex addressing mode
add     0x0(%r13,%rax,4),%r15d  //complex addressing mode
cmp     %ecx,%r15d
cmovge  %r15d,%ecx
mov     %ecx,0x4(%rdx)
```

ARM:

```
add   x5, x5, #0x4
ldr   w11, [x1,x5]
ldr   w4, [x26,x5]
add   w11, w11, w4
cmp   w11, w12
csel  w11, w11, w12, ge
str   w11, [x6,#4]
```

Alpha:

```
ldq   t3,160(sp)
addq  t3,t9,t2
ldl   t1,0(t2)
ldl   t0,0(a5)
addl  t1,t0,t1
cmple t1,t3,t2
cmovne t2,t3,t1
stl   t1,4(t10)
```

As can be seen in Listing 4.5, x86 makes use of complex addressing modes resulting in

a lower number of instructions (and also μ-ops) compared to the other ISAs. Even though x86 has a lower number of total μ-ops, on less aggressive out-of-order cores there is more congestion for x86 in the instruction queue resulting into lower performance on A15-like and Alpha-21264-like cores.

libquantum:

libquantum is a benchmark for the simulation of a quantum computer. Figure 4.14 shows the behavior of ISAs on one of the simpoints of *libquantum* for Haswell core. There are two main phases in this simpoint, which are circled in Figure 4.14. While on phase 1 all ISAs show similar behavior, on phase 2 the behavior of ISAs is very different from each other. In case of IO cores like Atom, as shown in Figure 4.15, on phase 2 ARM and Alpha show similar behavior but, x86 takes much more number of cycles as compared to them.

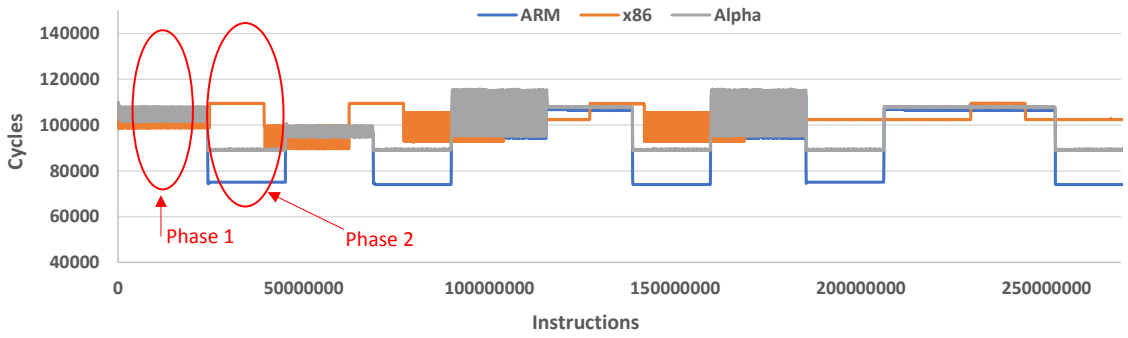

Figure 4.14: Cycles over windows of 50k instsructions for *libquantum* on Haswell core

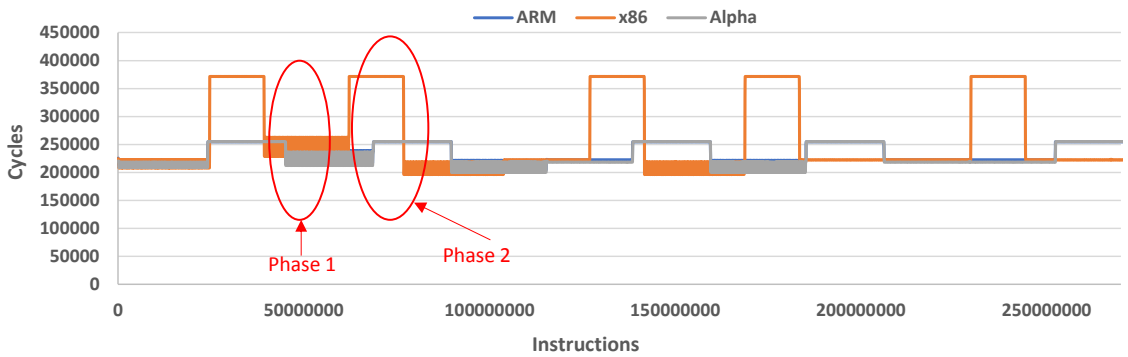

Figure 4.15: Cycles over windows of 50k instructions for *libquantum* on Atom core

During phase 2 in Figure 4.14, a function *quantum_sigma_x* executes and runs a loop repeatedly. In this loop x86 has one more uop compared to ARM as shown in Listing 4.6. This could result in a higher number of cycles per window of instructions for x86.

Listing 4.6: Example from *quantum_sigma_x* function

C/C++ Code:

```
for(i=0; i<reg->size; i++)
   {
  /* Flip the target bit of each basis state */
   reg->node[i].state ^= ((MAX_UNSIGNED) 1 << target);
   }
```

x86:

```
mov %rdx,%rcx
add 0x10(%rbx),%rcx   // inst. will be decoded into 2 μ-ops
add $0x1,%eax
add $0x10,%rdx
xor %r8,0x8(%rcx)     // r8 = 1 << target, inst. will be decoded into 3 μ-ops
cmp %eax,0x4(%rbx)    // inst. will be decoded into 2 μ-ops
jg  401ee0 <quantum_sigma_x+0x60>
```

ARM:

```
ldr x4, [x19,#16]
add w3, w3, #0x1
add x4, x4, x2
ldr x5, [x4,#8]
add x2, x2, #0x10
eor x5, x5, x0      // x0 = 1 << target
str x5, [x4,#8]
ldr w4, [x19,#4]
cmp w4, w3
b.gt 401bdc <quantum_sigma_x+0x6c>
```

Alpha:

```
ldq   t0,16(s0)
addl  t4,0x1,t4
addq  t0,t3,t0
lda   t3,16(t3)     // not a memory reference
ldq   t1,8(t0)
xor   t1,a0,t1      // a0 = 1 << target
stq   t1,8(t0)
ldl   t2,4(s0)
cmple t2,t4,t2
beq   t2,120001600 <quantum_sigma_x+0x80>
```

mcf:

mcf is a memory intensive benchmark and Alpha takes a larger number of cycles to execute this benchmark. It has approximately 40% more instruction queue full events and 28% more ROB full events compared to ARM on Haswell-like core. An example of extra dependency in *primal_bea_mpp function* function (the most critical function) in case of Alpha is shown in Listing 4.7. x86 makes use of complex addressing modes and results into one less instruction compared to the other ISAs. Alpha also suffers from higher register pressure as it has a higher number of loads and stores as shown in Figure 4.5 compared to the other ISAs for this benchmark. An example of higher register pressure is observed at the end point of *primal_iminus* function (the second most called function), where Alpha restores (or loads) double the number of callee-saved registers as compared to the other ISAs.

Listing 4.7: Example from *primal_bea_mpp* function

C Code:

```
perm[next]->a = arc;
```

x86:

```
mov     0x6b4d60(,%r10,8),%rsi
mov     %rax,(%rsi)
```

Alpha:

```
s8addq  t8,s1,t2
ldq     t0,0(t2)
stq     t5,0(t0)
```

ARM:

```
lsl     x5, x10, #3
ldr     x9, [x7,x5]
str     x0, [x9]
```

sjeng:

sjeng is an AI related benchmark. All ISAs perform similar for this benchmark.

SPEC-CPU2006 FLOATING POINT BENCHMARKS

lbm:

This program uses "Lattice Boltzman Method" to simulate fluids. *lbm* is another memory intensive benchmark like *mcf*. For *lbm*, x86 takes a larger number of cycles compared to the other ISAs. ARM and Alpha have significantly lower number of μ-ops than x86. Listing 4.8 shows an example from the most critical function of the benchmark, *LBM_performStreamCollide*. Since there are several additions performed with many intermediate sums, this piece of code needs several registers, x86 spills some registers (temporary loaded values) onto stack and later use them for addition. This finding is similar to what Venkat et al. found in [20]. Although x86 takes on average a high number of cycles compared to the other ISAs, on one of the 5 studied simpoints (simpoint 3) all ISAs take almost the same number of cycles. x86 has higher number of rename registers full events on all simpoints, except simpoint 3, compared to the other ISAs.

Listing 4.8: Example from *LBM_performStreamCollide*

C/C++ Code:

```
rho = + SRC_C ( srcGrid ) + SRC_N ( srcGrid )
      + SRC_S ( srcGrid ) + SRC_E ( srcGrid )
      + ..... // this continues
        // a total of 19 additions are performed
```

x86:

```
movsd   (%rdi),%xmm7
and     $0x2,%edx
movsd   0x8(%rdi),%xmm13
movapd  %xmm7,%xmm0   // final sum in xmm0
movsd   %xmm7,0x20(%rsp)
movsd   0x10(%rdi),%xmm7    // xmm7 loaded
addsd   %xmm13,%xmm0
movsd   0x18(%rdi),%xmm15
movsd   %xmm7,(%rsp)   // xmm7 pushed to stack
movsd   0x20(%rdi),%xmm4
movsd   0x28(%rdi),%xmm3
addsd   (%rsp),%xmm0    // xmm7 on stack + xmm0
movsd   %xmm4,0x8(%rsp)
.......                 // same pattern follows
```

ARM:

```
ldr    d10, [x0]
ldr    d23, [x0,#8]
ldr    d22, [x0,#16]
ldr    d25, [x0,#24]
fadd   d9, d10, d23    // final sum in d9
ldr    d24, [x0,#32]
fadd   d9, d9, d22
.......                // same pattern follows; no stack additions
```

Alpha:

```
ldt   $f10,0(s1)
stt   $f10,168(sp)
ldt   $f15,8(s1)
ldt   $f11,16(s1)
ldt   $f13,24(s1)
ldt   $f12,32(s1)
ldt   $f20,40(s1)
ldt   $f14,48(s1)
ldt   $f22,56(s1)
ldt   $f23,64(s1)
addt  $f10,$f15,$f10    // final sum in f10
.......                 // same pattern follows; no stack additions
```

milc:

milc focuses on simulations of four dimensional lattice gauge theory. x86 takes the most number of cycles on all configurations. An example of why it might have led to more μ-ops on x86 as compared to the other ISAs is shown in Listing 4.9.

Listing 4.9: Example from *mult_su3_na* function

C/C++ Code:

```
ar=a->e[i][0].real; ai=a->e[i][0].imag;
br=b->e[j][0].real; bi=b->e[j][0].imag;
cr=ar*br; t=ai*bi; cr += t;
ci=ai*br; t=ar*bi; ci -= t;
```

x86:

```
movsd   (%rdi),%xmm3  //'ar' in xmm3
movsd   0x8(%rdi),%xmm4  //'ai' in xmm4
movsd   (%rbx),%xmm0
add     $0x30,%rdi
movsd   0x8(%rbx),%xmm2
movapd  %xmm3,%xmm1    //'ar' copied to xmm3
movapd  %xmm4,%xmm5    //'ai' copied to xmm5
mulsd   %xmm0,%xmm1
mulsd   %xmm2,%xmm5
mulsd   %xmm4,%xmm0
mulsd   %xmm3,%xmm2
```

ARM:

```
ldr   d3, [x0]
ldr   d2, [x0,#8]
ldr   d5, [x1,#8]
ldr   d0, [x1]
fmul  d16, d2, d5
fmul  d7, d3, d5
fmadd   d16, d3, d0, d16
fnmsub  d7, d2, d0, d7
```

Alpha:

```
ldt   $f12,0(s0)
ldt   $f13,8(s0)
ldt   $f26,0(s2)
ldt   $f27,8(s2)
mult  $f12,$f27,$f11
mult  $f13,$f26,$f10
mult  $f12,$f26,$f12
mult  $f13,$f27,$f13
```

This Listing shows the code from the most critical function, *mult_su3_na*, of the benchmark. One of the main sources of extra instructions and more dependent operations is two copying operations, which are pointed out in Listing 4.9. The registers containing values of *ai* and *ar* will not retain their values after performing multiplication in the 3rd line of C/C++ code, so these are copied to two other registers to use these values to perform multiplication operations shown in the 4th line of C/C++ code. While on average Alpha is doing better than x86 this is not true for all simpoints. One out of the 5 studied simpoints is shown in Figure 4.16. There are two main phases in this simpoint as shown in the figure. x86 and Alpha alternate their behavior in those phases. During phase 1 x86 has the highest number of memory reads per window of instructions, while during phase 2 Alpha has the highest number of memory reads per window of instructions.

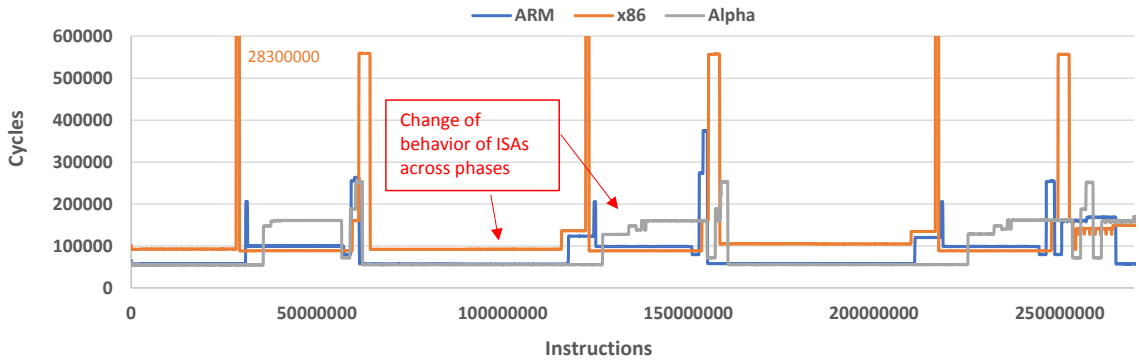

Figure 4.16: Cycles over windows of 50k instructions for a simpoint of *milc*

namd:

On *namd*, x86 takes much higher number of μ-ops than the other ISAs, which causes x86 to take the most number of cycles to execute this benchmark on all cores. Generally, in floating-point benchmarks, x86 leads to high number of μ-ops which hurts its performance in comparison to the other ISAs. Listing 2.1 in Appendix B shows an example code for namd. As shown in Listing 2.1 x86 has clearly less number of independent operations in this piece of code com-

pared to other ISAs, which shows an example of low performance for x86. It should be noted that the independent operations, in this example, refer to operations that are not dependent on any other operations within the shown code sequence.

povray:

povray is a ray tracing benchmark that performs rendering on reference input images. On average for this benchmark, ARM always takes the least number of cycles. Alpha takes the most number of cycles on most of the configurations except A15 and 21264.

sphinx:

This is a speech recognition benchmark. Like most of the other floating-point benchmarks, x86 takes the most number of cycles on most of the cores. For OoO cores, there are much more physical register full-events for x86 compared to the other ISAs. A code example is shown in Listing 4.10. x86 uses two operations *unpcklps* and *cvtps2pd* to convert single precision values to double precision values in this code example. As a result there is an extra operation in this piece of code for x86 compared to the other ISAs.

Figure A.1 to A.21 in Appendix A show generated kiviat plots for all benchmarks. Each axis in these kiviat plots represent performance of each ISA normalized to x86 for each microarchitecture. ISA with the lowest area of the resulting hexagon will be the one performing best for all microarchitecutres on average.

Listing 4.10: Example from *mgau_eval* function

C/C++ Code:

```
diff1 = x[i] - m1[i];
dval1 -= diff1 * diff1 * v1[i];
```

x86:

```
movss   0x0(%rbp,%rdx,4),%xmm0
movss   (%rsi,%rdx,4),%xmm2
subss   (%rcx,%rdx,4),%xmm0
add     $0x1,%rdx
cvtps2pd %xmm2,%xmm2
cmp     %edx,%ebx
unpcklps %xmm0,%xmm0    // 1st op. of single to double precision conversion
cvtps2pd %xmm0,%xmm0    // 2nd op. of single to double precision conversion
mulsd   %xmm0,%xmm0
mulsd   %xmm2,%xmm0
subsd   %xmm0,%xmm1
```

ARM:

```
ldr     s3, [x19,x1]
ldr     s1, [x2,x1]
ldr     s4, [x3,x1]
fsub    s1, s3, s1
fcvt    d1, s1
add     x1, x1, #0x4
fmul    d1, d1, d1
fcvt    d3, s4
cmp     x1, x20
fmsub   d2, d1, d3, d2
```

Alpha:

```
lds   $f10,0(t0)
lds   $f11,0(t1)
lds   $f12,0(t2)
subs  $f10,$f11,$f10
fmov  $f10,$f10
mult  $f10,$f10,$f10
mult  $f10,$f12,$f10
subt  $f13,$f10,$f13
```

4.3 Other Microarchitecture Dependent Statistics

This section examines other microarchitecture dependent statistics like branch mispredictions and cache misses across ISAs for all benchmarks. Figures 4.17 and 4.18 show instruction cache misses for ARM and Alpha relative to x86 for OoO and IO cores respectively. On average ARM has the lowest instruction cache misses and there are observed differences across ISAs. However, instruction cache misses are very low in number for most of the benchmarks on these studied cores. It can be concluded that the sizes of used L1 instruction caches are enough to mitigate any ISA bottlenecks related to code size.

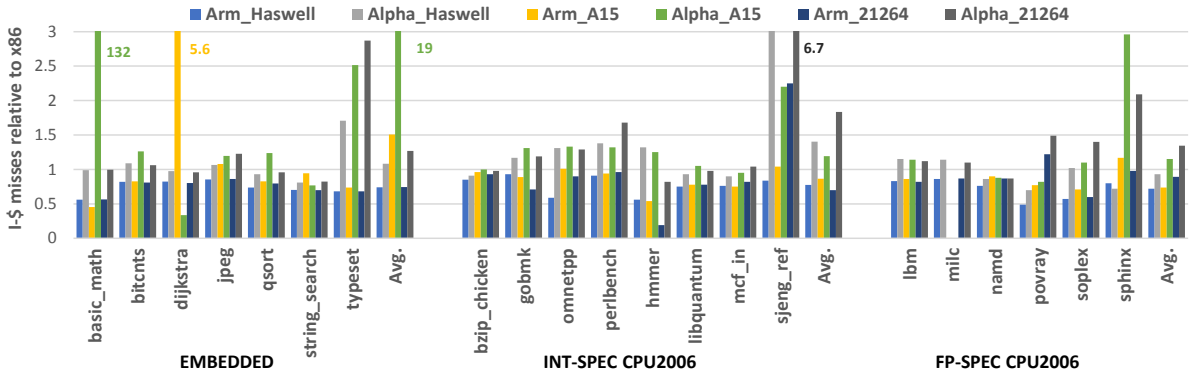

Figure 4.17: I-Cache Misses relative to x86 for OoO Cores relative

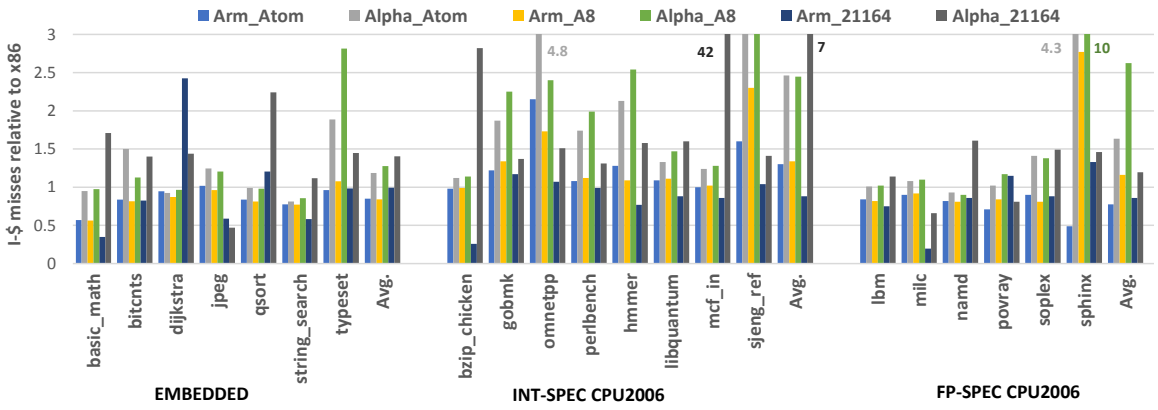

Figure 4.18: I-Cache Misses relative to x86 for IO Cores

Figures 4.19 and 4.20 show L1-data cache misses for ARM and Alpha relative to x86 for OoO and IO cores respectively. As shown in Figure 4.20, in case of IO cores the number of L1-data cache misses resemble across ISAs. However, in case of OoO cores the number of L1-data cache misses can differ considerably across ISAs as shown in Figure 4.19. One possible

explanation for this behavior is because the timing of cache accesses for OoO cores do not match for all ISAs. This is based on the dependencies in the dynamic instruction sequence for that ISA, which can lead to a different number of L1 data cache misses.

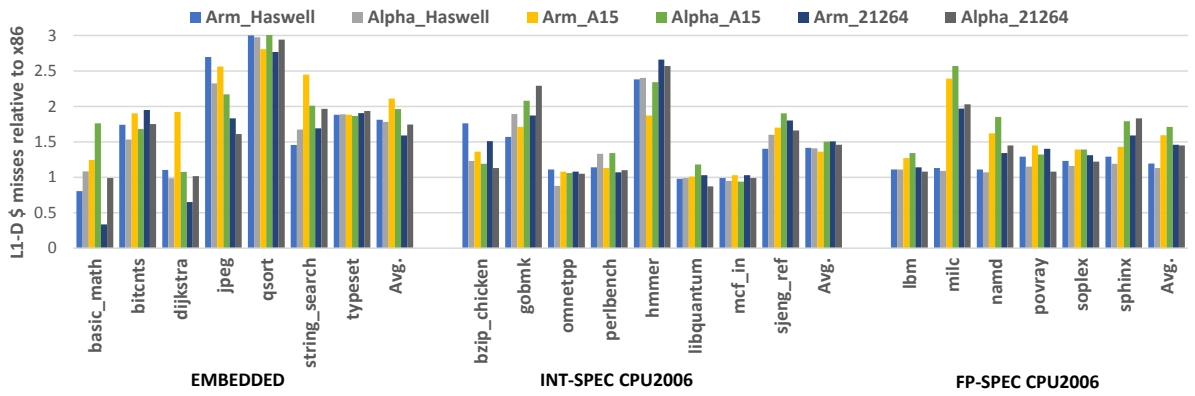

Figure 4.19: Normalized L1-d cache misses for OoO cores

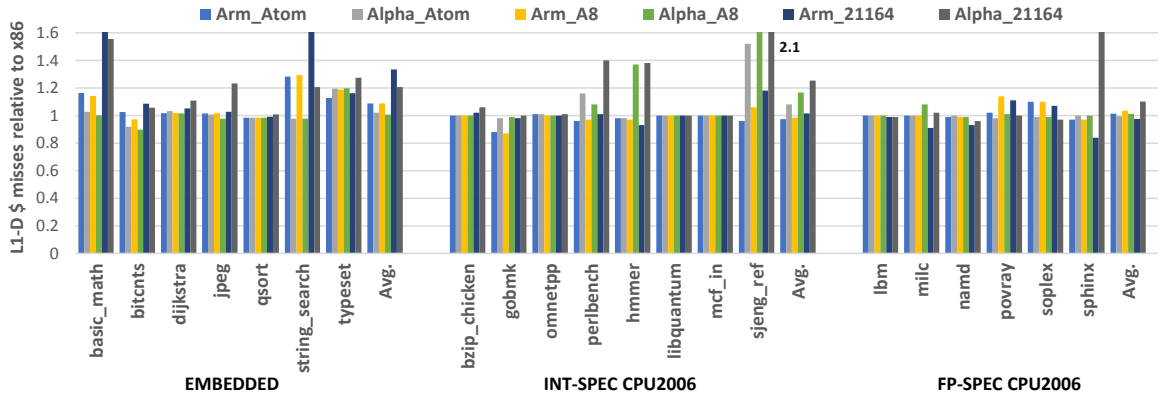

Figure 4.20: Normalized L1-d cache misses for IO cores

Figures 4.21 and 4.22 show last level cache (L3 for Haswell and Alpha-21164 and L2 for other cores) misses for ARM and Alpha relative to x86 for OoO and IO cores respectively. As shown in these figures the number of last-level cache misses are very close across ISAs for both OoO and IO cores respectively.

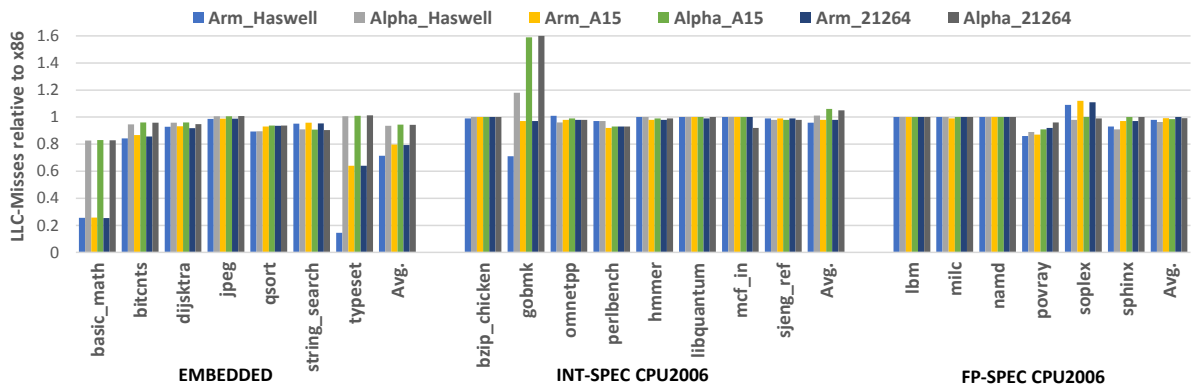

Figure 4.21: Normalized LLC cache misses for OoO cores

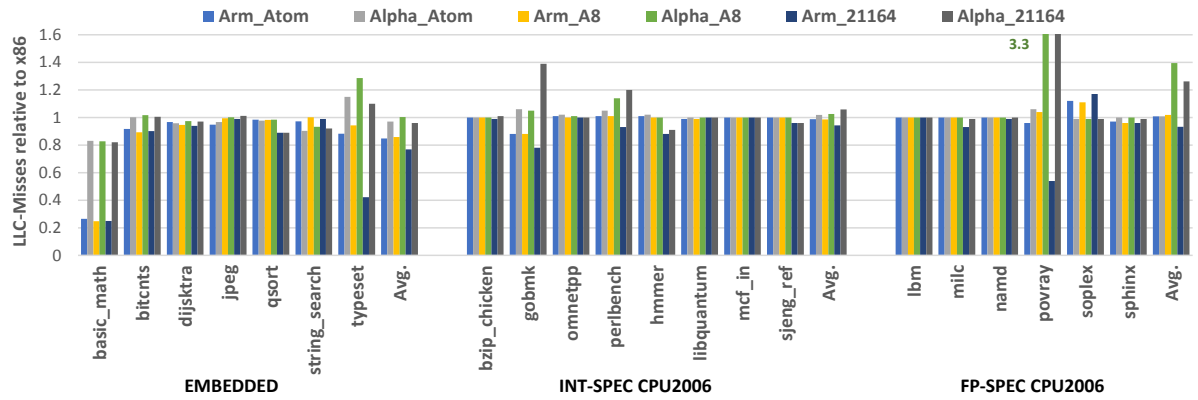

Figure 4.22: Normalized LLC cache misses for IO cores

Figures 4.23 and 4.24 show branch predictor misses for ARM and Alpha relative to x86 for OoO and IO cores respectively. For most of the benchmarks, the number of branch mispredictions are very close across ISAs for both OoO and IO cores. There are few exceptions as well like *gobmk*, *qsort* and *povray*. Having high number of branch operations and differing in exact number of branch operations across ISAs (Figure 4.5) could explain this behavior for the aforementioned benchmarks.

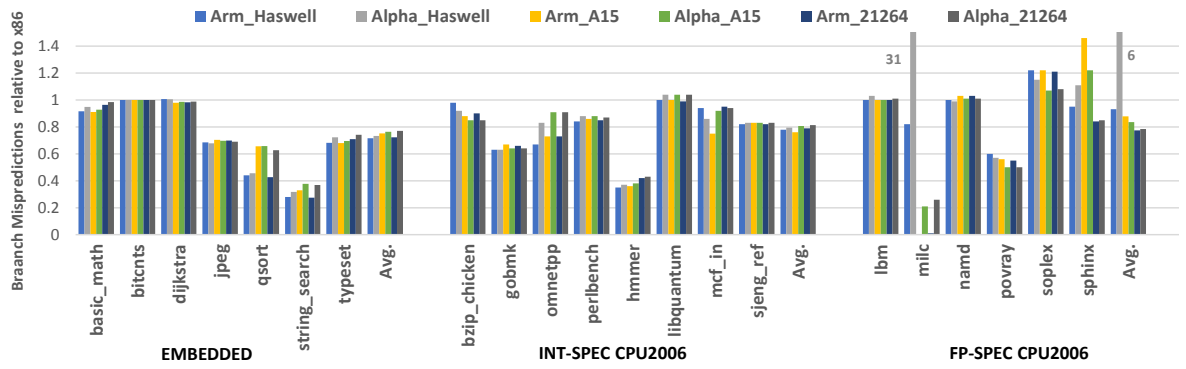

Figure 4.23: Branch predictor misses relative to x86 for OoO cores

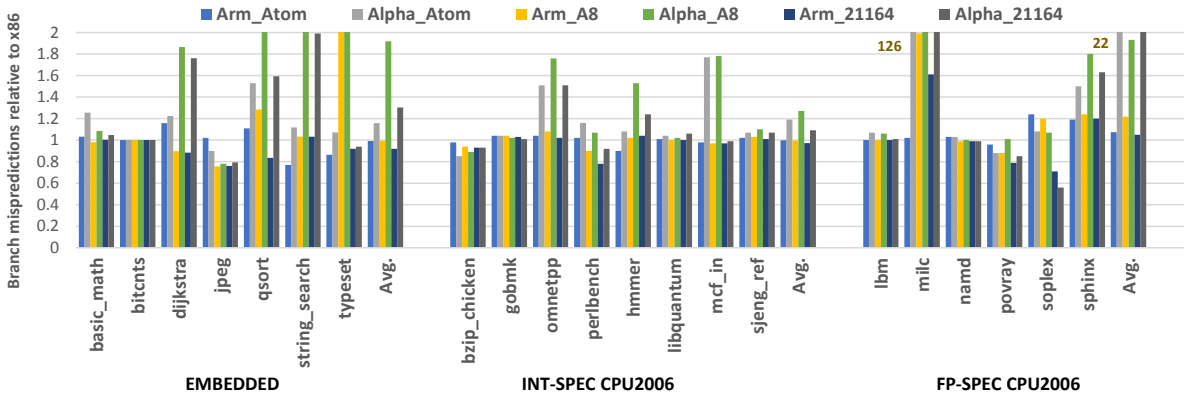

Figure 4.24: Branch predictor misses relative to x86 for IO cores

4.4 Performance Analysis Across Microarchitectures

This section analyzes the differences in performance across microarchitectures rather than ISAs. Figures 4.25, 4.26 and 4.27 show time taken to execute each benchmark on any microarchitecture and ISA combination relative to ARM_Haswell (combination of ARM ISA and Haswell microarchitecture) for embedded, SPEC-CPU2006 integer and floating point benchmarks respectively. As shown from these figures, the effect of *significant* changes in microarchitecture on perfromance is bigger than the effect of change of an ISA on a particular microarchitecture (Figure 4.1 and 4.2). For example, going from Haswell-like core to A15-like core affected performance more than an ISA change affected on each of the two cores.

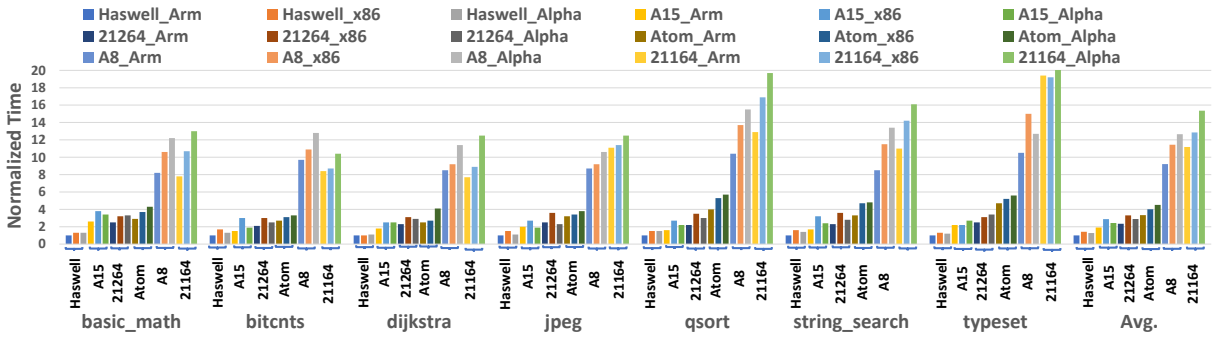

Figure 4.25: Execution time relative to ARM_Haswell for embedded benchmarks

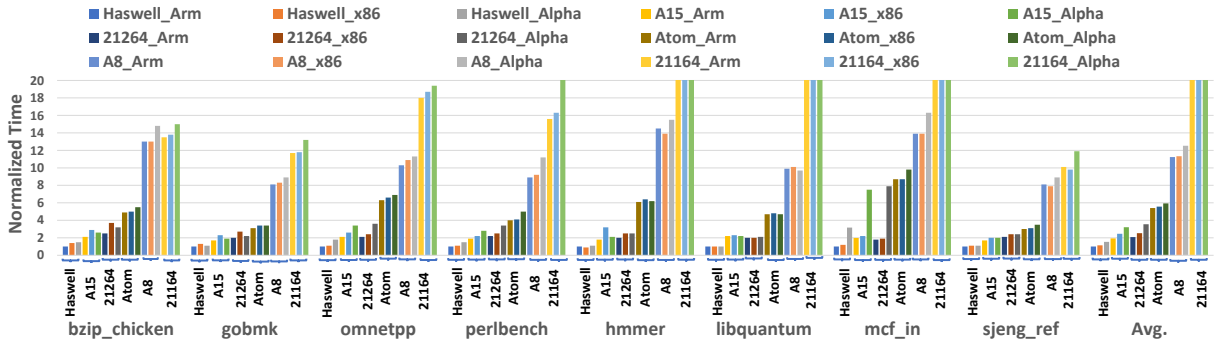

Figure 4.26: Execution time relative to ARM_Haswell for integer benchmarks

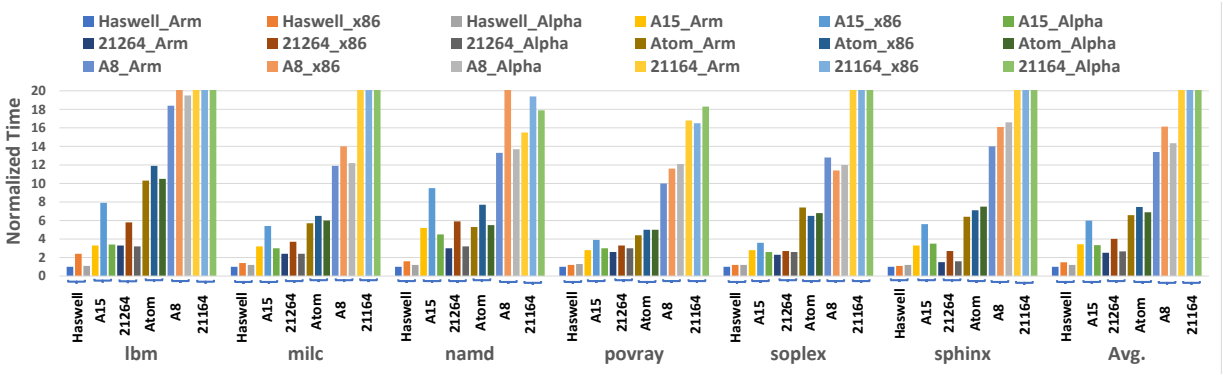

Figure 4.27: Execution time relative to ARM_Haswell for floating point benchmarks

4.5 Microarchitectural Optimizations for x86

Modern x86 microarchitectures like Intel Haswell employ many microarchitecture optimizations to improve performance, such as μ-op fusion. In this section, we study the impact of

two optimizations on OoO cores used in this work. We examined the effect of μ-op fusion and μ-op cache on performance of OoO cores and compared improved x86 performance with other ISAs.

μ-op fusion refers to the technique of generating a fused combination of μ-ops by the decoder stage in modern Intel pipelines. The fused μ-op occupies a single entry in reorder buffer and reservation stations. The operations in a fused μ-op are issued separately when they are ready to execute. To implement μ-op fusion in the simulator we performed an optimistic approximation. As shown in [36], the number of fusible operations range from 6% to 29% in SPEC-CPU2006 benchmarks. Thus, the maximum number of μ-ops that can be fused is 29%. Keeping this in mind, the maximum possible benefit from μ-op fusion can be observed by scaling up the sizes of pipeline structures (like ROB and reservation stations) and stages (rename, decode, dispatch) that deal with fused μ-ops by 14.5%. Thus, we scaled up their sizes by 14.5% to examine the benefit of μ-op fusion.

Contemporary Intel cores also use a μ-op cache, which caches μ-ops when instructions are decoded into μ-ops at decode stage. When new instructions have to be fetched from Instruction cache, the μ-op cache is also checked in parallel. In case of a μ-op cache hit, μ-ops are directly used from the μ-op cache. We added a μ-op cache in the simulator following the design discussed in [91]. The implemented cache is capable of holding 1.5K μ-ops and it is an 8-way associative cache. Figure 4.28 shows the percentage improvement in performance seen for each benchmark using both μ-op fusion and μ-op cache together for x86. The average improvement is up to 3% (only exception is A15 for embedded benchmarks).

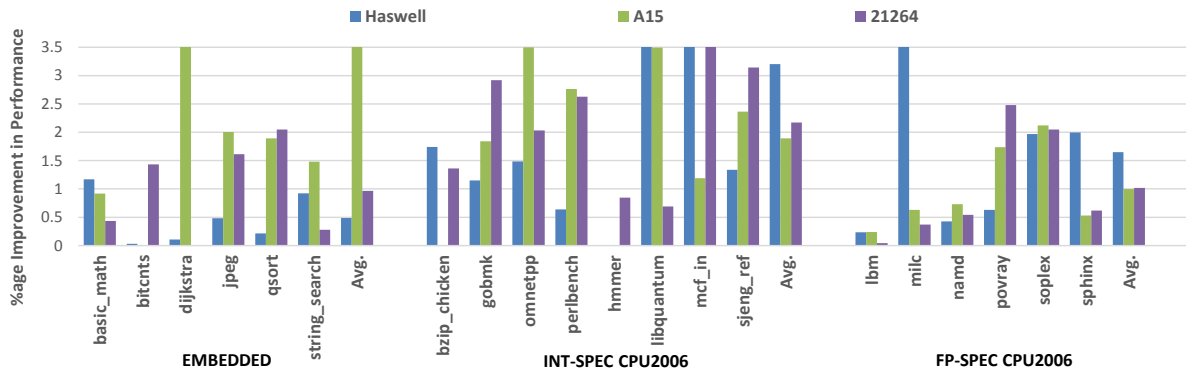

Figure 4.28: Percentage improvement for x86 using μ-op fusion and μ-op cache

Figure 4.29 shows cycle counts for ARM and Alpha relative to x86 with the implemented microarchitecure optimizations for OoO cores. Because of the small percentage improvement in performance the total execution time is not affected much by these optimizations and the figure is similar to Figure 4.1 (cycle counts relative to normal x86 cores).

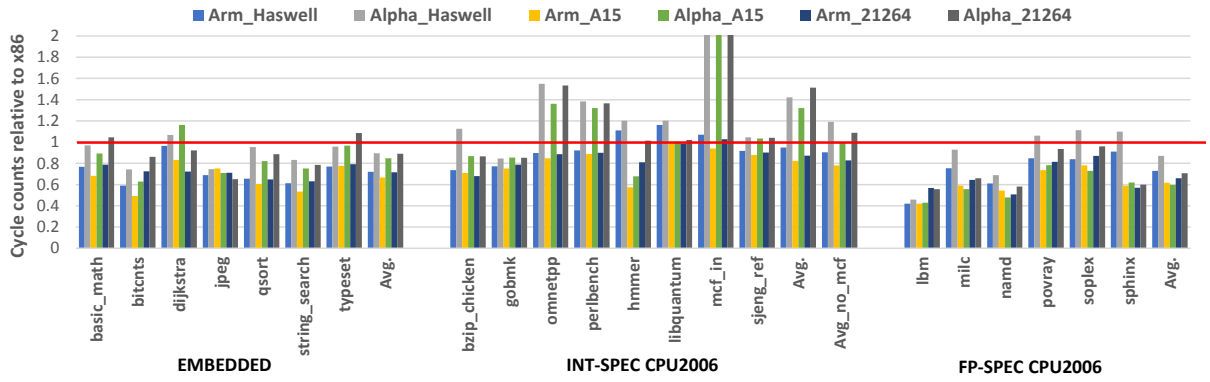

Figure 4.29: Cycles relative to x86 (with μ-architecture optimizations) for OoO Cores

4.6 Summary of the Findings

This section discusses a summary of the findings based on the aformentioned examples and results.

1. On average, ARMv8 outperforms other ISAs on similar microarchitectures, as it offers better instruction-level parallelism and has lower number of dynamic μ-ops compared to the other ISAs in most of the cases.

2. The average behavior of ISAs can be very different from their behavior for a particular phase of execution, which agrees with Venkat and Tullsen's findings [20].

3. The performance differences across ISAs are significantly reduced in in-order cores compared to out-of-order cores.

4. On average, x86 has the highest number of dynamic μ-ops. This agrees with previous findings when compared to Alpha [20]. There are few examples where Alpha exceeds

x86 in the number of μ-ops, but ARMv8 always has lower or equal number of μ-ops when compared to x86.

5. x86 seems to have over-serialized code due to ISA limitations, such as use of implicit operands and overlap of one of the source and destination registers as observed by [21]. x86 has the highest average degree of use of registers (the average number of instructions, which consume the value generated by a particular instruction).

6. The total number of L1-instruction cache misses is very low across all ISAs for the studied cores. This infers that the sizes of L1 instruction caches that are used are sufficient to eliminate any ISA bottlenecks related to code size for the studied benchmarks.

7. Based on our results, the number of L1-data cache misses are similar across all ISAs in case of in-order cores, but the numbers can vary significantly in case of out-of-order cores.

8. On average, the number of branch mispredictions are very close across ISAs for all cores with few exceptions such as *gobmk*, *qsort* and *povray*.

9. μ-ops to instructions ratio on x86 is usually less than 1.3, as observed by Blem et al [40]. However, the overall instructions count and mixes are ISA-dependent, which contradicts Blem et al's [40] conclusion that instruction counts do not depend on ISAs.

10. Significant microarchitectural changes affect performance more than an ISA change does on a particular microarchitecture.

11. According to Blem et al's study [40], performance differences on studied platforms are mainly because of microarchitectures. We see performance differences on exactly similar microarchitectures; which means ISAs are responsible for those performance differences. Moreover, since performance differences across ISAs are different for different microarchitectures, we can conclude that the behavior of ISA depends on microarchitecture as well but they certainly have a particular role in performance.

Chapter 5

Conclusion and Future Work

Modern developments in ISAs and their implementations, in addition to the conflicting claims regarding the role of ISAs in performance of a processor, demand for a review of this historical debate. This thesis studies and analyzes the effect of three ISAs on performance of many benchmarks using six microarchitectures. The adopted methodology ensures that the non-ISA factors are kept constant across ISAs for all experiments. This work relates the observed performance differences across ISAs to the benchmarks' assembly code for each ISA. Specific examples of such code blocks are shown to explain the noticed differences. We also relate our findings to the related work found in literature that studied impact of ISAs on performance. Our results indicate that ISAs can affect performance, and that the amount of effect differs based on the microarchitecture. Moreover, programs often exhibit phases of execution, which can be more affine to one ISA than the other. We also observed that the difference in performance among the studied ISAs is insignificant for in-order (IO) cores, compared to out-of-order (OoO) cores.

One future direction for this work is to explore the impact of ISAs on diverse types of workloads such as workloads related to server and cloud computing, artificial intelligence, cryptography, multimedia and ultra-low power systems. It will be interesting to see if some ISAs are more suitable for a particular category of workloads. Moreover, more diverse ISAs can be selected to examine the maximum possible performance differences across ISAs. Since, low power consumption has become a stringent need of modern microprocessors, it is important to study the impact of ISAs on power consumption given a particular microarchitecture. Differences across ISAs in terms of performance and power consumption (possibly for specific

types of workloads) can then be exploited to design heterogeneous ISA multicore systems.

Other interesting thing to look at is the possibility of building some models to predict the effect of ISAs on certain applications. These models can be built using machine learning or fuzzy reasoning based techniques. Such models can be used not only to improve ISAs, but also to help scheduling of applications across different cores in heterogeneous ISA multi-core systems.

Appendix A

Kiviat Plots for All Benchmarks

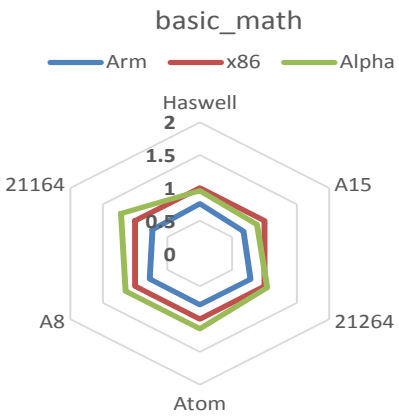

Figure A.1: kiviat plot for basic_math

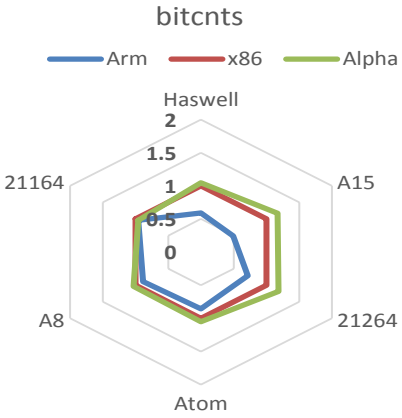

Figure A.2: kiviat plot for bitcnt

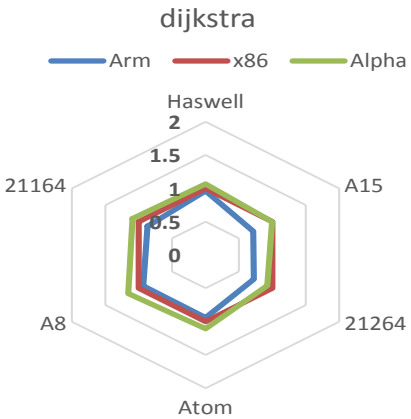

Figure A.3: kiviat plot for dijkstra

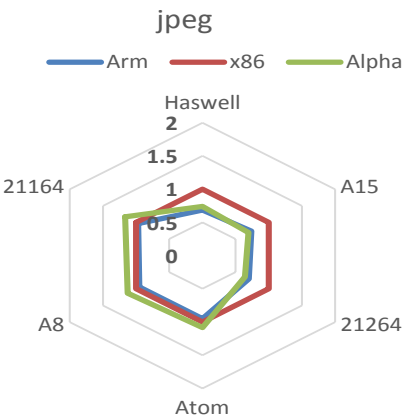

Figure A.4: kiviat plot for jpeg

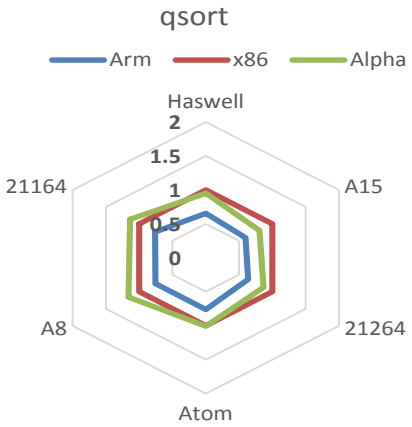

Figure A.5: kiviat plot for qsort

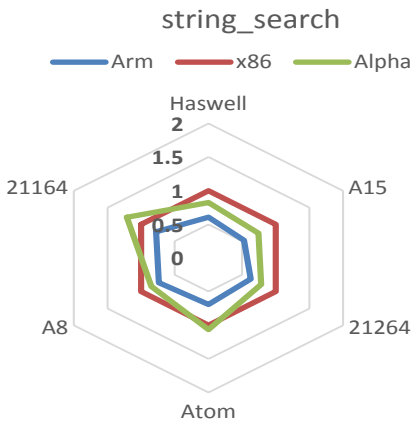

Figure A.6: kiviat plot for string_search

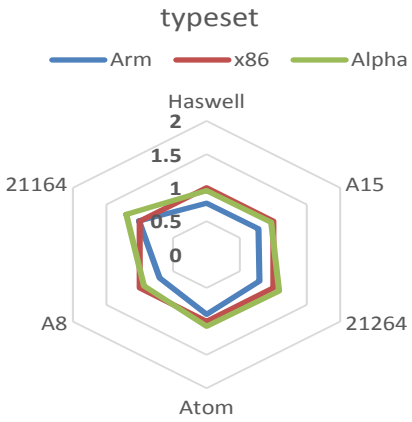

Figure A.7: kiviat plot for typeset

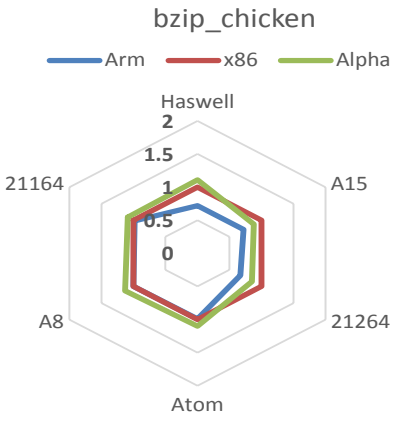

Figure A.8: kiviat plot for bzip_chicken

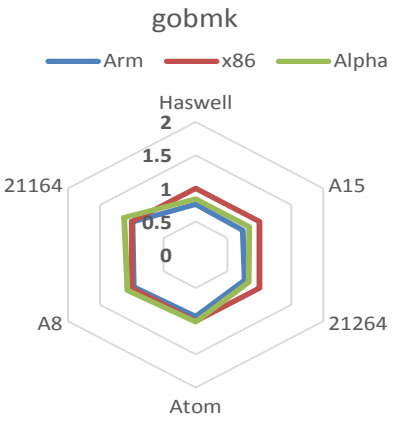

Figure A.9: kiviat plot for gobmk

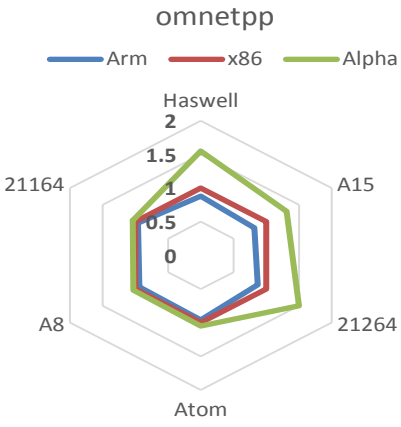

Figure A.10: kiviat plot for omnetpp

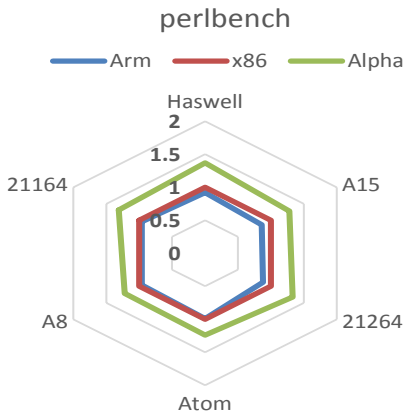

Figure A.11: kiviat plot for perlbench

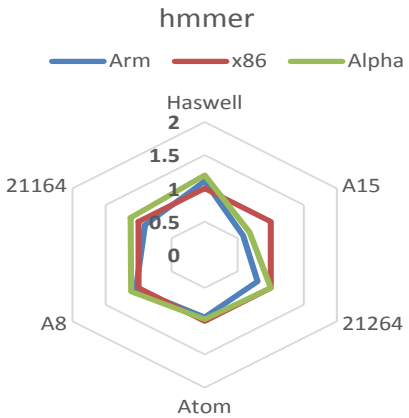

Figure A.12: kiviat plot for hmmer

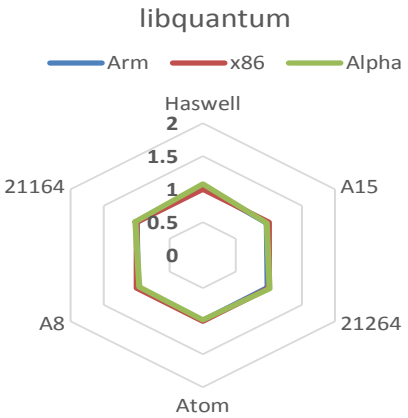

Figure A.13: kiviat plot for libquantum

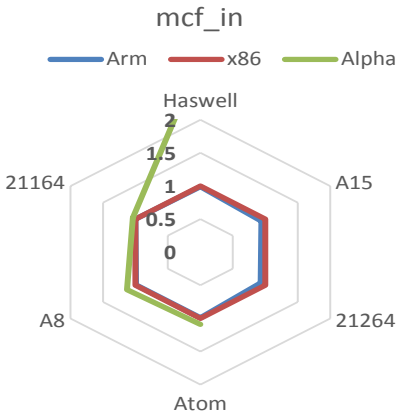

Figure A.14: kiviat plot for mcf_in

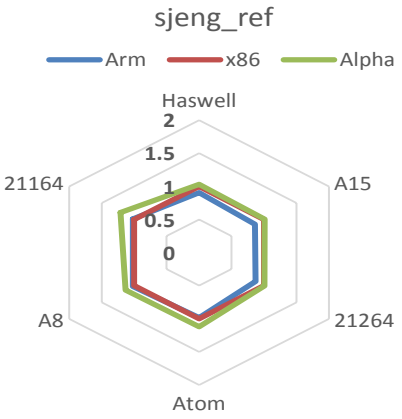

Figure A.15: kiviat plot for sjeng_ref

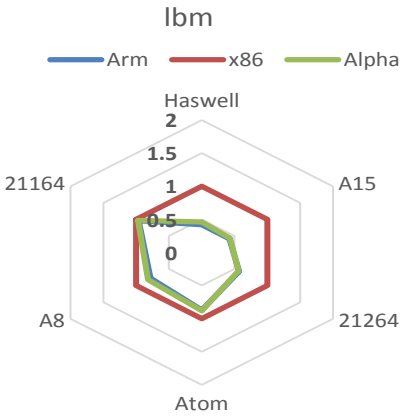

Figure A.16: kiviat plot for lbm

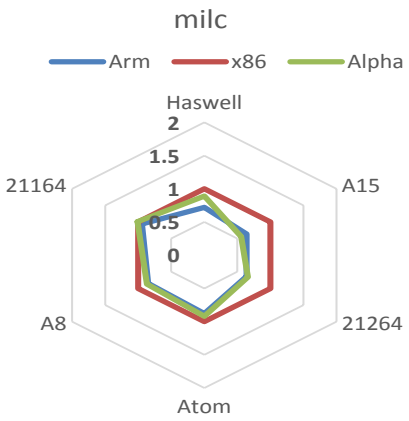

Figure A.17: kiviat plot for milc

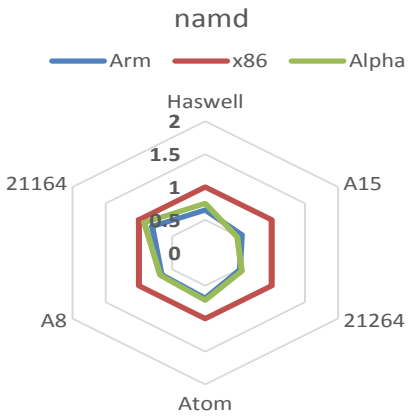

Figure A.18: kiviat plot for namd

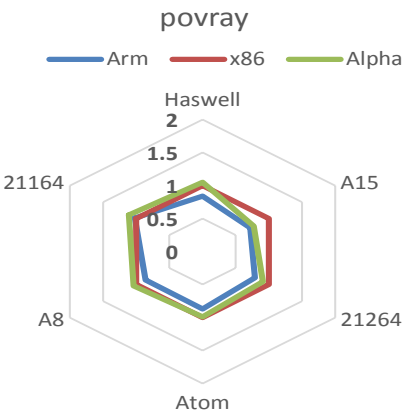

Figure A.19: kiviat plot for povray

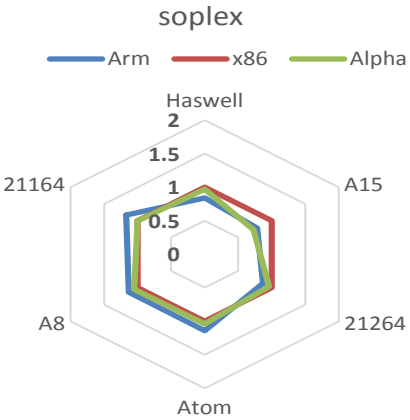

Figure A.20: kiviat plot for soplex

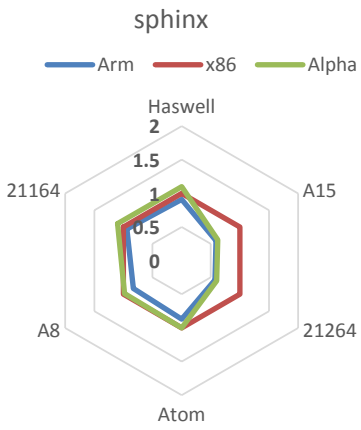

Figure A.21: kiviat plot for sphinx

Appendix B

Example Code Blocks

Listing 2.1: Example from *create* function

C/C++ Code:

```
Vector shift = (i%3-1) * a1 + ((i/3)%3-1) * a2 + (i/9-1) * a3;
 for( int j = 0; j < n; ++j ) {
    dt[j] = d[j];
    dt[j].position += shift;
   }
```

x86:

```
mov     %r12d,%eax              // independent operation
mov     $0x38e38e39,%edx        // independent operation
mov     %r12d,%edi              // independent operation
imul    %edx
sar     $0x1f,%edi
mov     $0x55555556,%ecx        // independent operation
mov     %r12d,%eax
movsd   0x28(%rbp),%xmm2        // independent operation
movsd   0x10(%rbp),%xmm5        // independent operation
sar     %edx
movsd   0x40(%rbp),%xmm1        // independent operation
sub     %edi,%edx
movsd   0x8(%rbp),%xmm6         // independent operation
sub     $0x1,%edx
cvtsi2sd %edx,%xmm3
imul    %ecx
mov     %edx,%esi
sub     %edi,%esi
mov     %esi,%eax
imul    %ecx
mov     %esi,%eax
```

```
mov     %esi,%ecx
sar     $0x1f,%eax
sub     %eax,%edx
lea     (%rdx,%rdx,2),%eax
mulsd   %xmm3,%xmm1
sub     %eax,%ecx
mov     %ecx,%eax
sub     $0x1,%eax
cvtsi2sd %eax,%xmm0
lea     (%rsi,%rsi,2),%eax
sub     %eax,%r12d
sub     $0x1,%r12d
test    %r14d,%r14d
cvtsi2sd %r12d,%xmm4
mulsd   %xmm0,%xmm2
mulsd   %xmm4,%xmm5
mulsd   %xmm4,%xmm6
mulsd   0x0(%rbp),%xmm4       // inst. will be decoded into 2 μ-ops
addsd   %xmm5,%xmm2
movsd   0x20(%rbp),%xmm5
mulsd   %xmm0,%xmm5
mulsd   0x18(%rbp),%xmm0      // inst. will be decoded into 2 μ-ops
addsd   %xmm2,%xmm1
movsd   0x38(%rbp),%xmm2
mulsd   %xmm3,%xmm2
addsd   %xmm6,%xmm5
mulsd   0x30(%rbp),%xmm3      // inst. will be decoded into 2 μ-ops
addsd   %xmm4,%xmm0
addsd   %xmm5,%xmm2
addsd   %xmm0,%xmm3
```

ARM:

```
mov     w3, #0x5556                 // independent operation
mov     w1, #0x3                    // independent operation
sdiv    w1, w19, w1
movk    w3, #0x5555, lsl #16        // independent operation
smull   x3, w1, w3
lsr     x3, x3, #32
mov     w2, #0x8e39                 // independent operation
sub     w3, w3, w1, asr #31
movk    w2, #0x38e3, lsl #16        // independent operation
smull   x4, w19, w2
add     w6, w3, w3, lsl #1
mov     x2, x22                     // independent operation
add     w5, w1, w1, lsl #1
lsr     x4, x4, #32
sub     w1, w1, w6
ldr     d18, [x2],#48
```

```
sub    w5, w19, w5
sub    w1, w1, #0x1
asr    w4, w4, #1
ldr    d6, [x22,#24]              // independent operation
ldr    d4, [x22,#32]              // independent operation
ldr    d19, [x22,#40]             // independent operation
scvtf  d2, w1
sub    w19, w4, w19, asr #31
sub    w1, w5, #0x1
ldr    d17, [x22,#8]              // independent operation
ldr    d16, [x22,#16]             // independent operation
scvtf  d1, w1
fmul   d6, d2, d6
fmul   d4, d2, d4
sub    w19, w19, #0x1
ldr    d3, [x22,#48]              // independent operation
ldr    d7, [x2,#8]
fmul   d2, d2, d19
ldr    d5, [x2,#16]
scvtf  d0, w19
fmadd  d6, d1, d18, d6    // inst. will be decoded into 2 μ-ops
fmadd  d4, d1, d17, d4    // inst. will be decoded into 2 μ-ops
cmp    w23, wzr
fmadd  d1, d1, d16, d2    // inst. will be decoded into 2 μ-ops
fmadd  d3, d0, d3, d6     // inst. will be decoded into 2 μ-ops
fmadd  d4, d0, d7, d4     // inst. will be decoded into 2 μ-ops
fmadd  d5, d0, d5, d1     // inst. will be decoded into 2 μ-ops
```

Alpha:

```
ldah   t4,21845           // independent operation
s8subq  s0,s0,t1          // independent operation
ldt    $f25,8(s1)         // independent operation
ldt    $f14,0(s1)         // independent operation
lda    t4,21846(t4)
sll    t1,0x6,t0
ldt    $f13,16(s1)        // independent operation
ldt    $f10,56(s1)        // independent operation
mulq   s0,t4,t3
addq   t1,t0,t1
ldt    $f23,64(s1)        // independent operation
ldt    $f22,48(s1)        // independent operation
sra    s0,0x1f,t5         // independent operation
s8addq  t1,s0,t1
ldt    $f24,24(s1)        // independent operation
ldt    $f26,32(s1)        // independent operation
sll    t1,0xf,t0
```

```
ldt   $f15,40(s1)
subq  t0,t1,t0
s8addq  t0,s0,t0
sra   t0,0x21,t0
subq  t0,t5,t0
srl   t3,0x20,t3
subl  t0,0x1,t0
subq  t3,t5,t3
itoft  t0,$f11
sextl  t3,t1
s4subq  t3,t3,t5
mulq  t1,t4,t4
sra   t1,0x1f,t1
subq  s0,t5,t2
subl  t2,0x1,t2
cvtqt  $f11,$f12
itoft  t2,$f27
cvtqt  $f27,$f11
mult  $f12,$f23,$f23
mult  $f12,$f22,$f22
srl   t4,0x20,t4
mult  $f12,$f10,$f12
subq  t4,t1,t4
s4subq  t4,t4,t4
mult  $f11,$f13,$f13
subq  t3,t4,t3
mult  $f11,$f14,$f14
subl  t3,0x1,t3
mult  $f11,$f25,$f11
itoft  t3,$f25
cvtqt  $f25,$f10
mult  $f10,$f15,$f15
mult  $f10,$f24,$f24
mult  $f10,$f26,$f10
addt  $f13,$f15,$f13
addt  $f14,$f24,$f14
addt  $f11,$f10,$f11
addt  $f13,$f23,$f13
addt  $f14,$f22,$f15
addt  $f11,$f12,$f14
```

Bibliography

[1] M. Martin and A. Roth, *Instruction Set Architecture*. Available: https://www.cis.upenn.edu/~milom/cis501-Fall05/lectures/02_isa.pdf [Online; accessed 1-June-2017].

[2] O. Mutlu, *ISA Tradeoffs*. Available: https://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=onur-447-spring15-lecture3-isa-tradeoffs-afterlecture.pdf [Online; accessed 1-June-2017].

[3] M. V. Wilkes, "The Growth of Interest in Microprogramming: A Literature Survey," *ACM Computing Surveys (CSUR)*, vol. 1, no. 3, pp. 139–145, September 1969.

[4] S. Terpe, *Why Instruction Sets No Longer Matter*. Available: http://ethw.org/Why_Instruction_Sets_No_Longer_Matter [Online; accessed 1-June-2017].

[5] E. W. Pugh, *IBM System/360*. Available: http://ethw.org/IBM_System/360 [Online; accessed 1-June-2017].

[6] *RISC Architecture*. Available: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/risc/ [Online; accessed 1-June-2017].

[7] J. Cocke and V. Markstein, "The evolution of RISC technology at IBM," *IBM Journal of research and development*, vol. 34, no. 1, pp. 4–11, January 1990.

[8] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Electronics*, vol. 38, pp. 114–117, April 1965.

[9] M. Bohr, *MOORE'S LAW LEADERSHIP*. Available: https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2017/03/Mark-Bohr-2017-Moores-Law.pdf [Online; accessed 1-June-2017].

[10] *risc vs. cisc*. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/ [Online; accessed 1-June-2017].

[11] M. Kerner and N. Padgett, *A History of Modern 64-bit Computing*, 2007. Available: http://courses.cs.washington.edu/courses/csep590/06au/projects/history-64-bit.pdf [Online; accessed 1-June-2017].

[12] *Intel Timeline: A History of Innovation*. Available: https://www.intel.com/content/www/us/en/history/historic-timeline.html?iid=about+ln_history [Online; accessed 1-June-2017].

[13] C. Lomont, *Introduction to x64 Assembly*. Available: https://software.intel.com/en-us/articles/introduction-to-x64-assembly [Online; accessed 1-June-2017].

[14] B. C. Lopes, R. Auler, L. Ramos, E. Borin, and R. Azevedo, "SHRINK: Reducing the ISA Complexity via Instruction Recycling," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 311–322, June 2015.

[15] *AMD vs Intel Market Share*. Available: https://www.cpubenchmark.net/market_share.html [Online; accessed 1-June-2017].

[16] *TOP500*. Available: https://www.top500.org/ [Online; accessed 1-June-2017].

[17] *TOP500 Supercomputers by Processor Family*. Available: https://commons.wikimedia.org/wiki/File:Processor_families_in_TOP500_supercomputers.svg [Online; accessed 1-June-2017].

[18] *Intel Streaming SIMD Extensions Technology Defined*. Available: https://www.intel.com/content/www/us/en/support/processors/000005779.html [Online; accessed 20-June-2017].

[19] *Introduction to Intel Advanced Vector Extensions*. Available: https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions [Online; accessed 20-June-2017].

[20] A. Venkat and D. M. Tullsen, "Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, pp. 121–132, Minneapolis, MN, 14-18 June 2014.

[21] R. Rico, J.-I. Pérez, and J. A. Frutos, "The impact of x86 instruction set architecture on superscalar processing," *Journal of Systems Architecture*, vol. 51, no. 1, pp. 63–77, January 2005.

[22] A. S. Waterman, *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California, Berkeley, 2016.

[23] *Guide to x86-64*. Available: https://web.stanford.edu/class/cs107/guide_x86-64.html [Online; accessed 20-January-2017].

[24] *Evolution of the ARM Architecture*. Available: http://www.eit.lth.se/fileadmin/eit/courses/eitf20/ARM_RG.pdf [Online; accessed 1-June-2017].

[25] *Q4/FY2016 Roadshow Slides*. Available: https://www.arm.com/company/investors [Online; accessed 1-June-2017].

[26] A. Shah, *Qualcomm's new chip may be too late as ARM server market fades*. Available: http://www.pcworld.com/article/3148251/data-center-cloud/qualcomms-new-chip-may-be-too-late-as-arm-server-market-fades.html [Online; accessed 1-June-2017].

[27] *NEON*. Available: https://developer.arm.com/technologies/neon [Online; accessed 20-June-2017].

[28] *64-bit ARM (Aarch64) Instructions Boost Performance by 15 to 30% Compared to 32-bit ARM (Aarch32) Instructions*. Available: http://www.cnx-software.com/2016/03/01/64-bit-arm-aarch64-instructions-boost-performance-by-15-to-30-compared-to-32-bit-arm-aarch32-instructions/ [Online; accessed 1-June-2017].

[29] *Introduction to ARMv8 64-bit Architecture*. Available: https://quequero.org/2014/04/introduction-to-arm-architecture/ [Online; accessed 1-June-2017].

[30] A. Prakash, *A study of the Alpha 21364 Processor*. Available: http://www.cs.utah.edu/~arul/projects/alpha.pdf [Online; accessed 20-January-2017].

[31] R. E. Kessler, "The Alpha 21264 Microprocessor," *IEEE micro*, vol. 19, no. 2, pp. 24–36, March/April 1999.

[32] *The evolution of the power workstation*. Available: http://triosdevelopers.com/jason.eckert/blog/Entries/2015/2/18_The_evolution_of_the_power_workstation.html [Online; accessed 1-June-2017].

[33] *Alpha Assembly Language Guide*. Available: https://www.cs.cmu.edu/afs/cs/academic/class/15213-f98/doc/alpha-guide.pdf [Online; accessed 20-January-2017].

[34] D. Bhandarkar, "RISC versus CISC: A Tale of Two Chips," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 1, pp. 1–12, March 1997.

[35] D. Bhandarkar and D. W. Clark, "Performance from Architecture: Comparing a RISC and a CISC With Similar Hardware Organization," vol. 19, no. 2, pp. 310–319, April 1991.

[36] C. Isen, L. K. John, and E. John, "A Tale of Two Processors: Revisiting the RISC-CISC Debate," in *Proceedings of SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pp. 57–76, Springer-Verlag, 2009.

[37] D. Ye, J. Ray, C. Harle, and D. Kaeli, "Performance Characterization of SPEC CPU2006 Integer Benchmarks on x86-64 Architecture," in *Proceedings of IEEE International Symposium on Workload Characterization (ISWC)*, pp. 120–127, San Jose, CA, October 2006.

[38] J. Goodacre and A. N. Sloss, "Parallelism and the ARM Instruction Set Architecture," *IEEE Computer*, vol. 38, no. 7, pp. 42–50, July 2005.

[39] V. M. Weaver and S. A. McKee, "Code Density Concerns for New Architectures," in *IEEE International Conference on Computer Design (ICCD)*, pp. 459–464, Lake Tahoe, CA, February 2009.

[40] E. Blem, J. Menon, T. Vijayaraghavan, and K. Sankaralingam, "ISA Wars: Understanding the Relevance of ISA being RISC or CISC to Performance, Power, and Energy on Modern Architectures," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 1, pp. 3:1–3:34, March 2015.

[41] E. Blem, J. Menon, and K. Sankaralingam, "A Detailed Analysis of Contemporary Arm and x86 Architectures," *UW-Madison Technical Report*, 2013.

[42] V. M. Weaver, *ll: Exploring the Limits of Code Density*. Available: `http://web.eece.maine.edu/~vweaver/papers/iccd09/ll_document.pdf` [Online; accessed 1-June-2017].

[43] H. Lozano and M. Ito, "Increasing the Code Density of Embedded RISC Applications," in *Proceedings of IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 182–189, York, UK, May, 2016.

[44] R. Durán and R. Rico, "Quantification of ISA Impact on Superscalar Processing," in *IEEE International Conference on Computer as a Tool, EUROCON*, vol. 1, pp. 701–704, Belgrade, Serbia, November 2005.

[45] B. C. Lopes, L. Ecco, E. C. Xavier, and R. Azevedo, "Design and Evaluation of Compact ISA Extensions," *Microprocessors and Microsystems*, vol. 40, pp. 1–15, 2016.

[46] S. Bartolini, R. Giorgi, and E. Martinelli, "Instruction Set Extensions for Cryptographic Applications," in *Springer Cryptographic Engineering*, pp. 191–233, 2009.

[47] R. B. Lee, "MULTIMEDIA EXTENSIONS FOR GENERAL-PURPOSE PROCESSORS," in *IEEE Workshop on Signal Processing Systems.*, pp. 9–23, Leicester, UK, November 1997.

[48] N. T. Slingerland and A. J. Smith, "Multimedia Extensions for General Purpose Microprocessors: A Survey," *Elsevier Microprocessors and Microsystems*, vol. 29, no. 5, pp. 225–246, June 2005.

[49] A. Jundt, A. Cauble-Chantrenne, A. Tiwari, J. Peraza, M. A. Laurenzano, and L. Carrington, "Compute Bottlenecks on the New 64-bit ARM," in *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing*, no. 6, Austin, TX, November 2015.

[50] K. Mayank, H. Dai, J. Wei, and H. Zhou, "Analyzing Graphics Processor Unit (GPU) Instruction Set Architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 155–156, Philadelphia, PA, March 2015.

[51] I.-J. Huang and A. Despain, "Synthesis of Application Specific Instruction Sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 6, pp. 663–675, June 1995.

[52] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution Migration in a Heterogeneous-ISA Chip Multiprocessor," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 261–272, March 2012.

[53] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, p. 29, Bordeaux, France, April 2015.

[54] C. Celio, P. Dabbelt, D. Patterson and K. Asanović, "The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V," Tech. Rep. UCB/EECS-2016-130, Electrical Engineering and Computer Sciences, University of California at Berkeley, July 2016.

[55] *The RISC-V Instruction Set Architecture*. Available: https://riscv.org/ [Online; accessed 1-June-2017].

[56] J. Engblom, *Does ISA Matter for Performance?* Available: http://jakob.engbloms.se/archives/1801 [Online; accessed 1-June-2017].

[57] *RISC vs. CISC: the Post-RISC Era*. Available: http://archive.arstechnica.com/cpu/4q99/risc-cisc/rvc-6.html [Online; accessed 1-June-2017].

[58] A. Akram and L. Sawalha, "The Impact of ISAs on Performance," in *Workshop on Duplicating, Deconstructing and Debunking (WDDD) co-located with 44th International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, June 2017. Available: https://drive.google.com/file/d/0By_qFMnj-9GLRXZPYm8yWDh4TE0/view [Online; accessed 1-July-2017].

[59] 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," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

[60] G. Black, N. Binkert, S. K. Reinhardt, and A. Saidi, "Modular ISA-Independent Full-System Simulation," in *Processor and System-on-Chip Simulation*, pp. 65–83, Springer, 2010.

[61] A. Akram and L. Sawalha, "x86 Computer Architecture Simulators: A Comparative Study," in *IEEE International Conference on Computer Design (ICCD)*, pp. 638–645, Pheonix, AZ, October 2016.

[62] K. Hoste and L. Eeckhout, "Microarchitecture-independent Workload Characterization," *IEEE Micro*, vol. 27, no. 3, pp. 63–72, August 2007.

[63] A. Fog, *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, [Online, accessed 3 September, 2015].

[64] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pp. 475–486, Tel-Aviv, Israel, June 2013.

[65] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation," in *ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1 – 12, Seattle, WA, 2011.

[66] Zsim, *Zsim Tutorial Validation*, 2015. Available: http://zsim.csail.mit.edu/tutorial/slides/validation.pdf [Online; accessed 5-June-2017].

[67] *ISA POWER STRUGGLES*. Available: http://research.cs.wisc.edu/vertical/wiki/index.php/Isa-power-struggles/Isa-power-struggles [Online; accessed 5-June-2017].

[68] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, December 1995.

[69] *Products formerly Haswell*. Available: https://ark.intel.com/products/codename/42174/Haswell [Online; accessed 1-June-2017].

[70] *Introuction to Intel Architecture*. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf [Online; accessed 1-June-2017].

[71] *INTEL ATOM PROCESSORS*. Available: https://www.intel.com/content/www/us/en/products/processors/atom.html [Online; accessed 1-June-2017].

[72] *Cortex-A15*. Available: https://developer.arm.com/products/processors/cortex-a/cortex-a15 [Online; accessed 1-June-2017].

[73] *Cortex-A8*. Available: https://developer.arm.com/products/processors/cortex-a/cortex-a8 [Online; accessed 1-June-2017].

[74] *Alpha 21164 Microprocessor Data Sheet*. Available: http://www.cs.cmu.edu/afs/cs/academic/class/15740-f03/public/doc/alpha-21164-data-sheet.pdf [Online; accessed 1-June-2017].

[75] *Intel's Haswell CPU Micrarchitecture*. Available: http://www.realworldtech.com/haswell-cpu/ [Online; accessed 1-June-2017].

[76] *Intel's Haswell Architecture Analyzed: Building a New PC and a New Intel*. Available: http://www.anandtech.com/show/6355/intels-haswell-architecture/6 [Online; accessed 1-June-2017].

[77] *Inside Atom Architecture*, 2015. http://www.hardwaresecrets.com/inside-atom-architecture/2/ [Online; accessed 1-June-2017].

[78] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[79] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of Error in Full-System Simulation," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 13–22, Monterey, CA, March 2014.

[80] F. A. Endo, D. Couroussé, and H.-P. Charles, "Micro-architectural Simulation of Embedded Core Heterogeneity with Gem5 and McPAT," in *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pp. 7:1–7:6, Amsterdam, Holland, January 2015.

[81] *ARM Cortex-A8*. Available: http://www.7-cpu.com/cpu/Cortex-A8.html [Online; accessed 5-June-2017].

[82] R. E. Kessler, E. J. McLellan, and D. A. Webb, "The Alpha 21264 Microprocessor Architecture," in *Proceedings of International Conference on Computer Design: VLSI in Computers and Processors*, pp. 90–95, Austin, Tx, 5-7 October 1998.

[83] Z. Cvetanovic and D. Bhandarkar, "Performance characterization of the alpha 21164 microprocessor using tp and spec workloads," in *Proceedings of Second IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 270–280, San Jose, CA, February 1996.

[84] *SPEC CPU 2006*. Available: `https://www.spec.org/cpu2006/` [Online; accessed 1-June-2017].

[85] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *IEEE 4th Annual Workshop on Workload Characterization*, pp. 3–14, Austin, TX, December 2001.

[86] *crosstool-NG*. Available: `http://crosstool-ng.github.io/` [Online; accessed 1-June-2017].

[87] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using Simpoint for Accurate and Efficient Simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, pp. 318–319, June 2003.

[88] *M5ops*. Available: `http://gem5.org/M5ops` [Online; accessed 1-June-2017].

[89] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A Call Graph Execution Profiler," in *ACM Sigplan Notices*, vol. 17, pp. 120–126, 1982.

[90] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 1–17, September 2006.

[91] B. Solomon, A. Mendelson, R. Ronen, D. Orenstien, and Y. Almog, "Micro-operation Cache: A Power Aware Frontend for Variable Instruction Length ISA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 5, pp. 801–811, 2003.