Relocating Machine Instructions by Currying

Norman Ramsey
University of Virginia

Relocation adjusts machine instructions to account for changes in the locations 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.

An alternative is to use SLED (Specification Language for Encoding and Decoding) to specify representations of machine instructions. 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. From a SLED specification, the New Jersey Machine-Code Toolkit can generate functions that encode instructions in the native binary representation. For instructions with relocatable operands, the toolkit also computes relocating transformations. Tool writers can create machine-independent software that uses these transformations to relocate machine instructions. For example, `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 an instruction with 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 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 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 15–25% for the Alpha, MIPS, SPARC, and PowerPC, and by about 40% for the Pentium. It also makes the second λs equivalent to relocating transformations named in standard object-file formats.

Categories and Subject Descriptors: D.4.9 [**Operating Systems**]: Systems Programs and Utilities—*Linkers*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classifications—*specialized application languages*

General Terms: Relocation, Linking, Currying

Additional Key Words and Phrases: higher-order functions

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

---

Authors' current address: Norman Ramsey, Department of Computer Science, University of Virginia, Charlottesville, VA, 22903. Email `nr@cs.virginia.edu`. This work supported in part by NSF grant number ASC-9612756.

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 their eventual placement at absolute addresses and also for the placements of the external symbols on which they depend.

Applying these transformations is called *relocation*.

Current object-code formats force tool writers to handle relocation in a machine-dependent way. Given an instruction-set architecture, a human being examines the instructions 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 transform object code. 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 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 Fernández 1997], which reads a compact machine description and generates functions that encode instructions. The machine description is written in SLED (Specification Language for Encoding and Decoding), which relates three representations of instructions: a symbolic representation akin to assembly language, assembly language itself, and the binary representation used by the hardware. Real instruction sets can be specified with modest effort; our Alpha, MIPS, SPARC, and Pentium specifications [Ramsey and Fernández 1994] are 118, 127, 193, and

460 lines. In SLED, an instruction is represented symbolically by its name and a list of its operands; the collection of all instructions resembles an algebraic data type. The machine description indicates which operands are relocatable addresses, and 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 12–25% for the Alpha, MIPS, SPARC, and PowerPC, and by about 40% 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 REPRESENTATIONS

SLED 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 the 32-bit displacement 612 and the 8-bit immediate operand 33.

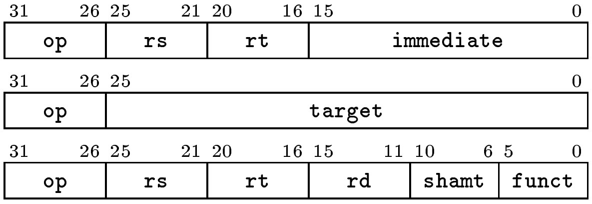
Each token in an instruction is partitioned into *fields*; a field is a contiguous range of bits within a token. On RISC machines, different instruction formats are represented by different partitions of the instruction token. Fields contain opcodes, operands, modes, or other information. Opcodes and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

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, which complete the API.

Tokens and fields

A machine description includes the names, sizes, and positions of the fields used to form tokens. This information can be found in architecture manuals. For example, the MIPS manual [Kane 1988, p A-3] uses a picture to specify fields:



The picture can be formalized in SLED 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. This paper uses 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 constructor specification gives the instruction's name, operands, and assembly-language syntax. The right-hand side contains a pattern that describes the instruction's binary representation. 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, since the first operand is named `rd`. The same rule applies to the uses of `rs` and `rt` on the right-hand side.

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; the equations appear in braces after the operands. 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]

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

where I use C notation for bit manipulation. $1 << 26$ encodes the `op = 1` constraint, $16 << 16$ encodes the `cond = 16` constraint, and $rs << 21$ puts the operand `rs` into the instruction. The remaining disjunct expresses the computation needed to compute the value of the `offset` field given a target address $addr$ and a program counter PC. The computation includes arithmetic, a shift, and a narrowing to 16 bits. PC 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 a "relocation entry" that tells the linker how to adjust the displacement bits when the relevant locations become known. The relocation entry names the instruction, the address on which it depends, and the transformation needed to adjust the displacement bits.

[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.

The toolkit can discover relocating transformations from SLED's description of an instruction. The description tells 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 relocating transformation that can be used to compute the final instruction once the location is known. This procedure amounts to currying the encoding function. We must know which operands are relocatable addresses, since theirs are the values that 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.

Currying the `bltzal` encoding function yields

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

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. These references take the form $\mathcal{R}[i]$, which denotes the ith element (numbered from 0) of closure record $\mathcal{R}$. The closure becomes an explicit argument to the λ-term, so after the transformation, the λ-term has no free variables.

In relocation, the inner λ-terms describe relocating transformations. When an encoding function is curried, 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 is a pair of the form

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

but different closures may have different values 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 [Appel 1992, Chapter 8]. 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 et al. 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 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 $e = f(e_1, e_2, \dots e_n)$ with a fresh variable v, which is free by definition. To remember what v stands for, create the substitution $\sigma = v \mapsto f(e_1, e_2, \dots e_n)$. 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 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] use an equivalent technique to improve the effectiveness of partial-redundancy elimination in a traditional optimizing compiler; the rank they assign to each variable corresponds to the number of λs between a free occurrence of a variable and its binding instance.

Rearranging associative and commutative operators gives the following relocation closure for `bltzal`:[2]

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

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:

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

The literal -4 has moved back into the λ-term, and the closure record is back down to 2 elements.

In the examples given so far, each instruction is represented by a single token. If an instruction has operands x and y, and if its binary representation is computed by the function $f(x, y)$, we can characterize the hoisting transformation as moving part of this computation outside the λ-abstraction:

$$\lambda x.\lambda y.\mathrm{emit}(f(x,y)) \Longrightarrow \lambda x.\langle \lambda y.\mathrm{emit}(f''(f'(x),y)), f'(x)\rangle,$$

where the angle brackets $\langle\cdots\rangle$ stand for closure creation, and $f''(f'(x), y) = f(x, y)$. (Too keep the inner λ simple, I have inlined references to the closure record.) When an instruction is represented by a sequence of tokens, as is common on CISC machines, there is an opportunity for further improvement; we can move the

[2]The astute reader may wonder why the literal $2^{16}-1$ used in masking is not moved to the closure. The toolkit's intermediate form restricts masking operations to constants of the form $2^k - 1$, and the constant k is attached directly to the & operator, so $2^{16}-1$ is not a free expression in the sense defined above. For similar reasons, the 2 in $(\cdots) >> 2$ is not moved to the closure.

sequence operator itself outside the λ-abstraction:

$$\begin{aligned}&\lambda x.\lambda y.\mathbf{seq}\{\mathrm{emit}(f_1(x,y));\cdots;\mathrm{emit}(f_n(x,y))\} \Longrightarrow\\ &\quad\lambda x.\mathbf{seq}\{\langle\lambda y.\mathrm{emit}(f_1''(f_1'(x),y)), f_1'(x)\rangle;\cdots;\langle\lambda y.\mathrm{emit}(f_n''(f_n'(x),y)), f_n'(x)\rangle\}.\end{aligned}$$

In other words, instead of creating one closure to relocate all the tokens in the sequence, we can create a separate closure to relocate each token. Although it may create more closures, this improvement yields smaller closures, because each closure holds information about only one token, and it yields fewer unique closure functions, because it creates opportunities for sharing closure functions between different instructions. The improvement is especially useful on the Pentium, where normally only one token in a sequence depends on the relocatable address, and the others can be emitted immediately, requiring no further relocation. Formally, when $f(x,y)$ depends only on x, we rewrite

$$\langle\lambda y.\mathrm{emit}(f_i''(f_i'(x),y)), f_i'(x)\rangle \Longrightarrow \mathrm{emit}(f(x,\perp)),$$

which emits the token and creates no closure. The measurements for hoisting in Section 6 incorporate this improvement.

5. 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!

The encoding functions and relocation closures generated by the toolkit treat relocatable addresses as values of an abstract data type with two operations: **known** and **force**.[3] **Force** takes a relocatable address and produces an (integer) absolute address. **Known** tells whether **force** can be applied. The relocatable address 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," which is part of the global state of the toolkit's encoding library. 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

[3]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.

⟨*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⟩
}
```

Fig. 1. Representing closures in C

variables, we would store it in the closure, but since it is already 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:

$$\begin{aligned}\lambda(rs, addr).\langle\lambda(\mathcal{R}).\mathrm{emit\_at}(&\mathcal{R}[1],\\ &\mathcal{R}[3] \mid ((\mathbf{force}\,\mathcal{R}[2] - \mathbf{force}\,\mathcal{R}[1] - 4) >> 2)\ \&\ (2^{16}-1)),\\ &PC,\\ &addr,\\ &1 << 26 \mid 16 << 16 \mid rs << 21\rangle.\end{aligned}$$

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 I, 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, again omitting the tests of conditions, is

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

The first line takes the relocatable address from the closure, subtracts 4, and subtracts the location of the instruction being relocated, computing $addr - PC - 4$. The second line shifts right 2 bits, narrows to 16 bits, and combines the result with the rest of the instruction, as stored in the closure. The third line stores the instruction, which is 4 bytes wide, at the proper location.

The toolkit generates a table that associates the function pointers used in closures with machine-independent strings like the one shown above. A machine-independent object file might include one one copy of each transformation. To minimize the space required to store these transformations, the toolkit can encode the transformation in a specialized but unreadable bytecode. It packs the full transformation in the `bltzal` closure, including the tests of conditions, into the 24-byte string given by the C literal

```
"\r\x15\v'\n\x03n]\x15\v'\xcb\xec\x80%\x07\x85-~]\x01-M?"
```

The upper half of Table I, in the next section, shows how much space is needed to hold the bytecodes for all the transformations on each of five target machines.

6. EXPERIMENTAL RESULTS

I have implemented currying and hoisting in the New Jersey Machine-Code Toolkit [Ramsey and Fernández 1997]. `mld` [Fernández 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

Targets

	Alpha		MIPS		SPARC		PPC 604		Pentium	
	Plain	Hoist	Plain	Hoist	Plain	Hoist	Plain	Hoist	Plain	Hoist
Instructions	300		167		276		451		606	
Relocatable insts.	18		21		99		56		393	
Closure functions	18	2	21	2	99	5	56	4	1988	5
Bytecode size	—	49	—	59	—	62	—	96	—	22
Largest closure	1	1	1	1	1	1	1	1	4	0

Hosts

Hosts	Alpha Plain	Alpha Hoist	MIPS Plain	MIPS Hoist	SPARC Plain	SPARC Hoist	PPC 604 Plain	PPC 604 Hoist	Pentium Plain	Pentium Hoist
Alpha object code	92.5K	82.0K	64.9K	51.5K	229.8K	173.0K	157.7K	124.2K	2928.1K	1723.7K
Ratio	1.00	**0.89**	1.00	**0.79**	1.00	**0.75**	1.00	**0.79**	1.00	**0.59**
MIPS object code	94.5K	82.6K	68.5K	53.5K	262.6K	198.3K	162.0K	125.2K	3597.7K	2174.7K
Ratio	1.00	**0.87**	1.00	**0.78**	1.00	**0.76**	1.00	**0.77**	1.00	**0.60**
SPARC object code	61.3K	52.9K	46.7K	35.8K	182.1K	137.2K	109.0K	82.4K	2497.9K	1529.0K
Ratio	1.00	**0.86**	1.00	**0.77**	1.00	**0.75**	1.00	**0.76**	1.00	**0.61**
RS/6000 object code	96.6K	84.6K	68.9K	53.8K	240.8K	173.7K	168.2K	130.2K	3870.5K	2285.8K
Ratio	1.00	**0.88**	1.00	**0.78**	1.00	**0.72**	1.00	**0.77**	1.00	**0.59**
I386 object code	38.8K	32.7K	30.8K	22.8K	127.0K	93.6K	72.5K	52.8K	1778.3K	1029.3K
Ratio	1.00	**0.84**	1.00	**0.74**	1.00	**0.74**	1.00	**0.73**	1.00	**0.58**

Instruction counts give the sizes of the instruction sets.

Sizes in K are code sizes needed to encode and relocate all instructions.

All routines were compiled by `gcc`, except the Pentium encoding routines on the Alpha host, which triggered a bug in `gcc`. Those routines were compiled with `lcc` [Fraser and Hanson 1995].

Table I. Space savings from hoisting optimization might use the generated encoding and relocating code include assemblers, linkers, whole-program optimizers, and object-code transformers.

One can imagine several measures of the performance of a relocation method: the space required to store the relocation code, the time required to execute it, the space required to store object modules, and the time required to execute the relocated binary code. Ideally, one could use these measures to compare currying with standard methods of relocation, but `mld` is the only linker that uses relocation by currying, and `mld`'s assumptions make meaningful comparisons difficult. For example, `mld` uses no object modules, so it is impossible to measure their sizes. This section focuses on the size of the relocation code and the time required to execute it. There is no need to compare the speed of binary codes as relocated by currying or by hand-written code, since both methods result in identical executable binaries.

This section also compares SPARC relocating transformations discovered by the toolkit with transformations defined by the ELF object-code standard.

Code size and hoisting

I used the toolkit to generate encoding and relocating code for the Alpha, MIPS, SPARC, and Pentium, as specified in Ramsey and Fernández [1994], and also for the PowerPC 604, as specified by Doug Currie of Flavors Technology. Table I shows the amount of space consumed in an application by generated encoding functions and relocating transformations. The column labels across the top name the specifications of the target machines for which object code can be generated or relocated. The results in Table I depend only on the specifications and on the toolkit itself; they are independent of the program being relocated.

The upper part of Table I describes properties of the instruction-set specifications and of the code generated to implement them. Each instruction accounts for an encoding function, as does each addressing mode. A "relocatable instruction" has an operand that is or contains a relocatable address. The table 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 effective addresses, which come in 8 modes, of which 5 involve relocatable addresses.

The "bytecode size" in Table I shows how many bytes of machine-independent bytecode are needed to represent the closure functions, as described above. This size is shown only for the hoisted functions; before hoisting, the closure functions don't have a meaningful bytecode representation, because the bytecode isn't rich enough to handle the more complicated closures. The last line in the top half of Table I shows the number of extra words (in addition to the location) stored in the largest closure. Only on the Pentium does relocating one token at a time result in smaller closures. This result makes sense because the Pentium is the only machine that uses sequences of tokens in which not all tokens depend on a relocatable address.

The toolkit supports cross-architecture assembly and linking. The lower part of Table I shows how much space the encoding and relocating functions take up for every combination of host and target machine.[4] Each row label 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 15-25% on the RISC specifications to about 40% on the Pentium specification. The differences in savings are explained by the differences in the proportion of instructions that use relocatable addresses.

When generating encoding functions, the toolkit trades space for time, generating specialized code for every combination of instruction and addressing mode. The toolkit does not encode an effective address until it knows in what instruction the address is used. Because of the inline expansion of addressing modes, this tactic is spectacularly costly on the Pentium. A meaningful measurement of the value of hoisting on the Pentium will have to await the elimination of this code bloat. Practical applications, like `mld`, use a subset of the full Pentium specification.

Hoisting reduces the size of relocation code to less than two percent of the size of encoding code. The sizes of relocation functions in SPARC object files are

Alpha	MIPS	SPARC	PPC 604	Pentium
0.6K	0.6K	0.9K	1.1K	0.7K

To compare one of these sizes with hand-written code, I examined the SPARC relocation code used in the GNU and Solaris linkers. These linkers implement all 23 of the transformations in the ELF standard, but the toolkit implements only 5. The GNU code is table-driven; the Solaris code is not.

The sizes of the three different implementations of relocation are comparable. The toolkit uses 1012 bytes of code to implement 5 transformations. The GNU linker uses 1420 bytes of code to interpret table entries, and the table requires 106 bytes per transformation, for a total of 1950 bytes for 5 transformations. These measurements describe the GNU code as altered to work inside `mld`; as part of the alteration I made several simplifying assumptions and eliminated code accordingly. The alterations are described in greater detail below. The Solaris linker uses 5192 bytes of code to implement 23 transformations.

Speed of relocation

I estimated differences in relocation speed by transplanting GNU and Solaris relocation code into `mld`. I removed significant portions of the GNU code to try to make the link-time assumptions like `mld`'s assumptions. For example, I eliminated support for multiple symbol tables and for generation of relocatable object code, and I removed many sanity checks and assertions. The GNU code relocates into and out of specialized "sections," but `mld` works directly in memory, so where possible I modified the GNU code to use memory addresses directly instead of sections and offsets. The Solaris code required similar, but less sweeping modifications.

To use the modified code in `mld`, I translated the toolkit's relocation closures into ELF-style relocation entries. I used `mld` to link four of the SPEC benchmarks (`eqntott`, `li`, `gcc`, and `espresso`), doing relocation all three ways. The

[4]Not having access to a PowerPC to act as a host machine, I used an RS/6000, which has a similar instruction set.

three methods yield identical instructions. Using `spix` from the Shade distribution [Cmelik and Keppel 1994], I measured the number of SPARC instructions needed to relocate these programs. Relocation by currying takes 22–26% fewer instructions than the GNU relocation code, but the Solaris relocation code takes 26–29% fewer instructions than relocation by currying. These measurements should not be taken too seriously, because the foreign relocation code is far removed from its original context, and the three methods vary in the assumptions they make and the amount of work they do. For example, relocation by currying checks to see if symbols are defined, the GNU code makes some less stringent sanity tests, and the Solaris code makes no checks at all. The measurements do indicate that relocation by currying costs about the same as standard methods.

The most interesting variation in method may be that the GNU and Solaris code require that the contents of each "section" be stored in contiguous memory. This requirement is awkward for `mld`, because it uses a lifetime-based memory allocator [Hanson 1990], and it does not know section sizes in advance. Relocation by currying permits the contents of sections to be split into any number of contiguous blocks, but there is a performance penalty. Every time an instruction is relocated, the relocation code must search for the contiguous block containing that instruction. If the searching is done in advance and the search time not counted, the cost of relocation by currying drops by about 20%. This change simulates the operation of a linker of object modules, in which sections are always contiguous because they are made so by an assembler.

Relationship to standards

The relocating transformations discovered by the toolkit are equivalent to those used in standard object formats. For example, the toolkit discovers five transformations for the SPARC, and they are equivalent to the transformations named `R_SPARC_WDISP30`, `R_SPARC_WDISP22`, `R_SPARC_HI22`, `R_SPARC_LO10`, and `R_SPARC_32` 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.) The toolkit discovers only five transformations because the SLED specification for the SPARC designates fewer operands as relocatable than does standard SPARC assembly language. We can make the toolkit discover more transformations simply by making more operands relocatable; adding a few lines to the SPARC specification helps the toolkit discover `R_SPARC_13` and `R_SPARC_22`. If we add constructors to store relocatable addresses in 8-bit and 16-bit tokens, the toolkit discovers `R_SPARC_8` and `R_SPARC_16`.

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.

Size of object code

Because `mld` creates no object modules, I cannot directly measure the effects of relocation by currying on the sizes of object modules, but I can make predictions based on the experiment of embedding GNU relocation code in `mld`. Each relocation closure can be translated into a relocation entry using the standard ELF representation. The only auxiliary data structure needed is a mapping of small integers to λ-terms; instead of being fixed by a machine-dependent object-code standard, this mapping must be stored in an object file. As shown in Table I, one could use the toolkit's byte-coded representation of λ-terms to store this mapping in no more than 100 bytes per object file. It might be possible to reduce the size of the bytecodes by using Proebsting's "superoperator" technique [Proebsting 1995].

7. DISCUSSION

Relocation by currying is a simple, abstract, machine-independent model of relocation. Abstract relocatable addresses have a cost; exposing the "label + offset" representation at code-generation time would enable extra savings. Offsets are always available at encoding time, and they could be hoisted out of closure functions. The closures would take less space, because a label occupies at most half the space of a (label, offset) pair. (The ELF object-code standard enables such space optimizations by providing for relocation entries both with and without offsets.) Exposing the representation of relocatable addresses 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. In particular, by keeping the number of distinct relocating transformations small, hoisting makes a machine-independent object code practical. 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. If the tool includes an interpreter for the bytecode representation of relocating transformations, it can relocate instructions for any machine—even a machine that doesn't exist when it is released.

Acknowledgements

Mary Fernández helped create the toolkit on which this work is based, and she put together an `mld` I could use for performance measurements. Peter Sestoft provided helpful pointers to the literature on partial evaluation and functional programming. Mary Fernández, Vince Russo, Zhong Shao, and Michal Young criticized the manuscript in helpful ways.

REFERENCES

Appel, A. W. 1992. *Compiling with Continuations.* Cambridge University Press, Cambridge.

Ball, T. and Larus, J. R. 1992. Optimally profiling and tracing programs. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages.* Albuquerque, NM, 59–70.

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

Cmelik, B. and Keppel, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 128–137.

Fernández, M. F. 1995. 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 (June), 103–115.

Fraser, C. W. and Hanson, D. R. 1982. A machine-independent linker. *Software—Practice & Experience 12,* 4 (Apr.), 351–366.

Fraser, C. W. and Hanson, D. R. 1995. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, Redwood City, CA.

Gircys, G. R. 1988. *Understanding and Using COFF.* Nutshell Handbooks. O'Reilly & Associates, Sebastopol, CA.

Hanson, D. R. 1990. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice & Experience 20,* 1 (Jan.), 5–12.

Hastings, R. and Joyce, B. 1992. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*. San Francisco, CA, 125–136.

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

Jones, D. W. 1983. Assembly language as object code. *Software—Practice & Experience 13,* 8 (Aug.).

Jones, N. D., Sestoft, P., and Søndergaard, H. 1989. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation 2,* 1, 9–50.

Kane, G. 1988. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ.

Larus, J. R. and Schnarr, E. 1995. EEL: machine-independent executable editing. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices 30,* 6 (June), 291–300.

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

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

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

Proebsting, T. A. 1995. Optimizing an ANSI C interpreter with superoperators. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*. San Francisco, California, 322–332.

Ramsey, N. 1992. A retargetable debugger. Ph.D. thesis, Princeton University, Department of Computer Science. Also Technical Report CS-TR-403-92.

Ramsey, N. and Fernández, M. F. 1994. New Jersey Machine-Code Toolkit architecture specifications. Tech. Rep. TR-470-94, Department of Computer Science, Princeton University. Oct. Revised December, 1996.

Ramsey, N. and Fernández, M. F. 1997. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*. To appear.

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

Srivastava, A. and Eustace, A. 1994. ATOM: A system for building customized program analysis tools. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices 29,* 6 (June), 196–205.

Srivastava, A. and Wall, D. W. 1993. 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, T. G. 1978. Assembling code for machines with span-dependent instructions. *Communications of the ACM 21,* 4 (Apr.), 300–308.

Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. 1993. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles.* 203–216.