Fast Machine-Adaptable Dynamic Binary Translation

Mark Probst
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien
`http://www.complang.tuwien.ac.at/schani/`

Abstract

Binary translation is the process of translating machine code from one instruction set architecture to another. It can be done statically or dynamically, i.e., on the fly. Most binary translation systems have been tailored to a specific source-target architecture combination. There are currently three published machine-adaptable binary translation systems, one of which is dynamic. Because it uses several intermediate representations to go from source to target code, its translation overhead is very high.

We present a machine-adaptable dynamic binary translation system currently in development which uses no intermediate representation at all, enabling it to translate very fast, while still producing good code.

1 Introduction

Binary translation is the process of translating machine code (binaries) from one instruction set architecture to another. Most binary translation systems currently in use are migration tools which aid the transition from old architectures to newer ones. We can expect a new wave of such migration tools when hardware vendors like Hewlett-Packard and Compaq switch from their current architectures to IA64.

Binary translation can be done statically by translating a whole binary to a new binary for the target platform, or dynamically, by translating code on the fly. For von Neumann architectures, where code and data reside in the same memory, static binary translation is never a complete solution, due to problems such as dynamic linking and self-modifying code. Dynamic binary translation overcomes these problems, while at the same time creating new ones. Most prominently, dynamic binary translators must be fast. Otherwise the translation overhead would be higher than the cost of running the translated code.

Until recently, static as well as dynamic binary translation system were always bound to a specific source-target architecture combination.

This paper presents `bintrans`, a machine-adaptable dynamic binary translator currently in development. The ultimate goal of the project is to produce a binary translation system capable of translating code between contemporary and future CISC, RISC and VLIW architectures, based primarily on machine descriptions.

The main contribution of this paper is a technique for generating a binary translator from two machine descriptions. The resulting translator does not use any intermediate representation, making it very fast, while still producing good code.

Section 2 gives a brief overview over the structure of `bintrans`. Section 3 outlines the language used for describing instruction sets. Section 4 presents the algorithm used in `bintrans` to find target instructions for source instructions at compiler-compile time. Section 5 presents and discusses performance results and statistics. Section 6 concludes and outlines some of our plans for the future.

1.1 Related work

The most prominent use of binary translation is in aiding migration from an old architecture to a new one. Well known published migration binary translators are Digital's VEST, mx [11] and FX!32 [4], which migrate VAX, MIPS and i386 code to the Alpha. Tandem used binary translation [2] to migrate their NonStop Systems series from a CISC instruction set to a RISC one.

Another area where dynamic binary translation has been applied frequently is profiling tools. A good example is Shade [7] which can do profiling at the level of single machine instructions with very high speeds. It generates code for source instructions at runtime and inserts profiling code before and/or after the instruction code. The paper on Shade [7] features a very good related work section. Out of Shade evolved Embra [15] which is a pure instruction set simulator using the techniques of Shade.

A retargetable static binary translator designed primarily for the purpose of profiling is described by Zhu and Gajski [16]. They use a quadruple-like intermediate representation. They report a performance loss of only a factor 1.1–2.5 when compared to native execution on the machine which is used for simulation, although they do not relate which source instruction set they translate from.

UQBT [6] is a resourceable and retargetable static binary translator. It takes as input descriptions of the source and target architectures at various levels of abstraction and compiles an executable for the source architecture to the target architecture by going through several intermediate representations. UQBT has a dynamic cousin, named UQDBT [13], which performs the translation dynamically. Since UQDBT uses the same translation logic as UQBT, its translation overhead is very high. It reportedly [13] uses 180,000 machine cycles to translate one byte of Pentium code to SPARC code, on average.

To achieve source and target independence, UQBT and UQDBT use three specification languages to specify binary formats [12], machine instruction syntax [10] and semantics [5].

UQDBT has been enhanced with the ability to produce edge profiles and based on that profiles select larger pieces of code (called "hot paths") which are then translated as a whole. The reported speedup [14] is about 15 percent.

2 System overview

The `bintrans` system consists of several parts. The main part is the translator generator, which takes two machine descriptions and produces a translator in C. The other platform independent parts are the dispatcher and the register allocator.

The system call interface and the memory management are mostly source and target operating system dependent, but depend little on the instruction set architectures.

The architecture dependent parts of our system (apart from the machine descriptions) are:

- The glue code for switching from the dispatcher to the generated code and back (250 lines of assembler for the Alpha).

- The instruction decoder for architectures with variable length instructions (130 lines of Lisp and 150 lines of C for the i386).

- Functions for generating register spilling instructions (50 line of C for the Alpha).

- Functions for generating code for loading large integers for target architectures which do not support large immediates (like all RISC architectures; 40 lines of C for the Alpha).

- The code for handling conditionals; see section 4.3.4 (140 lines of Lisp for the Alpha).

Unlike UQDBT, we use only one description language, which expresses syntax as well as semantics of the source and target machine instructions. The code for loading source binaries is hand-written. Up to now, we only support statically linked ELF binaries. We emulate the Linux system call interface.

2.1 Translation scheme

Our translation scheme is currently very simplistic: Whenever an untranslated instruction to be executed is encountered, the block beginning with that instruction up to and including the next branch instruction (direct or indirect) is compiled and then executed. When a block is compiled, its translation ends with instructions which transfer control back to the dispatcher, which looks up the target address. If it is found, execution continues with the target block. If the branch was a direct branch, the instructions which jumped to the dispatcher are changed to jump directly to the target block. If it is not found, the target block is translated and execution continues there. Block lookup is done through a hash table.

Each source instruction is translated to one or more target instructions. The translator generator processes the machine descriptions for the source and target architectures and determines which target instructions to use for the translation of a source instruction. This is done by matching the instruction descriptions, which is described in detail in Section 4.

For most source instructions, the matching process finds more than one target instruction sequence. Some of those target sequences are only applicable under some circumstances. The translator generator produces C code which determines for a given source instruction which of the instruction sequences is the most efficient applicable one and generates the machine code for that sequence.

2.2 Register allocation

Source registers can be mapped to target registers in one of two different ways. Direct mapping maps each source register to its own exclusive target register. This only works if there are more target registers than source registers. We use this mapping when translating from i386 to Alpha. Register caching keeps the source registers in memory but can cache them within a translation block in target registers which are allocated at instruction compile time. At the end of a block, modified registers have to be written back to their memory locations.

2.3 Memory management

A binary translator which emulates an architecture with the same address space size as the target architecture must be very careful about memory layout and memory management. We have not yet implemented such memory management, so we can currently only emulate 32 bit architectures on 64 bit architectures.

For this case, memory management is easy. The low 4 gigabytes of the address space are dedicated to the emulated program. The `bintrans` code, translated code and data structures are kept in the memory above.

Pages on the Alpha are 8 kilobytes long, while they are only 4 kilobytes on the PowerPC and the i386. If two emulated pages with different access rights fall on the same native page, that native page is granted the union of the access rights of the two pages. The consequence is that programs which rely on getting a segmentation violation signal for such pages will not work.

2.4 Supported architectures

We currently support i386 [8] and PowerPC [9] as source architectures and the Alpha [1] as the sole target architecture.

3 Machine descriptions

A machine description for an instruction set architecture specifies the semantics and (for RISC architectures) the syntax of that instruction set.

The most important contents of a machine description are

- Definitions of the register classes.
- Definitions of named subregisters.
- Definitions of instruction fields.
- Instruction descriptions.

Registers sharing the same properties are grouped into register classes. The PowerPC description, for example, defines three register classes, namely `gpr` (general purpose registers), `fpr` (floating point registers) and `spr` (special purpose registers). The definition of the `spr` class, for example, is

```
(define-register-class 'spr 'integer 32
  '(lr cr xer ctr fpscr))
```

Named subregisters are simply a way of accessing often-used subparts of registers in a convenient way. The i386 description, for example, defines the subregister `ax` as `(ax eax 0 15)`.

On RISC machines, which have fixed-sized instructions, instruction fields are specified with their location within the instruction word, so that decoders and encoders can be generated automatically. Since we do not yet provide this functionality for CISC architectures like the i386, instruction fields are only declared in those descriptions but must be decoded or encoded by hand-written code.

An instruction description contains several parts:

- Contents of instruction fields.
- Description of the instruction effect.
- Assembler syntax of the instruction.

The syntax of instruction effects is very similar to Lisp syntax. Expressions are written in parenthesized prefix (polish) notation. The expression $1 + 2 * 3$, for example, would be written as `(+ 1 (* 2 3))`. The language has several built-in operators such as `+`, `-` and `*` for describing two's complement arithmetic, `logand`, `logxor` and `shiftr` for bit manipulation, and so on.

In order to describe parts of the state of the machine, the language provides primitives such as `reg`, `subreg` and `mem`. `reg`, for example, takes a number and a register class name and describes the register with that number within that class.

The operator which introduces effects is `set`. Its arguments are an lvalue and a right-hand side expression. Setting the general purpose register with the number specified by the instruction field `ra` to zero would be written as `(set (reg ra gpr) 0)`. We call a `set` expression an "effect".

All expressions and all subexpressions within them have associated widths. These widths must match in certain ways. For example, the widths of the lvalue and of the right-hand side of a `set` expression must be the same. Similarly, many operands, like `+` and `logand` demand that their operands and their results have the same widths. For most subexpressions, the widths need not be specified by hand, because they can be inferred from the context. Where necessary, widths can be specified with the `width` operator. The widths of expressions can be expanded using the operators `zex` and `sex`, which describe zero-extension and sign-extension, respectively.

In addition, a `let` operator is provided for binding temporary values. For example, consider the PowerPC instruction `lwzu`, which loads a word from a memory address specified by a base register and an immediate offset into a target register and then sets the base register to the address of the loaded word. We describe these effects as

```
(let ((ea (width 32 (+ (reg ra gpr)
                       (sex d)))))
  (set (reg rd gpr) (mem ea))
  (set (reg ra gpr) ea))
```

Consider the case of `ra` and `rd` specifying the same register. Such an instruction is essentially equivalent to adding a signed immediate to a register, with the difference that it could generate a memory access violation. Without the `let` construct above, the described effect would instead be that of first setting the register to the value found at the memory address given by the sum of the register's old value and the immediate. After that, the immediate would be added to the new value of the register.

Given such a machine description, it is straightforward to generate an interpreter for the instruction set, a disassembler for debugging purposes and functions for emitting machine code given the name of the instruction and the source operands.

4 Matching instruction effects

For each instruction in the source machine description the translator generator matches the effects of the instruction against target instruction effects in order to find at least one sequence of target instructions with the same semantics as the source instruction.

First, conditionals whose conditions depend only on values which will be known at instruction translation time are eliminated. This is done by splitting them up into alternatives. Which of these alternatives applies for a given source instruction is determined by the generated compiler at instruction compile time (by examining the values of the conditionals).

The resulting alternatives are changed to semantically equivalent forms which use 64 bit registers instead of 32 bit registers. This step would be unnecessary if the source and target instructions had the same register widths.

The effects are then matched using an algorithm which can produce one or more target sequences, some of which may only be applicable under certain circumstances.

The target sequences are then ranked and C code for an instruction compiler is generated.

4.1 Decomposing machine instructions

Source and target instructions are split up by assuming all possible truth value assignments to conditionals whose conditions depend only on values known at instruction compile time. Those values are constants, the contents of instructions fields and the value of the program counter at the time of the execution of the instruction in question. For the remainder of this paper, those values will be called "translation-constant". For example, the effect of the `bis` (bitwise or) instruction with two register source operands on the Alpha is

```
(if (= rc 31)
  (nop)
  (set (reg rc gpr)
       (logor (if (= ra 31)
                0 (reg ra gpr))
              (if (= rb 31)
                0 (reg rb gpr)))))
```

The three conditions in this expression are `(= rc 31)`, `(= ra 31)` and `(= rb 31)`. Since `ra`, `rb`, and `rc` are instruction fields, they are translation-constants, so all three conditions can be resolved statically. By assuming all possible truth value assignments, the translator generator splits this instruction into five:

```
(nop)
(set (reg rc gpr) (logor 0 0))
(set (reg rc gpr) (logor (reg ra gpr) 0))
(set (reg rc gpr) (logor 0 (reg rb gpr)))
(set (reg rc gpr) (logor (reg ra gpr)
                         (reg rb gpr)))
```

Additionally, all other translation-constant subexpressions in the target effects are replaced by placeholders, which we call "inputs". For example, consider this expression (which is one of the decompositions of the Alpha `zap` (zero bytes) instruction with an immediate operand):

```
(set (reg rc gpr)
     (logand (reg ra gpr)
             (bitneg (maskmask 8 imm))))
```

The subexpression (bitneg (maskmask 8 imm))[1] is translation-constant, so it is replaced by an input, which is matched as a whole at compiler-compile time.

4.2 Compensating for different register widths

Before matching source instruction effects against target instruction effects, it may be necessary to modify the source effects. This is the case when the register widths of the architectures differ. Consider, for example, the PowerPC `add` instruction, which adds two 32 bit registers and produces a 32 bit value:

```
(set (reg rd gpr)
     (+ (reg ra gpr) (reg rb gpr)))
```

The effect of this instruction is virtually identical to the effect of the Alpha `addq` instruction, which, however, adds 64 bit registers, producing a 64 bit value.

There are two straightforward ways of storing a 32 bit value in a 64 bit register: zero-extended and sign-extended. Which way is the more efficient one is dependent on the target instruction set. On the Alpha, for example, sign-extending 32 bit values is the canonical way of operating on these values. Our system does not determine automatically which way to use. Instead, this must be specified for each target architecture.

The transformation of the `add` effect to 64 bits with sign-extension is this:

```
(set (reg rd gpr)
     (sex (+ (subreg 0 31 ra gpr)
             (subreg 0 31 rb gpr))))
```

This effect can be directly matched with the `addl` Alpha instruction.

4.3 A simple matching algorithm

The decomposed and possibly transformed (for register width compensation) source instruction effects are split up into individual `set` statements. Each of these atomic effects is matched separately against the effects of all target instructions. Those target instructions which match successfully are considered for generation. Section 4.4 describes how the matches are further processed.

If a target instruction has more than one atomic effect (consider i386 instructions which, apart from modifying a register or memory location, change lots of condition bits), only the "primary" effect is considered. An effect in a target instruction is primary if it modifies a memory location or a register which can potentially hold the contents of an emulated register. Target instructions with more than one primary effect (like the i386 `push` or the PowerPC `stwu` instructions) are not matched against.

Before actually matching an atomic effect, it is simplified. Simplification is just constant-folding. This is necessary because constants can be introduced in the decomposition step. For example, the atomic effect

```
(set (reg rc gpr) (logor 0 0))
```

would be simplified to

```
(set (reg rc gpr) 0)
```

Matching is then done for the lvalues as well as for the RHSs of the source and target `set` statements. The matching algorithm is recursive. Given the predicates `const?` and `input?`, which return whether an expression is translation-constant or an input, respectively, and the accessors `width`, `op` and `args`, which return the width, the operator and the arguments of an expression, respectively, a simple algorithm to match a source with a target expression can be written as:

```
(defun match (s t)
  (and (= (width s) (width t))
       (cond ((member (op t) '(reg subreg))
              (match-with-reg s t))
             ((and (const? s) (input? t))
              (match-with-input s t))
             ((eq (op s) (op t))
              (every #'match (args s)
                             (args t))))))
```

In words: the source expression matches the target only if their widths are the same. If they do, three cases are distinguished between.

4.3.1 Matching against a register

If an expression is matched against a register or a subregister in the target effect, the value of the expression must be put into that register or subregister before the target instruction can be executed. The function `match-with-reg` takes care of that case. If the

[1] `maskmask` stretches a bit-mask. (`maskmask` n m) evaluates to m with all zero bits replaced by n zero bits and all one bits replaced by n one bits. (`maskmask 4 5`), for example, evaluates to hexadecimal `f0f`.

source expression is itself a register in the same class as required in the target expression, no additional instruction is needed. Otherwise, a new temporary `set` statement is created and the whole matching process is recursively applied to it.

4.3.2 Matching constants with inputs

The case of a translation-constant matched against an input is a bit more complex. The value of the translation-constant can be easily computed from the instruction fields at instruction translation time. The input is usually a compound expression consisting of some constants and zero, one or more instruction fields.

In the general case, matching the constant with the input has to be delayed to instruction translation time, which is what we do. If the input does not contain an instruction field, and is thus a constant, matching is trivial: the values just have to be compared. If they are equal, the match is successful, otherwise it fails.

In case the input contains one instruction field, the matching at instruction translation time becomes similar to equation-solving. Consider, for example, the PowerPC instruction `mulli` (multiply with immediate) with the following effect (transformed to 64 bits):

```
(set (reg rd gpr)
     (sex (* (subreg 0 31 ra gpr)
             (sex simm))))
```

The `simm` field is 16 bits wide. The `mull` instruction with an immediate operand on the Alpha has this effect:

```
(set (reg rc gpr)
     (sex (* (subreg 0 31 ra gpr)
             (zex imm))))
```

The `imm` field is 8 bits wide. When the `mulli` effect is matched against the `mull` effect, the match depends on the outcome of matching the translation-constant `(sex simm)` against the input `(zex imm)`. Depending on the value of the `simm` field in the source instruction, this match can be successful or it can fail. The instruction matcher generates C code, which, at instruction translation time, calculates the value of `(sex simm)` and checks whether this value can be computed by zero-extending an 8 bit quantity. If it can, the match succeeds, otherwise it fails.

We do not yet handle matching against an input with more than one instruction field.

4.3.3 Matching ordinary expressions

If neither of these two cases apply, the operands of both expressions must be the same, otherwise the match fails. If the operands are the same, the arguments of the source and the target expressions are matched in pairs. The match is successful only if all source-target pairs match.

4.3.4 Special cases

The `let` operator, which introduces a temporary value binding is handled in the simplest way: each binding introduced in the `let` form is allocated a temporary register. The effects within the body of the `let` are matched against target effects with occurrences of the introduced bindings treated as registers.

Conditionals with conditions which are not translation-constant, i.e., which cannot be decomposed at compiler-compile time cannot be matched with our generator and have to be handled in architecture dependent code. The code to do this for the Alpha is less than 140 lines of Lisp.

Some very high-level constructs available on some machines may not be available on others. Code for emulating their effects must be hand-coded. The only example of this in our system so far is integer division, which is not implemented in hardware on the Alpha. We simply call C functions performing the division from the translated code.

4.4 Ranking the matches

The question arises what to do if a match fails at instruction translation time. Obviously, there must always be a fall-back, otherwise the source instruction could no be translated. In matching a source effect against all possible target effects, the compiler-compiler often finds several alternative ways of translating an instruction. In such cases, it sorts all alternatives by preferring the shorter ones (requiring less target instructions) to the longer ones and at the same length those which are always successful to those which can fail. For the resulting list it generates C code for the delayed matching and stops after the first alternative which can not fail.

In the above case of generating code for the `mulli` instruction, the generated compiler would first try whether the second operand fits into the `imm` field and if so, it would emit a `mull` instruction with that immediate operand. Otherwise, it would try to put the second operand into a register, which can always be done with the `lda` instruction. Then it would use the `mull` instruction to multiply the two register contents.

4.5 Improvements to the algorithm

The simple algorithm presented above has two serious drawbacks. Firstly, it does not find any matches for some source effects. Secondly, it sometimes does not realize that matching a translation-constant with an input will always succeed, so it includes one or more "fall-back" solutions in the translator code which are actually never used.

The first problem can be solved by introducing transformations. A transformation is a rule by which an expression can be rewritten as a semantically equivalent form, in the hope that this other form can be successfully matched against a target effect. As a simple example, consider the transformation

```
(define-transformation
  (neg (match x))
  (- 0 x))
```

It says that a value can be negated by subtracting it from 0, which makes it possible, for example, to match a 32 bit negation with the Alpha instruction `subl`. Most transformations in our system are concerned with subregisters. Specifically, there are transformations which specify how to extract a subregister from a register and how to change a subregister within a register without having specific instructions to do that. In other words, transformations are just a way to express semantic equivalences of expressions which cannot be derived by the translator (for example the fact that a rotation can be implemented by performing shifts and bitwise or).

When an expression is to be matched, our system tries to match not only that expression but also all expressions resulting from applying all applicable transformations to that expression. To avoid endless loops, each transformation can be applied at most once to each subexpression.

The second problem arises when a source and a target instruction do similar things in a slightly different way. Consider, for example, the PowerPC `srawi` (arithmetic shift right with immediate) instruction. The primary effect of this instruction is

```
(set (reg ra gpr) (ashiftr (reg rs gpr)
                           (zex sh)))
```

The `sh` field is 5 bits wide. The corresponding Alpha instruction is the immediate form of `sra`, whose effect is

```
(set (reg rc gpr)
     (ashiftr (reg ra gpr)
              (logand (zex imm) 63)))
```

Obviously, it is always possible to find a value for the 8 bit `imm` field such that the value of its low 6 bits is equal to the value of the `sh` field in the source instruction. To make the algorithm for matching translation-constants with inputs notice this, we use a simple recursive function which, given an expression, calculates which bits of this expression can be set and which cannot. For example, the value of the expression `(zex sh)` can only have its low 5 bits set. When matching against `(logand (zex imm) 63)`, this fact is taken into account, insofar as if only the low 5 bits can be set, the operation of bitwise conjunction of the value with 63 does not alter the value, hence the match always succeeds.

Some architectures, like the i386, set lots of condition bits in many instructions, most of which are not used later on. Some of these condition bits, like the parity flag on the i386, are expensive to compute on most target machines.

To achieve good performance when emulating such architectures, `bintrans` offers the possibility of doing simple liveness analysis for single bits of specified registers. From a machine description we generate an instruction decoder capable of computing which bits of the specified registers are used and which are overwritten by an instruction. Liveness analysis is then done with a single linear pass through the translation block. In order to save computing all bits at the end of a translation block, we also check the next few instructions of each (there are at most two) following block, if the block in question ends with a direct branch.

Additionally, `bintrans` can keep single bits (which must be specified by hand) of source registers in target registers of their own. If the target architecture offers many more registers than the source architecture, this makes setting as well as checking for these bits more efficient. We do this for the i386→Alpha translator for the CF, ZF, OF and SF bits of the EFLAGS register.

4.6 An example

Figure 1 contains actual generated C code (slightly simplified and commented) for translating the PowerPC `stwx` (store word indexed) instruction to Alpha machine code. The effect of `stwx` is

```
(set (mem (+ (if (= ra (width 5 0))
                 0 (reg ra gpr))
              (reg rb gpr)))
     (reg rs gpr))
```

This is special insofar as the register `r0` cannot be specified as the first operand to the effective address calculation. Instead, specifying `r0` means that the first operand is zero. This is reflected in the generated code, which first checks whether the `ra` field is zero. If it is, a single `stl` instruction with an immediate displacement of zero is generated. Otherwise, an `addl` instruction which calculates the effective address is generated before generating `stl`.

In the generator code the variable `insn` contains the PowerPC instruction word. It is often bit-manipulated to access instruction fields. The variables beginning with `field_` are to contain the values of Alpha instruction fields. The function `ref_gpr_reg_for_reading` makes sure that the value in the register whose index it is passed is contained in a native general purpose register and returns that native register. The number 5 is added to the numbers of the PowerPC general purpose registers because the index of the first PowerPC register happens to be 5 (because the five special purpose registers were declared before the general purpose registers in the machine description). The function `ref_gpr_reg_for_writing` has two purposes. When called with `-1` it simply allocates and returns a temporary native register. For other values, it announces that a PowerPC register is to be written to and a native register is needed to hold its value. That native register is returned. The function `unref_gpr_reg` indicates to the register allocator that the native register which it is passed is not needed anymore.

5 Preliminary results

Table 1 presents some performance results. We have run the go and compress benchmarks of the SPECINT95 suite on the PowerPC→Alpha and the i386→Alpha translators.

The go benchmark results given in the table are for the 5stone21 input file only. The results are similar for the other two input files.

The rows “Emulated run time” and “Native run time” give the total run times in seconds of the program in question when run in the binary translator (including all overhead imposed by the translator) and natively on the Alpha, respectively. The run-times are wall clock time averages of 5 runs on an unloaded Alpha 21164PC machine with 533MHz. The PowerPC, i386 and Alpha binaries of the benchmarks have been compiled with GCC with the switch `-O3`.

The row “Translation time” gives the time in milliseconds spent in the binary translator. It does not include program loading time, block lookup time for indirect branches and system call processing time.

The next two rows give the number of blocks and the number of source instructions which have been translated for the benchmark in question. Since the benchmarks are deterministic, these numbers are the same for all runs.

The last two rows give the total number of Alpha instructions generated and the number of instructions which load or store cached registers. As with the two rows above, these numbers are the same for all runs.

6 Conclusion and further work

We have presented a dynamic binary translation system which is resourceable and retargetable. Our system does not use an intermediate representation to generate code. Instead, the code generator is generated from two machine descriptions and instruction effects are matched at that time. At runtime our system uses no dynamic storage and almost no static storage for the purpose of code generation. Code generation is very fast.

Apart from the machine descriptions, very little architecture dependent code is needed in `bintrans`. About 140 lines of Lisp, 90 lines of C and 250 lines of Assembler are Alpha specific, about 130 lines of Lisp and 150 lines of C are i386 specific and no architecture specific code is needed for the PowerPC (which will change when we use the PowerPC as a target architecture).

The `bintrans` system is not without problems. Especially when translating PowerPC code to the Alpha, the generated code is often mediocre at best. The main reason is the condition code system on the PowerPC. Compare instructions on the PowerPC set 4 condition bits at once and those bits are in most cases live at the end of their translation blocks, so we cannot simply eliminate their calculations as we most often can for the i386. Furthermore, our short translation blocks lead to frequent register reloads, which are not necessary for the i386, because we use a fixed mapping there.

We plan to remedy both problems by making translation blocks larger with a scheme similar to Dynamo's [3]. This also opens the possibility for speed gains from liveness analysis and constant folding, for example.

6.1 System complexity

The translator generator is about 3500 lines of Lisp. The machine descriptions for the Alpha, the PowerPC

```
if (((insn >> 16) & 0x1F) == 0) { /* ra == 0 */
    word_5 tmp_114, field_ra, tmp_118, field_rb;
    word_64 tmp_120 = 0;
    word_16 field_memory_disp = 0;
    /* allocate native registers for rb and rs */
    tmp_118 = ref_gpr_reg_for_reading(5 + ((insn >> 11) & 0x1F));
    tmp_114 = ref_gpr_reg_for_reading(5 + ((insn >> 21) & 0x1F));
    field_ra = tmp_114; field_rb = tmp_118;
    emit(COMPOSE_STL(field_ra, field_memory_disp, field_rb));
    unref_gpr_reg(tmp_114);
    unref_gpr_reg(tmp_118);
} else { /* ra != 0 */
    word_5 tmp_114, field_ra, tmp_118, field_rb;
    word_64 tmp_120 = 0;
    word_16 field_memory_disp = 0;
    {   /* generate code for performing the addition of ra and rb */
        word_5 tmp_1154, field_ra, tmp_1156, field_rb, tmp_1160, field_rc;
        /* allocate native registers for ra and rb */
        tmp_1154 = ref_gpr_reg_for_reading(5 + ((insn >> 16) & 0x1F));
        tmp_1156 = ref_gpr_reg_for_reading(5 + ((insn >> 11) & 0x1F));
        /* allocate a temporary register for the result */
        tmp_1160 = tmp_118 = ref_gpr_reg_for_writing(-1);
        field_ra = tmp_1154; field_rb = tmp_1156; field_rc = tmp_1160;
        emit(COMPOSE_ADDL(field_ra, field_rb, field_rc));
        unref_gpr_reg(tmp_1156);
        unref_gpr_reg(tmp_1154);
    }
    /* allocate a native register for rs */
    tmp_114 = ref_gpr_reg_for_reading(5 + ((insn >> 21) & 0x1F));
    field_ra = tmp_114; field_rb = tmp_118;
    emit(COMPOSE_STL(field_ra, field_memory_disp, field_rb));
    unref_gpr_reg(tmp_114);
    unref_gpr_reg(tmp_118);
}
```

Figure 1: Slightly simplified and commented generated code for translating `stwx`

	go		compress	
	PowerPC	i386	PowerPC	i386
Native run time	187 s		318 s	
Emulated run time	998 s	653 s	946 s	823 s
Translation time	1533 ms	4754 ms	62 ms	175 ms
# blocks translated	9848	11044	856	920
# instructions translated	72486	62080	5391	4666
# instructions generated	481199	442716	35308	34565
# reg load/stores	114492	0	8240	0

Table 1: PowerPC→Alpha and i386→Alpha performance

and the i386 are about 1000 lines each. Of these, however, the Alpha description is the only one which is fairly complete. The PowerPC and i386 descriptions still lack a lot of instructions.

The hand-written parts of the translator are about 4700 lines of C, of which about 1400 lines are concerned with handling system calls and 800 lines with memory management.

6.2 Obtaining the source code

The source code for the `bintrans` system can be downloaded from `http://www.complang.tuwien.ac.at/schani/bintrans/`.

Acknowledgements

I thank Andreas Krall, Anton Ertl and the anonymous referees for their helpful comments on this paper.

References

[1] Alpha Architecture Committee. *Alpha Architecture Reference Manual*. Digital Press, 1998.

[2] Kristy Andrews and Duane Sand. Migrating a CISC computer family onto RISC via object code translation. *ACM SIGPLAN Notices*, 27(9):213–222, September 1992.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[4] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March/April 1998. Presented at Hot Chips IX, Stanford University, Stanford, California, August 24–26, 1997.

[5] C. Cifuentes and S. Sendall. Specifying the semantics of machine instructions. In *6th International Workshop on Program Comprehension - IWPC'98, Ischia, Italy, June 24-26 1998*, pages 126–133. IEEE Computer Society, 1998.

[6] Cristina Cifuentes, Mike Van Emmerik, and Norman Ramsey. The design of a resourceable and retargetable binary translator. In *Proceedings: Sixth Working Conference on Reverse Engineering*, pages 280–291. IEEE Computer Society Press, 1999.

[7] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[8] Intel Corporation. Intel architecture software developer's manual—volume 1: Basic architecture, 2000.

[9] Motorola Inc. *PowerPC Microprocessor Family: The Programming Environments For 32-Bit Microprocessors*. Motorola Inc., 1997.

[10] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[11] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[12] D. Ung and C. Cifuentes. SRL – A simple retargetable loader. In *Proceedings of the Australia Software Engineering Conference*, pages 60–69. IEEE CS Press, September 1997.

[13] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM Sigplan Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO-00)*, volume 35.7 of *ACM SIGPLAN NOTICES*, pages 41–51, N.Y., January 18–18 2000. ACM Press.

[14] David Ung and Cristina Cifuentes. Optimising hot paths in a dynamic binary translator. In *Second Workshop on Binary Translation, Oct 19, 2000, Philadelphia, Pennsylvania*, 2000.

[15] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer SYstems*, volume 24,1 of *ACM SIGMETRICS Performance Evaluation Review*, pages 68–79, New York, May 23–26 1996. ACM Press.

[16] Jianwen Zhu and Daniel D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Proceedings of the Design Automation and Test Conference In Europe*, 1999.