rv8: a high performance RISC-V to x86 binary translator

Michael Clark
The rv8 contributors
michaeljclark@mac.com

Bruce Hoult
The rv8 contributors
bruce@houlton.org

ABSTRACT
Dynamic binary translation has a history of being used to ease transi-
tions between CPU architectures [7], including micro-architectures. Modern x86 CPUs, while maintaining binary compatibility with their legacy CISC instruction set, have internal micro-architectures that resemble RISC. High performance x86 micro-architectures have long used a CISC decoder front-end to crack complex instructions into smaller micro-operations. Recently macro-op fusion [17][6] has been used to combine several instructions into one micro-op. Both techniques change the shape of the ISA to match the internal µop micro-architecture. Well-known binary translators also use micro-op internal representations to provide an indirection between the source and target ISAs as this makes the addition of new instruction sets much easier.

We present rv8, a high performance RISC-V simulation suite contain-
ing a RISC-V JIT (Just In Time) translation engine specifically target-
ing the x86-64 instruction set. Achieving the best possible mapping from a given source to target ISA pair requires a purpose designed internal representation. This paper explores a simple and fast translation from RISC-V to x86-64 that exploits knowledge of the geometries of the source and target ISAs, ABIs, and current x86-64 micro-architectures with a goal of producing a near optimal mapping from the 31 register source ISA to the 16 register target ISA. Techniques are explored that exploit the CISC encoding to increase instruction density, coalescing micro-ops into CISC instructions with the goal of generating the minimum number of micro-ops at the target micro-architectural level.

CCS CONCEPTS
• Computer systems organization → Reduced instruction set computing; Complex instruction set computing.

KEYWORDS
dynamic binary translation, RISC, CISC

Reference Format:
Michael Clark and Bruce Hoult. 2017. rv8: a high performance RISC-V to x86 binary translator. First Workshop on Computer Architecture Research with RISC-V, Boston, MA, USA, October 2017 (CARRV 2017), 7 pages.

1 INTRODUCTION
RISC-V [20] is a modern, elegant and extensible open source instruc-
tion set architecture originally designed for computer architecture research. Similar to the Linux kernel two decades earlier in operating systems, RISC-V provides a standard and open base ISA and competitive reference hardware designs that anyone can freely use as-is or as a base for further innovation. This makes RISC-V uniquely suitable for the next phase of development in microproces-
sor hardware architectures much as the Linux kernel serves as the software foundation for embedded, mobile and server computing on a huge variety of CPU architectures and with a proliferation of operating systems such as Android, Tizen, Red Hat, Ubuntu, Fedora, Debian and many more.

Dynamic binary translation has frequently [7] [14] [2] [1] been used to provide binary compatibility for applications targeting legacy architectures during transitions to new architectures, however it can also be used to enable compatibility for binaries from newer architectures to allow for their execution on legacy hardware. Given x86 is the dominant architecture in cloud computing environments, dynamic binary translation provides a convenient means to enable RISC-V binary compatibility on existing hardware.

For binary translation to be acceptable as a mechanism to run RISC-V application images on legacy x86 hardware in the cloud, the performance must be similar to that of native code and there must be compelling advantages beyond performance, such as increased security [23]. While RISC-V is designed as an instruction set for hardware its features also make it an excellent platform abstraction for high performance virtualization. With this in mind, the goal of rv8 is to provide a binary translation platform that is able to achieve near native performance to enable secure RISC-V virtual machines in cloud computing environments.

Given Intel’s and AMD’s access to the latest process nodes [3], 4+ GHz clock speeds [22], superscalar execution, several dozen cores [15] and hundreds of GB of memory in a server, a near native speed RISC-V binary translator is likely to be the fastest RISC-V implementation and most practical build environment for things such as operating system distributions for some years to come.

2 PRINCIPLES
The rv8 binary translation engine has been designed as a hybrid interpreter and JIT compiler. The translator first interprets code and profiles it for hot paths [12]. Hot paths are then JIT translated to native code. The translation engine maintains a call stack to allow runtime inlining of hot functions. A jump target cache is used to accelerate returns and indirect calls through function pointers. The translator supports mixed binary translation and interpretation to handle instructions that do not yet have native translations. Currently, RISC-V ‘IM’ code is translated while ‘AFD’ is interpreted. The rv8 translator also supports RVC compressed code [19].
3 OPTIMISATIONS

The rv8 binary translator performs JIT translation of RISC-V code to x86-64 code. This is a challenging problem because the shapes of the instruction set architectures differ in a number of ways. RISC-V is a 31 register load/store reduced instruction set architecture while x86-64 is a 16 register complex instruction set architecture where loads and stores can be embedded in most instructions using indirect indexed memory operands. The challenge for a high performance JIT translator is to achieve dense mapping from RISC-V to x86-64 micro-ops via the legacy CISC decoder.

3.1 Register allocation

If the emulated ISA has significantly fewer registers than the host ISA – for example emulating x86 on a typical RISC – then register allocation is trivial. Just permanently assign each emulated register to a host register. In other cases translators normally either store all emulated registers in a memory array, or run a register allocation algorithm similar to a conventional compiler, treating source code registers like high level language variables or intermediate language virtual registers. The former approach is easy to write and translates quickly, but the translated code runs slowly. The latter approach slows translation but the generated code can run quickly – except possibly at control flow joins (phi nodes); and a number of move instructions may be required to resolve different register allocations on different execution paths.

rv8 approaches the register set size problem by making the observation that register usage is not evenly distributed, but is heavily biased to the ABI function call argument registers, stack pointer, and function return address register. We statically allocate each RISC-V register to either one of twelve fixed host x86-64 registers or to memory. Dynamic execution profiles were captured from long running programs (such as the Linux kernel) to find the dominant register set. The dominant set appears to be relatively stable. Refer to Figure 1 for the static allocation currently used.

The ‘C’ compressed instruction extension choice of eight registers is based on the same observation, though in that case it is the static distribution of register usage that matters, not dynamic. The difference is small. If future compilers are taught to favour the RVC registers to get better code compression our translator speed will also benefit.

RISC-V registers that are not mapped to an x86-64 register are accessed indirectly using the rbp register. e.g. qword [rbp+0xF8] would be used to access the RISC-V t4 register.

Frequently accessed registers that are not in the statically allocated set will reside in L1 cache so accesses to spilled registers are relatively fast (approximately 1 nanosecond or 3 cycles latency), nevertheless in cases where the compiler allocates several hot registers outside of the statically allocated register set, substantial performance can be lost.

Registers that are allocated in memory is one place where the translator can take advantage of x86’s complex memory operands, allowing it to embed references to spilled registers directly within instructions rather than issuing separate loads and stores. This helps the translator maintain instruction density which lowers instruction cache pressure and helps improve throughput.

It is too late for the translator to rearrange stack memory for optimal stack spills as memory accesses must be translated precisely. RISC-V programs are optimised for 31 registers so stack spills are much less frequent. A future version of the translator may use a dynamic register allocator with live range splitting or potentially single static assignment [10] [4] [13]. This is discussed in Section 5.

3.2 CISC vs RISC

rv8 makes use of CISC memory operands to access registers residing in the memory backed register spill area. The complex memory operands end up being cracked into micro-ops in the CISC pipeline, however their use helps increase instruction density, which increases performance due to better use of instruction cache. Figure 2 shows various embeddings for resident and spilled registers.

There are many combinations of instruction expansions depending on whether a register is in memory or whether there are two or three operands. Three operand RISC-V instruction is translated into a move and a destructive two operand x86-64 instruction. Temporary registers are used if both operands are memory backed. Future versions of the translator may perform passes to coalesce loads, ALU ops and stores into ALU ops with memory operands. This is discussed in Section 5.

3.3 Translator registers

The rv8 translator needs to use several host registers to point to translator internal structures and for use as temporary registers during emulation of some instructions.
3.4 Prologue and epilogue

The interpreter is normal C code that can use any host register, and so always uses the memory array to store RISC-V registers. Therefore an assembly-language prologue loads RISC-V registers from memory to their assigned host registers when transitioning from the interpreter to native code and an epilogue saves RISC-V registers from host registers back to memory when transitioning from native code to the interpreter.

The emitted epilogue and prologue is relatively large so the JIT translator works to minimize the number of transitions from native code back to the translator. Nested loops run completely in native code without any returns to the interpreter.

3.5 Indirect branch acceleration

Indirect unconditional branches, such as calls through function pointers, cannot be statically translated as the target address of their host code is not known at the time of translation. rv8 maintains a hashtable mapping guest program addresses to translated host native code addresses. A full lookup is relatively slow because it requires saving caller-save registers and calling into C++ code.

To accelerate indirect calls through function pointers, a small assembly stub looks up the target address in a sparse 1024 entry direct mapped cache, and falls back to a slow call into the interpreter if the RISC-V target address is not found. The cache is then updated so that later indirect calls to the same RISC-V target address can be accelerated [18].

The direct mapped translation cache is indexed by bits [10:1] of the guest address as shown in Figure 3. Bit zero can be ignored because RISC-V instructions must start on a 2-byte boundary.

Some RISC-V indirect branches can be interpreted as direct branches which makes their translation more efficient. In particular the RISC-V CALL and TAIL macros for relative far unconditional branches. The CALL and TAIL macro expansions include the jump and link register instruction. The CALL macro expands to:

1: auipc ra, %pcrel_hi(sym)
jalr ra, %pcrel_lo(1b)(ra)

The jump and link register instruction would normally be considered an indirect branch, however the relative immediate value

---

**Table 1: Static register allocation**

<table>
<thead>
<tr>
<th>RISC-V</th>
<th>x86-64</th>
<th>Spill slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>ra</td>
<td>[rbp + 16]</td>
</tr>
<tr>
<td></td>
<td>sp</td>
<td>[rbp + 24]</td>
</tr>
<tr>
<td></td>
<td>gp</td>
<td>[rbp + 32]</td>
</tr>
<tr>
<td></td>
<td>tp</td>
<td>[rbp + 40]</td>
</tr>
<tr>
<td></td>
<td>t0</td>
<td>[rbp + 48]</td>
</tr>
<tr>
<td></td>
<td>t1</td>
<td>[rbp + 56]</td>
</tr>
<tr>
<td></td>
<td>t2</td>
<td>[rbp + 64]</td>
</tr>
<tr>
<td></td>
<td>s0</td>
<td>[rbp + 72]</td>
</tr>
<tr>
<td></td>
<td>s1</td>
<td>[rbp + 80]</td>
</tr>
<tr>
<td></td>
<td>a0</td>
<td>[rbp + 88]</td>
</tr>
<tr>
<td></td>
<td>a1</td>
<td>[rbp + 96]</td>
</tr>
<tr>
<td></td>
<td>a2</td>
<td>[rbp + 104]</td>
</tr>
<tr>
<td></td>
<td>a3</td>
<td>[rbp + 112]</td>
</tr>
<tr>
<td></td>
<td>a4</td>
<td>[rbp + 120]</td>
</tr>
<tr>
<td></td>
<td>a5</td>
<td>[rbp + 128]</td>
</tr>
<tr>
<td></td>
<td>a6</td>
<td>[rbp + 136]</td>
</tr>
<tr>
<td></td>
<td>a7</td>
<td>[rbp + 144]</td>
</tr>
<tr>
<td></td>
<td>s2</td>
<td>[rbp + 152]</td>
</tr>
<tr>
<td></td>
<td>s3</td>
<td>[rbp + 160]</td>
</tr>
<tr>
<td></td>
<td>s4</td>
<td>[rbp + 168]</td>
</tr>
<tr>
<td></td>
<td>s5</td>
<td>[rbp + 176]</td>
</tr>
<tr>
<td></td>
<td>s6</td>
<td>[rbp + 184]</td>
</tr>
<tr>
<td></td>
<td>s7</td>
<td>[rbp + 192]</td>
</tr>
<tr>
<td></td>
<td>s8</td>
<td>[rbp + 200]</td>
</tr>
<tr>
<td></td>
<td>s9</td>
<td>[rbp + 208]</td>
</tr>
<tr>
<td></td>
<td>s10</td>
<td>[rbp + 216]</td>
</tr>
<tr>
<td></td>
<td>s11</td>
<td>[rbp + 224]</td>
</tr>
<tr>
<td></td>
<td>t3</td>
<td>[rbp + 232]</td>
</tr>
<tr>
<td></td>
<td>t4</td>
<td>[rbp + 240]</td>
</tr>
<tr>
<td></td>
<td>t5</td>
<td>[rbp + 248]</td>
</tr>
<tr>
<td></td>
<td>t6</td>
<td>[rbp + 256]</td>
</tr>
</tbody>
</table>

---

- Store instructions require the use of two temporary registers if both register operands are in the register spill area.
- x86 variable offset shift instructions require the shift amount to be in the cl register.
- x86 instructions such as imul and idiv require the use of two temporary registers as they take implicit operands and clobber rax and rdx.

The registers rax and rcx were chosen over rax and rdx to optimise for variable offset shift instructions as these are more frequent and more latency sensitive than multiplies and divides. The translator uses rbp to point at the register spill area and rsp points at the host stack.

Four x86 host registers are reserved leaving twelve registers available for mapping directly to RISC-V registers. See Table 2 for details on translator reserved registers.
Table 2: Translator reserved registers

<table>
<thead>
<tr>
<th>x86-64</th>
<th>Translator register purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp</td>
<td>pointer to the register spill area and the L1 translation cache</td>
</tr>
<tr>
<td>rsp</td>
<td>pointer to the host stack for procedure calls</td>
</tr>
<tr>
<td>rax</td>
<td>general purpose translator temporary register</td>
</tr>
<tr>
<td>rcx</td>
<td>general purpose translator temporary register</td>
</tr>
</tbody>
</table>

Table 3 shows macro-op fusion patterns currently implemented by rv8.

rv8 currently only macro-op fuses instruction sequences where the destination register of an earlier RISC-V instruction is overwritten by a later instruction in the sequence. As the same restriction also applies to hardware macro-op fusion implementations, and macro-op fusion is likely to be a key technique in high performance RISC-V processors, we expect future compiler code generation to be tweaked to maximize fusion opportunities.

A technique known as deoptimisation [11] can be employed to allow elision of writes to temporary registers in macro-op fusion patterns assuming the translator sees the register overwritten within its translation window. Deoptimisation requires that the optimised translation has an accompanying deoptimisation sequence to fill in elided register values, and this is played back in the case of a fault (device or debug interrupt) so that the visible machine state precisely matches that which the ISA dictates. rv8 does not presently implement deoptimisation, however it may be necessary to allow more sophisticated optimisations.

3.8 Branch tail dynamic linking

The translator performs lazy translation of the source program during tracing. When it reaches a branch, the translator can only link both sides of the branch if there is an existing native code translation for the not taken side of the branch.

To accelerate branch tail exits, the translator emits a relative branch to a trampoline which is used to return the guest program counter to the tracer main loop. The tracer adds the branch to a table of branch fixup addresses indexed by target guest address. If the branch direction becomes hot, once it has been translated, all relative branches that point to the branch tail trampolines will be relinked to branch directly to the translated native code.

3.9 Sign extension versus zero extension

rv8 has to make sure that all 32-bit operations on registers have their result sign extended instead of zero-extended. The normal behaviour of 32-bit operations on x86-64 is to zero fill bits 32 to bit 63 whereas RISC-V requires implementations to *sign extend* bit 31 to bit 63 [21].
One potential optimisation is lazy sign extension. It may be possible in a future version of the JIT translation engine to elide redundant sign extension operations, however it is important that the register state precisely matches the ISA semantics before executing any instructions that may cause faults e.g. loads and stores.

### 3.10 Bit manipulation intrinsics

The benchmarks contain compression algorithms, digest algorithms and cryptographic ciphers which can take advantage of bit manipulation instructions such as rotate and bswap. Present day compilers detect rotate and byte swap bitwise logical operations by matching intermediate representation patterns that can be lowered directly to bit manipulation instructions such as ROR, ROL, BSWAP on x86-64.

Intermediate representation detection of bit manipulation intrinsics has the benefit of accelerating code that does not use inline assembly or compiler builtin functions. RISC-V currently lacks bit manipulation instructions however there are proposals to add them in the ‘B’ extension. The following is a typical byte swap pattern.

- 32-bit integer byteswap pattern
  
  \[
  (rs1 >> 24) \& 0x000000ff | (rs1 << 8 ) \& 0x00ff0000 \| (rs1 >> 8 ) \& 0x0000ff00 | (rs1 >> 24) \& 0xff000000
  \]

  rv8 implements rotate macro-op fusion which can translate two shift instructions and one ‘or’ instruction with the correct offsets into one shift and one rotate. The rotate macro-op fusion needs to preserve the residual temporary register side effects so that the register file contents are precisely matched, as it can’t easily prove the residual temporary register is not later used. Deoptimisation would be required to elide the temporary register write.

- 32-bit rotate right or left pattern
  
  \[
  (rs1 >> (shamt)) \| (rs1 << (32 - shamt))
  \]

- 64-bit rotate right or left pattern
  
  \[
  (rs1 >> (shamt)) \| (rs1 << (64 - shamt))
  \]

### 4 BENCHMARKS

To assess results of the rv8 binary translation strategies, a suite of computationally intensive benchmarks [8] were run comparing rv8 to QEMU riscv64 and QEMU aarch64. The rv8 benchmark suite includes the AES cipher, dhrystone v1.1, miniz compression and decompression, the NORX cipher, prime number generation, qsort and the SHA-512 digest algorithm.

This table summarises the performance ratio geomeans:

<table>
<thead>
<tr>
<th>optimisation</th>
<th>qemu aarch64</th>
<th>qemu riscv64</th>
<th>rv8 riscv64</th>
<th>native x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O3</td>
<td>4.68</td>
<td>4.57</td>
<td>2.62</td>
<td>1.00</td>
</tr>
<tr>
<td>-O0</td>
<td>4.62</td>
<td>5.20</td>
<td>2.47</td>
<td>1.00</td>
</tr>
</tbody>
</table>

See benchmark runtime bar charts and tables in Figure 5, Figure 6, Table 7 and Table 8, performance ratios in Table 9 and Table 10, and finally millions of instructions per second for emulated RISC-V and native x86-64 in Table 11 and Table 12.

### 5 CONCLUSION AND FUTURE WORK

The translation strategy employed by rv8 appears to have borne fruit when compared to other open source dynamic binary translators. Translation is simple and fast and the generated code is good enough for the overall performance to be on on average approximately 75% faster than the leading competitor. On some benchmarks we are approximately 30% slower than native x86-64 code. However we believe there is still a lot of remaining optimization potential in our approach that has not yet been realized.

rv8 implements a user mode simulator and an interpreted full system emulator modelling the RISC-V privileged ISA with an MMU, interrupts, MMIO devices and privilege levels. The current version of the rv8 JIT translator is presently only applied to user mode simulation so future work will explore applying dynamic binary translation to the full system emulator with hardware accelerated MMU using shadow paging. Future work may include:

- Exploring dynamic register allocation algorithms.
- Coalescing loads and stores into complex memory operands.
- Register write elision for pipelined operations with short live spans, to make better use of translator temporary registers.
- Hardware accelerated MMU using shadow paging.

### ACKNOWLEDGMENTS

The authors would like to thank Petr Kobalicek for providing the excellent AsmJit [16] library used for rv8’s x86-64 back-end support.
Figure 5: Runtimes 64-bit -O3

Figure 6: Runtimes 64-bit -Os

Figure 7: Benchmark Runtimes -O3 (seconds)

Figure 8: Benchmark Runtimes -Os (seconds)

Figure 9: Performance Ratio -O3

Figure 10: Performance Ratio -Os

Figure 11: Millions of Instructions Per Second -O3

Figure 12: Millions of Instructions Per Second -Os
REFERENCES


