The Impact of ISAs on Performance

Ayaz Akram and Lina Sawalha

Department of Electrical and Computer Engineering,
Western Michigan University, Michigan, USA
{ayaz.akram, lina.sawalha}@wmich.edu

ABSTRACT

Recent advances in different ISAs (instruction set architectures) and the way those ISAs are implemented have revived the debate on the role of ISAs in overall performance of a processor. Many people believe that with the advances in compilers and microarchitectures, the choice of ISAs does not remain a decisive matter anymore, while others 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 already been introduced. In this work, we evaluate applications' behavior using different RISC (Reduced Instruction Set Computers) and CISC (Complex Instruction Set Computers) ISAs with different microarchitectures. We correlate performance differences for the same applications across ISAs to certain ISA features. Our work shows that instruction set architectures can affect the overall performance of applications.

1. INTRODUCTION

Instruction Set Architecture (ISA) serves as an abstraction layer between the hardware and the software of a computer system. Most modern ISAs can be classified into two classes: RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer). In this work, we explore the impact of ISAs on performance of an application using a specific microarchitecture. We chose to study and compare three different ISAs: 64-bit ARM (ARM-v8), x86-64 and Alpha. We compared the behavior of the ISAs using six different microarchitectures for different benchmarks.

Fundamentally, the distinguishing features that characterize the CISC and RISC ISAs are given in Table 1. Processor technology has been in continuous evolution and changed significantly since the introduction of RISC ISAs. Microarchitectures (hardware implementation of ISAs) are continuously being optimized as well. 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 decode complex instructions into simpler RISC-like instructions, called micro-operations (μ-ops), to make pipelining feasible. As Moore's law continues to hold, more transistors can fit in a single chip, giving RISC architectures the opportunity to incorporate more complicated CISC-like instructions. The present time is viewed as "post-RISC" era, implying that the current architectures are neither fundamentally CISC nor RISC [1].

Table 1: Features of CISC and RISC ISAs

CISC	RISC
Complex instructions	Simple instructions
Emphasis on hardware	Emphasis on software
Can incorporate load and store in instructions	Load and store are independent instructions
Smaller code size	Large code size
Variable length instructions	Fixed length instructions
High number of addressing modes	Limited number of addressing modes
Complex encoding of instructions	Simple encoding of instructions
Have specialized instructions	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

1.1 Overview of Instruction Sets Under Analysis

This section provides an overview of the features of the ISAs studied in this paper: x86-64, ARMv8, and Alpha.

1.1.1 x86-64

x86-64 instruction set is a 64-bit version of x86 instruction set (CISC ISA), which is largely used in desktops and servers, and recently in mobile devices. Originally the 64-bit version of x86 ISA was introduced by AMD in 2000, but it is used by both AMD and Intel currently. x86-64 is characterized by variable length complex instructions ranging from 1 byte to 15 bytes in size. Being CISC in nature, x86-64 is considered to have higher code density than its RISC counterparts resulting in lower static code size. Instructions in x86 architectures are decoded into simple μ-ops at run-time. This decoding provides higher opportunity for instruction-level parallelism (ILP) and has become necessary due to high ILP demands of modern deep pipeline microarchitectures. x86-64 provides SIMD support through SSE/AVX extensions. The x86-64 instruction set has sixteen 64-bit registers for integer operations and sixteen 128-bit registers for floating-point and SIMD operations. x86-64 supports absolute memory addressing, sub-register ad-

dressing and register-to-register spills. Such features normally lead to lower register pressure [2]. x86 uses implicit operands for various instructions. For example, the destination operand of a multiply instruction (MUL) is an implied operand located in register AL, AX or EAX. The use of implicit operands results into extra dependencies in some cases, leading to negative impact on available parallelism [3].

1.1.2 ARMv8

The 64-bit variant of ARM architecture (ARMv8) targets low power servers market, along with embedded systems. 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. Overall the ISA is complex and supports more than one thousand instructions. ARMv8 has a fixed sized 32-bit long instructions. ARMv8 supports 8 different addressing modes, but still remains to be a load/store architecture, i.e. instruction operands cannot be values residing in memory. It has 31 64-bit general purpose registers. ARMv8 does not support the compact Thumb instruction encoding. SIMD processing is supported through NEON extensions. In fact, NEON is mandatory for ARMv8 and no software floating-point ABI (Application Binary Interface) is provided. ARMv8 is compact but does not compete well in code size with ISAs that have variable-length instructions [4]. Unlike ARMv7, an integer division instruction has been added. ARMv8 instructions are supposed to boost performance by 15% to 20% in comparison to ARMv7 instructions [5].

1.1.3 Alpha

Designed by Digital Equipment Corporation in the early 1990s, Alpha is another 64-bit RISC ISA. Alpha was designed for high-performance systems and was simple to implement. Alpha has fixed size instructions; 32 bits long with six different instruction formats. It supports 32 integer and 32 floating point registers, each register is 64 bits long. It does not support any compressed ISA extension. Alpha follows an imprecise floating-point trap model. 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. To support SIMD operations, an extension was added to the ISA, called Motion Video Instructions (MVI) [6]. MVI is composed of simple instructions that operate on integer data types. When Compaq purchased DEC in the late 1990s, they discontinued the development of Alpha in favor of Intel's Itanium. As a result Alpha ISA has died out now. Last implementation of Alpha was developed in 2004.

2. RELATED WORK

Most of ISA studies [7, 8, 9, 10] are old and do not include state-of-the-art developments in ISAs and their implementations. There are other studies, which either focus on only one ISA [11] or target only particular ISA features [12]. Two recent ISA studies performed by Venkat and Tullsen [2], and Blem et al. [13] have conflicting claims regarding the role of ISA in performance of a processor. Venkat and Tullsen [2] suggest that ISAs can affect performance significantly based on their features. On the other hand, Blem et al. [13] conclude that microarchitecture is the main reason for performance differences across different platforms and ISA effects are indistinguishable. Our work focuses on 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 [7, 8]. 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. [9] performed a comparison of performance between Power5+ and Intel Woodcrest, finding that both match in performance. They indicated that with the aggressive microarchitectural techniques CISC ISAs can have similar performance as RISC ISAs.

In a recent study Blem et al. [14] provide a detailed analysis of x86 and ARM ISAs. The authors claim that the different ISAs are optimized for different performance gains and no ISA is fundamentally more energy efficient than the other. Blem et al. suggest that the primary reason for performance differences is microarchitecture. Later, they included MIPS ISA and some other hardware test platforms in their study to conclude that the previous findings still hold true [13].

Weaver and Mckee [12, 15] studied the effect of ISA on code density. Their study includes more than 20 ISAs (including x86_64, ARM64 and Alpha). Their work finds x86 to be one of the most dense ISAs. 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.

Duran and Rico used graph theory techniques to quantify the impact of ISAs on superscalar processing [16]. Rico et al. also studied the impact of x86 specifically on superscalar processing [3]. 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. [11] 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. Ye et al. [10] characterized the performance of different x86-64 applications and compared them to those of 32-bit x86 applications. They showed that for integer benchmarks, 64-bit binaries perform better than 32-bit ones by an average of 7%. However, this is not true for all benchmarks, as some perform slower in 64-bit mode. They show that memory-intensive benchmarks, which use long and pointer data types extensively suffer from performance degradation in a 64-bit mode.

Lopes et al. evaluated different compact ISA extensions including Thumb2 and MicroMIPS [17]. They also proposed SPARC-16, a 16-bit extension to SPARC processor. Lee [18] provides an overview of various multimedia extensions for different ISAs (e.g. MAX, MMX, VIS). Similarly, Slingerland and Smith [19] surveyed existing multimedia instruction sets and examined the mapping of their functionality to a set of computationally important kernels. Bartolini et al. [20] analyzed various existing instruction set exten-

sions for cryptographic applications. They reviewed the associated benefits and limitations of such extensions. Ing and Despain [21] researched instruction sets designed for application specific needs and have a tighter integration with the underlying hardware. They outlined automatic instruction set generation for these application specific designs. This technique is called Automatic Synthesis of Instruction Set Architectures (ASIA). It outperforms manually designed instruction sets. However, it has some limitations (e.g. the need of hardware resources specifications by the designer).

DeVuyst et al. [22] developed a mechanism to migrate program execution on cores of different ISAs with minimum cost. Based on this execution migration methodology, Venkat and Tullsen [2] have 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 as compared to a single-ISA heterogeneous CMP.

Celio et al. [23] compared RISC-V [24], 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 at run time (macro-op fusion). The authors claim that using microarchitectural techniques, such as macro-op fusion, can make one ISA targeted for both high-end and low-end processors.

Steve Terpe researched the historical "RISC vs CISC" debate in [25]. 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. Terpe concluded that it does not really matter for an ISA to be RISC or CISC or a combination of both. Instead, other technology developments (like caching, pipelining, register renaming) play more significant role towards determining the overall performance of a system. Jakob [26] argues that 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 cleaner instruction set. Jon Stokes [27] 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. Instead, current platforms should be evaluated on their own merits.

3. METHODOLOGY

To compare and analyze the behavior of ISAs, we need to: keep all ISA-independent microarchitectural features the same across all ISAs for all runs, try a diverse set of microarchitectural configurations close to real implementations of the studied ISAs, keep the compilation infrastructure the same for all ISAs, and study the same phases of execution across all ISAs. We used gem5 [28] simulator for our experiments, which ensured that we were analyzing the behavior of different ISAs using the exact same microarchitectures. gem5 isolates the underlying simulated hardware (microarchitecture) from ISAs [29], making it a good fit for our study. We modified the source code of gem5 to make x86 instructions to μ-ops decoding more realistic.

We chose a diverse set of microarchitectures for our study, including three OoO (out-of-order) and three IO (in-order) cores. The simulated cores are based on Intel Haswell (OoO), Intel Atom (IO), ARM Cortex A15 (OoO), ARM Cortex A8 (IO), Alpha 21264 (OoO) and Alpha 21164 (IO). Detailed configurations of the selected microarchitectures are shown in Table 2.

We used C/C++ benchmarks of SPEC-CPU2006 benchmarks [30] and embedded benchmarks from MiBench [31] suite in our study, compiled for each ISA. We used gnu gcc version 4.8.5 to compile these benchmarks for x86 and ARM and 4.3.5 for Alpha, instead of using any vendor-specific compiler, to control compiler optimizations used. We built the gcc cross compilers using *crosstools-ng* version 1.22. While SPEC-CPU2006 benchmarks do not have SIMD code, the auto-vectorization feature of gcc can result in SIMD instructions in the compiled binary. We did not disable the auto-vectorization feature of the compiler.

Following is a brief summary of the methodology adopted to carry out all the experiments: We used simpoint tool [32] with x86 binaries of SPEC-CPU2006 benchmarks and came up with 5 simpoint intervals each of approximately 500 million x86 instructions. To map these x86 phases to other ISAs, we marked critical functions of the benchmarks. We also profiled SPEC-CPU2006 benchmarks using gprof [33] tool to identify critical functions. Four functions for each program were chosen 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. We inserted gem5 pseudo instructions in all four functions to mark the functions for each benchmark and ran the marked binaries to calculate the total number of marked-function calls at the starting and ending point of each phase for x86. This call count was used to identify the exact start point and end point of simulation for each phase on x86 and other ISAs. 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 phase boundary. In case of embedded benchmarks, we ran the entire benchmarks for all ISAs. We measured different microarchitectural performance statistics like cache misses, branch mispredictions, blocks of different stages etc. and some microarchitecture independent statistics like register dependency distance and instruction mixes for each phase for each ISA. All statistics were measured for windows of 50 000 instructions over time. Finally, we compared the differences in performance for various ISAs across these phases of execution.

Table 2: 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	90/256	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
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-$ size	8MB	N/A	N/A	N/A	N/A	4MB
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
Branch Predictor (Global Table/ 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						

4. RESULTS AND ANALYSIS

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

Figures 1 and 2 show the cycle counts for ARM and Alpha ISAs (relative to x86) for out-of-order (OoO) and in-order (IO) cores respectively. In case of SPEC-CPU2006 benchmarks, the cycle counts are cumulative cycles for all studied phases. On average, ARM takes the least number of cycles for all kinds of benchmarks on out-of-order cores. As the figures show, the ISAs behave differently for the various benchmarks and microarchitectures. For example, *gobmk* always takes less number of cycles on Alpha than x86 because x86 suffers from increased number of branch mispredictions. However, *perlbench* always takes less number of 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 less number of cycles than x86 on other out-of-order cores. In case of in-order cores, the difference in execution cycles for x86 and ARM is reduced significantly for *hmmer*. Similarly, the average difference is reduced on IO cores.

Figures 3 to 7 show various microarchitecture-independent metrics. Figure 3 shows the dynamic instruction counts for Alpha and ARM ISAs normalized to x86 instructions. Figure 4 shows the dynamic μ-op counts for Alpha and ARM ISAs normalized to x86 μ-op counts for all benchmarks. As the figures show, the final number of dynamic μ-ops depends on the ISA, and x86 has the most number of μ-ops in most cases. Figure 5 shows the number of different types of operations for all ISAs relative to x86. As the figures show, the total number of μ-ops of a particular type depends on the ISA as well. Figure 6 shows the average probability of register dependency distance (the number of instructions between the instruction that writes a register and the instruction that reads that register) [34]. Figure 7 shows the average values for degree of use of registers (number of instructions that consume the value of a register once it is written) [34]. x86 has the highest degree of use of registers which results into extra instruction queue blockings in out-of-order cores for some cases as compared to other ISAs. To better understand specific reasons for the differences among ISAs we studied each benchmark separately. We show some examples below.

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 most number of μ-ops to execute this benchmark as well. Another interesting thing to observe is the behavior of this benchmark over time. Figure 8 shows the number of cycles taken to execute each interval of 50,000 instructions for all ISAs on a 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 the following phases x86 takes a larger number of cycles in comparison to the other ISAs because of increased dependent operations. Listing 1

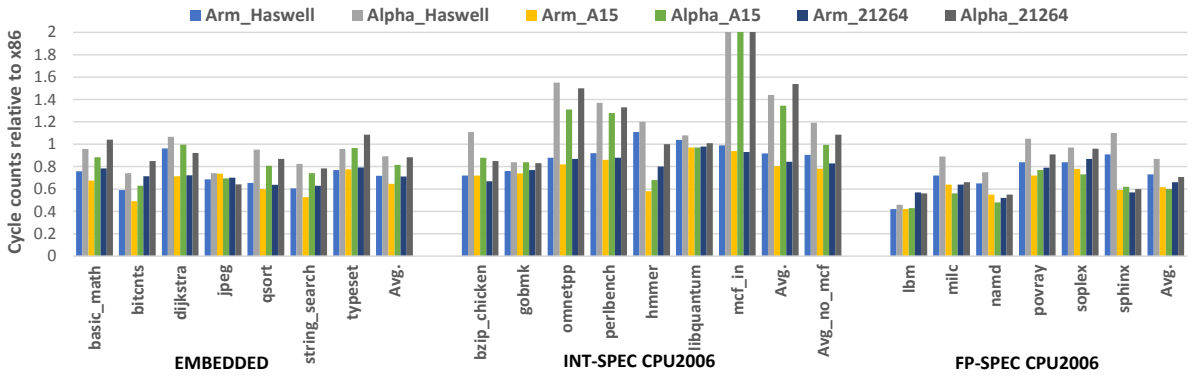

Figure 1: Cycle counts normalized to x86 for OoO Cores

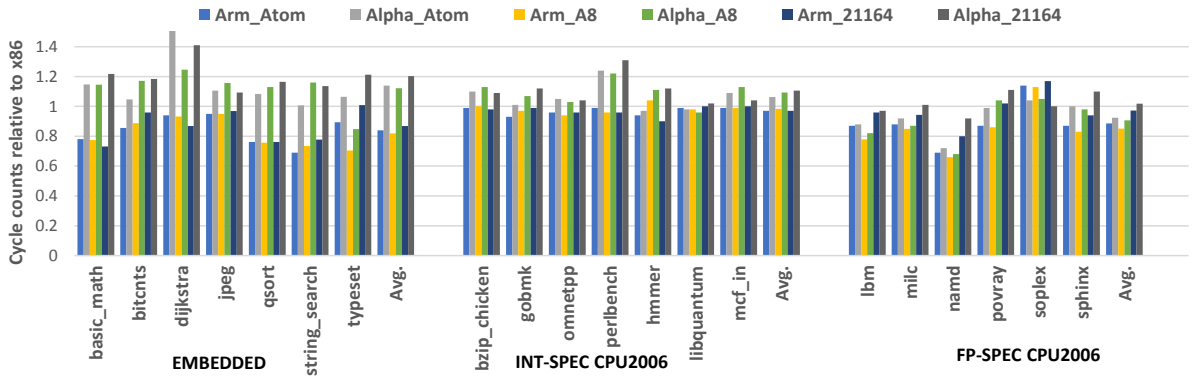

Figure 2: Cycle counts normalized to x86 for IO Cores

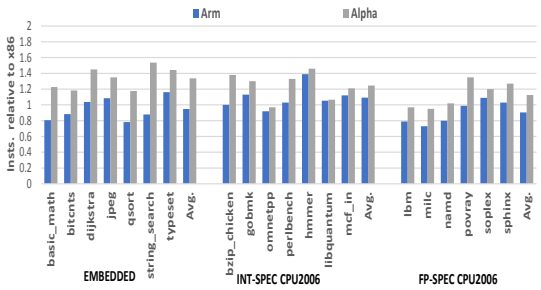

Figure 3: Instruction counts normalized to x86

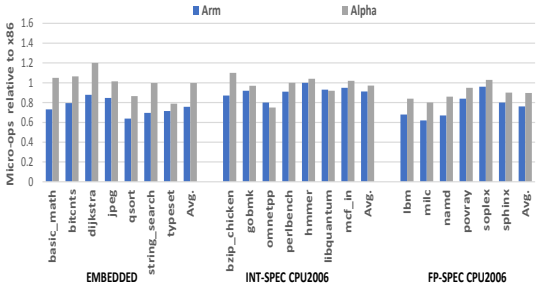

Figure 4: μ-op counts normalized to x86

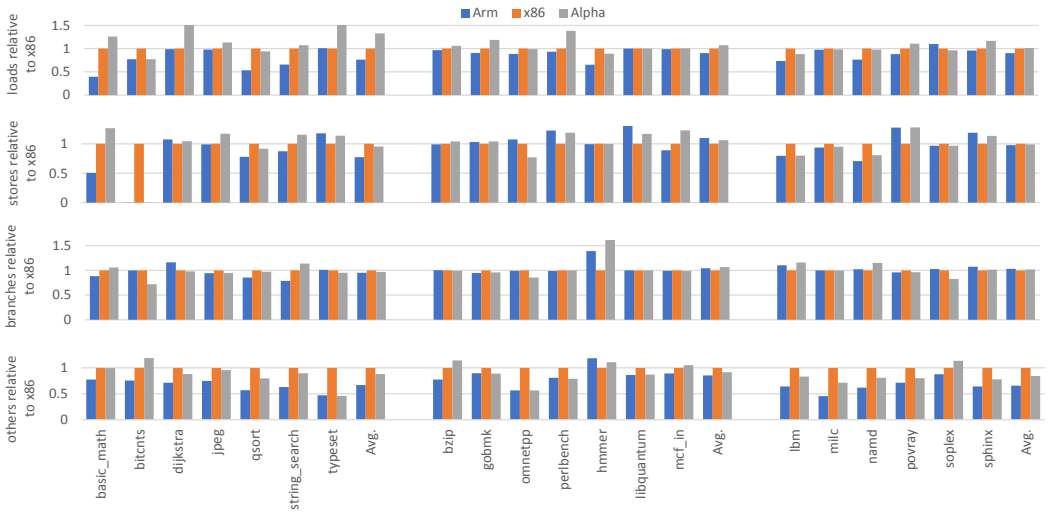

Figure 5: Types of μ-ops normalized to x86

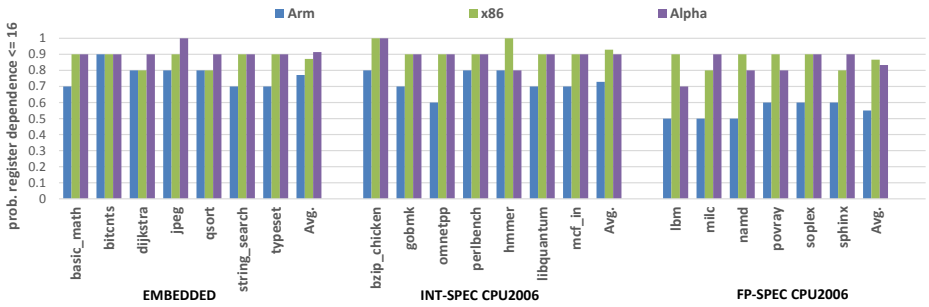

Figure 6: Register dependency distance for each ISA

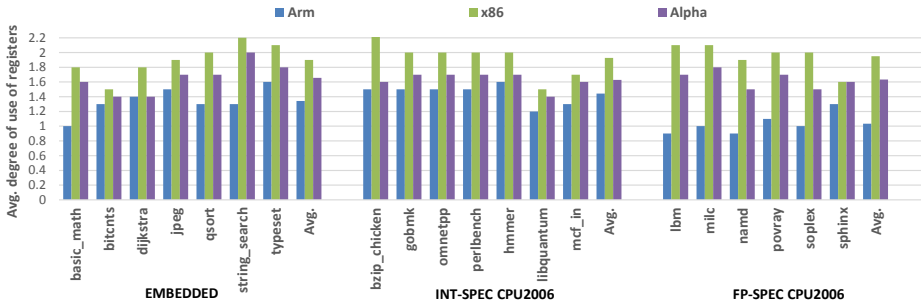

Figure 7: Degree of use of registers for each ISA

shows an example of assembly code that leads to reduced performance on aggressive out-of-order cores for x86 compared to the other ISAs for *bitcnt*.

Listing 1: 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
```

This code snippet is taken from a function 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 8. As can be seen in the Listing 1, 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 the out-of-order cores as our results indicate.

For *dijkstra* benchmark, Alpha suffers from high register pressure and has a greater number of load operations as compared to the other ISAs as shown in Figure 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.

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/instruction count for Alpha. Figure 9 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, during the second phase, x86 spends the highest number of cycles for each window of instructions.

bzip2 is a SPEC-CPU2006 integer benchmark, which performs the compression and decompression of an input file. While Alpha takes the most number of cycles on a Haswell-like microarchitecture and less on the other out-of-order cores on average, this behavior is not true for all phases. For example, in one of the phases Alpha always performs the worst. 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

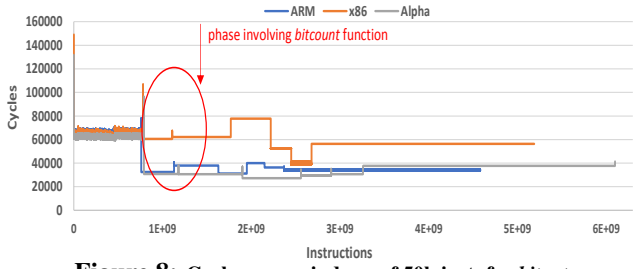

Figure 8: Cycles over windows of 50k inst. for *bitcnts*

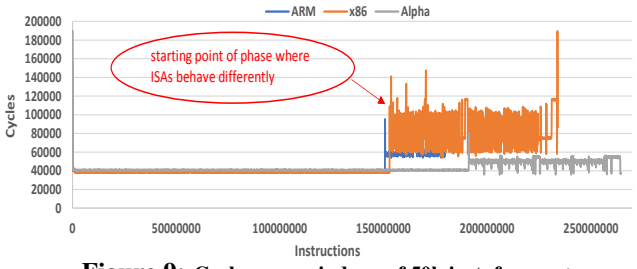

Figure 9: Cycles over windows of 50k inst. for *qsort*

required to clear the upper 32 bits of the index variables as pointed out in Listing 2. The particular phase, in which Alpha performs the worst, uses these functions frequently. The same behavior was also observed by Celio et al. in [23].

In case of *gobmk*, x86 suffers from a high number of branch mispredictions in all out-of-order cores and thus takes a larger number of cycles in comparison to the other ISAs. Interestingly, Alpha and x86 take the same number of cycles on one of the 5 studied phases, even in case of out-of-order cores. During this particular phase, Alpha takes almost 12% more dynamic μ-ops as compared to x86.

Listing 2: 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>
```

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 due to benchmark's nature. However, Alpha has significantly larger number of stack operations as 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 as compared to the other ISAs. This results into lower performance of Alpha on all phases in comparison to the other ISAs.

Listing 3: 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 addr. mode
add    0x0(%r13,%rax,4),%r15d  //complex addr. 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)
```

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 as compared to x86. *P7Viterbi* is a function where *hmmer* spends most of the time and contains many 'if' statements around store operations. An example of such statements with corresponding assembly instructions for all ISAs is shown in Listing 3. As can be seen in the Listing 3, x86 makes use of complex addressing modes resulting into less number of instructions (and also μ-ops) as 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 Alpha21264-like cores.

Figure 10 shows the behavior of ISAs on one of the phases

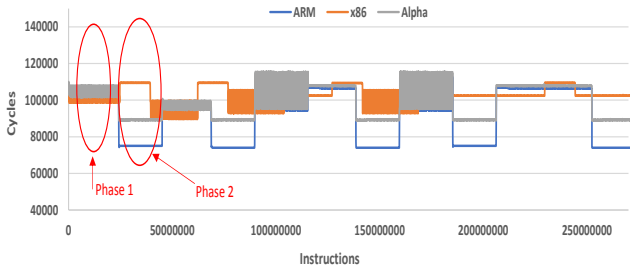

Figure 10: **Cycles over windows of 50k inst. for a phase of *libquantum***

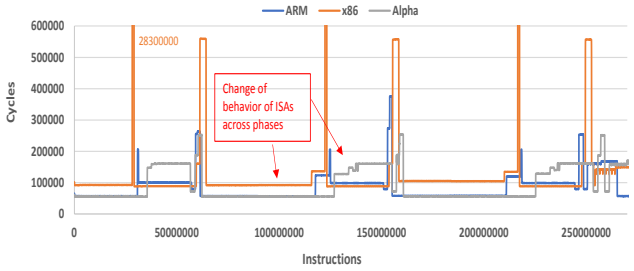

Figure 11: **Cycles over windows of 50k inst. for a phase of *milc***

of *libquantum*. There are two main sub-phases in this phase, which are circled in Figure 10. While on Phase 1 all ISAs show similar behavior, on Phase 2 the behavior of ISAs is very different from each other.

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 a 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. x86 makes use of complex addressing mode 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 5. An example of higher register pressure is observed at the end point of *primal_iminus*, the second most called function, where Alpha restores (or loads) double the number of callee-saved registers compared to the other ISAs.

lbm is another memory intensive benchmark, where x86 takes a larger number of cycles compared to the other ISAs. ARM and Alpha have significantly less number of μ-ops compared to x86. Listing 5 shows an example from the most critical function, *LBM_perf- ormStreamCollide*. Since this code contains several additions with many intermediate sums, this piece of code requires several registers, x86 spills some registers onto stack and later use them for addition. This finding is similar to Venkat et al. findings [2]. Although x86 takes a high number of cycles compared to the other ISAs on average, all ISAs take a similar number of cycles for one of the phases.

Listing 4: 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]
```

milc simulates 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 6. The Listing shows the code from the function where it spends most of the time, *mult_su3_na*. One of the main sources of extra instructions and dependent operations is two copying operations, which are pointed out in Listing 6. The registers containing values of *ai* and *ar* will not retain their values after performing multiplication in the 3rd line of the C/C++ code, so these are copied to two other registers to use their values for multiplication operations shown in line 4 of the C/C++ code. While on average Alpha is doing better than x86 this is not true on all phases. The cycles for one of the 5 phases is shown in Figure 11. There are two main sub-phases in this phase as shown in the figure. x86 and Alpha take the most number of cycles in those phases alternatively.

Listing 5: 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
```

On *namd*, x86 takes a higher number of μ-ops compared to 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 results in a higher number of μ-ops which hurts its performance in comparison to the other ISAs. On *povray*, Alpha takes the highest number of cycles for most of the configurations.

As shown in the examples above, ISAs can affect the performance of a benchmark depending on the microarchitecture. Next, we have a summary of findings based on the aforementioned examples and results.

Listing 6: 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
```

Summary of the Findings:

1. On average, ARMv8 outperforms other ISAs on similar microarchitectures, as it offers better instruction-level parallelism and has a lower number of dynamic μ-ops compared to the other ISAs for most 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 [2].
3. The performance differences across ISAs are significantly reduced in IO cores compared to OoO cores.
4. On average, x86 has the highest number of dynamic μ-ops. This agrees with previous findings when compared to Alpha [2]. There are few examples where Alpha exceeds x86 in the number of μ-ops, but ARMv8 always has a lower or equal number of μ-ops compared to x86.
5. x86 seems to have over-serialized code due to ISA limitations, such as implicit operands and overlapping source and destination registers, as was also observed by Rico et al. [3]. x86 has the highest average degree of use of registers.
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 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 IO cores, but the numbers can vary significantly in case of OoO cores. One possible explanation for this behavior is because timing of cache accesses for OoO cores does not match that for all ISAs (based on the dependencies in the dynamic instruction sequence for that ISA), which can lead to a different number of L1 data cache misses.
8. On average, the number of branch mispredictions are similar across ISAs for all of the microarchitectures. However, there are few exceptions like *gobmk*, *qsort* and *povray*. These benchmarks having a high number of branch operations, and differing in the exact number of branch operations across the ISAs (Figure 5), can explain the aforementioned behavior.
9. From our study, we found that μ-ops to instructions ratio on x86 is usually less than 1.3. This was also observed by Blem et al [13]. However, overall instructions count and mixes are ISA-dependent, which contradicts Blem et al's [13] conclusion of instruction counts being independent of ISAs.
10. Changing microarchitecture significantly affects performance more than changing ISA on a particular microarchitecture. For example, going from a Haswell-like core to an A15-like core affected performance more than an ISA change did on one of the two cores.
11. According to Blem et al's study [13], performance differences on studied platforms are mainly because of microarchitectures. We observed performance differences on exactly similar microarchitectures, which me-ans that ISAs are responsible for those performance

differences. Moreover, since performance differences across ISAs vary for different microarchitectures, we can conclude that the behavior of an ISA depends on the associated microarchitecture, but they certainly affect performance.

5. CONCLUSION

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. In this work, we studied the effect of three ISAs on performance of many benchmarks using six microarchitectures. We related the observed performance differences across ISAs to the generated assembly code for each ISA. Our results indicate that ISAs can affect performance and 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 of ISAs is insignificant for IO cores, compared to OoO cores.

Acknowledgement

The authors would like to thank Brandon Arrendondo and Tyler Bayne for their help with this work and the anonymous reviewers for their valuable feedback.

6. REFERENCES

[1] "risc vs. cisc." `https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/`. [acc. 6/1/2017].

[2] A. Venkat and D. M. Tullsen, "Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor," in *ISCA, 2014*, pp. 121–132, 14-18 June, Minneapolis, MN.

[3] 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, 2005.

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

[5] "64-bit ARM (Aarch64) Instructions Boost Performance by 15 to 30% Compared to 32-bit ARM (Aarch32) Instructions." `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/`. [acc. 6/1/2017].

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

[7] D. Bhandarkar, "RISC versus CISC: A Tale of Two Chips," *SIGARCH Comput. Arch. News*, vol. 25, no. 1, pp. 1–12, 1997.

[8] D. Bhandarkar and D. W. Clark, "Performance from Architecture: Comparing a RISC and a CISC With Similar Hardware Organization," in *SIGARCH Comput. Arch. News*, vol. 19, pp. 310–319, 1991.

[9] 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.

[10] D. Ye, J. Ray, C. Harle, and D. Kaeli, "Performance Characterization of SPEC CPU2006 Integer Benchmarks on x86-64 Architecture," in *Proceedings of IEEE ISWC*, 2006.

[11] B. C. Lopes, R. Auler, L. Ramos, E. Borin, and R. Azevedo, "SHRINK: Reducing the ISA Complexity via Instruction Recycling," *SIGARCH Comput. Arch. News*, vol. 43, pp. 311–322, June 2015.

[12] V. M. Weaver and S. A. McKee, "Code Density Concerns for New Architectures," in *IEEE ICCD*, pp. 459–464, 2009.

[13] 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*, vol. 33, pp. 3:1–3:34, Mar. 2015.

[14] E. Blem, J. Menon, and K. Sankaralingam, "A detailed analysis of contemporary arm and x86 architectures," *UW-Madison Technical Report*, 2013.

[15] V. M. Weaver, "ll: Exploring the limits of code density." `http://web.eece.maine.edu/~vweaver/papers/iccd09/ll_document.pdf`. [acc. 6/1/2017].

[16] 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, 2005.

[17] 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.

[18] R. B. Lee, "Multimedia extensions for general-purpose processors," in *IEEE Workshop on Signal Processing Systems.*, pp. 9–23, 1997.

[19] N. T. Slingerland and A. J. Smith, "Multimedia extensions for general purpose microprocessors: A survey," *Microprocessors and Microsystems*, vol. 29, no. 5, pp. 225–246, 2005.

[20] S. Bartolini, R. Giorgi, and E. Martinelli, "Instruction set extensions for cryptographic applications," in *Cryptographic Engineering*, pp. 191–233, Springer, 2009.

[21] 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, pp. 663–675, Jun 1995.

[22] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution Migration in a Heterogeneous-ISA Chip Multiprocessor," *SIGARCH Comput. Archit. News*, vol. 40, pp. 261–272, Mar. 2012.

[23] 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.

[24] "The RISC-V Instruction Set Architecture." `https://riscv.org/`. [acc. 6/1/2017].

[25] S. Terpe, "Why Instruction Sets No Longer Matter." `http://ethw.org/Why_Instruction_Sets_No_Longer_Matter`. [acc. 6/1/2017].

[26] J. Engblom, "Does ISA Matter for Performance?." `http://jakob.engbloms.se/archives/1801`. [acc. 6/1/2017].

[27] "RISC vs. CISC: the Post-RISC Era." `http://archive.arstechnica.com/cpu/4q99/risc-cisc/rvc-6.html`. [acc. 6/1/2017].

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Comput. Arch. News*, vol. 39, pp. 1–7, Aug. 2011.

[29] 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.

[30] "SPEC CPU 2006." `https://www.spec.org/cpu2006/`. [acc. 6/1/2017].

[31] 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, 2 December 2001.

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *SIGARCH Comp. Arch. News*, vol. 30, pp. 45–57, 2002.

[33] 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.

[34] K. Hoste and L. Eeckhout, "Microarchitecture-independent Workload Characterization," *IEEE Micro*, vol. 27, no. 3, 2007.