Binary Translation Using Peephole Superoptimizers

Sorav Bansal
Computer Systems Lab
Stanford University
sbansal@cs.stanford.edu

Alex Aiken
Computer Systems Lab
Stanford University
aiken@cs.stanford.edu

Abstract

We present a new scheme for performing binary translation that produces code comparable to or better than existing binary translators with much less engineering effort. Instead of hand-coding the translation from one instruction set to another, our approach automatically learns translation rules using superoptimization techniques. We have implemented a PowerPC-x86 binary translator and report results on small and large compute-intensive benchmarks. When compared to the native compiler, our translated code achieves median performance of 67% on large benchmarks and in some small stress tests actually outperforms the native compiler. We also report comparisons with the open source binary translator Qemu and a commercial tool, Apple's Rosetta. We consistently outperform the former and are comparable to or faster than the latter on all but one benchmark.

1 Introduction

A common worry for machine architects is how to run existing software on new architectures. One way to deal with the problem of software portability is through *binary translation*. Binary translation enables code written for a *source architecture* (or instruction set) to run on another *destination architecture*, without access to the original source code. A good example of the application of binary translation to solve a pressing software portability problem is Apple's Rosetta, which enabled Apple to (almost) transparently move its existing software for the Power-based Macs to a new generation of Intel x86-based computers [1].

Building a good binary translator is not easy, and few good binary translation tools exist today. There are four main difficulties:

1. Some performance is normally lost in translation. Better translators lose less, but even good translators often lose one-third or more of source architecture performance for compute-intensive applications.

2. Because the instruction sets of modern machines tend to be large and idiosyncratic, just writing the translation rules from one architecture to another is a significant engineering challenge, especially if there are significant differences in the semantics of the two instruction sets. This problem is exacerbated by the need to perform optimizations wherever possible to minimize problem (1).

3. Because high-performance translations must exploit architecture-specific semantics to maximize performance, it is challenging to design a binary translator that can be quickly retargeted to new architectures. One popular approach is to design a common intermediate language that covers all source and destination architectures of interest, but to support needed performance this common language generally must be large and complex.

4. If the source and destination architectures have different operating systems then source system calls must be emulated on the destination architecture. Operating systems' large and complex interfaces combined with subtle and sometimes undocumented semantics and bugs make this a major engineering task in itself.

Our work presents a new approach to addressing problems (1)-(3) (we do not address problem (4)). The main idea is that much of the complexity of writing an aggressively optimizing translator between two instruction sets can be eliminated altogether by developing a system that automatically and systematically learns translations. In Section 6 we present performance results showing that this approach is capable of producing destination machine code that is at least competitive with existing state-of-the-art binary translators, addressing problem (1). While we cannot meaningfully compare the engineering effort needed to develop our research project with what goes into commercial tools, we hope to convince the reader that, on its face, automatically learning translations must require far less effort than hand coding translations between architectures, addressing problem (2). Similarly, we believe our approach helps resolve the tension between performance and retargetability: adding a new architecture requires only a parser for the binary format and a description of the instruction set semantics (see Section 3). This is the minimum that any binary translator would require to incorporate a new architecture; in particular, our approach has no intermediate language that must be expanded or tweaked to accommodate the unique features of an additional architecture.

Our system uses *peephole rules* to translate code from one architecture to another. Peephole rules are pattern matching rules that replace one sequence of instructions by another equivalent sequence of instructions. Peephole rules have traditionally been used for compiler-optimizations, where the rules are used to replace a sub-optimal instruction sequence in the code by another equivalent, but faster, sequence. For our binary translator, we use peephole rules that replace a source-architecture instruction sequence by an equivalent destination architecture instruction sequence. For example,

```
ld [r2]; addi 1; st [r2] =>
inc [er3]  { r2 = er3 }
```

is a peephole translation rule from a certain accumulator-based RISC architecture to another CISC architecture. In this case, the rule expresses that the operation of loading a value from memory location `[r2]`, adding 1 to it and storing it back to `[r2]` on the RISC machine can be achieved by a single in-memory increment instruction on location `[er3]` on the CISC machine, where RISC register `r2` is emulated by CISC register `er3`.

The number of peephole rules required to correctly translate a complete executable for any source-destination architecture pair can be huge and manually impossible to write. We automatically learn peephole translation rules using *superoptimization* techniques: essentially, we exhaustively enumerate possible rules and use formal verification techniques to decide whether a candidate rule is a correct translation or not. This process is slow; in our experiments it required about a processor-week to learn enough rules to translate full applications. However, the search for translation rules is only done once, off-line, to construct a binary translator; once discovered, peephole rules are applied to any program using simple pattern matching, as in a standard peephole optimizer. Superoptimization has been previously used in compiler optimization [5, 10, 14], but our work is the first to develop superoptimization techniques for binary translation.

Binary translation preserves execution semantics on two different machines: whatever result is computed on one machine should be computed on the other. More precisely, if the source and destination machines begin in equivalent states and execute the original and translated programs respectively, then they should end in equivalent states. Here, *equivalent states* implies we have a mapping telling us how the states of the two machines are related. In particular, we must decide which registers/memory locations on the destination machine emulate which registers/memory locations of the source machine.

Note that the example peephole translation rule given above is conditioned by the *register map* `r2 = er3`. Only when we have decided on a register map can we compute possible translations. The choice of register map turns out to be a key technical problem: better decisions about the register map (e.g., different choices of destination machine registers to emulate source machine registers) lead to better performing translations. Of course, the choice of instructions to use in the translation also affects the best choice of register map (by, for example, using more or fewer registers), so the two problems are mutually recursive. We present an effective dynamic programming technique that finds the best register map and translation for a given region of code (Section 3.3).

We have implemented a prototype binary translator from PowerPC to x86. Our prototype handles nearly all of the PowerPC and x86 opcodes and using it we have successfully translated large executables and libraries. We report experimental results on a number of small compute-intensive microbenchmarks, where our translator surprisingly often outperforms the native compiler. We also report results on many of the SPEC integer benchmarks, where the translator achieves a median performance of around 67% of natively compiled code and compares favorably with both Qemu [17], an open source binary translator, and Apple's Rosetta [1]. While we believe these results show the usefulness of using superoptimization as a binary translation and optimization tool, there are two caveats to our experiments that we discuss in more detail in Section 6. First, we have not implemented translations of all system calls. As discussed above under problem (4), this is a separate and quite significant engineering issue. We do not believe there is any systematic bias in our results as a result of implementing only enough system calls to run many, but not all, of the SPEC integer benchmarks. Second, our system is currently a static binary translator, while the systems we compare to are dynamic binary translators, which may give our system an advantage in our experiments as time spent in translation is not counted as part of the execution time. There is nothing that prevents our techniques from being used in a dynamic translator; a static translator was just easier to develop given our initial tool base. We give a detailed analysis of translation time, which allows us to bound the additional cost that would be incurred in a dynamic translator.

In summary, our aim in this work is to demonstrate the ability to develop binary translators with competitive performance at much lower cost. Towards this end, we make the following contributions:

- We present a design for automatically learning binary translations using an off-line search of the space of candidate translation rules.
- We identify the problem of selecting a register map and give an algorithm for simultaneously computing the best register map and translation for a region of code.
- We give experimental results for a prototype PowerPC to x86 translator, which produces consistently high performing translations.

The rest of this paper is organized as follows. We begin with a discussion on the recent applications of binary translation (Section 2). We then provide a brief overview of peephole superoptimizers followed by a discussion on how we employ them for binary translation (Section 3). We discuss other relevant issues involved in binary translation (Section 4) and go on to discuss our prototype implementation (Section 5). We then present our experimental results (Section 6), discuss related work (Section 7), and finally conclude (Section 8).

2 Applications

Before describing our binary translation system, we give a brief overview of a range of applications for binary translation. Traditionally, binary translation has been used to emulate legacy architectures on recent machines. With improved performance, it is now also seen as an acceptable portability solution.

Binary translation is useful to hardware designers for ensuring software availability for their new architectures. While the design and production of new architecture chips complete within a few years, it can take a long time for software to be available on the new machines. To deal with this situation and ensure early adoption of their new designs, computer architects often turn to software solutions like virtual machines and binary translation [7].

Another interesting application of binary translation for hardware vendors is backward and forward compatibility of their architecture generations. To run software written for older generations, newer generations are forced to support backward compatibility. On the flip side, it is often not possible to run newer generation software on older machines. Both of these problems create compatibility headaches for computer architects and huge management overheads for software developers. It is not hard to imagine the use of a good binary-translation based solution to solve both problems in the future.

Binary translation is also being used for machine and application virtualization. Leading virtualization companies are now considering support for allowing the execution of virtual machines from multiple architectures on a single host architecture [20]. Hardware vendors are also developing virtualization platforms that allow people to run popular applications written for other architectures on their machines [16]. Server farms and data centers can use binary translation to consolidate their servers, thus cutting their power and management costs.

People have also used binary translation to improve performance and reduce power consumption in hardware. Transmeta Crusoe [12] employs on-the-fly binary translation to execute x86 instructions on a VLIW architecture thereby cutting power costs [11]. Similarly, in software, many Java virtual machines perform on-the-fly translation from Java bytecode to the host machine instructions [25] to improve execution performance.

3 Binary Translation Using Peephole Superoptimizers

In this section we give a necessarily brief overview of the design and functionality of peephole superoptimizers, focusing on the aspects that are important in the adaptation to binary translation.

3.1 Peephole Superoptimizers

Peephole superoptimizers are an unusual type of compiler optimizer [5, 10, 14], and for brevity we usually refer to a peephole superoptimizer as simply an optimizer. For our purposes, constructing a peephole superoptimizers has three phases:

1. A module called the *harvester* extracts *target instruction sequences* from a set of training programs. These are the instruction sequences we seek to optimize.
2. A module called the *enumerator* enumerates all possible instruction sequences up to a certain length. Each enumerated instruction sequence s is checked to see if it is equivalent to any target instruction sequence t. If s is equivalent to some target sequence t and s is cheaper according to a cost function (e.g., estimated execution time or code size) than any other sequence known to be equivalent to t (including t itself), then s is recorded as the

best known replacement for t. A few sample peephole optimization rules are shown in Table 1.

3. The learned (target sequence, optimal sequence) pairs are organized into a lookup table indexed by target instruction sequence.

Once constructed, the optimizer is applied to an executable by simply looking up target sequences in the executable for a known better replacement. The purpose of using harvested instruction sequences is to focus the search for optimizations on the code sequences (usually generated by other compilers) that appear in actual programs. Typically, all instruction sequences up to length 5 or 6 are harvested, and the enumerator tries all instruction sequences up to length 3 or 4. Even at these lengths, there are billions of enumerated instruction sequences to consider, and techniques for pruning the search space are very important [5]. Thus, the construction of the peephole optimizer is time-consuming, requiring a few processor-days. In contrast, actually applying the peephole optimizations to a program typically completes within a few seconds.

The enumerator's equivalence test is performed in two stages: a fast execution test and a slower boolean test. The execution test is implemented by executing the target sequence and the enumerated sequence on hardware and comparing their outputs on random inputs. If the execution test does not prove that the two sequences are different (i.e., because they produce different outputs on some tested input), the boolean test is used. The equivalence of the two instruction sequences is expressed as boolean formula: each bit of machine state touched by either sequence is encoded as a boolean variable, and the semantics of instructions is encoded using standard logical connectives. A SAT solver is then used to test the formula for satisfiability, which decides whether the two sequences are equal.

Using these techniques, *all* length-3 x86 instruction sequences have previously been enumerated on a single processor in less than two days [5]. This particular superoptimizer is capable of handling opcodes involving flag operations, memory accesses and branches, which on most architectures covers almost all opcodes. Equivalence of instruction sequences involving memory accesses is correctly computed by accounting for the possibility of aliasing. The optimizer also takes into account live register information, allowing it to find many more optimizations because correctness only requires that optimizations preserve live registers (note the live register information qualifying the peephole rules in Table 1).

Target Sequence	Live Registers	Equivalent Enumerated Sequence
`movl (%eax), %ecx` `movl %ecx, (%eax)`	`eax,ecx`	`movl (%eax), %ecx`

`sub %eax, %ecx`
`mov %ecx, %eax`
`dec %eax`
	`eax`	`not %eax`
		`add %ecx, %eax`

`sub %eax, %ecx`
`test %ecx, %ecx`
`je .END`
`mov %edx, %ebx`
`.END:`
	`eax,ecx,`
	`edx,ebx`
		`sub %eax, %ecx`
		`cmovne %edx, %ebx`

Table 1: Examples of peephole rules generated by a superoptimizer for x86 executables

3.2 Binary Translation

We discuss how we use a peephole superoptimizer to perform efficient binary translation. The approach is similar to that discussed in Section 3.1, except that now our target sequences belong to the source architecture while the enumerated sequences belong to the destination architecture.

The binary translator's harvester first extracts target sequences from a training set of source-architecture applications. The enumerator then enumerates instruction sequences on the destination architecture checking them for equivalence with any of the target sequences. A key issue is that the definition of equivalence must change in this new setting with different machine architectures. Now, equivalence is meaningful only with respect to a *register map* showing which memory locations on the destination machine, and in particular registers, emulate which memory locations on the source machine; some register maps are shown in Table 2. A register in the source architecture could be mapped to a register or a memory location in the destination architecture. It is also possible for a memory location in the source architecture to be mapped to a register in the destination architecture.

A potential problem is that for a given source-architecture instruction sequence there may be many valid register maps, yielding a large number of (renamed) instruction sequences on the target-architecture that must be tested for equivalence. For example, the two registers used by the PowerPC register move instruction `mr r1,r2` can be mapped to the 8 registers of the x86 in 8*7=56 different ways. Similarly, there may be many variants of a source-architecture instruction sequence. For example, on the 32 register PowerPC, there are 32*31=992 variants of `mr r1, r2`. We avoid these problems by eliminating source-architecture sequences

Register Map	Description
`r1`→`eax`	Maps PowerPC register to x86 register
`r1`→M_1	Maps PowerPC register to a memory location
M_s→`eax`	Maps a memory location in source code to a register in the translated code
`r1`→`eax` `r2`→`eax`	Invalid. Cannot map two PowerPC registers to the same x86 register
M_s→M_t	Maps one memory location to another (e.g. address space translation)

Table 2: Some valid (and invalid) register maps from PowerPC-x86 translation (M_i refers to a memory location).

PowerPC Sequence	Live Registers	State Map	x86 Instruction Sequence
`mr r1,r2`	`r1,r2`	`r1`→`eax` `r2`→M_1	`movl` M_1`,eax`
`lwz r1,(r2)`	`r1,r2`	`r1`→`eax` `r2`→`ecx`	`mov (ecx),eax` `bswap eax`

`lwz r1,(r2)`
`stw r1,(r3)`
	`r1,r2,`
	`r3`
		`r1`→`eax`
		`r2`→`ecx`
		`r3`→`edx`
			`movl (ecx),eax`
			`movl eax,(edx)`

`mflr r1`	`r1,lr`	`r1`→`eax` `lr`→`ecx`	`movl ecx,eax`
`lis r1,C0` `ori r1,r1,C1`	`r1`	`r1`→`eax`	`mov $C0C1,eax`

`subfc r1,r2,r1`
`adde r1,r1,r3`
	`r1,r2`
	`r3`
		`r1`→`eax`
		`r2`→`ecx`
		`r3`→`edx`
			`subl ecx,eax`
			`adcl edx,eax`

Table 3: Examples of peephole translation rules from PowerPC to x86. The x86 sequences are written in AT&T syntax assembly with % signs omitted before registers.

that are register renamings of one canonically-named sequence and by considering only one register map for all register maps that are register renamings of each other. During translation, a target sequence is (canonically) renamed before searching for a match in the peephole table and the resulting translated sequence is renamed back before writing it out. Further details of this *canonicalization* optimization are in [5].

When an enumerated sequence is found to be equivalent to a target sequence, the corresponding peephole rule is added to the translation table together with the register map under which the translation is valid. Some examples of peephole translation rules are shown in Table 3.

Once the binary translator is constructed, using it is relatively simple. The translation rules are applied to the source-architecture code to obtain destination-architecture code. The application of translation rules is more involved than the application of optimization rules. Now, we also need to select the register map for each code point before generating the corresponding translated code. The choice of register map can make a noticeable difference to the performance of generated code. We discuss the selection of optimal register maps at translation time in the next section.

3.3 Register Map Selection

Choosing a good register map is crucial to the quality of translation, and moreover the best code may require changing the register map from one code point to the next. Thus, the best register map is the one that minimizes the cost of the peephole translation rule (generates the fastest code) plus any cost of switching register maps from the previous program point—because switching register maps requires adding register move instructions to the generated code to realize the switch at run-time, switching register maps is not free.

We formulate a dynamic programming problem to choose a minimum cost register map at each program point in a given code region. At each code point, we enumerate all register maps that are likely to produce a translation. Because the space of all register maps is huge, it is important to constrain this space by identifying only the relevant register maps. We consider the register maps at all predecessor code points and extend them using the registers used by the current target sequence. For each register used by the current target sequence, all possibilities of register mappings (after discarding register renamings) are enumerated. Also, we attempt to avoid enumerating register maps that will produce an identical translation to a register map that has already been enumerated, at an equal or higher cost. For each enumerated register map M, the peephole translation table is queried for a matching translation rule T and the corresponding translation cost is recorded. We consider length-1 to length-3 instruction sequences while querying the peephole table for each enumerated register map. Note that there may be multiple possible translations at a given code point, just as there may be multiple valid register maps; we simply keep track of all possibilities. The dynamic programming problem formulation then considers the translation produced for each sequence length while determining the optimal translation for a block of code. Assume for simplicity that the code point under consideration has only one predecessor, and the possible register maps at the predecessor are $P_1, \ldots, P_n$. For simplicity, we also assume that we are translating one instruction at a time and that T is the minimum cost translation for that instruction if register map M is used. The best cost register map M is then the one that minimizes the cost of switching from a predecessor map P_i to M, the cost of the instruction sequence T, and, recursively, the cost of P_i:

$$cost(M) = cost(T) + min_i(cost(P_i) + switch(P_i, M))$$

To translate multiple source-architecture instructions using a single peephole rule, we extend this approach to consider translations for all length-1 to length-3 sequences that end at the current code point. For example, at the end of instruction i_4 in an instruction sequence (i_1, i_2, i_3, i_4), we search for possible translations for each of the sequences (i_4), (i_3, i_4) and (i_2, i_3, i_4) to find the lowest-cost translation. While considering a translation for the sequence (i_2, i_3, i_4), the predecessor register maps considered are the register maps at instruction i_1. Similarly, the predecessor register maps for sequences (i_3, i_4) and (i_4) are maps at instructions i_2 and i_3 respectively. The cost of register map M is then the minimum among the costs computed for each sequence length.

We solve the recurrence in a standard fashion. Beginning at start of a code region (e.g., a function body), the cost of the preceding register map is initially 0. Working forwards through the code region, the cost of each enumerated register map is computed and stored before moving to the next program point and repeating the computation. When the end of the code region is reached, the register map with the lowest cost is chosen and its decisions are backtracked to decide the register maps and translations at all preceding program points. For program points having multiple predecessors, we use a weighted sum of the switching costs from each predecessor. To handle loops, we perform two iterations of this computation. Interesting examples are too lengthy to include here, but a detailed, worked example of register map selection is in [4].

This procedure of enumerating all register maps and then solving a dynamic programming problem is computationally intensive and, if not done properly, can significantly increase translation time. While the cost of finding the best register map for every code point is not a problem for a static translator, it would add significant overhead to a dynamic translator. To bound the computation time, we prune the set of enumerated register maps at each program point. We retain only the n lowest-cost register maps before moving to the next program point. We allow the value of n to be tunable and refer to it as the *prune size*. We also have the flexibility to trade computation time for lower quality solutions. For example, for code that is not performance critical we can consider code regions of size one (e.g., a single instruction) or even use a fixed register map. In Section 6 we show that the cost of computing the best register maps for frequently executed instructions is very small for our benchmarks. We also discuss the performance sensitivity of our benchmarks to the prune size.

4 Other Issues

In this section, we discuss the main issues relevant to our approach to binary translation.

4.1 Static vs Dynamic Translation

Binary translation can either be performed statically (compile-time) or dynamically (runtime). Most existing tools perform binary translation dynamically for its primary advantage of having a complete view of the current machine state. Moreover, dynamic binary translation provides additional opportunities for runtime optimizations. The drawback of dynamic translation is the overhead of performing translation and book-keeping at runtime. A static translator translates programs offline and can apply more extensive (and potentially whole program) optimizations. However, performing faithful static translation is a slightly harder problem since no assumptions can be made about the runtime state of the process.

Our binary translator is static, though we have avoided including anything in our implementation that would make it impractical to develop a dynamic translator (e.g., whole-program analysis or optimizations) using the same algorithms. Most of the techniques we discuss are equally applicable in both settings and, when they are not, we discuss the two separately.

4.2 Endianness

If the source and destination architectures have different endianness, we convert all memory reads to destination endianness and all memory writes to source endianness. This policy ensures that memory is always in source endianness while registers have destination endianness. The extra byte-swap instructions needed to maintain this invariant are only needed on memory accesses; in particular, we avoid the additional overhead of shuffling bytes on register operations.

While dealing with source-destination architecture pairs with different endianness, special care is required in handling OS-related data structures. In particular, all executable headers, environment variables and program arguments in the program's address space need to be converted from destination endianness to source endianness before transferring control to the translated program. This step is necessary because the source program assumes source endianness for everything while the OS writes the data structures believing that the program assumes destination endianness. In a dynamic translator, these conversions are performed inside the translator at startup. In a static translator, special initialization code is emitted to perform these conversions at runtime.

4.3 Control Flow Instructions

Like all other opcodes, control flow instructions are also translated using peephole rules. Direct jumps in the source are translated to direct jumps in the translated code, with the jump destination being appropriately adjusted to point to the corresponding translated code. Our superoptimizer is capable of automatically learning translations involving direct jump instructions.

To handle conditional jumps, the condition codes of the source architecture need to be faithfully represented in the destination architecture. Handling condition codes correctly is one of the more involved aspects of binary translation because of the divergent condition-code representations used by different architectures. We discuss our approach to handling condition codes in the context of our PowerPC-x86 binary translator; see Section 5.3. The handling of indirect jumps is more involved and is done differently for static and dynamic translators. We discuss this in detail in Section 5.4.

4.4 System Calls

When translating across two different operating systems, each source OS system call needs to be emulated on the destination OS. Even when translating across the same operating system on different architectures, many system calls require special handling. For example, some system calls are only implemented for specific architectures. Also, if the two architectures have different endianness, proper endianness conversions are required for all memory locations that the system call could read or write. There are other relevant issues to binary translation that we do not discuss here: full system vs. user-level emulation, address translation, precise exceptions, misaligned memory accesses, interprocess communication, signal handling, etc. These problems are orthogonal to the issues in peephole binary translation and our solutions to these issues are standard. In this paper, our focus is primarily on efficient code-generation.

5 Implementation

We have implemented a binary translator that allows PowerPC/Linux executables to run in an x86/Linux environment. The translator is capable of handling almost all PowerPC opcodes (around 180 in all). We have tested our implementation on a variety of different executables and libraries.

The translator has been implemented in C/C++ and O'Caml [13]. Our superoptimizer is capable of automatically inferring peephole translation rules from PowerPC to x86. Because we cannot execute both the target sequence and the enumerated sequence on the same machine, we use a PowerPC emulator (we use Qemu in our experiments) to execute the target sequence. Recall from Section 3.1 that there are two steps to determining which, if any, target instruction sequences are equivalent to the enumerated instruction sequence: first a fast execution test is used to eliminate all but few plausible candidates, and then a complete equivalence check is done by converting both instruction sequences to boolean formulas and deciding a satisfiability query. We use zChaff [15, 26] as our backend SAT solver. We have translated most, but not all, Linux PowerPC system calls. We present our results using a static translator that produces an x86 ELF 32-bit binary executable from a PowerPC ELF 32-bit binary. Because we used the static peephole superoptimizer described in [5] as our starting point, our binary translator is also static, though as discussed previously our techniques could also be applied in a dynamic translator. A consequence of our current implementation is that we also translate all the shared libraries used by the PowerPC program.

In this section, we discuss issues specific to a PowerPC-x86 binary translator. While there exist many architecture-specific issues (as we discuss in this section), the vast bulk of the translation and optimization complexity is still hidden by the superoptimizer.

5.1 Endianness

PowerPC is a big-endian architecture while x86 is a little-endian architecture, which we handle using the scheme outlined in Section 4.2. For integer operations, there exist three operand sizes in PowerPC: 1, 2 and 4 bytes. Depending on the operand size, the appropriate conversion code is required when reading from or writing to memory. We employ the convenient `bswap` x86 instruction to generate efficient conversion code.

5.2 Stack and Heap

On Linux, the stack is initialized with `envp`, `argc` and `argv` and the stack pointer is saved to a canonical register at load time. On x86, the canonical register storing the stack pointer is `esp`; on PowerPC, it is `r1`. When the translated executable is loaded in an x86 environment (in the case of dynamic translation, when the translator is loaded), the `esp` register is initialized to the stack pointer by the operating system while the emulated `r1` register is

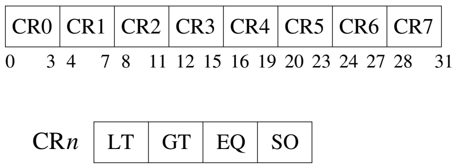


Figure 1: PowerPC architecture has support for eight independent sets of condition codes `CR0-CR7`. Each 4-bit `CR`*n* register uses one bit each to represent less than (`LT`), greater (`GT`), equal (`EQ`) and overflow-summary (`SO`). Explicit instructions are required to read/write the condition code bits.

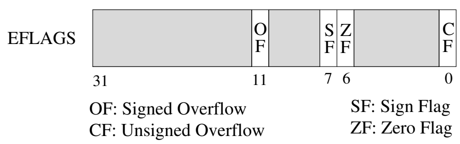


Figure 2: The x86 architecture supports only a single set of condition codes represented as bits in a 32-bit `EFLAGS` register. Almost all x86 instructions overwrite these condition codes.

left uninitialized. To make the stack visible to the translated PowerPC code, we copy the `esp` register to the emulated `r1` register at startup. In dynamic translation, this is done by the translator; in static translation, this is done by the initialization code. The handling of the heap requires no special effort since the `brk` Linux system call used to allocate heap space is identical on both x86 and PowerPC.

5.3 Condition Codes

Condition codes are bits representing quantities such as carry, overflow, parity, less, greater, equal, etc. PowerPC and x86 handle condition codes very differently. Figures 1 and 2 show how condition codes are represented in PowerPC and x86 respectively.

While PowerPC condition codes are written using separate instructions, x86 condition codes are overwritten by almost all x86 instructions. Moreover, while PowerPC compare instructions explicitly state whether they are doing a signed or an unsigned comparison and store only one result in their flags, x86 compare instructions perform both signed and unsigned comparisons and store both results in their condition bits. On x86, the branch instruction then specifies which comparison it is interested in (signed or unsigned). We handle these differences by allowing the PowerPC condition registers (`cr0`-`cr7`) to be mapped to x86 flags in the register map. For example, an entry `cr0`→`SF` in the register map specifies that, at that program point, the contents of register `cr0` are encoded in the x86 signed flags (`SF`). The translation of a branch instruction then depends on whether the condition register being used ($\mathtt{cr}_i$) is mapped to signed (`SF`) or unsigned (`UF`) flags.

5.4 Indirect Jumps

Jumping to an address in a register (or a memory location) is an *indirect* jump. Function pointers, dynamic loading, and case statements are all handled using indirect jumps. Since an indirect jump could jump almost anywhere in the executable, it requires careful handling. Moreover, because the destination of the indirect jump could assume a different register-map than the current one, the appropriate conversion needs to be performed before jumping to the destination. Different approaches for dealing with indirect jumps are needed in static and dynamic binary translators.

Handling an indirect jump in a dynamic translator is simpler. Here, on encountering an indirect jump, we relinquish control to the translator. The translator then performs the register map conversion before transferring control to the (translated) destination address.

Handling an indirect jump in a static translator is more involved. We first identify all instructions that can be possible indirect jump targets. Since almost all well-formed executables use indirect jumps in only a few different code paradigms, it is possible to identify possible indirect jump targets by scanning the executable. We scan the read-only data sections, global offset tables and instruction immediate operands and use a set of pattern matching rules to identify possible indirect jump targets. A lookup table is then constructed to map these jump targets (which are source architecture addresses) to their corresponding destination architecture addresses. However, as we need to perform register map conversion before jumping to the destination address at runtime, we replace the destination addresses in the lookup table with the address of a code fragment that performs the register-map conversion before jumping to the destination address.

The translation of an indirect jump involves a table lookup and some register-map conversion code. While the table lookup is fast, the register-map conversion may involve multiple memory accesses. Hence, an indirect jump is usually an expensive operation.

Although the pattern matching rules we use to identify possible indirect jump targets have worked extremely well in practice, they are heuristics and are prone to adversarial attacks. It would not be difficult to construct an executable that exploits these rules to cause a valid PowerPC program to crash on x86. Hence, in an adver-

ppc	x86	Comparison
`bl`	`call`	`bl` (branch-and-link) saves the instruction pointer to register `lr` while `call` pushes it to stack
`blr`	`ret`	`blr` (branch-to-link-register) jumps to the address pointed-to by `lr`, while `ret` pops the instruction pointer from the stack and jumps to it

Table 4: Function call and return instructions in PowerPC and x86 architectures

sarial scenario, it would be wise to assume that all code addresses are possible indirect jump targets. Doing so results in a larger lookup table and more conversion code fragments, increasing the overall size of the executable, but will have no effect on running time apart from possible cache effects.

5.5 Function Calls and Returns

Function calls and returns are handled in very different ways in PowerPC and x86. Table 4 lists the instructions and registers used in function calls and returns for both architectures.

We implement function calls of the PowerPC architecture by simply emulating the link-register (`lr`) like any other PowerPC register. On a function call (`bl`), the link register is updated with the value of the next PowerPC instruction pointer. A function return (`blr`) is treated just like an indirect jump to the link register.

The biggest advantage of using this scheme is its simplicity. However, it is possible to improve the translation of the `blr` instruction by exploiting the fact that `blr` is always used to return from a function. For this reason, it is straightforward to predict the possible jump targets of `blr` at translation time (it will be the instruction following the function call `bl`). At runtime, the value of the link register can then be compared to the predicted value to see if it matches, and then jump accordingly. This information can be used to avoid the extra memory reads and writes required for register map conversion in an indirect jump. We have implemented this optimization; while this optimization provides significant improvements while translating small recursive benchmarks (e.g., recursive computation of the fibonacci series), it is not very effective for larger benchmarks (e.g., SPEC CINT2000).

5.6 Register Name Constraints

Another interesting challenge while translating from PowerPC to x86 is dealing with instructions that operate

Opcode	Registers	Description
`mul` *reg32*	`eax, edx`	Multiplies *reg32* with eax and stores the 64-bit result in `edx:eax`.
`div` *reg32*	`eax, edx`	Divides `edx:eax` by *reg32* and stores result in `eax`.
any 8-bit insn	`eax, ebx` `ecx, edx`	8-bit operations can only be performed on these four registers.

Table 5: Examples of x86 instructions that operate only on certain fixed registers.

only on specific registers. Such instructions are present on both PowerPC and x86. Table 5 shows some such x86 instructions.

To be able to generate peephole translations involving these special instructions, the superoptimizer is made aware of the constraints on their operands during enumeration. If a translation is found by the superoptimizer involving these special instructions, the generated peephole rule encodes the name constraints on the operands as *register name constraints*. These constraints are then used by the translator at code generation time.

5.7 Self-Referential and Self-Modifying Code

We handle self-referential code by leaving a copy of the source architecture code in its original address range for the translated version. To deal with self-modifying code and dynamic loading, we would need to invalidate the translation of a code fragment on observing any modification to that code region. To do this, we would trap any writes to code regions and perform the corresponding invalidation and re-translation. For a static translator, this involves making the translator available as a shared library—a first step towards a full dynamic translator. While none of our current benchmarks contain self-modifying code, it would be straightforward to extend our translator to handle such scenarios.

5.8 Untranslated Opcodes

For 16 PowerPC opcodes our translator failed to find a short equivalent x86 sequence of instructions automatically. In such cases, we allow manual additions to the peephole table. Table 6 describes the number and types of hand additions: 9 are due to instructions involving indirect jumps and 7 are due to complex PowerPC instructions that cannot be emulated using a bounded length straight-line sequence of x86 instructions. For some

Number of Additions	Reason
2	Overflow/underflow semantics of the divide instruction (`div`)
2	Overflow semantics of `srawi` shift instruction
1	The rotate instruction `rlwinm`
1	The `cntlzw` instruction
1	The `mfcr` instruction
9	Indirect jumps referencing the jumptable

Table 6: The distribution of the manual translation rules we added to the peephole translation table.

more complex instructions mostly involving interrupts and other system-related tasks, we used the slow but simple approach of emulation using C-code.

5.9 Compiler Optimizations

An interesting observation while doing our experiments was that certain compiler optimizations often have an adverse effect on the performance of our binary translator. For example, an optimized PowerPC executable attempts to use all 8 condition-registers (`cr0-cr7`). However, since x86 has only one set of flags, other condition registers need to be emulated using x86 registers causing extra register pressure. Another example of an unfriendly compiler optimization is instruction scheduling. An optimizing PowerPC compiler separates two instructions involving a data dependency to minimize pipeline stalls, while our binary translator would like the data-dependent instructions to be together to allow the superoptimizer to suggest more aggressive optimizations. Our implementation reorders instructions within basic blocks to minimize the length of dependencies prior to translation.

6 Experimental Results

We performed our experiments using a Linux machine with a single Intel Pentium 4 3.0GHz processor, 1MB cache and 4GB of memory. We used `gcc` version 4.0.1 and `glibc` version 2.3.6 to compile the executables on both Intel and PowerPC platforms. To produce identical compilers, we built the compilers from their source tree using exactly the same configuration options for both architectures. While compiling our benchmarks, we used the `-msoft-float` flag in `gcc` to emulate floating point operations in software; our translator currently does not translate floating point instructions. For all our benchmarks except one, emulating floating point in software makes no difference in performance. All the executables were linked statically and hence, the libraries were also converted from PowerPC to x86 at translation time. To emulate some system-level PowerPC instructions, we used C-code from the open source emulator Qemu [17].

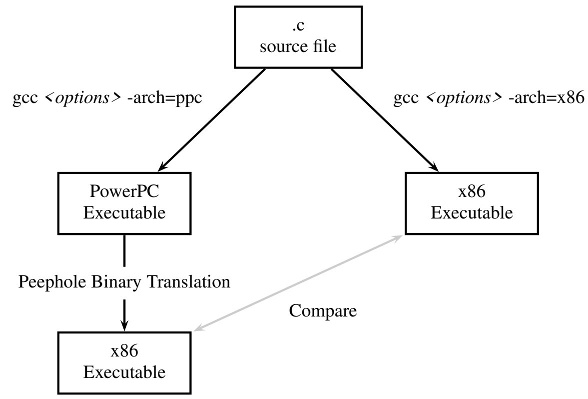


Figure 3: Experimental Setup. The translated binary executable is compared with the natively-compiled x86 executable. While comparing, the same compiler optimization options are used on both branches.

In our experiments, we compare the executable produced by our translator to a natively-compiled executable. The experimental setup is shown in Figure 3. We compile from the C source for both PowerPC and x86 platforms using `gcc`. The same compiler optimization options are used for both platforms. The PowerPC executable is then translated using our binary translator to an x86 executable. And finally, the translated x86 executable is compared with the natively-compiled one for performance.

One would expect the performance of the translated executable to be strictly lower than that of the natively-compiled executable. To get an idea of the state-of-the-art in binary translation, we discuss two existing binary translators. A general-purpose open-source emulator, Qemu [17], provides 10–20% of the performance of a natively-compiled executable (i.e., 5–10x slowdown). A recent commercially available tool by Transitive Corporation [22] (which is also the basis of Apple's Rosetta translator) claims "typically about 70–80%" of the performance of a natively-compiled executable on their website [18]. Both Qemu and Transitive are dynamic binary translators, and hence Qemu and Rosetta results include the translation overhead, while the results for our static translator do not. We estimate the translation overhead of our translator in Section 6.1.

Table 7 shows the performance of our binary translator on small compute-intensive microbenchmarks. (All

Benchmark	Description	`-O0`	`-O2`	`-O2ofp`
`emptyloop`	A bounded for-loop doing nothing	98.56 %	128.72 %	127 %
`fibo`	Compute first few Fibonacci numbers	118.90 %	319.13 %	127.78 %
`quicksort`	Quicksort on 64-bit integers	81.36 %	92.61 %	90.23 %
`mergesort`	Mergesort on 64-bit integers	83.22 %	91.54 %	84.35 %
`bubblesort`	Bubble-sort on 64-bit integers	75.12 %	70.92 %	64.86 %
`hanoi1`	Towers of Hanoi Algorithm 1	84.83 %	70.03 %	61.96 %
`hanoi2`	Towers of Hanoi Algorithm 2	107.14 %	139.64 %	143.69 %
`hanoi3`	Towers of Hanoi Algorithm 3	81.04 %	90.14 %	80.15 %
`traverse`	Traverse a linked list	69.06 %	67.67 %	67.15 %
`binsearch`	Perform binary search on a sorted array	65.38 %	61.24 %	62.15 %

Table 7: Performance of the binary translator on some compute-intensive microbenchmarks. The columns represent the optimization options given to `gcc`. '`-O2ofp`' expands to '`-O2 -fomit-frame-pointer`'. '`-O2ofp`' omits storing the frame pointer on x86. On PowerPC, '`-O2ofp`' is identical to '`-O2`'. The performance is shown relative to a natively compiled application (the performance of a natively compiled application is 100%).

	O0			O2		
	native (secs)	peep (secs)	% of native	native (secs)	peep (secs)	% of native
`bzip2`	311	470	66.2 %	195	265	73.7 %
`gap`	165	313	52.5 %	87	205	42.5 %
`gzip`	264	398	66.3 %	178	315	56.5 %
`mcf`	193	221	87.3 %	175	184	94.7 %
`parser`	305	520	58.7 %	228	338	67.3 %
`twolf`	2184	1306	167.2 %	1783	1165	153.0 %
`vortex`	193	463	41.7 %	161	-	-

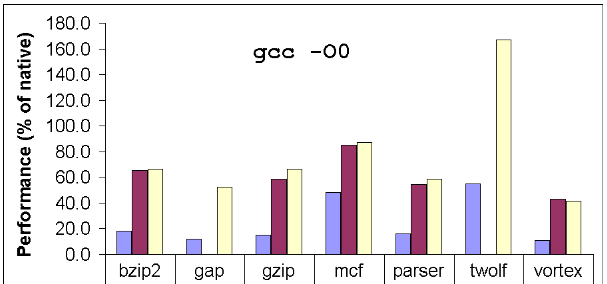
Table 8: Performance of the binary translator on SPEC CINT2000 benchmark applications. The percentage (% of native) columns represent performance relative to the x86 performance (the performance of a natively compiled application is 100%). '–' entries represent failed translations.

reported runtimes are computed after running the executables at least 3 times.) Our microbenchmarks use three well-known sorting algorithms, three different algorithms to solve the Towers of Hanoi, one benchmark that computes the Fibonacci sequence, a link-list traversal, a binary search on a sorted array, and an empty for-loop. All these programs are written in C. They are all highly compute-intensive and hence designed to stress-test the performance of binary translation.

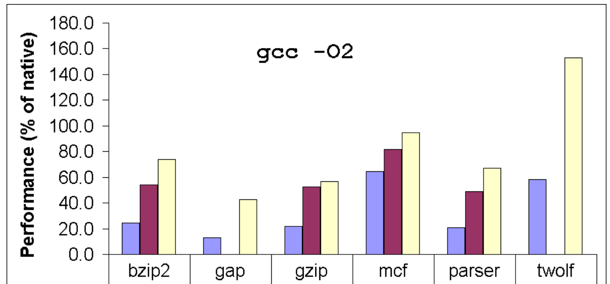
The translated executables perform roughly at 90% of the performance of a natively-compiled executable on average. Some benchmarks perform as low as 64% of native performance and many benchmarks outperform the natively compiled executable. The latter result is a bit surprising. For unoptimized executables, the binary translator often outperforms the natively compiled executable because the superoptimizer performs optimizations that are not seen in an unoptimized natively compiled executable. The bigger surprise occurs when the translated executable outperforms an already optimized executable (columns `-O2` and `-O2ofp`) indicating that even mature optimizing compilers today are not producing the best possible code. Our translator sometimes outperforms the native compiler for two reasons:

- The `gcc`-generated code for PowerPC is sometimes superior to the code generated for x86. This situation is in line with the conventional wisdom that it is easier to write a RISC optimizer than a CISC optimizer.

- Because we search the space of all possible translations while performing register mapping and instruction-selection, the code generated by our translator is often superior to that generated by `gcc`.



	bzip2	gap	gzip	mcf	parser	twolf	vortex
qemu	18.0	11.8	14.9	48.2	16.2	55.0	11.0
rosetta	65.3		58.7	84.8	54.4		43.1
peep	66.2	52.5	66.3	87.3	58.7	167.2	41.7



	bzip2	gap	gzip	mcf	parser	twolf
qemu	24.6	13.2	21.8	64.4	21.0	58.3
rosetta	54.0		52.5	81.5	49.0	
peep	73.7	42.5	56.5	94.7	67.3	153.0

Figure 4: Performance comparison of our translator (`peep`) with open source binary translator Qemu (`qemu`), and a commercial binary translator Apple Rosetta (`rosetta`). The bars represent performance relative to a natively compiled executable (higher is better). Missing bars are due to failed translations.

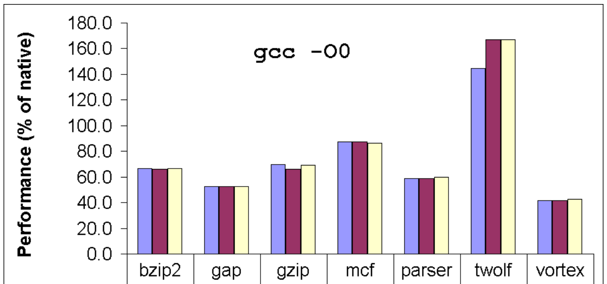
When compared with Apple Rosetta, our translator consistently performs better than Rosetta on all these microbenchmarks. On average, our translator is 170% faster than Apple Rosetta on these small programs.

A striking result is the performance of the `fibo` benchmark in the `-O2` column where the translated executable is almost three times faster than the natively-compiled and optimized executable. On closer inspection, we found that this is because `gcc`, on x86, uses one dedicated register to store the frame pointer by default. Since the binary translator makes no such reservation for the frame pointer, it effectively has one extra register. In the case of `fibo`, the extra register avoids a memory spill present in the natively compiled code causing the huge performance difference. Hence, for a more equal comparison, we also compare with the '`-fomit-frame-pointer`' `gcc` option on x86 (`-O2ofp` column).

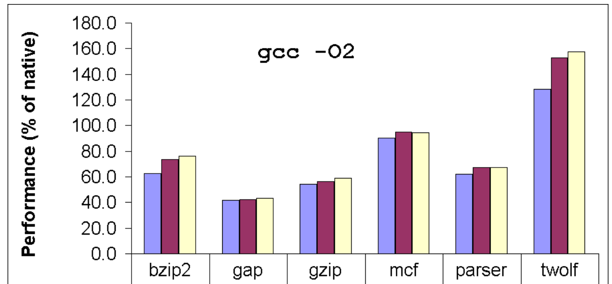
Table 8 gives the results for seven of the SPEC integer benchmarks. (The other benchmarks failed to run correctly due to the lack of complete support for all Linux system calls in our translator). Figure 4 compares the performance of our translator to Qemu and Rosetta. In our comparisons with Qemu, we used the same PowerPC and x86 executables as used for our own translator. For comparisons with Rosetta, we could not use the same executables, as Rosetta supports only Mac executables while our translator supports only Linux executables. Therefore, to compare, we recompiled the benchmarks on Mac to measure Rosetta performance. We used exactly the same compiler version (`gcc` 4.0.1) on the two platforms (Mac and Linux). We ran our Rosetta experiments on a Mac Mini Intel Core 2 Duo 1.83GHz processor, 32KB L1-Icache, 32KB L1-Dcache, 2MB L2-cache and 2GB of memory. These benchmarks spend very little time in the kernel, and hence we do not expect any bias in results due to differences in the two operating systems. The differences in the hardware could cause some bias in the performance comparisons of the two translators. While it is hard to predict the direction and magnitude of this bias, we expect it to be insignificant.

Our peephole translator fails on `vortex` when it is compiled using `-O2`. Similarly, Rosetta fails on `twolf` for both optimization options. These failures are most likely due to bugs in the translators. We could not obtain performance numbers for Rosetta on `gap` because we could not successfully build `gap` on Mac OS X. Our peephole translator achieves a performance of 42–164% of the natively compiled executable. Comparing with Qemu, our translator achieves 1.3–4x improvement in performance. When compared with Apple Rosetta, our translator performs 12% better (average) on the executables compiled with `-O2` and 3% better on the executables compiled with `-O0`. Our system performs as well or better than Rosetta on almost all our benchmarks, the only exceptions being `-O0` for `vortex` where the peephole translator produces code 1.4% slower than Rosetta, and `-O2` for `vortex`, which the peephole translator fails to translate. The median performance of the translator on these compute-intensive benchmarks is 67% of native code.

A very surprising result is the performance of the `twolf` benchmark where the performance of our translator is significantly better than the performance of natively compiled code. On further investigation, we found that `twolf`, when compiled with `-msoft-float`, spends a significant fraction of time ($\sim 50\%$) in the floating point emulation library (which is a part of `glibc`). The x86 floating point emulation library functions contain a redundant function call to determine the current instruction pointer, while the PowerPC floating point emulation code contains no such function call. This is the



	bzip2	gap	gzip	mcf	parser	twolf	vortex
No Reorder	66.8	52.4	69.8	87.3	58.8	144.8	41.8
Default	66.2	52.5	66.3	87.3	58.7	167.2	41.7
With profile	66.4	52.5	69.3	86.4	60.0	166.8	42.5



	bzip2	gap	gzip	mcf	parser	twolf
No Reorder	62.4	41.9	54.2	90.3	62.2	128.5
Default	73.7	42.5	56.5	94.7	67.3	153.0
With profile	76.1	43.1	59.1	94.3	67.5	157.6

Figure 5: Performance comparison of the default peephole translator with variants `No-Reorder` and `With-Profile`. The bars represent performance relative to a natively compiled executable (higher is better).

default `glibc` behavior and we have not found a way to change it. Coupled with the optimizations produced by our translator, this extra overhead in natively compiled x86 code leads to better overall performance for translated code. We do not see this effect in all our other benchmarks as they spend an insignificant fraction ($< 0.01\%$) of time in floating point emulation. The complete data on the running times of natively compiled and translated benchmarks is available in [4].

Next, we consider the performance of our translator on SPEC benchmarks by toggling some of the optimizations. The purpose of these experiments is to obtain insight into the performance impact of these optimizations. We consider two variants of our translator:

1. `No-Reorder`: Recall that, by default, we cluster data-dependent instructions inside a basic block for better translation (refer Section 5.9). In this variant, we turn off the re-ordering of instructions.

2. `With-Profile`: In this variant, we profile our executables in a separate offline run and record the profiling data. Then, we use this data to determine appropriate weights of predecessors and successors during register map selection (see Section 3.3).

Figure 5 shows the comparisons of the two variants relative to the default configuration. We make two key observations:

- The re-ordering of instructions inside a basic block has a significant performance impact on executables compiled with `-O2`. The PowerPC optimizing compiler separates data-dependent instructions to minimize data stalls. To produce efficient translated code, it helps to "de-optimize" the code by bringing data-dependent instructions back together. On average, the performance gain by re-ordering instructions inside a basic block is 6.9% for `-O2` executables. For `-O0` executables, the performance impact of re-ordering instructions is negligible, except `twolf` where a significant fraction of time is spent in precompiled optimized libraries.

- From our results, we think that profiling information can result in small but notable improvements in performance. In our experiments, the average improvement obtained by using profiling information is 1.4% for `-O2` executables and 0.56% for `-O0` executables. We believe our translator can exploit such runtime profiling information in a dynamic binary translation scenario.

Our superoptimizer uses a peephole size of at most 2 PowerPC instructions. The x86 instruction sequence in a peephole rule can be larger and is typically 1–3 instructions long. Each peephole rule is associated with a cost that captures the approximate cycle cost of the x86 instruction sequence.

We compute the peephole table offline only once for every source-destination architecture pair. The computation of the peephole table can take up to a week on a single processor. On the other hand, applying the peephole table to translate an executable is fast (see Section 6.1). For these experiments, the peephole table consisted of approximately 750 translation rules. Given more time and resources, it is straightforward to scale the number of peephole rules by running the superoptimizer on longer length sequences. More peephole rules are likely to give better performance results.

The size of the translated executable is roughly 5–6x larger than the source PowerPC executable. Of the total size of the translated executable, roughly 40% is occupied by the translated code, 20% by the code and data sections of the original executable, 25% by the indirect jump lookup table and the remaining 15% by other management code and data. For our benchmarks, the average size of the code sections in the original PowerPC executables is around 650 kilobytes, while the average size of the code sections in the translated executables is around 1400 kilobytes. Because both the original and translated executables operate on the same data and these benchmarks spend more than 99.99% of their time in less than 2% of the code, we expect their working set sizes to be roughly the same.

6.1 Translation Time

Translation time is a significant component of the run-time overhead for dynamic binary translators. As our prototype translator is static, we do not account for this overhead in the experiments in Section 6. In this section we analyze the time consumed by our translator and how it would fit in a dynamic setting.

Our static translator takes 2–6 minutes to translate an executable with around 100K instructions, which includes the time to disassemble a PowerPC executable, compute register liveness information for each function, perform the actual translation including computing the register map for each program point (see Section 3.3), build the indirect jump table and then write the translated executable back to disk. Of these various phases, computing the translation and register maps accounts for the vast majority of time.

A dynamic translator, on the other hand, typically translates instructions when, and only when, they are executed. Thus, no time is spent translating instructions that are never executed. Because most applications use only a small portion of their extensive underlying libraries, in practice dynamic translators only translate a small part of the program. Moreover, dynamic translators often trade translation time for code quality, spending more translation time and generating better code for hot code regions.

To understand the execution characteristics of a typical executable, we study our translator's performance on `bzip2` in detail. (Because all of our applications build on the same standard libraries, which form the overwhelming majority of the code, the behavior of the other applications is similar to `bzip2`.) Of the 100K instructions in `bzip2`, only around 8–10K instructions are ever executed in the benchmark runs. Of these, only around 2K instructions (hot regions) account for more than 99.99% of the execution time. Figure 6 shows the time spent in translating the hot regions of code using our translator.

We plot the translation time with varying prune sizes; because computing the translation and register maps dominate, the most effective way for our system to trade code quality for translation speed is by adjusting the prune size (recall Section 3.3). We also plot the performance of the translated executable at these prune sizes. At prune size 0, an arbitrary register map is chosen where

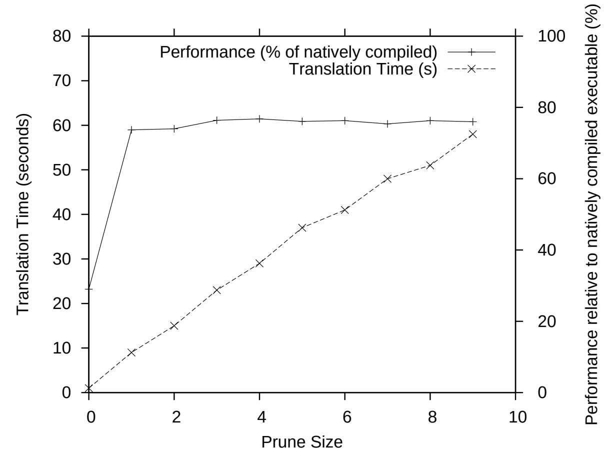


Figure 6: Translation time overhead with varying prune size for `bzip2`.

all PowerPC registers are mapped to memory. At this point, the translation time of the hot regions is very small (less than 0.1 seconds) at the cost of the execution time of the translated executable. At prune size 1 however, the translation time increases to 8 seconds and the performance already reaches 74% of native. At higher prune sizes, the translation overhead increases significantly with only a small improvement in runtime (for `bzip2`, the runtime improvement is 2%). This indicates that even at a small prune size (and hence a low translation time), we obtain good performance. While higher prune sizes do not significantly improve the performance of the translator on SPEC benchmarks, they make a significant difference to the performance of tight inner loops in some of our microbenchmarks.

Finally, we point out that while the translation cost reported in Figure 6 accounts for only the translation of hot code regions, we can use a fast and naive translation for the cold regions. In particular, we can use an arbitrary register map (prune size of 0) for the rarely executed instructions to produce fast translations of the remaining code; for `bzip2` it takes less than 1 second to translate the cold regions using this approach. Thus we estimate that a dynamic translator based on our techniques would require under 10 seconds in total to translate `bzip2`, or less than 4% of the 265 seconds of run-time reported in Table 8.

7 Related Work

Binary translation first became popular in the late 1980s as a technique to improve the performance of existing emulation tools. Some of the early commercial binary translators were those by Hewlett-Packard to migrate their customers from its HP 3000 line to the new Precision architecture (1987), by Digital Equipment Corporation to migrate users of VAX, MIPS, SPARC and x86 to Alpha (1992), and by Apple to run Motorola 68000 programs on their PowerMAC machines(1994).

By the mid-1990's more binary translators had appeared: IBM's DAISY [8] used hardware support to translate popular architectures to VLIW architectures, Digital's FX!32 ran x86/WinNT applications on Alpha/WinNT [7], Ardi's Executor [9] ran old Macintosh applications on PCs, Sun's Wabi [21] executed Microsoft Windows applications in UNIX environments and Embra [24], a machine simulator, simulated the processors, caches and other memory systems of uniprocessors and cache-coherent multiprocessors using binary translation. A common feature in all these tools is that they were all designed to solve a specific problem and were tightly coupled to the source and/or destination architectures and operating systems. For this reason, no meaningful performance comparisons exist among these tools.

More recently, the moral equivalent of binary translation is used extensively in Java just-in-time (JIT) compilers to translate Java bytecode to the host machine instructions. This approach is seen as an efficient solution to deal with the problem of portability. In fact, some recent architectures especially cater to Java applications as these applications are likely to be their first adopters [2].

An early attempt to build a general purpose binary translator was the UQBT framework [23] that described the design of a machine-adaptable dynamic binary translator. The design of the UQBT framework is shown in Figure 7. The translator works by first decoding the machine-specific binary instructions to a higher level RTL-like language (RTL stands for register transfer lists). The RTLs are optimized using a machine-independent optimizer, and finally machine code is generated for the destination architecture from the RTLs. Using this approach, UQBT had up to a 6x slowdown in their first implementation. A similar approach has been taken by a commercial tool being developed at Transitive Corporation [22]. Transitive first disassembles and decodes the source instructions to an intermediate language, performs optimizations on the intermediate code and finally assembles it back to the destination architecture. In their current offerings, Transitive supports SPARC-x86, PowerPC-x86, SPARC-x86/64-bit and SPARC-Itanium source-destination architecture pairs.

A potential weakness in the approach used by UQBT and Transitive is the reliance on a well-designed intermediate RTL language. A universal RTL language would need to capture the peculiarities of all different machine architectures. Moreover, the optimizer would need to understand these different language features and be able to exploit them. It is a daunting task to first design a good and universal intermediate language and then write an

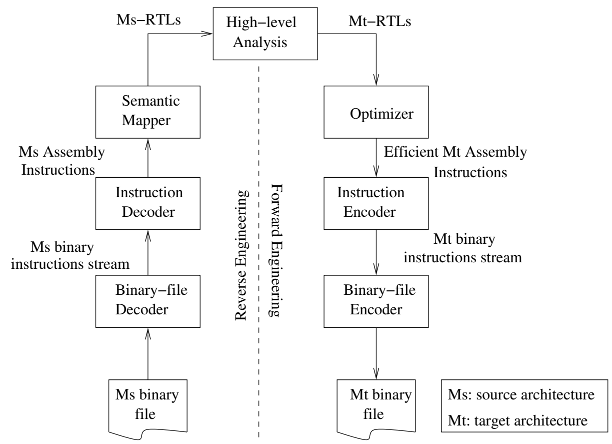


Figure 7: The framework used in UQBT binary translation. A similar approach is taken by Transitive Corporation.

optimizer for it, and we believe using a single intermediate language is hard to scale beyond a few architectures. Our comparisons with Apple Rosetta (Transitive's PowerPC-x86 binary translator) suggest that superoptimization is a viable alternative and likely to be easier to scale to many machine pairs.

In recent years, binary translation has been used in various other settings. Intel's IA-32 EL framework provides a software layer to allow running 32-bit x86 applications on IA-64 machines without any hardware support. Qemu [17] uses binary translation to emulate multiple source-destination architecture pairs. Qemu avoids dealing with the complexity of different instruction sets by encoding each instruction as a series of operations in C. This allows Qemu to support many source-destination pairs at the cost of performance (typically 5–10x slowdown). Transmeta Crusoe [12] uses on-chip hardware to translate x86 CISC instructions to RISC operations on-the-fly. This allows them to achieve comparable performance to Intel chips at lower power consumption. Dynamo and Dynamo-RIO [3, 6] use dynamic binary translation and optimization to provide security guarantees, perform runtime optimizations and extract program trace information. Strata [19] provides a software dynamic translation infrastructure to implement runtime monitoring and safety checking.

8 Conclusions and Future Work

We present a scheme to perform efficient binary translation using a superoptimizer that automatically learns translations from one architecture to another. We demonstrate through experiments that our superoptimization-based approach results in competitive performance while significantly reducing the complexity of building a high performance translator by hand.

We have found that this approach of first learning several peephole translations in an offline phase and then applying them to simultaneously perform register mapping and instruction selection produces an efficient code generator. In future, we wish to apply this technique to other applications of code generation, such as just-in-time compilation and machine virtualization.

9 Acknowledgments

We thank Suhabe Bugrara, Michael Dalton, Adam Oliner and Pramod Sharma for reviewing and giving valuable feedback on earlier drafts of the paper. We also thank Ian Pratt (our shepherd) and the anonymous reviewers.

References

[1] Apple Rosetta. `http://www.apple.com/rosetta/`.

[2] Azul Systems. `http://www.azulsystems.com/`.

[3] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices 35*, 5 (2000), 1–12.

[4] BANSAL, S. *Peephole Superoptimization*. PhD thesis, Stanford University, 2008.

[5] BANSAL, S., AND AIKEN, A. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (October 21-25, 2006), pp. 394–403.

[6] BRUENING, D. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[7] CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. FX!32: A profile-directed binary translator. *IEEE Micro 18*, 2 (Mar/Apr 1998), 56–64.

[8] EBCIOGLU, K., AND ALTMAN, E. R. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)* (1997), pp. 26–37.

[9] Executor by ARDI. `http://en.wikipedia.org/wiki/Executor_(software)`.

[10] GRANLUND, T., AND KENNER, R. Eliminating branches using a superoptimizer and the gnu C compiler. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (June 1992), vol. 27, pp. 341–352.

[11] HALFHILL, T. Transmeta breaks x86 low-power barrier. *Microprocessor Report* (February 2000).

[12] KLAIBER, A. The technology behind Crusoe processors. Tech. rep., Transmeta Corp., January 2000.

[13] LEROY, X., DOLIGEZ, D., GARRIGUE, J., AND VOUILLON, J. The Objective Caml system. Software and documentation available at `http://caml.inria.fr`.

[14] MASSALIN, H. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)* (1987), pp. 122–126.

[15] MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)* (2001), pp. 530–535.

[16] PowerVM Lx86 for x86 Linux applications. `http://www.ibm.com/developerworks/linux/lx86/index.html`.

[17] Qemu. `http://fabrice.bellard.free.fr/qemu/`.

[18] QuickTransit for Power-to-X86. `http://transitive.com/products/pow_x86.htm`.

[19] SCOTT, K., AND DAVIDSON, J. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation* (2001).

[20] Server consolidation and containment with VMware Virtual Infrastructure and Transitive. `http://www.transitive.com/pdf/VMwareTransitiveSolutionBrief.pdf`.

[21] SunSoft Wabi. `http://www.sun.com/sunsoft/Products/PC-Integration-products/`.

[22] Transitive Technologies. `http://www.transitive.com/`.

[23] UNG, D., AND CIFUENTES, C. Machine-adaptable dynamic binary translation. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo)* (2000), pp. 41–51.

[24] WITCHEL, E., AND ROSENBLUM, M. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems* (1996), pp. 68–79.

[25] YANG, B.-S., MOON, S.-M., PARK, S., LEE, J., LEE, S., PARK, J., CHUNG, Y. C., KIM, S., EBCIOGLU, K., AND ALTMAN, E. R. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)* (1999), pp. 128–138.

[26] ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. W., AND MALIK, S. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)* (November 2001), pp. 279–285.