1995

Relocating Machine Instructions by Currying

Norman Ramsey

Report Number:
95-068

RELOCATING MACHINE INSTRUCTIONS BY CURRYING

Norman Ramsey

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR-95-068
October 1995

Relocating Machine Instructions by Currying

Norman Ramsey, Purdue University

October 23, 1995

Abstract

Relocation adjusts machine instructions to account for changes in the locations either of the instructions themselves or of external symbols to which they refer. Standard linkers implement a finite set of relocation transformations, suitable for a single architecture. These transformations are enumerated, named, and engraved in a machine-dependent object-file format, and linkers must recognize them by name. These names and their associated transformations are an unnecessary source of machine-dependence.

The New Jersey Machine-Code Toolkit is an application generator. It helps programmers create applications that manipulate machine code, including linkers. Guided by a short instruction-set specification, the toolkit generates the bit-manipulating code. Instructions are described by *constructors*, which denote functions mapping lists of operands to instructions' binary representations. Any operand can be designated as "relocatable," meaning that the operand's value need not be known at the time the instruction is encoded. For instructions with relocatable operands, the toolkit computes relocating transformations. Tool writers can use the toolkit to create machine-independent software that relocates machine instructions. `mld`, a retargetable linker built with the toolkit, needs only 20 lines of C code for relocation, and that code is machine-independent.

The toolkit discovers relocating transformations by currying encoding functions. An attempt to encode a relocatable operand results in the creation of a closure. The closure can be applied when the values of the relocatable operands become known. Currying provides a general, machine-independent method of relocation.

Currying rewrites a single λ-term into two nested λ-terms. The standard implementation has the first λ allocate a closure and store therein its operands and a pointer to the second λ. Using this simple strategy in the toolkit means that, when it builds an application, the toolkit generates code for many different inner λ-terms—one for each instruction that uses a relocatable address. Hoisting some of the computation out of the second λ into the first makes many of the second λs identical—a handful are enough for a whole instruction set. This optimization reduces the size of machine-dependent assembly and linking code by 15–20% for the MIPS, SPARC, and PowerPC, and by about 30% for the Pentium.

1 Introduction

Compiling whole programs is slow; compiling units separately and linking the compiled units into a program speeds up the edit-compile-go cycle. For separate compilation, a compiler must be able to emit instructions and data without knowing the exact locations either of the instructions and data the compiler itself emits, or of the instructions and data emitted by other compilations. Using assembly language makes this task easy, because in assembly language all locations are represented symbolically. Symbolic, assembly-like units can be linked to form programs (Fraser and Hanson 1982; Jones 1983), but the linker or loader must translate all units from symbolic form into the binary representation required by the target hardware. It is believed to be more efficient to translate each unit separately into a binary form called *relocatable object code.*

Object code must contain more than just instructions and data. To support delayed binding of locations, it must also represent

- The symbols defined in the object file and the locations to which they are bound.
- The symbols imported from other units, i.e., external symbols.
- The transformations that must be applied to the instructions and data to account for its eventual placement at an absolute address and also for the placements of the external symbols on which it depends.

Applying these transformations is called *relocation.*

Current object-code formats force tool writers to handle relocation in a machine-dependent way. For a particular architecture, a human being examines the instruction set and determines which operands can be relocatable addresses and what relocating transformations are needed. Each transformation is named, and linkers and other tools must recognize transformations by name. The names are informal and machine-dependent, so retargetable tools that manipulate object code must recognize each set of names on each machine.

This paper makes several contributions. It presents a machine-independent, automatic method of discovering relocating transformations. It presents an optimization that makes the cost of the automatic method comparable to the cost of hand-implemented methods and makes the discovered transformations equivalent to the transformations used in standard object-file formats. Finally, the paper gives a machine-independent representation of the transformations.

This new technique for relocating machine instructions is an enabling technology for building machine-independent tools for static, incremental, and dynamic linking. It will also simplify the construction of retargetable tools that implement object-code transformations. Object-code transformation, which is growing in importance, is used for profiling and tracing (Ball and Larus 1992), testing (Hastings and Joyce 1992), enforcing protection (Wahbe *et al.* 1993), optimization (Srivastava and Wall 1993), and binary translation (Sites *et al.* 1993). There are even some frameworks for creating applications that transform object code (Johnson 1990; Larus and Schnarr 1995; Srivastava and Eustace 1994).

The techniques presented here build on the New Jersey Machine-Code Toolkit (Ramsey and Fernandez 1995), which reads a compact machine description and generates functions that encode instructions. The machine description relates two representations of instructions: a symbolic representation akin to assembly language and the binary representation used by the hardware. The symbolic representation of an instruction includes its name, a list of operands, and a suggested assembly-language syntax. The author of the machine description indicates which operands are relocatable addresses. Currying the encoding function with respect to those operands results in a relocating transformation.

Currying rewrites the encoding function into two nested λ-terms. In the standard implementation, the outer λ allocates a closure and stores therein its operands and a pointer to the inner λ, which uses the contents of the closure to encode (relocate) the instruction. The inner λs are the relocating transformations discovered by the toolkit, and the closures take the place of "relocation entries" in traditional object files.

Using the standard implementation of currying, the toolkit generates code for many different inner λ-terms—one for each instruction that uses a relocatable address. Hoisting some of the computation out of the inner λ into the outer makes many of the inner λs identical—a handful are enough for a whole instruction set. This optimization is closely related to fully lazy lambda-lifting (Peyton Jones 1987). It reduces the size of machine-dependent assembly and linking code by 15–20% for the MIPS, SPARC, and PowerPC, and by about 30% for the Pentium. It also makes the relocating transformations discovered by the toolkit equivalent to those that are now implemented by hand. To support machine-independent use of these transformations, the toolkit associates each one with a string that can be interpreted to have the effect of applying the transformation. These strings can be used in an object file as meaningful, formal, machine-independent names.

2 Describing instruction encodings

The New Jersey Machine-Code Toolkit describes the binary representation of an instruction as a sequence of *tokens*. On a RISC machine, each instruction is a single 32-bit token. On a machine like the Pentium, formats vary; for example, the instruction `add 612[DX], 33` has an 8-bit opcode token, followed by another 8-bit token that has both opcode and address-mode bits, followed by an the 32-bit displacement 612 and the 8-bit immediate operand 33.

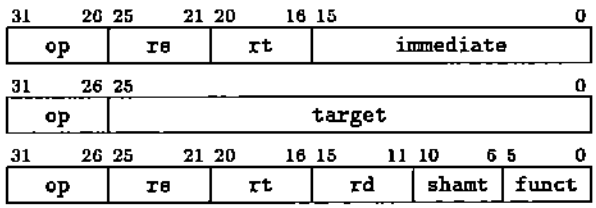
Each token in an instruction is partitioned into *fields*; a field is a contiguous range of bits within a token. Fields contain opcodes, operands, modes, or other information. Opcodes and operands can be distributed among multiple fields. On RISC machines, different instruction formats are represented by different partitions of the instruction token.

Patterns constrain the values of fields; they may constrain fields in a single token or in a sequence of tokens. They can be used to describe binary representations of opcodes, of whole instructions, and of groups of instructions.

Constructors connect the symbolic and binary representations of instructions. At a symbolic level, an instruction is an opcode (the constructor) applied to a list of operands. The result of the application is a sequence of tokens, which is described by a pattern. For each constructor, the toolkit derives an encoding function that emits the constructor's binary representation. We get relocating transformations by currying the encoding functions. The encoding functions generated from a machine description form part of an application-program interface (API) to an assembler for that machine. The toolkit includes a library of other functions that complete the API.

Tokens and fields

A machine description includes the names, sizes, and positions of the fields used to form tokens. The information can be found in architecture manuals. For example, the MIPS manual (Kane 1988, p A-3) gives this informal field specification:



This informal specification can be formalized in a machine description as follows:

```
fields of instruction (32)
 op 26:31 rs 21:25 rt 16:20 rd 11:15 shamt 6:10 funct 0:5
 target 0:25 immed 0:15 offset 0:15 base 21:25 cond 16:20
 breakcode 6:25 ft 16:20 fs 11:15 fd 6:10 format 21:24
```

This declaration defines not only the fields used in the formats pictured above but also `offset`, `cond`, and other synonyms that appear in the MIPS manual.

Patterns

Patterns constrain both the division of streams into tokens and the values of the fields in those tokens. They are composed from *constraints* on fields. A constraint fixes the range of values a field may have. The typical range has a single element, e.g., `op = 1`. Patterns may be composed by conjunction (`&`), concatenation (`;`), or disjunction (`|`). Conjoining patterns constrains fields within a single token; concatenating them constrains a sequence of tokens. In this paper we use patterns that constrain all the bits in a sequence of tokens; such patterns are equivalent to binary representations.

Constructors

A constructor connects the symbolic and binary representations of an instruction by mapping a list of operands to a pattern. The left-hand side of a con-

structor specification resembles the assembly-language syntax of the instruction specified. The right-hand side contains a pattern that describes the binary representation of the instruction. That pattern may contain free identifiers, which refer to the constructor's operands. For example, the following constructor describes the MIPS add instruction:

```
constructors
  add rd, rs, rt  is  op = 0 & funct = 32 & rd & rs & rt
```

where, on the right-hand side, `rd` is an abbreviation for the pattern constraining the field `rd` to be equal to the first operand, and similarly for `rs` and `rt`.

Some instructions have operands that cannot be used directly as field values. The most common are PC-relative branches, in which the operand is the target address, but the corresponding field contains the difference between the target address and the program counter. Constructor specifications may include equations that express relationships between operands and fields. For example, the specifications for the MIPS `bne` and `bltzal` instructions are:

```
constructors
  bltzal rs, addr
     { addr = L + 4 * offset! } is op = 1 & cond = 16 & rs & offset; L: epsilon
  bne rs, rt, addr
     { addr = L + 4 * offset! } is op = 5 & rs & rt & offset; L: epsilon
```

`epsilon` is the pattern specifying the empty sequence of tokens. Here it serves only as an anchor for the label `L`, which is bound to the location of the instruction following the branch. The exclamation point in `offset!` is a sign-extension operator. The equation in braces specifies the relationship between the target address `addr` and the `offset` used in the instruction's binary representation:

> *A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit* `offset`*, [sign-extended and multiplied by 4]* (Kane 1988, p A-23).

The toolkit solves this equation to compute `offset` as a function of `addr` and the program counter. The equation has a solution only when the target address and the program counter differ by a multiple of 4 and when the computed `offset` fits in 16 bits, and the generated encoding function checks these conditions.

3 Instruction encoding and relocation

The toolkit uses the field locations and constraints to figure out the bit manipulations needed to encode an instruction. If `addr` and the program counter are known at the time a `bltzal`, for example, is encoded, we can emit the binary representation directly by using the following function:[1]

$$\lambda(rs, addr).\text{emit}(1 << 26 \,|\, 16 << 16 \,|\, ((addr - PC - 4) >> 2) \,\&\, (2^{16} - 1) \,|\, rs << 21),$$

[1]To simplify the presentation, I have omitted such details as the check that the target address and program counter differ by a multiple of 4.

where I have used C notation for bit manipulation. PC, the program counter, represents the address at which the instruction is to be located. If PC and addr are unknown, we can't emit the instruction; we must create relocation information instead. A typical compiler or assembler emits the instruction with the displacement bits set to zero, along with "relocation information" that tells the linker how to adjust the displacement bits when the relevant locations become known. The relocation information names the instruction, the address on which it depends, and the transformation needed to adjust the displacement bits.

We can discover relocation transformations from the toolkit's description of an instruction. The description tells us how an instruction's operands determine its final, binary representation *after* relocation. When called upon to emit an instruction referring to an unknown location, an assembler must delay encoding, emit a partial instruction, and record a relocation transformation that can be used to compute the final instruction once the location is known. This procedure amounts to currying the relocation function. We must know which operands are relocatable addresses, since these are the operands the values of which may not be known when an object file is created. The MIPS specification contains the directive `relocatable addr`, which specifies that all operands named `addr` are relocatable addresses.

Here is a curried version of the `bltzal` encoding function:

$$\lambda rs.\lambda addr.\mathrm{emit}(1 << 26 \,|\, 16 << 16 \,|\, ((addr - PC - 4) >> 2) \,\&\, (2^{16} - 1) \,|\, rs << 21).$$

When applied to a particular rs, this encoding function returns a closure containing rs and the inner λ-term. To generate C or Modula-3 code, it helps to convert to an explicit closure-passing style (Appel 1992, Chapter 10). Converted functions, i.e., function values, are represented by closures. A closure is a record containing a λ-term, which represents the function's algorithmic content, and the values of the function's free variables. In the λ-term, the function's free variables are replaced by references to the closure, and the closure becomes an explicit argument to the term. After the transformation, the λ-term has no free variables. Different applications of the outer function create different closures; these closures share a λ-term, but they differ in the other contents of the closure record—the values of the free variables. For example, every `bltzal` closure has the form

$$\mathcal{R}_{\texttt{bltzal}} = (\lambda(\mathcal{R}, addr).\mathrm{emit}(1{<<}26|16{<<}16|((addr{-}PC{-}4){>>}2)\&(2^{16}{-}1)|\mathcal{R}[1]{<<}21), rs),$$

but different closures may differ in the value of rs.

Closure conversion also changes the way functions are invoked. In the original form, we could invoke a relocation closure $\mathcal{R} = \lambda addr.\cdots$ by simple function application $\mathcal{R}\ addr$. After closure conversion, we must fetch the λ-term out of the closure and pass the closure as an extra argument. If the closure-converted version is $\mathcal{R}_{\texttt{bltzal}}$, we invoke it by $\mathcal{R}_{\texttt{bltzal}}[0](\mathcal{R}_{\texttt{bltzal}}, addr)$.

The implementation of closure conversion is straightforward. We add a closure argument to each function. We discover the free variables in the body of the function and put each in the closure, and we replace each occurrence of a free variable in the body with code to get its value from the closure.

4 Optimizing relocation closures

In the scheme outlined above, each relocatable instruction needs its own closure function. Compiling these functions takes time, and they take up space in an application or an object file. We can reduce the number of closure functions by moving computation from the inner λ to the outer λ. I call this movement *hoisting*, by analogy with the CPS transformation that moves variable definitions from one scope to another. It simplifies the inner λs, creating opportunities for them to be shared. Hoisting is very closely related to fully lazy lambda-lifting (Peyton Jones 1987, Chapter 15), and the analysis required to implement it is reminiscent of the binding-time analyses used in partial evaluation (Jones, Sestoft, and Søndergaard 1989). Unlike these other techniques, hoisting is not intended to make programs run faster. Hoisting might result in marginally faster linking, but its purpose is to reduce the number of different λ-terms needed to implement relocation.

Hoisting is implemented by a variation on closure conversion. Operands of outer λs are available before those of inner λs. If we think in terms of binding times, λ-bound arguments are always "late," i.e., not available to compute with until the function is applied. Free variables are always "early," i.e., available to compute with when the closure is created. In ordinary closure conversion, free variables are replaced with references to the closure. To perform the hoisting transformation, we want to replace not only free variables to be early, but also terms that depend only on free variables. Such terms are called *free expressions* in Peyton Jones (1987), and a free expression that is not a proper subexpression of another free expression is said to be *maximal*. Fully lazy lambda-lifting rewrites λ-terms to make the maximal free expressions additional operands; hoisting moves them into the closure. For example, to convert the function $\lambda c.a+b+c$, we hoist $a+b$, creating a closure of the form $(\lambda(\mathcal{R}, c).\mathcal{R}[1]+c, a+b)$.

We can implement closure-conversion with hoisting by rewriting a function's abstract syntax tree in a bottom-up walk:

- Leaf nodes are free expressions unless they are variables bound by the innermost enclosing λ-abstraction.
- Internal nodes are free expressions if and only if all their children are free expressions. To simplify the computation, replace each free internal node $e = f(e_1, e_2, \ldots e_n)$ with a fresh variable v, which is free by definition. To remember what v stands for, create the substitution $\sigma = v \mapsto f(e_1, e_2, \ldots e_n)$. Compose these substitutions during the tree walk.

When we reach a λ-abstraction, all free expressions have been replaced with variables, and since no variable can be a proper subexpression of another, the free variables represent the maximal free expressions of the original λ-term. We could recover the original body of the λ-term by applying the substitution σ to it, but instead we closure-convert the rewritten form, then apply the substitution to the closure. Thus, in the example given above,

1. We begin with $\lambda c.a + b + c$.

2. We rewrite it to $\lambda c.v + c$, with substitution $\sigma = v \mapsto a + b$.

3. By ordinary closure conversion, we get $\mathcal{R} = (\lambda(\mathcal{R}, c).\mathcal{R}[1] + c, v)$.

4. We apply σ to the closure, producing $\mathcal{R} = (\lambda(\mathcal{R}, c).\mathcal{R}[1] + c, a + b)$. We can save a minor computation by applying σ only to the variables in the closure; applying it to the λ-term has no effect since after closure conversion the λ-term has no free variables.

To get better results with closures for machine instructions, we rewrite expressions involving associative and commutative operators to bring free expressions together. This rewriting step can reduce the number of maximal free expressions, resulting in simpler λs and smaller closures. The relevant operators include integer addition, assuming that it does not overflow, and bitwise or. Briggs and Cooper (1994) describes a related but more sophisticated technique used to improve the effectiveness of partial-redundancy elimination in a traditional optimizing compiler.

Rearranging associative and commutative operators gives the following relocation closure for `bltzal`:

$$\mathcal{R}_{\texttt{bltzal}} = (\lambda(\mathcal{R}, addr).\text{emit}(\mathcal{R}[1]|((addr - PC + \mathcal{R}[2]) >> 2) \& (2^{16} - 1)), 1 << 26 | 16 << 16 | rs << 21, -4).$$

The new λ-term can be shared with other relative-branch instructions, since all information about the opcode and about the register argument *rs* has been hoisted out of the λ-term and into the closure.

Hoisting moves integer literals, like -4 in this example, into closures. Such literals take up space, and we can improve the closures by using a heuristic: if a value to be stored in the closure is an integer literal, push it back into the λ-term instead of storing it in the closure. We *don't* push other constant expressions into the λ-term. The heuristic works because integer literals tend to arise from address computations, which are typically the same across instructions, but other constant expressions often come from opcodes, which are different for every instruction. To preserve the distinction, we delay constant folding until after hoisting.

Applying the heuristic to the `bltzal` instruction yields a smaller closure:

$$\mathcal{R}_{\texttt{bltzal}} = (\lambda(\mathcal{R}, addr).\text{emit}(\mathcal{R}[1]|((addr - PC - 4) >> 2) \& (2^{16} - 1)), 1 << 26 | 16 << 16 | rs << 21).$$

The literal -4 has moved back into the λ-term.

5 Realizing relocation closures in C

Creating efficient C code to perform relocation by currying requires some refinements. There is no need to put global variables in any closure, because globals are accessible to all functions. Therefore, there is no need to convert top-level functions to closure-passing style, because all their free variables are globals. This is just as well, since C programmers expect functions in an API to be implemented in standard C style, not in closure-passing style!

```
⟨type of closure⟩≡
  typedef struct O1_1_closure {
    ClosureHeader h;    /* contains lambda-term, etc ... */
    ClosureLocation loc;
    struct {  RAddr a1; unsigned u1; } v;
  } *O1_1_Closure;

⟨relocating transformation⟩≡
  static void _clofun_1(O1_1_Closure _c, Emitter emit_at) {
    emit_at(_c->loc,
            _c->v.u1 | location(_c->v.a1) - pc_location(_c->loc) - 4 >> 2 & 0xffff,
            4);
  }

⟨closure creation⟩≡
  { O1_1_Closure _c = (O1_1_Closure)  malloc(sizeof *_c);
    static struct closure_header _h = { _clofun_1, ... };
    _c->h = &_h;
    ⟨initialize _c->loc with current PC⟩
    _c->v.a1 = addr;
    _c->v.u1 = 1 << 26 | 16 << 16 | rs << 21;
    ⟨save closure _c for future use⟩
  }
```

Figure 1: Representing closures in C

The encoding functions and relocation closures generated by the toolkit treat relocatable addresses as values of an abstract data type.[2] An address may be known or unknown, and a known address may be forced to reveal its location, which is an integer. The address itself is supplied when the instruction is encoded; what may not yet be available is the actual location denoted by the address. We have to keep track of the address, so we can force it to a location at relocation time, and the easiest way is to store it in the closure.

Ordinary encoding functions, which create no relocation information, emit code at a "current location" that is part of the global state of the assembler. Relocation closures should not emit instructions at the current location, but at the location of the original encoding attempt. This location, too, is stored in the closure, and instead of "emit," which emits a token at the current location, we use "emit_at," which emits a token at a location given explicitly.

The program counter, PC, gets special treatment. It is another name for the location of the original encoding attempt, and we have to save this location so we know where to put the relocated instruction. If we handled PC as we handle other variables, we would store it in the closure, but since it is already

[2]The toolkit's library of machine-independent assembly and linking code represents a relocatable address as a label plus a constant offset. This representation is adequate for almost all Unix applications (Szymanski 1978), but application writers could substitute another representation.

in a special part of the closure, we rewrite references to PC to refer to that location.

Applying these refinements to the MIPS `bltzal` instruction produces an encoding function that can be represented as follows:

$$\begin{aligned}
&\lambda(rs, addr).(\lambda(\mathcal{R}).\text{emit_at}(\mathcal{R}[1],\\
&\qquad\qquad \mathcal{R}[3] \mid ((\textbf{force}\,\mathcal{R}[2] - \textbf{force}\,\mathcal{R}[1] - 4) >> 2)\ \&\ (2^{16} - 1)),\\
&\quad PC,\\
&\quad addr,\\
&\quad 1 << 26 \mid 16 << 16 \mid rs << 21\\
&\quad).
\end{aligned}$$

The real encoding function is still more complicated, since it emits the instruction directly when *addr* and PC are known, and it also checks the multiple-of-4 and fits-in-16-bits conditions.

The closure-converted form is easily represented in C, as shown in Figure 1. As with the other examples, Figure 1 omits all checking code, as well as such details as converting pointer types and recording the size of the closure. The C code binds `emit_at` as late as possible; the late binding enables different implementations in different applications. The final argument to `emit_at` is the size of the token being emitted; that size has been omitted from the other examples in this paper.

The closure shown in Figure 1 has the same information as a "relocation entry" used in standard object-code formats like COFF (Gircys 1988) and ELF (Prentice Hall 1993a). For example, a COFF relocation entry contains an `r_vaddr` that corresponds to the `loc` field; both store the location of the instruction to be relocated. It contains an `r_symndx` field that corresponds to the `v.a1` field; both store the relocatable address on which the relocation depends. Finally it contains an `r_type` field that corresponds to the `h` field; both identify the relocating transformation. ELF relocation entries are similar, except ELF combines `r_symndx` and `r_type` into a single word. Relocation entries in standard formats have nothing corresponding to the `v.u1` field of the closure shown in Figure 1; instead, they store that information in the space to be occupied by the instruction after relocation. The toolkit could use this space-saving trick, which would reduce the "largest closure" numbers in Table 1, but for the time being it seems more interesting to make relocation closures idempotent. Idempotent closures should be useful in tools that relocate instructions repeatedly, like incremental linkers.

A final refinement is needed to write relocation closures to disk. In memory, the relocating transformation is represented as a function pointer, which is neither machine-independent nor meaningful when written to disk. Instead, we describe relocating transformations using a subset of PostScript (Ramsey 1992), extended with special operators to get addresses and values out of closures. The machine-independent representation of the transformation in the `bltzal` closure is:

```
-4 1 cla force add cl-loc force sub
```

```
-2 bitshift 16 narrows 1 clv orb cl-loc force 4 emit-at
```

The toolkit generates a table that associates the function pointer with this string. The prolixity of this representation is not a problem. Only one copy of each transformation need appear in an object file, and it can go in a string table. It is true, however, that even a small subset of PostScript is overkill for such simple computations. Applications might be better served by a customized bytecode language and an interpreter for that language. Bytecodes would also yield a space savings; with a suitable choice of bytecodes, it would be easy to represent the `bltzal` transformation in a space as small as 13 bytes.

6 Experimental results

I have implemented currying and hoisting in the New Jersey Machine-Code Toolkit (Ramsey and Fernandez 1995). Both the optimized and unoptimized versions are effective. `mld` (Fernandez 1995), a retargetable, optimizing linker, uses encoding functions and relocating transformations generated by the toolkit. `mld` needs only 20 lines of C code for relocation, and it uses the same code on all platforms; the code keeps a list of relocation closures and applies them when the addresses on which they depend become known. Other applications that might use the generated encoding and relocating code include assemblers, linkers, whole-program optimizers, and object-code transformers.

Table 1 shows the amount of space consumed in an application by generated encoding functions and relocating transformations. I used the toolkit to generate encoding and relocating code for the MIPS, SPARC, and Pentium, as specified in Ramsey and Fernandez (1994), and also for the PowerPC 604, as specified by Doug Currie of Flavors Technology. The column labels across the top of Table 1 name the specifications of the target machines for which object code can be generated or relocated.

The upper part of Table 1 describes properties of the specifications and of the generated code. Each instruction accounts for an encoding function, and an encoding function is also generated for each addressing mode. A "relocatable instruction" is one having an operand that is or contains a relocatable address. The next line shows how hoisting reduces the number of closure functions. On the Pentium, the number of closure functions, without hoisting, is greater than the number of relocatable instructions, because the toolkit expands addressing modes inline and generates a different closure for each combination of instruction and addressing mode. Many instructions on the Pentium use one of 8 possible addressing modes, of which 5 involve relocatable addresses. The last line in the top half of Table 1 shows the number of extra words (in addition to the location) stored in the largest closure.

The toolkit supports cross-architecture assembly and linking. The lower part of Table 1 shows how much space the encoding and relocating functions take up for all available combinations of host and target machine.[3] Each row label

[3] I do not have access to a PowerPC to act as a host machine.

		Targets							
		MIPS		SPARC		PPC 604		Pentium	
		Plain	Hoist	Plain	Hoist	Plain	Hoist	Plain	Hoist
	Instructions	167		260		451		645	
	Relocatable insts.	21		97		56		427	
	Closure functions	21	3	97	2	56	4	1911	29
	Largest closure	1	2	0	1	1	1	4	6
Hosts	Object (SPARC)	43.9K	37.0K	153.5K	122.1K	101.0K	83.1K	2331.5K	1650.6K
	Ratio	1.00	**0.84**	1.00	**0.80**	1.00	**0.82**	1.00	**0.71**
	Object (MIPS)	61.0K	51.3K	207.3K	160.9K	142.9K	117.6K	3063.4K	2098.6K
	Ratio	1.00	**0.84**	1.00	**0.78**	1.00	**0.82**	1.00	**0.69**
	Object (Pentium)	27.0K	22.7K	102.0K	81.7K	62.8K	51.2K	1546.5K	1083.8K
	Ratio	1.00	**0.84**	1.00	**0.80**	1.00	**0.82**	1.00	**0.70**

Table 1: Savings from hoisting optimization

identifies a different host machine, on which the relocation code runs. The data in the table are the sizes as compiled with `gcc`, for code generated with and without hoisting. The savings from hoisting are shown in **bold**, as ratios. The reduction in object-code size ranges from 15–20% on the RISC specifications to about 30% on the Pentium specification. The savings is higher on the Pentium because proportionally more instructions take operands that include relocatable addresses.

The encoding functions generated by the toolkit take lots of space because the toolkit trades space for time, generating specialized code for every combination of instruction and addressing mode. This tradeoff is a poor one for the Pentium; the specialized code grows to staggering size because of the inline expansion of addressing modes. Practical applications, like `mld`, use a subset of the full Pentium specification.

The relocating transformations discovered by the toolkit are closely related to those used in standard object formats. For example, the toolkit discovers two transformations for the SPARC, and they are equivalent to the transformations named `R_SPARC_WDISP22` and `R_SPARC_WDISP30` in the ELF format for the SPARC (Prentice Hall 1993b), provided we represent the relocatable address as the sum of the label S and the offset A. (In ELF terminology, these values are called the symbol and the addend.) It discovers only two transformations because the toolkit specification for the SPARC designates fewer operands as relocatable than the standard SPARC assembly language. We can make the toolkit discover more transformations simply by making more operands relocatable; for the SPARC, it requires a 7-line change to a 200-line specification. The toolkit then discovers 253 relocating transformations, which are reduced to 6 by hoisting. The new transformations are `R_SPARC_13`, `R_SPARC_22`, `R_SPARC_HI22`,

and a combination of R_SPARC_HI22 and R_SPARC_LO10. If we add constructors to store relocatable addresses in 8-bit, 16-bit, and 32-bit tokens, the toolkit discovers R_SPARC_8, R_SPARC_16, and R_SPARC_32. In the presence of these extra relocatable operands and the extra relocating transformations, the savings from hoisting increases from 20% to 30%.

There are transformations the toolkit does not discover. Some are specialized versions of the ones that are discovered. For example, several ELF transformations are specialized to refer to locations relative to the start of a "global offset table" or a "procedure linkage table." Some relocation entries in standard object files cannot be discovered by the toolkit because they represent more than just transformations. For example, the R_SPARC_GLOB_DAT relocation entry names the same transformation as R_SPARC_32, but it also instructs the linker to create an entry in the global offset table.

7 Discussion

Hoisting works well on the RISC machines because most instructions occupy a single token. It works less well on the Pentium, because the toolkit creates a single closure for an encoding of an instruction even when that encoding is a sequence of tokens. The relocating transformation for that closure must relocate every token in the sequence. To minimize the number of relocating transformations functions, it would be better to create a closure for each token that depends on a relocatable address, since relocating transformations for single tokens can be reused more freely than those for sequences. Moreover, by splitting up sequences, we could emit some tokens immediately, not having to wait until relocation time. We could go further and create separate closures for different fields of a single token. This technique would be unlikely to reduce the number of transformations further, but it would make it easy to create idempotent transformations while using the encoded instruction itself to store part of the closure.

The abstract view of relocatable addresses has a cost. If we exposed the "label + offset" representation at code-generation time, we could realize extra savings. Offsets are always available at encoding time, and they could be hoisted out of closure functions. Storage requirements would be reduced because a label occupies no more than half the space of a (label, offset) pair. Exposing the representation would also make it possible to treat certain labels, like those of the ELF global offset table and procedure linkage table, as special cases. Such treatment would make it possible to shrink machine-independent object code by moving these special labels back into the λs.

Currying and hoisting make it possible to write efficient, machine-independent tools that relocate machine instructions. The New Jersey Machine-Code Toolkit can derive C implementations of relocating transformations from a set of machine descriptions, and a tool writer can incorporate those implementations to provide efficient relocation on a number of platforms. By including an interpreter for a bytecode representation of relocating transformations, the tool

writer can undertake to relocate instructions for any machine—even a machine that doesn't exist when the tool is released.

Acknowledgements

Mary Fernandez helped create the toolkit on which this work is based, and she tested the optimized closures. Peter Sestoft provided helpful pointers to the literature on partial evaluation and functional programming. Vince Russo, Zhong Shao, and Michal Young provided useful criticism of the manuscript.

References

Appel, Andrew W. 1992.
Compiling with Continuations.
Cambridge: Cambridge University Press.

Ball, Thomas and James R. Larus. 1992 (January).
Optimally profiling and tracing programs.
In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, NM.

Briggs, Preston and Keith D. Cooper. 1994 (June).
Effective partial redundancy elimination.
Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 29(6):159–170.

Fernandez, Mary F. 1995 (June).
Simple and effective link-time optimization of Modula-3 programs.
Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 30(6):103–115.

Fraser, Christopher W. and David R. Hanson. 1982 (April).
A machine-independent linker.
Software—Practice & Experience, 12(4):351–366.

Gircys, Gintaras R. 1988 (November).
Understanding and Using COFF. Nutshell Handbooks.
Sebastopol, CA: O'Reilly & Associates.

Hastings, Reed and Bob Joyce. 1992 (January).
Purify: Fast detection of memory leaks and access errors.
In *Proceedings of the Winter USENIX Conference*, pages 125–136.

Johnson, Stephen C. 1990.
Postloading for fun and profit.
In *Proceedings of the Winter USENIX Conference*, pages 325–330.

Jones, Neil D., Peter Sestoft, and Harald Søndergaard. 1989.
Mix: A self-applicable partial evaluator for experiments in compiler generation.
Lisp and Symbolic Computation, 2(1):9–50.

Jones, Douglas W. 1983 (August).
Assembly language as object code.
Software—Practice & Experience, 13(8).

Kane, Gerry. 1988.
MIPS RISC Architecture.
Englewood Cliffs, NJ: Prentice Hall.

Larus, James R. and Eric Schnarr. 1995 (June).
Eel: machine-independent executable editing.
In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

Peyton Jones, Simon L. 1987.
The Implementation of Functional Programming Languages. International Series in Computer Science.
Englewood Cliffs, NJ: Prentice Hall.

Prentice Hall. 1993a.
System V Application Binary Interface. Third edition.
Englewood Cliffs, NJ.
Unix Press.

Prentice Hall. 1993b.
System V Application Binary Interface, SPARC Architecture Processor Supplement. Third edition.
Englewood Cliffs, NJ.
Unix Press.

Ramsey, Norman and Mary F. Fernandez. 1994 (October).
New Jersey Machine-Code Toolkit architecture specifications.
Technical Report TR-470-94, Department of Computer Science, Princeton University.

———. 1995 (January).
The New Jersey machine-code toolkit.
In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA.

Ramsey, Norman. 1992 (December).
A Retargetable Debugger.
PhD thesis, Princeton University, Department of Computer Science.
Also Technical Report CS-TR-403-92.

Sites, Richard L., Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. 1993 (February).
Binary translation.
Communications of the ACM, 36(2):69–81.

Srivastava, Amitabh and Alan Eustace. 1994 (June).
Atom: A system for building customized program analysis tools.
In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205.

Srivastava, Amitabh and David W. Wall. 1993 (March).
A practical system for intermodule code optimization.
Journal of Programming Languages, 1:1–18.
Also available as WRL Research Report 92/6, December 1992.

Szymanski, Thomas G. 1978 (April).
Assembling code for machines with span-dependent instructions.
Communications of the ACM, 21(4):300–308.

Wahbe, Robert, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993 (December).
Efficient software-based fault isolation.
In *Proc. Fourteenth ACM Symposium on Operating System Principles*, pages 203–216.