Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed

by Andrew S. Waterman

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

_____

Professor D. Patterson
Research Advisor

_____

(Date)

* * * * * * *

_____

Professor K. Asanović
Second Reader

_____

(Date)

Abstract

Delivering the instruction stream can be the largest source of energy consumption in a processor, yet loosely-encoded RISC instruction sets are wasteful of instruction bandwidth. Aiming to improve the performance and energy efficiency of the RISC-V ISA, this thesis proposes RISC-V Compressed (RVC), a variable-length instruction set extension. RVC is a superset of the RISC-V ISA, encoding the most frequent instructions in half the size of a RISC-V instruction; the remaining functionality is still accessible with full-length instructions. RVC programs are 25% smaller than RISC-V programs, fetch 25% fewer instruction bits than RISC-V programs, and incur fewer instruction cache misses. Its code size is competitive with other compressed RISCs. RVC is expected to improve the performance and energy per operation of RISC-V.

1   Introduction

Power dissipation and energy efficiency are primary design constraints for processors both simple and complex. For the simpler processor cores in mobile devices in particular, delivering the instruction stream is often the single largest source of energy consumption. In the DEC StrongARM-110, for example, instruction address translation and cache access account for 36% of the chip's power dissipation [10]. In a more recent study [5], instruction cache access alone dissipated 40% of the energy in a five-stage RISC pipeline. Main memory accesses and processor stalls incurred upon instruction cache misses consume more energy still.

Instruction set architects have broadly used two techniques to reduce the relative energy cost of instruction stream delivery. One approach is to increase the amount of work performed by a single instruction. Vector machines, for example, reduce instruction bandwidth demands by expressing a large amount of SIMD parallelism in a single instruction. CISC machines do so by combining multiple simple operations into a single instruction and providing more addressing modes.

An alternative approach is to reduce the size of the instructions. CISC instruction sets generally have been composed of variable-length instructions: the simpler and more common ones are usually encoded in fewer bits than those that require more operands or occur less frequently. RISC ISAs initially forewent the code density advantages of variable-length instruction encodings in favor of simpler, fixed-length 32-bit encodings. Since then, RISC instruction set extensions have provided fixed-length 16-bit encodings [1, 9], although often at the expense of performance and access to some hardware features. Variable-length RISC ISAs [2] can obviate these drawbacks by encoding the most common instructions densely, while maintaining most or all of the functionality of the 32-bit base ISA.

In this thesis, I propose RISC-V Compressed (RVC), a variable-length instruction encoding extension to RISC-V [17], a modern RISC instruction set. RVC's goal is to improve energy per operation by reducing instruction fetch traffic, cache misses, off-chip memory accesses, and stall cycles. To that end, RVC encodes instructions that occur frequently, either in the static program binary or in the dynamic instruction stream, in 16 bits, half of a RISC-V instruction word. I then evaluate RVC's effectiveness at both static code size compression and dynamic instruction fetch

traffic reduction, and compare the code size of RVC programs with those of several commercial ISAs. Finally, I discuss the implications of RVC for energy efficiency, performance, and processor design.

2 Background and Related Work

Before the microcomputer era, small address spaces and expensive memories limited program size, even in large systems. Hence, dense instruction encodings have long been a feature of many scalar instruction sets. Microcoded control made it simple to decode instructions of variable length, and the ISAs and microarchitectures of the CISC minicomputers, like the DEC VAX [15], reflected this design choice. Instructions that had fewer operands or used simpler addressing modes were encoded in fewer bytes, reducing code size as compared to a fixed-length encoding with the same features. The Intel x86 instruction set, a variable-length CISC in the same vein, remains popular today.

As the CISC minicomputer gave way to the RISC microprocessor [12], simpler, more regular instruction sets that simplified hard-wired control and pipelining rose to prominence. These RISC ISAs were generally rather loosely encoded, typically with a fixed 32-bit instruction word. Furthermore, since RISC instructions generally performed only simple operations, a given task might have taken more RISC instructions than, for example, VAX instructions. As a result, a program was typically expressed by many more RISC instruction bytes than VAX instruction bytes.

Although cheaper memories and integrated caches mitigated the larger instruction footprint of RISCs in workstations, code size continued to be a primary constraint in embedded systems. As pin count and memory size directly impact both cost and power consumption, a compact instruction encoding was particularly desirable in this domain. To meet the demands of this market, some RISC vendors extended their instruction sets to support a subset of their base ISA's functionality using short instruction words, typically 16 bits. These short instructions have a straightforward mapping to one or more instructions in the base ISA. MIPS16 [9] and ARM Thumb [1] are notable examples of this approach.

Processors that support MIPS16 or Thumb also support the corresponding base ISA (MIPS or ARM). At any given time, only the base ISA instructions or the short instructions can be executed; switching between the two ISA modes occurs only at procedure-call boundaries, using special control-flow instructions. Almost all instructions in the short ISA mode are 16 bits long. To encode the requisite functionality in this constrained opcode space, immediate operands are shortened,

and most instructions can only address eight general-purpose registers[1]. Additionally, not all ISA features are available: for example, neither MIPS16 nor Thumb procedures have direct access to the floating-point hardware.

Although these short instruction word RISC ISAs can reduce code size substantially, they do so at the expense of performance. The reduced register space results in more register spills to the stack, thereby increasing instruction count and cache misses, and the unavailability of some hardware features and addressing modes further increases instruction count. This tradeoff may be acceptable when code size is a primary constraint, but the reduced performance of these ISAs make them unattractive in other situations.

Variable-length RISC ISAs provide a compromise between the loosely-encoded base ISA and the denser, less-performant short instruction word ISA. Frequent instructions with few operands or small immediates are represented with a short instruction word. These short instructions can be mixed with full-length instructions that support most or all of the base ISA functionality. ARM Thumb-2 [2] exemplifies this approach, as does Heads and Tails [11], a variable-length RISC ISA designed for efficient superscalar instruction decoding.

An alternative to adding short instructions to an existing ISA is to employ more general compression techniques. Dictionary-based compression [6] schemes, for example, replace frequent instructions or instruction sequences with an index into a dictionary, which stores the decompressed instructions. Though these schemes are effective at reducing code size, the dictionary lookup adds latency and offsets the energy reduction from compression. Additionally, as the dictionary is a per-program data structure, it adds to the architectural state.

---

[1]By comparison, the base MIPS ISA has 32 general-purpose registers, and the base ARM ISA has 16.

3   RVC: A Variable-Length RISC ISA

Variable-length RISC ISAs can reduce static and dynamic code size as compared to their fixed-length counterparts, yet they avoid the performance loss of a less-capable ISA that comprises only short instructions. RVC, short for RISC-V Compressed, aims to leverage the performance and energy advantages of a variable-length encoding in the RISC-V ISA. This section describes the RISC-V base ISA and the RVC variable-length instruction extension.

3.1   The RISC-V Base ISA

RISC-V is a new ISA designed to support computer architecture research and education. RISC-V has been architected to be straightforward to implement while supporting many features of contemporary commercial instruction sets. It is inspired by previous RISC ISAs [3, 4, 8, 12–14], although its design omits many architectural features that favor particular microarchitectural styles at the expense of others, like delay slots and condition codes. This section describes the RISC-V programmer-visible state and instruction encoding; a full description can be found in [17].

RISC-V supports 32-bit and 64-bit address spaces. The user-level architected state includes a program counter and 32 fixed-point registers[2], all either 32 or 64 bits wide, coincident with the size of the address space. Additionally, the ISA has 32 64-bit floating-point registers. RISC-V is a load-store architecture: memory is accessed only via loads and stores, and computational instructions operate on the registers.

In the base ISA, all instructions are 32 bits long and must be naturally aligned. However, the ISA encoding supports variable-length instruction extensions. Branch and jump targets need only be 2-byte-aligned. All 32-bit instructions have their least significant two bits set to `11`; bit patterns `00`, `01`, and `10` are reserved for 16-bit instructions[3]. This scheme simplifies the task of determining the boundaries between the instructions in the instruction stream, which is particularly important for efficient superscalar instruction decoding.

---

[2]Fixed-point register `0` is hard-wired to zero.

[3]Additionally, opcode space has been reserved for instructions longer than 32 bits; these instructions have their least significant five bits set to 11111. 48-bit and 64-bit instructions are likely useful for experimentation and for ISA extensions that require substantial opcode space.

31	27 26	22 21	17 16	12 11	10 9	7 6	0	
rd	rs1	rs2	funct10			opcode		R-type
rd	rs1	rs2	rs3	funct5		opcode		R4-type
rd	rs1	imm[11:7]	imm[6:0]		funct3	opcode		I-type
imm[11:7]	rs1	rs2	imm[6:0]		funct3	opcode		B-type
rd	LUI immediate[19:0]					opcode		L-type
jump offset [24:0]						opcode		J-type

Table 1: RISC-V base instruction formats. *rd* specifies a destination register, while *rs1*, *rs2*, and *rs3* specify source registers. *imm* is a 12-bit immediate operand. The *funct* fields are additional opcodes. [17]

```
size_t strlen(const char* str)    00: 29000003  lb   a1, 0(a0)
{                                 04: 19000013  addi v1,a0,0
  const char* p = str;            08: 01402063  beq  a1,x0,18
  while(*p)                       0c: 18c00413  addi v1,v1,1
    p++;                          10: 10c00003  lb   v0, 0(v1)
  return p - str;                 14: f881f0e3  bne  v0,x0,c
}                                 18: 10c90033  sub  v0,v1,a0
                                  1c: 004000eb  jalr x0,ra
```

Figure 1: C and RISC-V code to compute the length of a C string.

Table 1 shows the six basic instruction formats. The R-type and R4-type formats support register-register computation. The I-type format encodes register-immediate computation and loads; branches and stores are B-type. L-type instructions load large immediates. The J-type format comprises uncoditional jumps and procedure calls.

Some example RISC-V code, which uses a simple approach to calculate the length of a C string, is shown in Figure 1. It is interesting to note that 28 of the 64 nibbles in these 8 instructions are zeros, suggesting a loose, compressible encoding.

3.2 RVC Design Methodology

To determine a candidate set of RVC instructions and to evaluate RVC's effectiveness, we collected static and dynamic measurements from a subset of the SPEC CPU2006 benchmark suite [16].

Benchmark	Static Instructions	Dynamic Instructions (Millions)
bzip2	13038	2117
gcc	645973	3166
mcf	2050	3114
milc	22926	28560
gobmk	196743	19496
soplex	94297	1543
hmmer	57063	4077
sjeng	28881	20195
lbm	2503	1252
astar	7191	28635

Table 2: SPEC CPU2006 subset used for all experiments.

Table 2 lists the benchmarks and their static and dynamic instruction counts when compiled for RISC-V.

All benchmarks were compiled with a GCC 4.4.0/Newlib 1.18.0 cross-compiler, optimizing for size (`-Os`). Static measurements were obtained directly from the resulting executables and object code. Dynamic measurements were obtained from a RISC-V instruction set simulator, running the benchmarks to completion using their small input sets.

3.3 The RVC Extension

RVC's goal is to improve energy efficiency by expressing frequent instructions with 16 bits, or half the size of a base instruction, thereby reducing code size and fetch traffic. To do so without reducing performance, which would offset the energy savings, the 32-bit instructions in the RISC-V base ISA are always available. Consequently, the 16-bit instructions need not cover all functionality in the ISA, allowing the diminutive opcode space to be devoted only to the instructions that will most reduce static and dynamic code size.

The RVC instruction set design is motivated by four observations about the instruction composition of many programs:

- *A small number of opcodes account for most instructions in a program.* Table 3 shows the static and dynamic frequencies of the 20 most common instructions across the SPEC

8

Instruction	Static Frequency	Cumulative	Instruction	Dynamic Frequency	Cumulative
ADDI	25.1%	25.1%	ADDI	15.2%	15.2%
LD	9.7%	34.8%	ADD	7.9%	23.1%
SD	7.6%	42.4%	FLD	7.6%	30.7%
LW	6.2%	48.6%	LW	6.8%	37.5%
LUI	4.9%	53.5%	LD	6.7%	44.2%
JAL	4.7%	58.3%	BNE	5.3%	49.5%
J	4.5%	62.8%	SLLI	4.7%	54.2%
BEQ	4.5%	67.3%	SW	3.8%	58.1%
ADD	4.1%	71.3%	BEQ	3.8%	61.8%
BNE	3.8%	75.2%	ADDIW	3.7%	65.6%
SW	3.5%	78.7%	FSD	3.0%	68.5%
ADDIW	2.6%	81.3%	SD	2.4%	70.9%
SLLI	2.1%	83.4%	FADD.D	2.2%	73.1%
JALR	1.4%	84.8%	FMUL.D	1.9%	75.0%
ANDI	1.2%	86.0%	LUI	1.9%	76.9%
ADDW	1.1%	87.1%	BGE	1.8%	78.7%
FLD	1.0%	88.1%	SUB	1.7%	80.4%
SLTI	0.8%	88.9%	BLT	1.6%	82.0%
SB	0.8%	89.7%	SRLI	1.6%	83.6%
LBU	0.8%	90.5%	FSUB.D	1.5%	85.1%

Table 3: Top 20 most frequent RISC-V instructions, statically and dynamically, in a subset of the SPEC CPU2006 benchmark suite. ADDI is also used to synthesize constants and register moves in RISC-V.
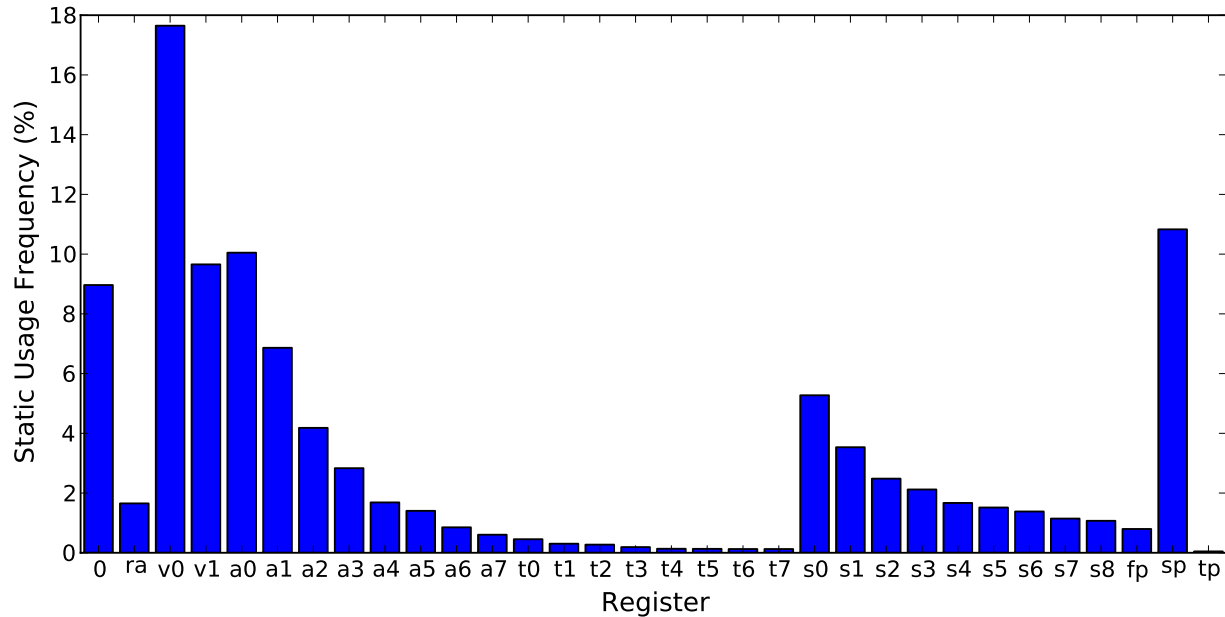
Figure 2: Static frequency of register usage, as either a source or destination operand, in a subset of the SPEC CPU2006 benchmark suite. `ra` is the link register. The `v`, `a`, and `t` registers are for return values, arguments, and temporaries, respectively; all are caller-saved. The `s` registers are callee-saved. `fp` is the frame pointer, `sp` is the stack pointer, and `tp` is the thread-local storage pointer.

CPU2006 benchmark subset. Statically, 10 opcodes account for 75% of instructions and 20 opcodes for 90%. Dynamically, 14 opcodes account for 75% of all instructions executed. ADDI alone accounts for one-quarter of instructions statically and one-seventh dynamically.

- *Many instructions have few unique operands.* For example, for I-type and R-type instructions in these programs, the destination register is the same as at least one source register 36% of the time statically, and 31% of the time dynamically.

- *Register accesses exhibit substantial locality of reference.* Figure 2 shows the static frequency of register usage across the same programs. 60% of register references are to eight registers: two return-value registers and four argument registers (which double as caller-saved temporaries), and two callee-saved registers. 11% of the remaining references are to the stack pointer and 9% to the zero register. The remaining 22 registers account for only
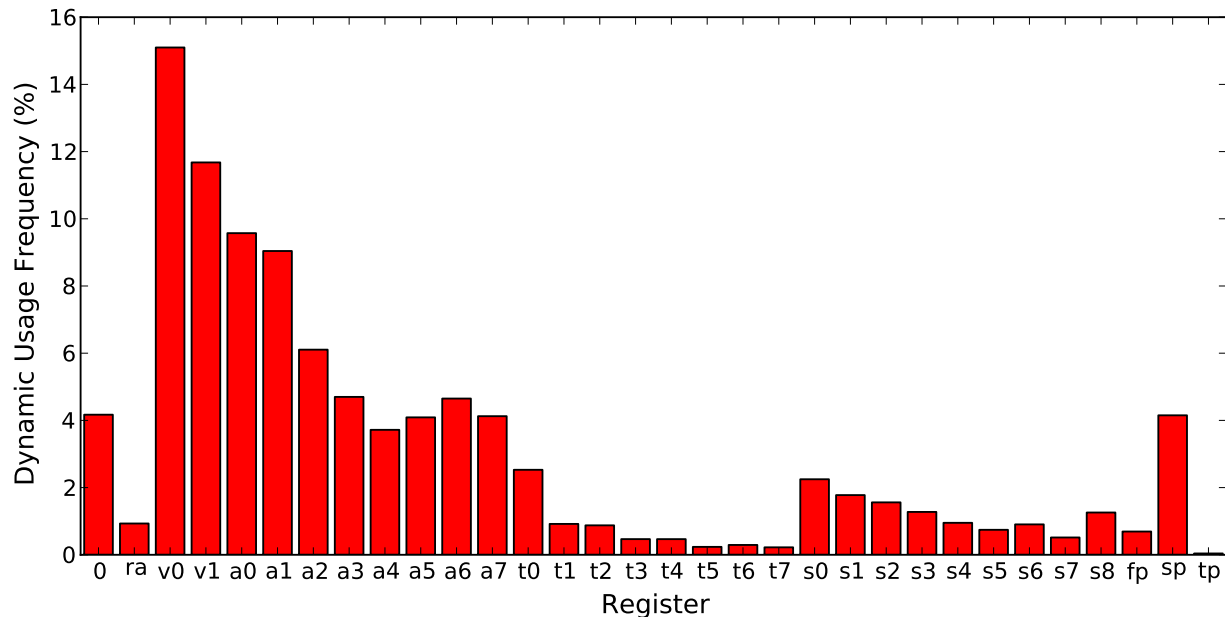
Figure 3: Dynamic frequency of register usage, as either a source or destination operand, in a subset of the SPEC CPU2006 benchmark suite.

20% of register references. This phenomenon is largely attributable to the calling convention but also to the compiler's register allocation scheme. The same eight registers also account for 60% of dynamic references (see Figure 3). Dynamically, the stack pointer and zero register are referenced less frequently; even so, the remaining 22 registers still account for only 33% of dynamic references.

- *Immediate operands are usually small.* Figure 4 shows the size of immedate operands in the same programs. Statically, more than two-thirds of immediates fit within 6 bits, and nearly half within 4. Dynamically, smaller immediates are slightly more common. Branch and jump displacements are larger, with half of these static instructions requiring at least 9 bits (see Figure 5). Dynamically, however, short branches are quite common.

These observations led to a design in which many instructions specify the same source and destination register or can only reference eight of the registers. With 3-bit register specifiers and 5- or 6-bit immediates, 11 bits worth of operands is sufficient to address two registers and either
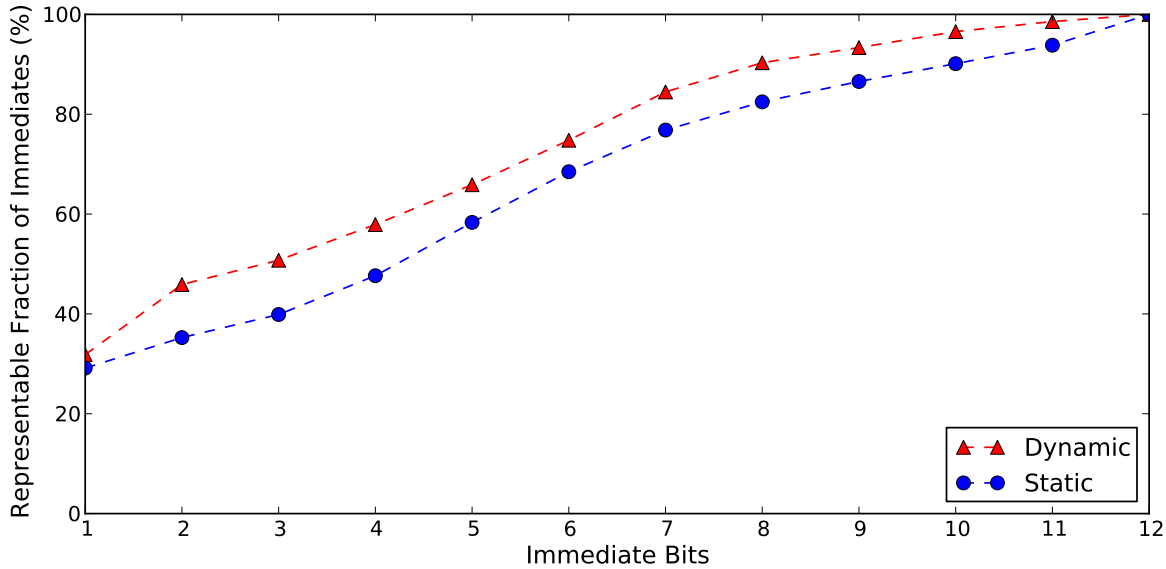
Figure 4: Cumulative distribution of immediate operand sizes in a subset of the SPEC CPU2006 benchmark suite. RISC-V immediates are at most 12 bits. Immediates are two's-complement values, so 0 and -1 are the 1-bit immediates.

an immediate or a third register, leaving 5 bits for the opcode[4].

Instructions were considered for compression if they occurred frequently either statically or dynamically with suitable operands. Table 5 lists the 33 instructions in RVC, along with the RISC-V instruction to which the RVC instruction maps, and whether the RVC instruction was included to reduce static code size, dynamic code size, or both. We included 14 fixed-point arithmetic instructions, the most common control-flow instructions, and several word and double-word loads and stores. Floating-point arithmetic and sub-word loads and stores are notably absent; both require substantial opcode space to achieve worthwhile savings. Floating-point loads and stores are included, however, as they are dynamically common in nearly all floating-point programs.

To simplify the implementation, all RVC instructions map to a single existing RISC-V instruction[5]. Some RVC opcodes map to the same RISC-V instruction: for example, RVC instructions to

---

[4]Only 24 of the 32 possible opcodes are available for RVC, since instructions whose two LSBs equal 11 must be 32 bits or longer.

[5]As a consequence, the compiler does not need to generate RVC instructions; rather, the assembler can substitute the RVC equivalent of a RISC-V instruction if one exists. Nevertheless, instructions that map to a sequence of multiple
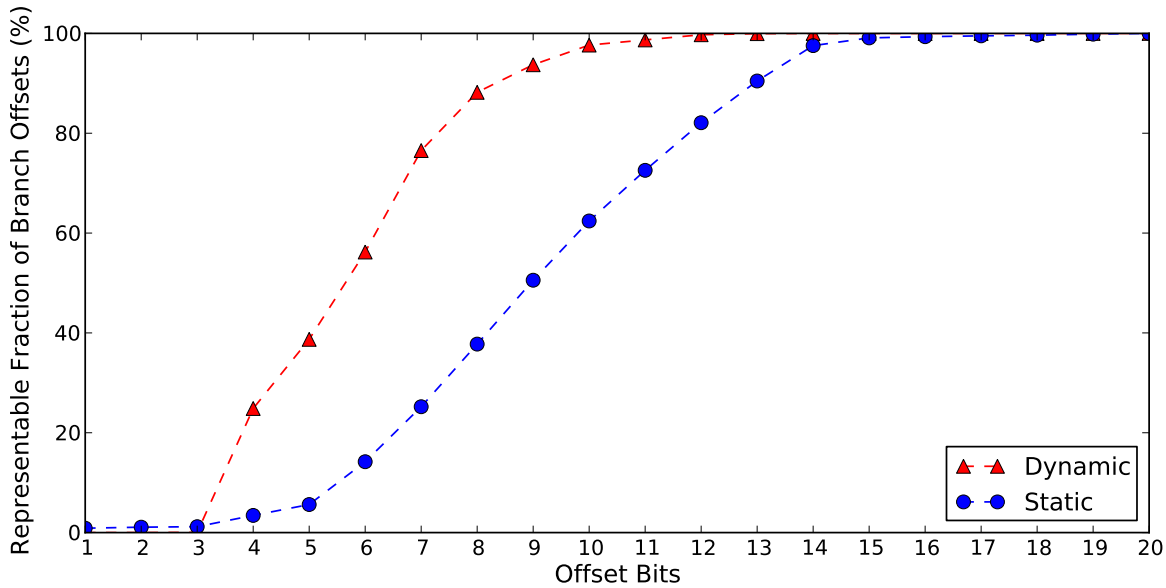
Figure 5: Cumulative distribution of branch and jump offset sizes in a subset of the SPEC CPU2006 benchmark suite. Instructions are 4 bytes large, but offsets are multiples of 2 bytes; the implicit 0 LSB is not counted. RISC-V conditional branch offsets are at most 12 bits. Unconditional jump offsets can be up to 25 bits, but offsets longer than 20 bits ($\pm$256K instructions) did not occur.

increment a register (C.ADDI), move a register (C.MOVE), or load an immediate (C.LI) all map to ADDI. Similarly, three RVC opcodes provide different addressing modes for the RISC-V load word instruction: stack-relative (C.LWSP), register-indirect (C.LW0), and displacement (C.LW).

The addition of RVC instructions does not complicate control flow in mixed RISC-V/RVC code. RISC-V branches and jumps target 2-byte-aligned addresses so can address a RVC or RISC-V instruction that would have been misaligned in the base ISA. Additionally, since RISC-V and RVC instructions can be mixed freely, instructions to switch between ISA modes are not necessary.

Table 6 shows the encoding of the RVC instructions. All instructions have a 5-bit opcode and at least one operand. Immediates are either 5 or 6 bits; branch offsets are 6 bits; and jump offsets are 10 bits long. Instructions that reference a register do so either by its full 5-bit specifier (*rd*, *rs1*, or *rs2*), or by a 3-bit specifier (*rs1a*, *rs1a*, *rs2a*, or *rda*). The mapping from these 3-bit fields to

RISC-V instructions were considered, such as load-multiple and store-multiple instructions, but were omitted for simplicity.

```
00: 29000003  lb   a1, 0(a0)      00: 29000003  lb    a1,0(a0)
04: 19000013  addi v1,a0,0        04: 1062      c.move v1,a0
08: 01402063  beq  a1,x0,18       06: f4b0      c.beq a1,x0,10
0c: 18c00413  addi v1,v1,1        08: 0461      c.addi v1,1
10: 10c00003  lb   v0, 0(v1)      0a: 10c00003  lb    v0,0(v1)
14: f881f0e3  bne  v0,x0,c        0e: ebb1      c.bne v0,x0,8
18: 10c90033  sub  v0,v1,a0       10: 4d9c      c.sub3 v0,v1,a0
1c: 004000eb  jalr x0,ra          12: 0401      c.jr  ra
```

Figure 6: RISC-V and RVC code to compute the length of a C string. The corresponding C code is the same as that of Figure 1.

RVC Register #	rs1a/rs2b/rda	rs2b
0	20 (s0)	20 (s0)
1	21 (s1)	21 (s1)
2	2 (v0)	2 (v0)
3	3 (v1)	3 (v1)
4	4 (a0)	4 (a0)
5	5 (a1)	5 (a1)
6	6 (a2)	6 (a2)
7	7 (a3)	0 (0)

Table 4: RVC mapping of 3-bit register specifiers to the full 32-register space.

RVC Instruction	RISC-V Equivalent	Description	Improves Static	Dynamic
C.ADDI	ADDI rd, rd, imm6	Increment register.	✓	✓
C.ADDIW	ADDIW rd, rd, imm6	Increment register (as 32-bit word).	✓	✓
C.LI	ADDI rd, x0, imm6	Load immediate.	✓	
C.LWSP	LW rd, imm6×4(sp)	Load word, stack-relative.	✓	
C.LDSP	LD rd, imm6×8(sp)	Load double-word, stack-relative.	✓	
C.SWSP	SW rs2, imm6×4(sp)	Store word, stack-relative.	✓	
C.SDSP	SD rs2, imm6×8(sp)	Store double-word, stack-relative.	✓	
C.LW0	LW rd, 0(rs1)	Load word, register-indirect.		✓
C.LD0	LD rd, 0(rs1)	Load double-word, register-indirect.		✓
C.ADD	ADD rd, rs1, rd	Add register, destructive.	✓	✓
C.SUB	SUB rd, rs1, rd	Subtract register, destructive.		✓
C.MOVE	ADDI rd, rs1, 0	Move register.	✓	✓
C.ADD3	ADD rda, rs1a, rs2a	Add register.		✓
C.SUB3	SUB rda, rs1a, rs2a	Subtract register.		✓
C.OR3	OR rda, rs1a, rs2a	Bitwise-OR register.		✓
C.AND3	AND rda, rs1a, rs2a	Bitwise-AND register.		✓
C.SLLI	SLLI rda, rda, shamt	Shift left logical.	✓	✓
C.SRLI	SRLI rda, rda, shamt	Shift right logical.	✓	✓
C.SRAI	SRAI rda, rda, shamt	Shift right arithmetic.	✓	✓
C.SLLIW	SLLIW rda, rda, shamt	Shift left logical (as 32-bit word).	✓	✓
C.LW	LW rda, imm5×4(rs1a)	Load word.	✓	✓
C.LD	LD rda, imm5×8(rs1a)	Load double-word.	✓	✓
C.SW	SW rs2b, imm5×4(rs1a)	Store word.	✓	✓
C.SD	SD rs2b, imm5×8(rs1a)	Store double-word.	✓	✓
C.FLW	FLW rda, imm5×4(rs1a)	Load floating-point word.		✓
C.FLD	FLD rda, imm5×8(rs1a)	Load floating-point double-word.		✓
C.FSW	FSW rs2b, imm5×4(rs1a)	Store floating-point word.		✓
C.FSD	FSD rs2b, imm5×8(rs1a)	Store floating-point double-word.		✓
C.BEQ	BEQ rs1a, rs2b, imm5	Branch if equal.	✓	✓
C.BNE	BNE rs1a, rs2b, imm5	Branch if not equal.	✓	✓
C.JR	JALR x0, rs1a	Indirect jump/subroutine return.	✓	
C.JALR	JALR ra, rs1a	Indirect subroutine call.	✓	
C.J	J imm10	Unconditional jump.	✓	

Table 5: RVC instruction listing.

5-bit register numbers is shown in Table 4.

Figure 6 shows RISC-V and RVC code to compute the length of a C string. Six of the eight RISC-V instructions can be represented with RVC instructions, reducing code size by $\frac{3}{8}$. RVC lacks a load byte opcode, so RISC-V `lb` instructions are used. Note that the second such instruction is not naturally aligned.

3.4   Implementation Considerations

RVC implementations are modestly more complex than RISC-V implementations. An RVC machine must be able to fetch misaligned 32-bit instructions, possibly across cache line and page boundaries[6]. Additional decode logic is needed to convert RVC instructions to their RISC-V equivalent (or to some internal format). Superscalar implementations must partially decode instructions to find the boundaries between them, in order to determine which bits to decode. Fortunately, this is a fast operation in RVC, as the first two bits of the instruction determine its length.

To leverage the reduction in fetch traffic, some implementations may require additional instruction buffering, so that when multiple instructions have been fetched and buffered, the instruction fetch unit can be selectively turned off.

---

[6]A design that required 32-bit alignment of 32-bit instructions would offset much of the savings of RVC. For example, if 50% of instructions are compressible, distributed uniformly at random, then in expectation $\frac{1}{3}$ of 32-bit instructions need 16 bits of padding, eliminating $\frac{1}{3}$ of the code size reduction.

15	14 13 12	11 10	9 8 7	5 4 0	
imm6			rd ≠ 0	ADDI	C.ADDI
imm6			rd	ADDIW	C.ADDIW
imm6			rd	LI	C.LI
imm6			rd	LWSP	C.LWSP
imm6			rd	LDSP	C.LDSP
imm6			rs2	SWSP	C.SWSP
imm6			rs2	SDSP	C.SDSP
0	rs1		rd	L0	C.LW0
1	rs1		rd	L0	C.LD0
0	rs1		rd	R2	C.ADD
1	rs1		rd	R2	C.SUB
0	rs1		rd	MOVE	C.MOVE
rda	rs1a	00	rs2a	R3	C.ADD3
rda	rs1a	01	rs2a	R3	C.SUB3
rda	rs1a	10	rs2a	R3	C.OR3
rda	rs1a	11	rs2a	R3	C.AND3
rda	00	shamt		SHIFT	C.SLLI
rda	01	shamt		SHIFT	C.SRLI
rda	11	shamt		SHIFT	C.SRAI
rda	10	0	shamt	SHIFT	C.SLLIW
rda	rs1a	imm5		LW	C.LW
rda	rs1a	imm5		LD	C.LD
rda	rs1a	imm5		FLW	C.FLW
rda	rs1a	imm5		FLD	C.FLD
rs2b	rs1a	imm5		SW	C.SW
rs2b	rs1a	imm5		SD	C.SD
rs2b	rs1a	imm5		FSW	C.FSW
rs2b	rs1a	imm5		FSD	C.FSD
rs2b	rs1a	imm5		BEQ	C.BEQ
rs2b	rs1a	imm5		BNE	C.BNE
0	rs1		00000	ADDI	C.JR
1	rs1		00000	ADDI	C.JALR
1	jump target			MOVE	C.J

Table 6: RVC instruction encodings.

4 Evaluation

To evaluate RVC's potential for improving code size, energy efficiency, and performance, we obtained static and dynamic measurements from the same SPEC CPU2006 programs listed in Table 2. All benchmarks were compiled with a GCC 4.4.0 cross-compiler for the respective ISA, optimizing for size (`-Os`). Dynamic measurements were obtained from an RVC instruction set simulator augmented with cache models. Some ISAs in this evaluation only support 32-bit address spaces, so all programs were compiled for 32-bit targets[7].

Compression from RISC-V to RVC instructions is implemented as a pass in the GNU assembler. This procedure is straightforward, except for branches. Branch offsets cannot be computed until the lengths of all intervening instructions are known, but whether an RVC branch can be used depends on the branch offset. So, we speculate that all branches are short enough to be encoded as RVC branches, then we relax them to longer RISC-V branches when offsets are determined not to fit.

4.1 Static Code Compression Results

Static code size impacts instruction memory costs and start-up overhead, and it correlates with instruction cache miss rates. One of our goals in defining RVC is thus to reduce static code size. We measure RVC's code compression efficacy by computing its compression ratio, i.e., RVC code size divided by RISC-V code size. Figure 7 shows the static compression ratios for the SPEC benchmarks. Compression ratios are generally about 75%, meaning that half of static instructions were compressed.

Of course, compression ratio is a figure of less merit than code size, as one could simply adopt a denser base ISA rather than compressing instructions in the base ISA. Figure 8 shows the code size for several instruction sets, normalized to the size of RVC code. The first three ISAs, RISC-V, MIPS, and ARM, are RISC ISAs with fixed-length 32-bit instructions. The next four, RVC, MIPS16, ARM Thumb, and ARM Thumb-2, are compressed RISC ISAs. Finally, x86 represents

---

[7]Compression ratios for 64-bit RVC programs tend to be 0.1%-0.2% higher than for 32-bit RVC programs, as immediates are larger.
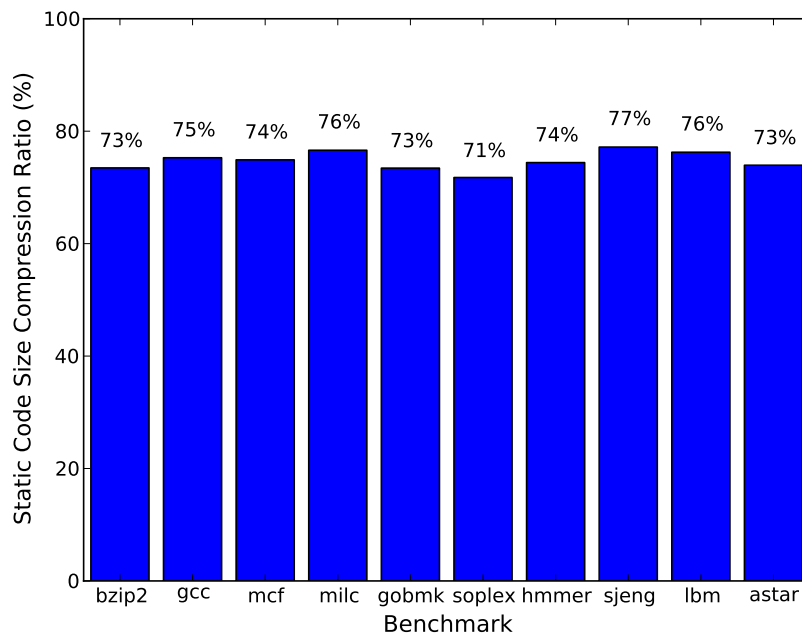
Figure 7: Static compression ratio of RVC code over RISC-V code, across a subset of the SPEC CPU2006 benchmark suite.
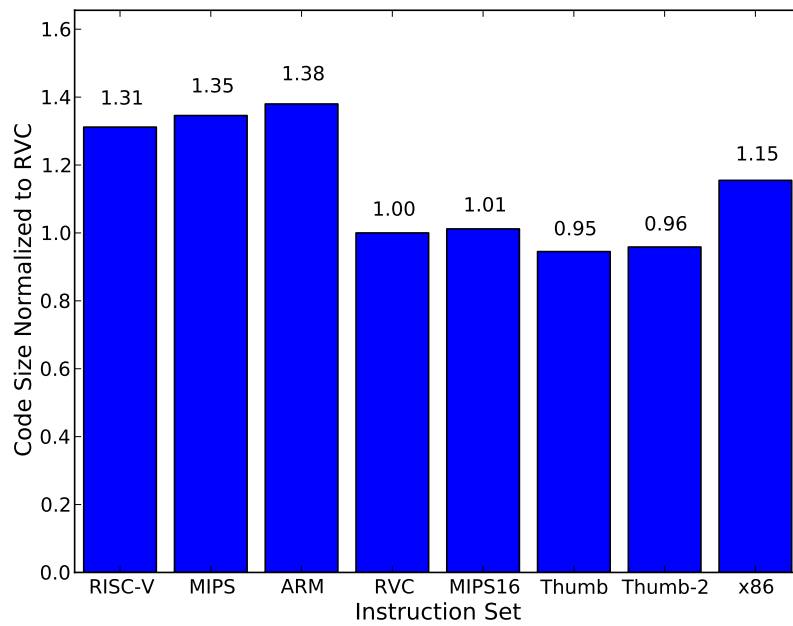
Figure 8: Static code size, normalized to RVC, averaged geometrically across a subset of the SPEC CPU2006 benchmark suite. Only the programs' object code is considered as the C library differs between platforms.

the variable-length CISCs.

The fixed-length ISAs have the largest code size. Interestingly, variable-length x86 code is less than 15% smaller than fixed-length RISC-V code, in spite of having instructions as short as one byte. The compressed RISC ISAs are substantially smaller; RVC and MIPS16 are 15% smaller than x86. RVC code is 6-7% larger than Thumb and Thumb-2 code; Chapter 5 discusses some reasons for their greater code density.

4.2 Dynamic Code Compression Results

In an idealized processor model, dynamic code size equals the amount of data fetched from instruction memory. Reducing dynamic code size can thus reduce instruction fetch energy, as fewer total bits are retrieved from instruction memory. Figure 9 shows the dynamic compression ratios for the same programs. RVC programs usually cause 25%-30% fewer instruction bits to be fetched than RISC-V programs.
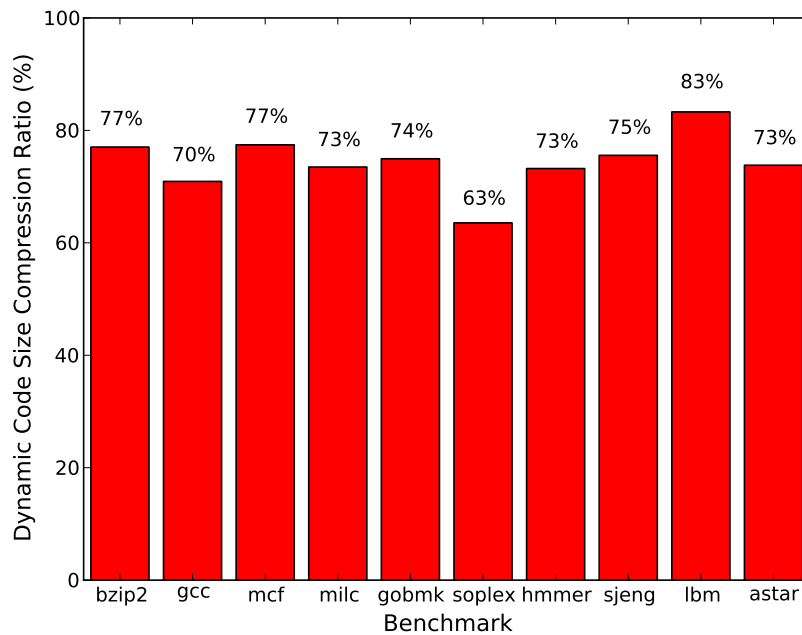
Figure 9: Dynamic compression ratio of RVC code over RISC-V code, across a subset of the SPEC CPU2006 benchmark suite.

We did not compare RISC-V and RVC dynamic code size with that of other ISAs, but we expect their static and dynamic code size to correlate to some degree.

4.3 Instruction Cache Performance

A reduced instruction working set will often incur fewer instruction cache misses. To study the effect of RVC on instruction cache performance, we simulated each program running with cache sizes ranging from 256 bytes to 32 KB, direct-mapped and two-way set-associative. All caches use 32-byte lines. Figures 10 through 19 plot the instruction cache miss rates for these programs versus the cache size. Two of the four curves are for the RISC-V (i.e. RV) program, one curve for a direct-mapped cache and the other for a two-way cache; the other two curves are for the RVC program, with the same cache configurations.

RVC reduces miss rates substantially in most programs, commensurate with the reduction in instruction working set size. For 7 of the 10 programs (all except `milc`, `lbm`, and `astar`), using RVC improves performance roughly as much as doubling the instruction cache size.
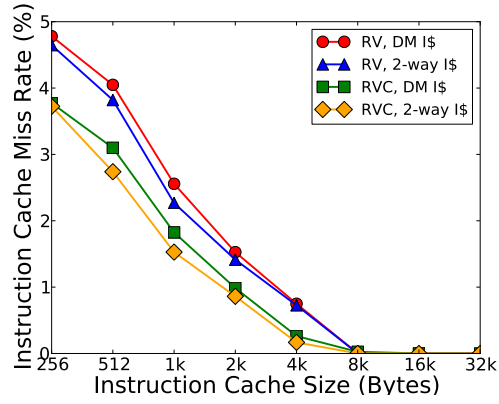
Figure 10: Instruction cache miss rates for bzip2.
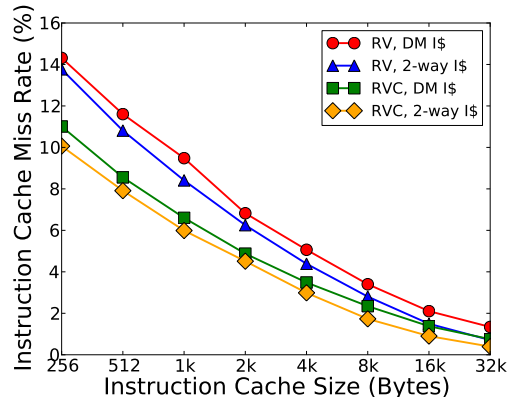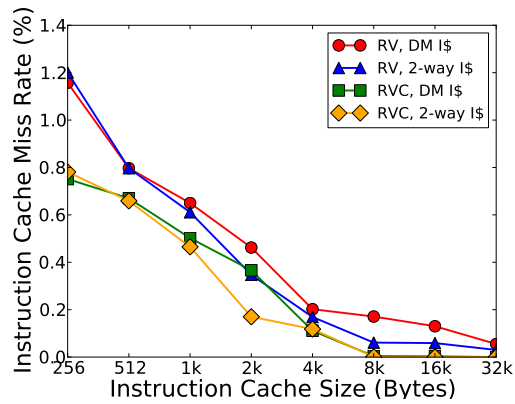

Figure 11: Instruction cache miss rates for gcc.
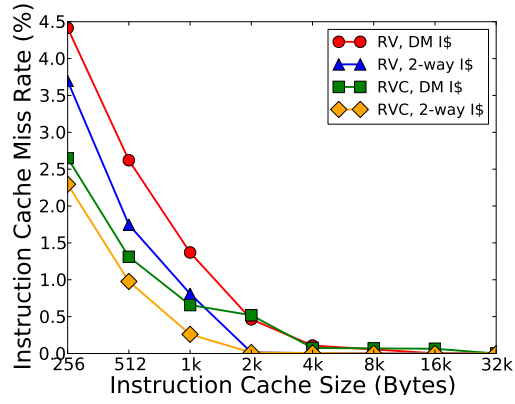

Figure 12: Instruction cache miss rates for mcf.
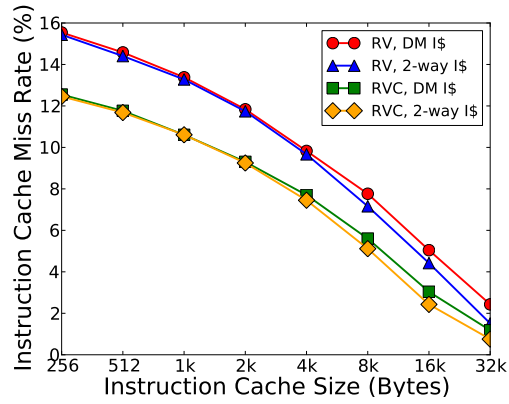
Figure 13: Instruction cache miss rates for `milc`.

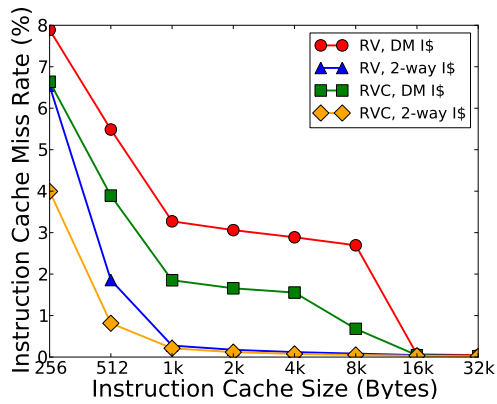Figure 14: Instruction cache miss rates for `gobmk`.
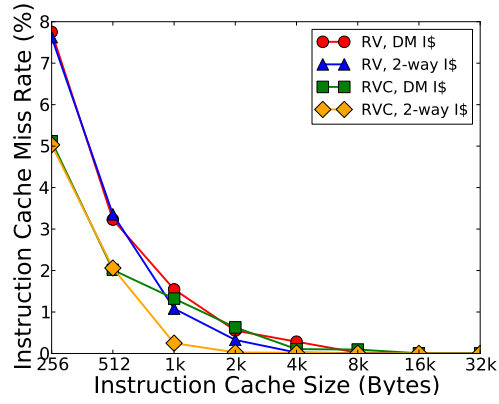

Figure 15: Instruction cache miss rates for `soplex`.

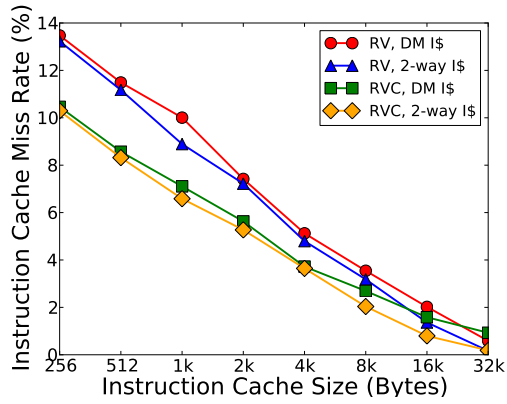Figure 16: Instruction cache miss rates for `hmmer`.

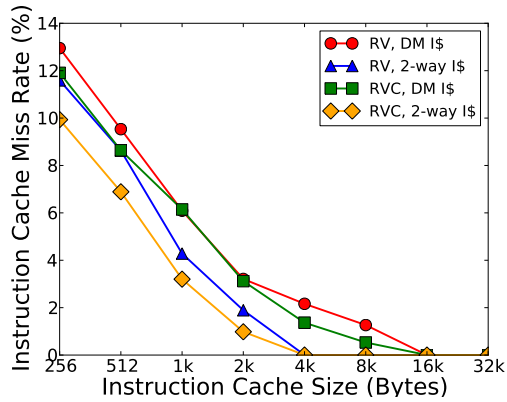Figure 17: Instruction cache miss rates for `sjeng`.

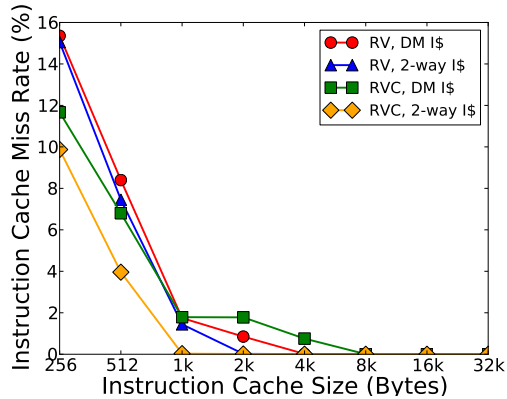Figure 18: Instruction cache miss rates for `lbm`.

Figure 19: Instruction cache miss rates for `astar`.

4.4   System Performance

The reduced instruction miss rates in RVC programs correspond to an improvement in overall performance. To measure this effect, we modeled a processor that executes one instruction per clock cycle, except for instruction and data cache and TLB misses; main memory accesses take 50 cycles. Figure 20 compares three configurations to a base RISC-V system with some instruction cache size: a system with twice the instruction cache capacity; a system with a two-way set-associative instruction cache; and an RVC system running an RVC program.

RVC improves performance substantially when the instruction working set does not fit in cache. For 6 of the 8 cache configurations, using RVC is more effective than doubling the associativity. Using RVC attains, on average, 80% of the speedup of doubling the cache size. A system with a 16 KB direct-mapped cache with RVC is 99% as fast as a system with a 32 KB direct-mapped cache without RVC.
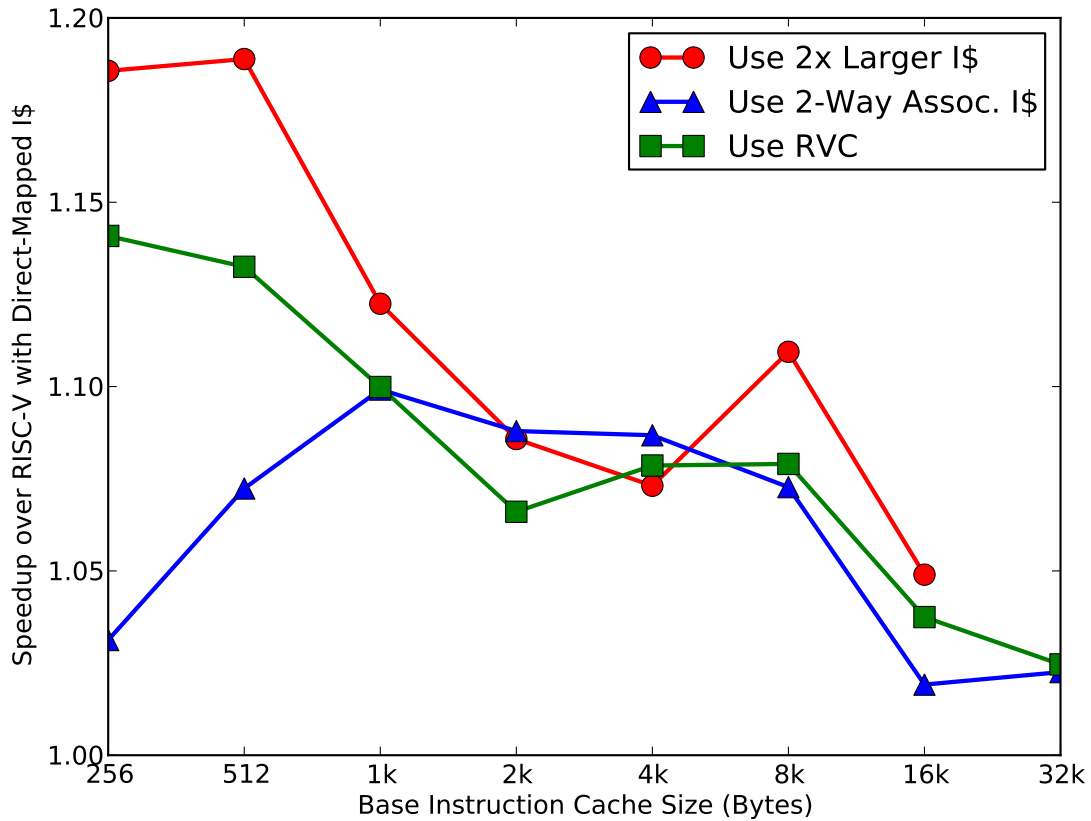
26

Figure 20: Geometric mean speedup of various configurations over a RISC-V system with a direct-mapped instruction cache of a given instruction cache size. All instructions execute in one clock cycle, except for cache misses, on which the processor blocks. Cache refills and writebacks take 50 cycles; TLB refills take 100. All configurations use a 32 KB 2-way set-associative data cache. All caches have 32-byte lines. Instruction and data TLBs are each 8 entries and are fully associative. The page size is 4 KB.

5 Discussion

Program binaries compiled for fixed-length RISC ISAs are large relative to their CISC counter-parts. But x86 code is substantially larger than that of the short-instruction-word and variable-length RISC ISAs. For embedded systems with severe constraints on instruction memory size, these ISAs are likely to be the most cost-effective.

Of the compressed RISCs, MIPS16 and RVC code are about the same size, while Thumb is about 6% smaller. Even so, RVC code is likely to outperform both. MIPS16 and Thumb suffer from lack of floating-point instructions and from access to only eight registers, increasing register spills. Static and dynamic instruction counts are thereby increased. MIPS16 and Thumb programs are composed almost exclusively of 16-bit instructions, whereas only about half of static RVC instructions are 16 bits. Consequently, RVC programs are encoded in vastly fewer static instructions than either MIPS16 or Thumb programs. Most RVC programs thus comprise fewer dynamic instructions than either MIPS16 or Thumb programs, likely improving performance over both.

Thumb-2 is a variable-length RISC ISA similar to RVC. Like ARM and Thumb, Thumb-2 has several ISA features that improve code size over RISC-V and RVC code. A PC-relative addressing mode enables many global variable accesses to be encoded as a single 16-bit instruction, whereas global variable accesses in RVC require two 32-bit instructions[8]. Conditional execution shortens some code paths. Load Multiple and Store Multiple instructions shorten function prologues and epilogues in particular. Finally, some arithmetic code sequences can be expressed by fewer Thumb-2 instructions than RVC instructions[9]. Although these features serve to make Thumb-2 code about 7% smaller than RVC code, it is worth noting that some of them significantly complicate certain implementation styles (e.g. conditional execution in a dynamically scheduled pipeline with register renaming).

In addition to reducing static code size, RVC shows promise for improving energy per operation. An RVC machine with small instruction caches is likely to outperform and use less energy

---

[8]In RISC-V, globals are loaded with a LUI/Load instruction pair. The load cannot be compressed into an RVC instruction because the immediate value is not known until link time.

[9]For example, the C expression `a + (b << 17)` might compile to C.SLLI/C.ADD3 in RVC, but just ADD.W in Thumb-2.

than a similarly-configured RISC-V machine. On a system with a 16 KB direct-mapped instruction cache and 32 KB 2-way data cache, 9% of all main memory accesses (instruction and data) are eliminated, performance is 4% better, and 25% fewer instruction bits need be fetched. With an 8 KB instruction cache, main memory accesses are reduced 14% and performance is 7% improved.

Perhaps more importantly, the smaller instruction footprint of RVC programs enables the use of smaller or simpler caches. For a 1% performance hit, a RISC-V implementation with a 32 KB instruction cache could add RVC support and halve its instruction cache size. And, of course, fewer bits need be fetched from this smaller cache. For implementations in which instruction fetch energy is a large component of total energy, using RVC and a smaller cache could improve energy efficiency significantly.

6   Conclusions and Future Work

It is difficult to project the actual energy savings of a variable-length instruction encoding like RVC without a complete implementation. For example, additional buffering and decode logic increase energy and could, for some processors, affect cycle time or require additional pipelining. On the other hand, if RVC allows using a smaller cache, the reduced cache access time might largely compensate for this effect. Forthcoming VLSI and FPGA implementations of the RVC instruction set should elucidate its power, area, and timing tradeoffs.

Additionally, a comparison of the dynamic instruction counts and fetch traffic of the various ISAs in this study would paint a clearer picture of the tradeoff between performance, code size, and energy. Virtutech Simics [7], a full-system simulator, supports all of the instruction sets in this study except RISC-V and RVC and would be suitable for these experiments.

Nevertheless, RVC is a promising means to improve energy efficiency. Its code size is competitive with the other compressed RISC ISAs in this study, which have proven quite popular. RVC programs need to fetch about 25% fewer instruction bits than RISC-V programs, reducing costly instruction fetches, and smaller code size can improve performance by reducing cache misses or can enable the use of smaller caches, further reducing energy and also reducing area. RVC will thus make RISC-V processors more efficient.

References

[1] ARM Ltd. ARMv6-M Architecture Reference Manual, 2008. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0419c/

[2] ARM Ltd. ARMv7-M Architecture Reference Manual, 2010. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0403c/

[3] M. Hill, S. Eggers, J. Larus, G. Taylor, G. Adams, B. K. Bose, G. Gibson, P. Hansen, J. Keller, S. Kong, C. Lee, D. Lee, J. Pendleton, S. Ritchie, D. A. Wood, B. Zorn, P. Hilfinger, D. Hodges, R. Katz, J. Ousterhout, and D. Patterson. Design Decisions in SPUR. *IEEE Computer*, 19, November 1986.

[4] M. Katevenis, R. Sherburne, D. Patterson, and C. Séquin. The RISC II Micro-Architecture. *Advances in VLSI and Computer Systems*, 1:138–152, October 1984.

[5] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Achitecture*, ISCA '11. ACM, 2011.

[6] C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving Code Density Using Compression Techniques. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 194–203, 1997.

[7] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 2002.

[8] MIPS Technologies Inc. MIPS64 Architecture for Programmers Volume I: Introduction to the MIPS64 Architecture, 2010. http://www.mips.com/products/architectures/mips64/

[9] MIPS Technologies Inc. MIPS64 Architecture for Programmers Volume IV-a: The MIPS16e Application-Specific Extension to the MIPS64 Architecture, 2010. http://www.mips.com/products/architectures/mips64/

[10] J. Montanaro, R. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Technical Journal*, 9:49–62, January 1997.

[11] H. Pan and K. Asanović. Heads and Tails: A Variable-Length Instruction Format Supporting Parallel Fetch and Decode. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '01. ACM, 2001.

[12] D. Patterson and C. Séquin. RISC I: A Reduced Instruction Set VLSI Computer. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, ISCA '81. IEEE Computer Society Press, 1981.

[13] A. Samples, M. Klein, and P. Foley. SOAR Architecture. Technical Report UCB/CSD-85-226, EECS Department, University of California, Berkeley, 1985.

[14] SPARC International, Inc. The SPARC Architecture Manual, Version 9, 2010. `http://www.sparc.org/standards/SPARCV9.pdf`

[15] W. Strecker. VAX-11/780—A Virtual Address Extension to the DEC PDP-11 Family. In *AFIPS Spring Conference*, 1978.

[16] System Performance Evaluation Cooperative. SPEC CPU2006 Benchmarks, 2006. `http://www.spec.org/cpu2006/`

[17] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.