A Transformational Approach to Binary Translation of Delayed Branches

NORMAN RAMSEY
Harvard University
and
CRISTINA CIFUENTES
Sun Microsystems Laboratories

A binary translator examines binary code for a source machine and generates code for a target machine. Understanding what to do with delayed branches in binary code can involve tricky case analyses, for example, if there is a branch instruction in a delay slot. This article presents a disciplined method for deriving such case analyses. The method identifies problematic cases, shows the translations for the nonproblematic cases, and gives confidence that all cases are considered. The method supports such common architectures as SPARC, MIPS, and PA-RISC, and it should apply to any tool that analyzes machine instructions. We begin by writing a very simple interpreter for the source machine's code. We then transform the interpreter into an interpreter for a target machine without delayed branches. To maintain the semantics of the program being interpreted, we simultaneously transform the sequence of source-machine instructions into a sequence of target-machine instructions. The transformation of the instructions becomes our algorithm for binary translation.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors; I.1.4 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation—*applications*

General Terms: Experimentation, Languages

Additional Key Words and Phrases: Binary translation, program analysis, program transformation

1. INTRODUCTION

Binary translation makes it possible to run code compiled for source platform S on target platform T. Unlike interpreted or emulated code, binary-translated code approaches the speed of native code on machine T.

The fundamental steps in binary translation are to distinguish code from data, to map data locations from the source to the target machine, and to translate instructions [Sites et al. 1993]. The problem of distinguishing code

This work has been supported by grant A49702762 from the Australian Research Council. N. Ramsey has additional support from National Science Foundation grants ASC-9612756 and CCR-9733974 and from DARPA contract MDA904-97-C-0247.
Authors' addresses: N. Ramsey, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA 01238; email: nr@eecs.harvard.edu; C. Cifuentes, Sun Microsystems Laboratories, Palo Alto, CA 94303; email: cristina.cifuentes@sun.com.

from data is difficult, but solutions are well known [Larus and Ball 1994]. This article focuses on translating instructions.

When a mapping from source locations to target locations has been established, finding the target instructions needed to achieve a particular effect is simply code generation. But the effect of a delayed-branch instruction is not always obvious, especially when a branch or call instruction appears in a delay slot. The contribution of this article is a disciplined method for understanding the effects of delayed branches, even in tricky combinations that are used only in kernel code [SPARC International 1992, §B.26] or instrumented code [Ronsse and De Bosschere 2001]. Because such tricky combinations are rare, performance is not an issue. A simple translation is good enough, especially since a simple translation is easily improved by a peephole optimizer. The benefit of our method is not to improve performance, but to identify problematic cases, give translations for them, and give the programmer confidence that all cases have been considered.

Our method uses register-transfer lists (RTLs) to reason about instructions on both the source and target machines [Ramsey and Davidson 1998]. We divide a machine's semantics into two parts. We specify semantics common to most instructions (e.g., the advancement of the program counter) as part of a simple imperative program representing the execution loop of a machine. We specify the unique effect of each instruction as a register-transfer list. The effect of executing a program is represented as the effect of running the execution loop on a sequence of instructions, or more precisely, on a sequence of register-transfer lists representing the semantics of the instructions.

To build a binary translator, we consider semantics for two machines. Each has an execution loop and a set of instructions. We transform the source machine's execution loop into the target machine's execution loop. To preserve the program's semantics, we simultaneously transform the sequence of source-machine instructions into a sequence of target-machine instructions. This second transformation is our algorithm for binary translation.

A quick reading of this article might suggest that the problem we solve is trivial. To build a flow graph representing a binary program, why not simply convert the delayed branch to a nondelayed branch and push the instruction in the delay slot along zero, one, or both sucessor edges? This simple approach can fail on an architecture such as PA-RISC, where a branching instruction (e.g., BLE) may have other computational effects. The simple approach also fails, on any architecture, when the instruction in the delay slot is itself a delayed branch, call, or other transfer of control. Despite the failures of the "pushing" approach, it appears to be the only approach known to many practitioners who are charged with developing binary translators and similar tools. Many such practitioners have told us that they find it "very tricky" to get their code to work correctly "for all cases." Our method translates all cases correctly.

Although the delayed branch is an architectural idea whose time has come and gone, legacy codes for SPARC, MIPS, and PA-RISC architectures will be with us for some time, and those who own such codes will need tools for analysis and translation. If a new version of SPARC or PA-RISC architecture is to eliminate delayed branches, it will be because the legacy codes can be mapped to the

new architecture using binary translation. The binary translator in question must deal with *all* codes written for the old machine, not just almost all codes.

The building blocks of this article are not new. Register-transfer languages have been used to describe instructions for years [Bell and Newell 1971; Barbacci and Siewiorek 1982]. Our program transformations draw from standard techniques in compiler optimization [Aho et al. 1986] and partial evaluation [Jones et al. 1993]. This article shows that these techniques, which were developed in an abstract, high-level setting, can be profitably applied even at the machine level.

2. SEMANTIC FRAMEWORK

We express source and target instructions using *register-transfer lists* (RTLs). Our full RTL formalism makes all machine-dependent computation explicit, including byte order [Ramsey and Davidson 1998]. For this work, we use a simplified version specified using the following syntax.

rtl	$\Rightarrow$ [*effect* { | *effect*}]		Multiple assignment
effect	$\Rightarrow$ [*exp* $\rightarrow$] *location*	:= *exp*	Guarded assignment
exp	$\Rightarrow$ *constant*		Constant
	| *location*		Fetch from a location
	| *exp binop exp*		Binary RTL operator
	| *operator* (*explist*)		RTL operator.

A register-transfer list is a list of guarded effects. Each effect represents the transfer of a value into a storage location,[1] that is, a store operation. The transfer takes place only if the guard (an expression) evaluates to **true**. Any two effects must either assign to distinct locations or have mutually exclusive guards. This restriction is imposed because effects in a list take place simultaneously, as in Dijkstra's multiple-assignment statement; an RTL represents a single change of state. The state is modeled as a collection of *storage spaces*. A storage space is an array of mutable cells; the space is identified by a single letter, such as m for memory, r for registers, and so on.

An expression is an integer literal, a fetch from a location, or an application of an *RTL operator* to a list of expressions. An RTL operator denotes a pure function on values. In this article, we assume that a location is a single cell in a mutable store.

As an example of a typical RTL, consider a SPARC load instruction using the displacement addressing mode, written in the SPARC assembly language as

```
ld [%sp-12], %i0
```

This load instruction computes an address by adding -12 to the stack pointer (register 14), then loads a word from that address into register `%i0` (register 24). The effect of the instruction might be written

$$\$r[24] := \$m[\$r[14] + sx(-12)].$$

[1]Storage locations represent not only memory but also registers and other processor state.

The notation $\$space[address]$ represents the cell numbered *address* in the storage space *space*. The sx operator sign-extends the 13-bit immediate constant -12 so it can be added to the 32-bit value fetched from register 14.

2.1 Processor State for Delayed Branches

When the processor executes a load instruction, it not only loads a value into register 24; it also advances the program counter to point to the next instruction. Advancing the program counter makes the processor execute instructions in sequence until it gets to a branch instruction. When it reaches an instruction I that causes a delayed branch to a location *target*, the processor first executes I's successor, then executes the instruction located at *target*. The location holding I's successor is called I's "delay slot." On some architectures, such as the SPARC architecture, the instruction I can "annul" its successor, in which case the successor is *not* executed; instead the processor stalls for a cycle before transferring control to *target*.

To model delayed branches with annuls, we use three pieces of processor state:

PC is the program counter, which identifies the instruction about to be executed.

nPC is the "next program counter," which identifies the instruction to be executed after the instruction at *PC*.

annul is the "annul status," which determines whether the processor executes the instruction at *PC* or ignores it.[2]

In this model, the processor sets $PC := nPC$ after execution of each instruction, and it increments *nPC* after a nonbranching instruction. A branch, which is delayed, is represented by an assignment to *nPC*. For example, a SPARC call instruction simultaneously assigns the target address to *nPC* and the current *PC* to register 15. (Register 15 is called %o7 in SPARC assembly language, and it is where the caller expects to find the return address minus 8 or 12.)

$$nPC := target \mid \$r[15] := PC.$$

As another example, we represent a conditional branch by a guarded assignment to *nPC*. The `BNE` (branch not equal) instruction tests the processor's Z (zero) bit:

$$\neg Z \rightarrow nPC := target.$$

2.2 A Canonical Form of RTLs

To isolate the part of instruction semantics that is relevant to control flow, we put each RTL into a canonical form:

$$b_I \rightarrow nPC := target_I \mid annul := a_I \mid I_c.$$

[2]Readers who are familiar with the SPARC architecture must distinguish the *annul* status, which is part of the processor state, from the a bit found in the binary representations of most branch instructions. The interpretation of the *annul* status is trivial: it tells whether to execute an instruction. The interpretation of the a bit is more involved, because there are special rules for some instructions. We abstract away from these special rules by associating each instruction I with a Boolean expression a_I (not necessarily a single bit) that tells the processor whether to annul the instruction's successor.

We interpret this form as follows.

b_I is a Boolean-valued expression that tells whether I branches. It is an *expression*, not a constant or a field of the instruction. For a nonbranching instruction, b_I is **false**. For a call or an unconditional branch, b_I is **true**. For a conditional branch, b_I is some other expression.

$target_I$ is an expression that identifies the target address to which I may branch. (If b_I is **false**, $target_I$ is arbitrary.) For a call or a PC-relative branch, $target_I$ is a constant. For an indirect branch, $target_I$ is a more complex expression, for example, one that fetches an address stored in a register.

a_I is a Boolean-valued expression that tells whether I annuls its successor. Like b_I, it is an expression; it is not the value of the `a` bit in an instruction's representation. For example, the SPARC `BNE` instruction annuls its successor if the `a` bit is set and if the branch is not taken, so a_I is $\texttt{a} \neq 0 \wedge Z$.

I_c is an RTL that represents I's "computational effect." I_c may be empty, or it may contain guarded assignments that do not change *annul*, *nPC*, or *PC*. A typical RISC instruction changes control flow or performs computation, but not both, so I_c tends to be nonempty only when b_I and a_I are **false**.

Here are a few example RTLs in canonical form; SPARC assembly language appears on the left, RTLs on the right. The mnemonic `ba,a` stands for "branch always, annul;" **skip** is the empty RTL, that is, the empty list of effects.

```
add rs1, rs2, rd
    false → nPC := any | annul := false | $r[rd] := $r[rs1] + $r[rs2]
ba,a addr   true → nPC := addr | annul := true | skip
call addr   true → nPC := addr | annul := false | $r[15] := PC.
```

2.3 Instruction Decoding and Execution on Two Platforms

We represent instruction decoding using a **let**-binding notation:

let $(b_I \rightarrow nPC := target_I \mid annul := a_I \mid I_c) \equiv src[PC]$
in ...
end

The **let** construct binds b_I, $target_I$, a_I, and I_c, which together determine the semantics of the instruction I found in the source memory *src*. Perhaps unusually, these identifiers are bound to *syntax* (either expressions or RTLs), not to *values*. The **let**-binding represents not only the process of using the binary representation to identify the instruction and its operands, but also the mapping from that representation into a register-transfer list. This mapping can be done statically, at binary-translation time; it can even be automated based on a combination of machine descriptions [Ramsey and Fernández 1997; Ramsey and Davidson 1998].

An RTL alone does not suffice to specify the behavior of a machine when it executes an instruction; we also require an *execution loop*. The source-machine

execution loop decodes an instruction and executes it as follows.

```
fun loop() ≡
  let (b_I → nPC := target_I | annul := a_I | I_c) ≡ src[PC]
  in  if annul then
        [[PC := nPC | nPC := succ_s(nPC) | annul := false]]
      else if [[b_I]] then
        [[PC := nPC | nPC := target_I | I_c | annul := a_I]]
      else
        [[PC := nPC | nPC := succ_s(nPC) | I_c | annul := a_I]]
      fi
      ;loop()
  end
```

This execution loop is written using a simple imperative metalanguage. We use several notational shortcuts. The most important of these are the brackets $[\![\cdot]\!]$, which represent evaluation of syntax; for example, $[\![b_I]\!]$ is short for $eval_e(b_I)$, which produces the value of the branch condition (true or false), given the current state of the machine. The notation $[\![r]\!]$ is short for $eval_r(r)$, which changes the state of the machine. The function $succ_s$ (resp., $succ_t$) finds the successor instruction on the source (resp., target) machine. We specify the repeated execution of the processor loop as a tail call, rather than as a loop, because that simplifies the program transformations to follow.

Our example target, the Pentium, has neither delayed branches nor annuling, so it has a simpler canonical form and a simpler execution loop:

```
fun simple() ≡
  let (b_I → PC := target_I | I_c) ≡ tgt[PC]
  in  if [[b_I]] then
        [[PC := target_I | I_c]]
      else
        [[PC := succ_t(PC) | I_c]]
      fi
      ;simple()
  end
```

2.4 Strategy for Translating Delayed Branches

We want to translate a source-machine program into a target-machine program such that when the two programs are executed by their respective execution loops, the target program simulates the source program. It would suffice to create a translation that represented the source *PC*, *nPC*, and *annul* explicitly on the target machine, but such a translation would be very inefficient. For example, the representation of *nPC* would have to be updated in software after every execution of a translated instruction. A better idea is to make the values of the source *PC*, *nPC*, and *annul* implicit in the value of the target *PC*. How to do this is not immediately obvious. The contribution of this article is to use well-known program-transformation techniques to eliminate *nPC* and

annul wherever possible, so that (almost all of) *loop* can be expressed using only the *PC*. Our transformation leads to suitable changes in the sequence of instructions executed, thus guiding a transformation from *src* to *tgt*. This latter transformation is an algorithm for binary translation of delayed-branch instructions.

3. TRANSFORMING THE EXECUTION LOOP

Our translation function examines $src[pc_s]$ and produces suitable instructions at a corresponding target location $tgt[pc_t]$. We cannot simply have $pc_t = pc_s$, because source and target instruction sequences may be different sizes. During translation, we build *codemap* such that $pc_t = codemap(pc_s)$.

We assume that when the source processor starts executing code at $src[pc_s]$, it is not "in the middle" of a delayed or annuled branch. Formally, we write

$$annul = \textbf{false} \wedge nPC = succ_s(PC).$$

We call a state *stable* if it satisfies this predicate. The processor ABI (application binary interface) guarantees that the processor is in a stable state at a program's start location [Prentice-Hall 1993], and procedure calling conventions guarantee that the processor is in a stable state at the entry point of each procedure.

We begin our transformation by defining a function *stable* that can be substituted for *loop* whenever $annul = \textbf{false} \wedge nPC = succ_s(PC)$.

```
fun stable() ≡
  ⟦annul := false | nPC := succ_s(PC)⟧;
  let (b_I → nPC := target_I | annul := a_I | I_c) ≡ src[PC]
  in  if annul then
        ⟦PC := nPC | nPC := succ_s(nPC) | annul := false⟧
      else if ⟦b_I⟧ then
        ⟦PC := nPC | nPC := target_I | I_c | annul := a_I⟧
      else
        ⟦PC := nPC | nPC := succ_s(nPC) | I_c | annul := a_I⟧
      fi
      ;loop()
  end
```

Our companion technical report [Cifuentes and Ramsey 2002] lists the transformations used to get from this definition to something much like *simple*.

We do not show every step in the transformation of *stable*. The first transformations move the initial assignments inside the **let**, propagate (by forward substitution) the assignments to *annul* and *nPC*, move *loop*() inside the **if**, replace *loop*() with *stable*() where possible, and drop the (now dead) assignments. The next stage unfolds *loop* in the first and second arms of the **if** statement. In the second arm, *annul* is **true**, so the call to *loop*() can be replaced by $PC := nPC; nPC := succ_s(nPC); stable()$. Transformation proceeds by combining these two fragments, moving the **let**s together, and flattening the nested

if statements. We then use "The Trick" from partial evaluation [Danvy et al. 1996]: whenever $[\![a_I]\!]$ is free in a statement S, we replace S with **if** $[\![a_I]\!]$ **then** S **else** S **fi**. The Trick enables us to replace several calls to *loop* with calls to *stable*. The result is

fun *stable*() ≡
 let $(b_I \to nPC := target_I \mid annul := a_I \mid I_c) \equiv src[PC]$
  $(b_{I'} \to nPC := target_{I'} \mid annul := a_{I'} \mid {I'}_c) \equiv src[succ_s(PC)]$
 in **if** $[\![b_I]\!] \wedge [\![a_I]\!]$ **then**
  $[\![I_c]\!]$;
  $[\![PC := target_I]\!]$
  ;*stable*()
 else if $[\![b_I]\!] \wedge \neg[\![a_I]\!] \wedge [\![b_{I'}]\!] \wedge [\![a_{I'}]\!]$ **then**
  $[\![I_c]\!]$;
  $[\![{I'}_c]\!]$;
  $[\![PC := target_{I'}]\!]$
  ;*stable*()
 else if $[\![b_I]\!] \wedge \neg[\![a_I]\!] \wedge [\![b_{I'}]\!] \wedge \neg[\![a_{I'}]\!]$ **then**
  $[\![I_c]\!]$;
  $[\![PC := target_I \mid nPC := target_{I'} \mid {I'}_c \mid annul :=$ **false**$]\!]$
  ;*loop*()
 else if $[\![b_I]\!] \wedge \neg[\![a_I]\!] \wedge \neg[\![b_{I'}]\!] \wedge [\![a_{I'}]\!]$ **then**
  $[\![I_c]\!]$;
  $[\![{I'}_c]\!]$;
  $[\![PC := succ_s(target_I)]\!]$
  ;*stable*()
 else if $[\![b_I]\!] \wedge \neg[\![a_I]\!] \wedge \neg[\![b_{I'}]\!] \wedge \neg[\![a_{I'}]\!]$ **then**
  $[\![I_c]\!]$;
  $[\![PC := target_I \mid {I'}_c]\!]$
  ;*stable*()
 else if $\neg[\![b_I]\!] \wedge [\![a_I]\!]$ **then**
  $[\![PC := succ_s(succ_s(PC)) \mid I_c]\!]$
  ;*stable*()
 else if $\neg[\![b_I]\!] \wedge \neg[\![a_I]\!]$ **then**
  $[\![PC := succ_s(PC) \mid I_c]\!]$
  ;*stable*()
 fi
end

This version of *stable* suffices to guide the construction of a translator. We consider the cases in order.

—A branch that annuls the instruction in its delay slot acts like an ordinary branch on a machine without delayed branches.
—A branch that does not annul, but that has an annuling branch in its delay slot, acts as if the first branch never happened, and the second is a nondelaying branch.

—A nonannuling branch with another nonannuling branch in its delay slot is not trivial to translate; this is the one case in which we cannot substitute *stable* for *loop*. Interestingly, the MIPS architecture manual specifies that the machine's behavior in this case is undefined [Kane 1988, Appendix A]. This case requires potentially unbounded unfolding of *loop*, which we discuss in Section 5.

—A nonannuling branch with an annuling nonbranch in its delay slot acts as a branch to the successor of the target instruction (note that the SPARC architecture has an annulling nonbranch, namely `BN,A`.).

—A nonannuling branch with a nonannuling nonbranch in its delay slot has the effect of delaying the branch by one cycle. This is the common case.

—An annuling nonbranch skips over its successor.

—A nonannuling nonbranch (i.e., an ordinary computational instruction) simply executes and advances the program counter to its successor.

4. APPLICATION

We have applied our method to the University of Queensland binary translator, which supports both the SPARC and PA-RISC instruction sets [Cifuentes et al. 1999; Cifuentes and Van Emmerik 2000]. Here we show some results from the SPARC instruction set, and we sketch the derivation of a translator.

4.1 Classification of SPARC Instructions

The three properties of instructions that govern the translation of control flow are b_I (must branch, may branch, may not branch), a_I (must annul, may annul, may not annul), and $target_I$ (static target, dynamic target, no target). There are 15 reasonable combinations of these three properties. On the SPARC archictecture, only 9 combinations are used.[3]

Instruction	b_I	a_I	$target_I$	I_c	*Class*
`BA`	**true**	**false**	static	**skip**	*SD*
`BN`	**false**	**false**	N/A	**skip**	*NCT*
`Bcc`	$test_{cc}(icc)$	**false**	static	**skip**	*SCD*
`BA,A`	**true**	**true**	static	**skip**	*SU*
`BN,A`	**false**	**true**	N/A	**skip**	*SKIP*
`Bcc,A`	$test_{cc}(icc)$	$\neg test_{cc}(icc)$	static	**skip**	*SCDA*
`CALL`	**true**	**false**	static	$\$r[15] := PC$	*SD*
`JMPL`	**true**	**false**	dynamic	$\$r[rd] := PC$	*DD*
`RETT`	**true**	**false**	dynamic	⟨*restore state*⟩	*DD*
`TN`	**false**	**false**	N/A	**skip**	*NCT*
`Ticc`	$test_{cc}(icc)$	$test_{cc}(icc)$	dynamic	⟨*save state*⟩	*TRAP*
`TA`	**true**	**true**	dynamic	⟨*save state*⟩	*TRAP′*
NCT	**false**	**false**	N/A	varies	*NCT*

[3]For readers unfamiliar with SPARC assembly language, the mnemonics of the sample instructions include branch always (`BA`), branch never (`BN`), branch on condition code (`Bcc`), branches with annul (e.g., `BA,A`), call, jump, and link (`JMPL`), return from trap (`RETT`), trap never (`TN`), trap on integer condition code (`Ticc`), and trap always (`TA`).

These combinations enable us to classify instructions. We name eight of the nine classes as follows.

NCT	Non-control-transfer instructions (arithmetic, etc.)
DD	Dynamic delayed (unconditional)
SD	Static delayed (unconditional)
SCD	Static conditional delayed
SCDA	Static conditional delayed, annuling
SU	Static unconditional (not delayed)
SKIP	Skip successor (implement as static unconditional)
TRAP	Trap

4.2 Derivation of a Translator

Our translator builds a simulation relation between source-machine states and target-machine states. A translation is correct if execution on the target machine simulates execution on the source machine. We present only a sketch here; the details can be found in a technical report [Cifuentes and Ramsey 2002]. The state of the machine is the contents of the storage locations. The simulation relation correlates locations, and for the most part, one state simulates another if the correlated locations hold the same value. The exception is the program counter; PC_t simulates PC_s if $PC_t = codemap(PC_s)$. We build *codemap* incrementally during translation.

Our translator, *trans*, uses the following auxiliary procedures:

$emit(pc_t, I)$	Emit target-machine instructions I at pc_t.
$codemap(pc_s)$	Given a program counter on the source, return the corresponding program counter on the target.
$queueForTranslation(pc_s, pc_t)$	Add a pair to the work queue.
$newBlock()$	Return a pointer to a fresh basic block.

Deriving *trans* is tedious but straightforward. For each class of instructions, we use a_I and b_I to simplify *stable*. If necessary, we also consider $a_{I'}$ and $b_{I'}$, where I' is the instruction in the delay slot. We transform the simplified *stable* as needed until it suggests an obvious translation, and finally we emit target-machine instructions. We show only a few representative cases; our technical report has more. Table I shows example SPARC and Pentium assembly language for each. These examples are shown unoptimized; in some cases, such as `be` followed by `mov`, it is possible to put I' before I, eliminating significant overhead. This restructuring is not possible in the general case, however, and particular cases may best be left to a general-purpose optimizer.

The easiest cases are ones in which as and bs are known statically. For non-control-transfer instructions, $b_I \equiv$ **false** and $a_I \equiv$ **false**, which corresponds to the last arm of *stable*, and the translation is

$$\langle \textit{cases for translation of } I \rangle \equiv$$
$$\mid NCT \Longrightarrow pc_t := emit(pc_t, \overline{I_c}); trans(succ_s(pc_s), pc_t).$$

Table I. Example Translations from SPARC to Pentium[a]

Class(I)	*Class*(I')	SPARC Instructions	Pentium Instructions

NCT	any	`add %i1, %i2, %i3`	`mov eax, SPARCI1`
			`add eax, SPARCI2`
			`mov SPARCI3, eax`

SU	any	`ba,a L`	`jmp L`

SCD	*NCT*	`ba L`
		`mov %o1, %o2`
		⋮
			`nop`
			`je BB`
			
			`mov eax, SPARCO1`
			`mov SPARCO2, eax`
			⋮

			`BB: mov eax, SPARCO1`
			`mov SPARCO2, eax`
			`jmp L`

SD	*SD*	`ba L1`
		`ba L2`
		`mov 3, %o0`
		⋮
		`L1: mov 2, %o0`
		⋮
		`L2: ...`
			`nop`
			`nop`
			`mov eax, 2`
			
			`jmp L2`

[a]SPARC assembly language puts the destination on the right, but Pentium assembly language puts the destination on the left. The SPARC architecture has more registers than the Pentium, so we map its registers onto memory locations. For example, SPARC register `%i1` becomes memory location `SPARCI1`. The translation of the *SCD*/*NCT* combination uses two basic blocks, which are separated by a horizontal line in the rightmost column.

A more interesting example, which involves dynamic conditions, is the *SCD* (static conditional delayed) class, in which b_I is dynamic and a_I is **false**. The translation depends on what is in the delay slot.

$\langle$*cases for translation of* $I\rangle + \equiv$
  | $SCD \Longrightarrow$
    let $(b_{I'} \rightarrow nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) \equiv src[succ_s(pc_s)]$
    in case *class*(I') **of**
      $\langle$*translation cases for class*(I')*, where class(I) = SCD*$\rangle$
    end

The most common delay instruction is a non-control-transfer instruction (class *NCT*), where $b_{I'} =$ **false** and $a_{I'} =$ **false**. In this case, *stable* reduces to

$\langle$*specialization of stable for SCD with NCT in the delay slot*$\rangle \equiv$
  if $[\![b_I]\!]$ **then**
    $[\![I_c]\!]; [\![PC := target_I \mid I'_c]\!]; stable()$
  else
    $[\![I_c \mid PC := succ_s(PC)]\!]; stable()$
  fi

We rewrite parallel to sequential composition. Because I_c never affects PC, we can move I_c outside the **if**. In general, no single target instruction implements

$[\![PC := target_I \mid I'_c]\!]$, so we rewrite it into the sequence $[\![I'_c]\!]; [\![PC := target_I]\!]$, and we put this sequence into a new "trampoline" basic block *bb*. Then *stable* becomes

$\langle$*final specialization of stable for SCD with NCT in the delay slot*$\rangle \equiv$
  $[\![I_c]\!]$;
  if $[\![b_I]\!]$ **then**
    $[\![PC := bb]\!]$;
  else
    $[\![PC := succ_s(PC)]\!]$;
  fi
  ; *stable*()

which we translate using an ordinary branch instruction:

$\langle$*translation cases for class*(I')*, where class(I) = SCD*$\rangle \equiv$
  $\mid NCT \Longrightarrow$
    local $bb := newBlock()$;
    $pc_t := emit(pc_t, \overline{I_c})$;
    $pc_t := emit(pc_t, \overline{b_I} \rightarrow PC := bb)$;
    $bb := emit(bb, \overline{I'_c})$;
    $bb := emit(bb, PC := codemap(target_I))$;
    $queueForTranslation(target_I, codemap(target_I))$;
    $trans(succ_s(pc_s), pc_t)$;

The most difficult cases arise when the instruction in the delay slot of a branch I is another delayed-branch instruction I'. We show just one of the many cases: putting an unconditional *SD* branch in the delay slot of another unconditional *SD* branch. This trick makes it possible to execute a single nonbranching *NCT* instruction "out of line." To make the case truly useful, the target of the first branch should be computed dynamically, but this change would complicate the exposition significantly.

When *SD* is in *SD*'s delay slot, $b_{I'} =$ **true** and $a_{I'} =$ **false**, and *stable* reduces to the following code.

$\langle$*specialization of stable for SD with SD in the delay slot*$\rangle \equiv$
  $[\![I_c]\!]$;
  $[\![PC := target_I \mid nPC := target_{I'} \mid I'_c \mid annul :=$ **false**$]\!]$
  ; *loop*()

We unfold the call to *loop* and substitute forward for *PC*, *nPC*, and *annul*. We remove dead assignments, move *loop*() inside the conditional, convert it to *stable*() in one branch, unfold, and so on. We wind up with four cases based on the values of $[\![a_{I''}]\!]$ and $[\![b_{I''}]\!]$, where I'' is the instruction at $target_I$. Because I is a static branch, the value of $[\![target_I]\!]$ is independent of the state of the machine, so we can find I'' and the expressions $[\![a_{I''}]\!]$ and $[\![b_{I''}]\!]$ statically. The simplest case is one in which I'' never branches (*NCT*), that is, where $b_{I''} =$ **false** and

$a_{I''} =$ **false**. This case reduces to

$\langle$*specialization of stable for SD with SD in the delay slot (class(I'') = NCT)*$\rangle \equiv$
$[\![I_c]\!];$
$[\![I'_c]\!];$
$[\![PC := target_{I'} \mid I''_c]\!]$
$;stable()$

As before, because I' is static, we can rewrite $[\![PC := target_{I'} \mid I''_c]\!]$ as the sequence $[\![I''_c]\!];[\![PC := target_{I'}]\!]$, and we read off the following translation.

$\langle$*translation cases for class(I''), where class(I) = SD and class(I') = SD*$\rangle \equiv$
$\mid NCT \Longrightarrow$
$pc_t := emit(pc_t, \overline{I_c});$
$pc_t := emit(pc_t, \overline{I'_c});$
$pc_t := emit(pc_t, \overline{I''_c});$
$pc_t := emit(pc_t, PC := codemap(target_{I'}));$
$queueForTranslation(target_{I'}, codemap(target_{I'}));$

Since on the SPARC, I_c and I'_c are no-ops, the translation executes the effect of I'', then branches to $target_{I'}$, as shown in the last example in Table I. The instruction in the delay slot of I' (`mov 3, %o0` in Table I) is not executed.

5. EXPERIENCE

We have used translators for delayed branches in two tools: a binary translator [Cifuentes et al. 1999; Cifuentes and Van Emmerik 2000; Cifuentes et al. 2002] and a decompiler [Cifuentes et al. 1998]. In both tools, we translate machine instructions into a low-level, machine-independent intermediate form *without* delayed branches. The binary translator uses this form to generate target code, applying standard optimization techniques. The decompiler analyzes the intermediate form to recover high-level information such as structured control flow. The translation into intermediate form includes simple optimizations not mentioned above.

Our first attempt at translating delayed branches was based on a case analysis of the SPARC's architecture manual. The analysis did not cover all cases, as there were many combinations whose meaning was not clear from a direct reading of the manual. It was difficult even to characterize the set of binary codes that could be analyzed. These difficulties motivated the work presented here.

All the transformations discussed in this article were done by hand. We investigated tools that might have helped us transform *stable*, but we were left with the impression that this is still a research problem [Shankar 1996], and it was easy enough to transform *stable* by hand. By contrast, it would be very useful to automate the derivation of the translator from *stable* and the discovery of the translations of the a_Is, b_Is, and I_cs. This work is not intellectually demanding, but it is tedious because there are many cases.

As presented in this article, a branch in a delay slot requires a recursive call to *loop*, not to *stable*. Most cases, including all those shown in the SPARC manual, can be handled by an additional unfolding of *loop*, which we have done in our implementation. The unfolding game can go on indefinitely; no

matter how many times we unfold *loop*, a single recursive call to *loop* remains, and it is always possible to write a program whose interpretation reaches this recursive call. Because a program that does this indefinitely is not useful (it does nothing but jump from one branch to another, never executing a computational instruction), we do not unfold beyond what is shown in this article. This level of unfolding handles the case of two branch instructions I_1 and I_2, where I_2 is in I_1's delay slot. If the target of I_1 is also a branch instruction, our system currently rejects the code. We have not decided whether it will eventually fall back on an interpreter, or whether we will develop a fallback translation algorithm to which both *nPC* and *PC* are parameters.

ACKNOWLEDGMENTS

We thank Jack Davidson for helpful discussions, Mike Van Emmerik for the implementation of the translator, and Mitch Wand for help with semantics. We also thank Benjamin Pierce for his guidance in preparing this manuscript for publication.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Reading, MA.

BARBACCI, M. R. AND SIEWIOREK, D. P. 1982. *The Design and Analysis of Instruction Set Processors*. McGraw-Hill, New York.

BELL, C. G. AND NEWELL, A. 1971. *Computer Structures: Readings and Examples*. McGraw-Hill, New York.

CIFUENTES, C. AND RAMSEY, N. 2002. A transformational approach to binary translation of delayed branches with applications to SPARC and PA-RISC instruction sets. Tech. Rep. TR-2002-104, Sun Microsystems Laboratories, Palo Alto, CA. January.

CIFUENTES, C., SIMON, D., AND FRABOULET, A. 1998. Assembly to high-level language translation. In *Proceedings of the International Conference on Software Maintenance*. IEEE-CS Press, Los Alamitos, CA, 228–237.

CIFUENTES, C. AND VAN EMMERIK, M. 2000. UQBT: Adaptable binary translation at low cost. *IEEE Computer 33*, 3 (March), 60–66.

CIFUENTES, C., VAN EMMERIK, M., AND RAMSEY, N. 1999. The design of a resourceable and retargetable binary translator. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'99)*, IEEE CS Press, Los Alamitos, CA, 280–291.

CIFUENTES, C., VAN EMMERIK, M., RAMSEY, N., AND LEWIS, B. 2002. Experience in the design, implementation, and use of a retargetable static binary translation framework. Tech. Rep. TR-2002-105, Sun Microsystems Laboratories, Palo Alto, CA. January.

DANVY, O., MALMKJÆR, K., AND PALSBERG, J. 1996. Eta-expansion does The Trick. *ACM Trans. Program. Lang. Syst. 18*, 6 (Nov.), 730–751.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, International Series in Computer Science, Englewood Cliffs, NJ.

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

LARUS, J. R. AND BALL, T. 1994. Rewriting executable files to measure program behavior. *Softw. Pract. Exper. 24*, 2 (Feb.), 197–218.

PRENTICE-HALL 1993. *System V Application Binary Interface, SPARC Architecture Processor Supplement*, third ed. Prentice-Hall, Englewood Cliffs, NJ.

RAMSEY, N. AND DAVIDSON, J. W. 1998. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*

(LCTES'98). Lecture Notes in Computer Science, vol. 1474, Springer Verlag, New York, 172–188.

RAMSEY, N. AND FERNÁNDEZ, M. F. 1997. Specifying representations of machine instructions. *ACM Trans. Program. Lang. Syst. 19*, 3 (May), 492–524.

RONSSE, M. AND DE BOSSCHERE, K. 2001. JiTI: A robust just in time instrumentation technique. *2000 Workshop on Binary Translation,* in *Comput. Arch. News 29*, 1 (March), 43–54.

SHANKAR, N. 1996. Steps towards mechanizing program transformations using PVS. *Sci. Comput. Program. 26*, 1–3 (May), 33–57.

SITES, R. L., CHERNOFF, A., KIRK, M. B., MARKS, M. P., AND ROBINSON, S. G. 1993. Binary translation. *Commun. ACM 36*, 2 (Feb.), 69–81.

SPARC INTERNATIONAL 1992. *The SPARC Architecture Manual, Version 8*. SPARC International, Englewood Cliffs, NJ.

Received April 2001; revised July 2002; accepted September 2002