A Transformational Approach to Binary Translation of Delayed Branches

Norman Ramsey

Department of Computer Science
University of Virginia, Charlottesville, VA 22903 USA
`nr@cs.virginia.edu`

Cristina Cifuentes

Department of Computer Science and Electrical Engineering
University of Queensland, Brisbane, QLD 4072 Australia
`cristina@csee.uq.edu.au`

Abstract

The fundamental steps in binary translation are distinguishing code from data, mapping data from source to target, and translating instructions. Translating instructions presents few problems, except when the source instruction set has features not present in typical compiler intermediate codes. The most common such feature is the delayed branch.

Standard code-generation technology can handle delayed branches in the target language, but not in the source. Translating delayed branches can involve tricky case analyses to figure out what happens if there is a branch instruction in a delay slot. This paper presents a disciplined method for deriving such case analyses. The method identifies problematic cases, shows the translations for the non-problematic cases, and gives confidence that all cases are considered. The method also applies to other tools that analyze machine instructions.

We begin by writing a very simple interpreter for the source machine. It specifies, at the register-transfer level, how the source machine executes instructions, including delayed branches. 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.

We show the translation is correct by using a correspondence between source and target states, and showing if the source and target machines begin execution in corresponding states, they reach new corresponding states in a few instructions.

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. Hardware vendors can use binary translation to provide a tempting array of software along with new machines. Hardware buyers can use binary translation to run old code on new platforms. This ability is particularly valuable when the old code is available only in binary form, e.g., when it has been purchased from a third party or its source code has been lost. Finally, binary translation is also an enabling technology for efficient simulation (Cmelik and Keppel 1994; Witchel and Rosenblum 1996).

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. Data must be translated differently from code, pointers must be translated differently from non-pointers, and code pointers (e.g., for indirect branches) must be translated differently from data pointers (Sites *et al.* 1993; Larus and Ball 1994). Von Neumann machines use a single representation for both code and data, and standard binary-file formats make it difficult to distinguish code from data, so in general, some code may not be identifiable until run time, and its translation (or interpretation) must be delayed until then. This paper addresses the problem of translating the instructions once they have been identified.

When a mapping from source data locations to target data locations has been established, translating instructions is mostly straightforward. Finding the target instructions needed to achieve a particular effect is simply code generation. It is not always obvious, however, what is the effect of a delayed-branch instruction, especially when a branch instruction appears in a delay slot. The contribution of this paper is a disciplined method for understanding the effects of delayed branches, even in tricky, rarely used combinations. This method identifies cases that are problematic for translation, shows the translations for the non-problematic cases, and gives confidence that all cases are considered. The results of the method are being applied to a binary translator, and they could profitably be applied to other tools that analyze machine instructions, including optimizers (Srivastava and Wall 1993), code instrumentors (Wall 1992; Larus and Ball 1994; Larus and Schnarr 1995), fault isolators (Wahbe *et al.* 1993), and decompilers (Cifuentes and Gough 1995; Hoffman 1997; Cifuentes, Simon, and Fraboulet 1998).

Our method uses register-transfer lists (RTLs) as a semantic framework in which 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.

We build a binary translator by considering semantics for two machines. Each has an execution loop and a set of instructions. We begin by transforming the source machine's execution loop into the target machine's execution loop. To maintain the proper semantics for a program, we simultaneously transform the sequence of source-machine instructions into a sequence of target-machine instructions. This transformation of the sequence of instructions becomes our algorithm for binary translation.

A quick reading of this paper 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 non-delayed branch and push the instruction in the delay slot along zero, one, or both sucessor edges? (The set of successors that should get copies of the instruction in the delay slot depends on whether the delayed branch "annuls" that instruction.) This simple approach is in fact correct, *except* when the instruction in the delay slot is itself a delayed branch. In that case, the "pushing" approach fails to execute the instruction that is the target of the first branch. The methods in this paper translate this case correctly. In practice, such cases occur rarely in user code, but they are recommended in kernel code as a way of returning from interrupts or otherwise switching contexts (SPARC 1992, §B.26).

The building blocks of this paper 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, Sethi, and Ullman 1986) and partial evaluation (Jones, Gomard, and Sestoft 1993). What is new is the application of these techniques to a new problem domain and their use in building a SPARC front end for a binary translator.

2 Semantic framework

Rather than translate source-machine instructions directly into target-machine instructions, we translate source instructions into register transfer lists (RTLs), transform the RTLs, optimize the RTLs, and translate the RTLs into target-machine instructions. RTLs provide a uniform framework that can express source instructions, target instructions, and their interpretations by the source and target processors.

2.1 Register transfer lists

Our RTL formalism is designed for use in tools and component generators, and it makes machine-dependent computation explicit (Ramsey and Davidson 1998). For this paper, we use a simplified version specified using an imperative syntax:

$\mathit{rtl} \Rightarrow [\mathit{effect}\ \{\mid\ \mathit{effect}\}]$	Multiple assignment
$\mathit{effect} \Rightarrow [\mathit{exp} \rightarrow]\ \mathit{location} := \mathit{exp}$	Guarded assignment
$\mathit{exp} \Rightarrow \mathit{constant}$	Constant
$\mid\ \mathit{location}$	Fetch from a location
$\mid\ \mathit{exp}\ \mathit{binop}\ \mathit{exp}$	Binary RTL operator
$\mid\ \mathit{operator}\ (\ \mathit{explist}\ )$	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] i.e., a store operation. The transfer takes place only if the guard (an expression) evaluates to **true**. Effects in a list take place simultaneously, as in Dijkstra's multiple-assignment statement; an RTL represents a single change of state.

Values are computed by expressions without side effects. Eliminating side effects simplifies analysis and transformation. Expressions may be integer constants, fetches from locations, or applications of *RTL operators* to lists of expressions. RTL operators are pure functions on values.

For purposes of this paper, we assume that locations are single cells in a mutable store, although the RTL formalism supports a more general view that makes byte order explicit.

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
```

The effect of this load instruction might be written

$\langle \mathit{RTL\ for\ sample\ instruction} \rangle \equiv$

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

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

because the stack pointer is register 14 and register `%i0` is register 24. The notation $*space*`[`*address*`]` specifies a cell in a mutable store. 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.

The load instruction not only loads a value into register 24; it also advances the program counter to point to the next instruction. Changing the program counter is intimately connected with branching; we separate the effect on the program counter in order to give it special treatment.

2.2 Processor state for delayed branches

A processor executing straight-line code executes one instruction after another, in sequence. A delayed-branch instruction causes the processor to depart from that sequence, but not immediately. When the processor executes 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 machines, like the SPARC, the instruction I can "annul" its successor, in which case the successor is *not* executed, but 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 current instruction.
annul	is the "annul status," which determines whether the processor executes the instruction at PC or ignores it.[2]

In this model, a delayed control transfer 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:

$\langle$*RTL for call*$\rangle\equiv$
  $nPC := \mathit{target} \mid \$r[15] := PC$

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

The *target* address in the semantics is distinct from the `target` field in the binary representation of the call instruction. In the case of the SPARC, we abstract away from the rule that says the target address is computed by extending the `target` field on the right with zeroes.

A call transfers control unconditionally; we represent a conditional branch by a guarded assignment to nPC. The `BNE` (branch not equal) instruction tests the Z (zero) bit in the condition codes:

$\langle$*rtl for conditional branch* `BNE`$\rangle\equiv$
  $\neg Z \rightarrow nPC := target$

Again we abstract the computation of the target address relative to the location of the instruction.

2.3 A canonical form of RTLs

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

$\langle$*RTL for generic instruction I*$\rangle\equiv$
  $b_I \rightarrow nPC := target_I \mid annul := a_I \mid I_c$

We interpret this form as follows:

b_I is a predicate that tells whether I branches. It is an *expression*, not a constant or a field of the instruction. For non-branching instructions, b_I is **false**. For calls and unconditional branches, b_I is **true**. For conditional branches, b_I is some other expression, the value of which depends on the state of the machine (e.g., on the values of the condition codes).

$target_I$ is an expression that identifies the target address to which I may branch. (If b_I is **false**, $target_I$ is arbitrary.) For calls and PC-relative branches, $target_I$ is a constant that statically identifies a target address. For indirect branches, $target_I$ may be a more complex expression, e.g., one that fetches an address stored in a register.

a_I is a predicate that tells whether I annuls its successor. It is an expression, not the value of the `a` bit in an instruction's representation. For most instructions, a_I is **false**. For conditional branches, a_I may be more complicated. 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. Typical RISC instructions change control flow or perform computation, but not both, so I_c tends to be non-empty only when b_I and a_I are **false**. On CISC architectures, however, an instruction like "decrement and skip if zero" might have both non-empty I_c (the decrement) and a nontrivial b_I (the test for zero).

An instruction can be expressed in this canonical form if, when executed, it branches to at most one *target*.[3] This is true of all instructions on all architectures with which we are familiar, including indirect-branch instructions (although the value of *target* may be different on different executions of an indirect branch).

Here are a few example RTLs in canonical form; SPARC assembly language appears on the left, RTLs on the right. **skip** is the empty RTL.

`add rs1, rs2, rd`

$\textbf{false} \rightarrow nPC := any \mid annul := \textbf{false} \mid \$r[\texttt{rd}] := \$r[\texttt{rs1}] + \$r[\texttt{rs2}]$

`ba,a addr` $\textbf{true} \rightarrow nPC := \texttt{addr} \mid annul := \textbf{true} \mid \textbf{skip}$

`call addr` $\textbf{true} \rightarrow nPC := \texttt{addr} \mid annul := \textbf{false} \mid \$r[15] := PC$

2.4 Instruction decoding and execution on two platforms

Given this canonical form for instructions, we represent instruction decoding using a **let**-binding notation:

```
⟨instruction decoding⟩≡
  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*. This **let**-binding represents not only the process of using the binary representation to identify the instruction and its operands, but also the abstraction from that representation into the RTL semantics. This abstraction from binary representation to semantics 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).

[3] And of course if the machine uses delayed branches.

The source-machine execution loop decodes an instruction and executes it as follows:

```
$\langle$*sparc execution loop*$\rangle\equiv$
  **fun** $loop() \equiv$
    **let** $(b_I \to nPC := target_I \mid annul := a_I \mid I_c) \equiv src[PC]$
    **in** **if** $annul$ **then**
           $PC := nPC \mid nPC := succ_s(nPC) \mid annul :=$ **false**
        **else if** $[\![b_I]\!]$ **then**
           $PC := nPC \mid nPC := [\![target_I]\!] \mid [\![I_c]\!] \mid annul := [\![a_I]\!]$
        **else**
           $PC := nPC \mid nPC := succ_s(nPC) \mid [\![I_c]\!] \mid annul := [\![a_I]\!]$
        **fi**
        ; $loop()$
    **end**
```

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.

The notation $[\![\bullet]\!]$ represents execution; for example, $[\![b_I]\!]$ is the value of the branch condition, given the current state of the machine. Executing the computational effect $[\![I_c]\!]$ changes the state of the machine.

The function $succ_s$ abstracts over the details of identifying the successor instruction on the source machine; $succ_t$ finds the successor on the target machine. In both cases, *succ* is computed as part of instruction decoding.

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

```
$\langle$*Pentium execution loop*$\rangle\equiv$
  **fun** $simple() \equiv$
    **let** $(b_I \to PC := target_I \mid I_c) \equiv tgt[PC]$
    **in** **if** $[\![b_I]\!]$ **then**
          $PC := [\![target_I]\!] \mid [\![I_c]\!]$
        **else**
          $PC := succ_t(PC) \mid [\![I_c]\!]$
        **fi**
        ; $simple()$
    **end**
```

2.5 Strategy for translating delayed branches

Both our formalism and the SPARC architecture manual give a clear semantics of delayed branches in terms of PC, nPC, and *annul*. To get an efficient target program, however, we wish *not* to represent the source PC, nPC, and *annul* explicitly, but to make all three *implicit* in the value of the target PC. How to do this based on the information in the architecture manual is not immediately obvious, but our semantic framework enables a new technique. We transform *loop*, eliminating nPC and *annul* wherever possible, so that (almost all of) *loop* can be expressed using only the PC.

3 Transforming the execution loop

We wish to develop a translation function that we can point at a location $src[pc_s]$ and that will produce suitable instructions at a corresponding target location $tgt[pc_t]$. We cannot simply have $pc_t = pc_s$; source program counters cannot be identical to target program counters, because source and target instruction sequences may be different sizes. During translation, we build *codemap*, a map that relates program counters on the two machines, so $pc_t = codemap(pc_s)$.

We assume that when the source processor starts executing code at $src[pc_s]$, it is not in the middle of a delayed or annulled branch, or formally,

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

Software conventions guarantee that the processor will be in such a state at a program's start location and at procedure entry points.

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

```
⟨stable execution loop⟩≡
  fun $stable() \equiv$
    $annul := \mathbf{false} \mid nPC := succ_s(PC)$;
    let $(b_I \rightarrow nPC := target_I \mid annul := a_I \mid I_c) \equiv src[PC]$
    in  if $annul$ then
          $PC := nPC \mid nPC := succ_s(nPC) \mid annul := \mathbf{false}$
        else if $[\![b_I]\!]$ then
          $PC := nPC \mid nPC := [\![target_I]\!] \mid [\![I_c]\!] \mid annul := [\![a_I]\!]$
        else
          $PC := nPC \mid nPC := succ_s(nPC) \mid [\![I_c]\!] \mid annul := [\![a_I]\!]$
        fi
        ; $loop()$
    end
```

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 result is:

```
⟨stable execution loop⟩+≡
  fun $stable() \equiv$
    let $(b_I \rightarrow nPC := target_I \mid annul := a_I \mid I_c) \equiv src[PC]$
    in  if $[\![b_I]\!]$ then
          $PC := succ_s(PC) \mid nPC := [\![target_I]\!] \mid [\![I_c]\!] \mid annul := [\![a_I]\!]$
          ; $loop()$
        else if $[\![a_I]\!]$ then
          $PC := succ_s(PC) \mid nPC := succ_s(succ_s(PC)) \mid [\![I_c]\!] \mid annul := \mathbf{true}$
          ; $loop()$
        else
          $PC := succ_s(PC) \mid [\![I_c]\!]$
          ; $stable()$
        fi
    end
```

The last arm of the **if** shows the execution of an instruction that never branches or annuls. It corresponds to the execution of a similar instruction on the *simple* target.

The next step is to unfold *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 $PC := nPC; nPC := succ_s(nPC); stable()$. The definition of *stable* reduces to

```
⟨stable execution loop⟩+≡
  fun $stable() \equiv$
    let $(b_I \rightarrow nPC := target_I \mid annul := a_I \mid I_c) \equiv src[PC]$
    in  if $[\![b_I]\!]$ then
          ⟨case where I branches⟩
        else if $[\![a_I]\!]$ then
          $PC := succ_s(succ_s(PC)) \mid [\![I_c]\!]$
          ; $stable()$
        else
          $PC := succ_s(PC) \mid [\![I_c]\!]$
          ; $stable()$
        fi
    end
```

where the interesting case is

```
⟨case where I branches⟩≡
  $PC := succ_s(PC) \mid nPC := [\![target_I]\!] \mid [\![I_c]\!] \mid annul := [\![a_I]\!]$;
  let $(b_{I'} \rightarrow nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) \equiv src[PC]$
  in  if $annul$ then
        $PC := nPC \mid nPC := succ_s(nPC) \mid annul :=$ false
      else if $[\![b_{I'}]\!]$ then
        $PC := nPC \mid nPC := [\![target_{I'}]\!] \mid [\![I'_c]\!] \mid annul := [\![a_{I'}]\!]$
      else
        $PC := nPC \mid nPC := succ_s(nPC) \mid [\![I'_c]\!] \mid annul := [\![a_{I'}]\!]$
      fi
      ; $loop()$
  end
```

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, Malmkjær, and Palsberg 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

```
⟨stable execution loop⟩+≡
  fun stable() ≡
    let $(b_I \rightarrow nPC := target_I \mid annul := a_I \mid I_c) \equiv src[PC]$
        $(b_{I'} \rightarrow nPC := target_{I'} \mid annul := a_{I'} \mid {I'}_c) \equiv src[succ_s(PC)]$
    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. Considering the cases in order,

- A branch that annuls the instruction in its delay slot acts just 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 non-delaying branch.
- A non-annuling branch with another non-annuling 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 is discussed in Section 6.
- A non-annuling branch with an annuling non-branch in its delay slot acts as a branch to the successor of the target instruction. (Note that the SPARC has an annulling non-branch, viz, `BN,A`.)
- A non-annuling branch with a non-annuling non-branch in its delay slot has the effect of delaying the branch by one cycle. This is the common case.
- An annuling non-branch skips over its successor.
- A non-annuling non-branch (i.e., an ordinary computational instruction) simply executes and advances the program counter to its successor.

We now apply this analysis to the SPARC.

4 Application to the SPARC instruction set

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, only 9 are used:

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	$\langle restore\ state \rangle$	*DD*
`TN`	**false**	**false**	N/A	**skip**	*NCT*
`Ticc`	$test_{cc}(icc)$	$test_{cc}(icc)$	dynamic	$\langle save\ state \rangle$	*TRAP*
`TA`	**true**	**true**	dynamic	$\langle save\ state \rangle$	*TRAP'*
NCT	**false**	**false**	N/A	varies	*NCT*

We name 8 of the 9 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, annulling
SU	Static unconditional (not delayed)
SKIP	Skip successor (implement as static unconditional)
TRAP	Trap

Our treatment of trap instructions may be surprising, since the architecture manual presents them as instructions that set both PC and nPC. Because nPC is always set to $PC + 4$ (SPARC 1992, §C.8), we can model this behavior as setting *nPC* to the address of the trap handler and setting *annul* to **true**. Our model introduces a stall before the trap is taken, but no interesting state changes during a stall, so there is no problem. For simplicity, we put the unconditional trap (*TRAP′*) in the same class as the conditional traps (*TRAP*). We can't do this with the branch instructions because of `BA,A`'s anomalous treatment of the `a` bit.

The table exposes a useful property of the SPARC instruction set; a_I is not arbitrary, but is always given by one of these four possibilities:

$a_I \equiv \textbf{false}$	Never annul.
$a_I \equiv \textbf{true}$	Always annul.
$a_I \equiv b_I$	Annul if branch taken.
$a_I \equiv \neg b_I$	Annul if branch not taken.

Whenever processor designers use this scheme, a_I can be eliminated at binary-translation time. A more general a_I would require a second test in the translated code.

4.2 Derivation of a translator

Correctness

To say what it means to have a correct translation, we reason about states, about values of expressions in states, and about state transitions. For notation, if a machine is in a state σ, we write $\mathcal{E}[\![e]\!]\sigma$ for the value of expression e in state σ; if executing instruction I causes a machine to make a transition from a state σ to a new state σ', we write $\sigma' = \mathcal{C}[\![I]\!]\sigma$, so $\mathcal{C}[\![I]\!]$ stands for the state-changing effect of I.

A translation is correct if execution on the target machine simulates execution on the source machine. The translator builds a map $\overline{\bullet}$ from source-machine states to target-machine states.[4] In a way made precise below, this map respects the operation of the machine. In our design, $\overline{\bullet}$ is *partial*—it is not defined when the source machine is "about to" execute a delayed branch or annulled instruction. To be precise, $\overline{\sigma}$ is defined if $\mathcal{E}[\![\neg annul \wedge nPC = succ_s(PC)]\!]\sigma$.

[4] Technically, the translator establishes not a map but a relation, because more than one target-machine state can be used to simulate a particular source-machine state. We nevertheless use the $\overline{\bullet}$ notation because it seems more intuitive. When we write $\overline{{}^s\sigma}$, we really mean "any state ${}^t\sigma$ such that ${}^t\sigma$ and ${}^s\sigma$ stand in a weak bisimulation relation."

The target machine is said to *simulate* the source machine if the following condition holds: if we start the source machine in a state ${}^s\sigma_1$, and the *loop* function takes it through a sequence of states ${}^s\sigma_1, {}^s\sigma_2, \ldots$, then there is a subsequence of such states ${}^s\sigma_{k_1}, {}^s\sigma_{k_2}, \ldots$ such that $\overline{{}^s\sigma_{k_1}}, \overline{{}^s\sigma_{k_2}}, \ldots$ is a subsequence of the states that the target machine goes through when started in state ${}^t\sigma_1 = \overline{{}^s\sigma_1}$. Informally, although the target machine may go through some intermediate states that don't correspond to any execution of the source, and though the source machine may go through some intermediate states that don't correspond to any execution of the target, when we remove those intermediate states, what's left of the executions correspond one to one.[5] We sketch a proof in Section 5.

Translations of expressions and computational effects

In the RTL framework, the state of the machine is the contents of all the storage locations. In a naïve translator, $\overline{\bullet}$ can mostly map locations to locations, without changing values. The exception is the program counter; its translation must use *codemap*, so $\mathcal{E}[\![PC]\!]\overline{\sigma} = \mathit{codemap}(\mathcal{E}[\![PC]\!]\sigma)$. Given a map $\overline{\bullet}$ on locations, we can easily extend it to expressions like a_I, b_I, and target_I. If e is an expression, then $\mathcal{E}[\![\overline{e}]\!]\overline{\sigma} = \mathcal{E}[\![e]\!]\sigma$.

We assume that translations can be found for the computational effects I_c, which do not affect PC, nPC, or *annul*. We require only that $\mathcal{C}[\![\overline{I_c}]\!]\overline{\sigma} = \overline{\mathcal{C}[\![I_c]\!]\sigma}$. In general, $\overline{I_c}$ will be a sequence of instructions, not exactly one instruction. We also assume that, given any condition b and address *target*, we can construct an instruction sequence implementing $b \rightarrow PC := \mathit{target}$ on the target machine.

Under these assumptions, we analyze source branch conditions b_I, annulment conditions a_I, and target addresses target_I, and we show how to construct branch conditions and target addresses for the target machine. In the process, we build the *codemap* function that takes source program counters to target program counters.

[5] In the terminology of Milner (1990), the transitions to these intermediate states are "silent."

Structure of the translator

Our translator works with one basic block at a time. *codemap* must be built incrementally, by the translator itself, because the only way to know the size of the target basic blocks is to translate the source basic blocks. The translator maintains a work queue of untranslated blocks, each of which is represented by a (pc_s, pc_t) pair. pc_s is the address of some code on the source machine. pc_t may be the corresponding target-machine address, or more likely a placeholder for a target-machine address, to be filled in later. (For example, pc_t might be a pointer to a basic block in a control-flow graph.) *codemap* contains pairs that have already been translated. We use the following auxiliary procedures:

$queueForTranslation(pc_s, pc_t)$	Add a pair to the work queue.
$codemap(pc_s)$	If a pair (pc_s, pc_t) is in *codemap*, return pc_t. Otherwise, let pc_t be a fresh placeholder, add (pc_s, pc_t) to *codemap*, and return pc_t. (We use *codemap* both as a function and as a collection of ordered pairs, but these usages are equivalent.)
$emit(pc_t, I)$	Emit target-machine instructions I at pc_t, returning a pointer to the location following the instructions. If I is a sequence of n instructions, $emit(pc_t, I)$ returns the result of applying $succ_t$ to pc_t, n times.
$newBlock()$	Return a pointer to a fresh placeholder.

Placeholders created with *codemap* correspond to basic blocks in the source program. Placeholders created with *newBlock* are artifacts of translation.

The translator loops, removing pairs from the work queue, and calling *trans* if those pairs have not already been translated. *trans* translates individual basic blocks. If an instruction branches, *trans* calls *queueForTranslation* with the target addresses (from source and target machines). If an instruction flows through to its successor, *trans* calls itself tail-recursively.[6] The outline of *trans* is

```
$\langle translator \rangle \equiv$
  **fun** $trans(pc_s, pc_t) \equiv$
    $\langle put\ (pc_s, pc_t)\ in\ codemap\ if\ they\ are\ not\ there\ already \rangle$
    **let** $I$ **as** $(b_I \to nPC := target_I \mid annul := a_I \mid I_c) \equiv src[pc_s]$
    **in case** $class(I)$ **of**
        $\langle cases\ for\ translation\ of\ I \rangle$
    **end**
```

[6] Recursive calls to *trans* could be replaced by calls to *queueForTranslation*. The converse is not true, because *trans* would recurse forever on loops.

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`

SD	*NCT*	`ba L`
		`add %i1, %i2, %i3`
			`nop`
			`mov eax, SPARCI1`
			`add eax, SPARCI2`
			`mov SPARCI3, eax`
			`jmp L`

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

SCDA	*NCT*	`be,a L`
		`mov %o1, %o2`
		⋮
			`nop`
			`je BB`
			⋮
			`BB: mov eax, SPARCO1`
			`mov SPARCO2, eax`
			`jmp L`

SPARC assembly language puts the destination on the right, but Intel assembly language puts the destination on the left. The SPARC has more registers than the Pentium, so we map onto them memory locations `SPARCI1` $= \overline{\texttt{\%i1}}$, `SPARCI2` $= \overline{\texttt{\%i2}}$, etc. The last two examples show the same `be` instruction with and without the `,a` suffix (annul when branch not taken).

Table 1
Example translations from SPARC to Pentium

Translations of SPARC instructions

Deriving a translation function 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. Space limitations allow us to show only a few representative cases. Table 1 shows example SPARC and Pentium assembly language for each.

The easiest cases are ones in which a's and b's are known statically. For non-control-transfer instructions, $b_I \equiv \mathbf{false}$ and $a_I \equiv \mathbf{false}$, which corresponds to the last arm of *stable*, and the translation is

```
⟨cases for translation of I⟩≡
  | $NCT \Longrightarrow pc_t := emit(pc_t, \overline{I_c}); trans(succ_s(pc_s), pc_t)$
```

The static unconditional branch with annul is just like an ordinary branch. $b_I \equiv \mathbf{true}$ and $a_I \equiv \mathbf{true}$, which corresponds to the first arm of *stable*, and the translation is

```
⟨cases for translation of I⟩+≡
  | $SU \Longrightarrow pc_t := emit(pc_t, PC := codemap(target_I));$
            $queueForTranslation(target_I, codemap(target_I));$
```

The next simplest cases are the static delayed (SD) class, with $b_I \equiv \mathbf{true}$ and $a_I \equiv \mathbf{false}$. These instructions include unconditional branches and calls, and the translation depends on what sort of instruction I' is found in the delay slot.

```
⟨cases for translation of I⟩+≡
  | $SD \Longrightarrow$
    let $(b_{I'} \to nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) \equiv src[succ_s(pc_s)]$
    in  case $class(I')$ of
        ⟨translation cases for class(I'), where class(I) = SD⟩
    end
```

In the common case, we have a non-control-transfer instruction in the delay slot, with $b_{I'} \equiv \mathbf{false}$ and $a_{I'} \equiv \mathbf{false}$. This corresponds to the fifth arm of *stable*, which executes $[\![I_c]\!]; PC := target_I \mid [\![I'_c]\!]$. Since $target_I$ is a constant, we can rewrite this as $[\![I_c]\!]; [\![I'_c]\!]; PC := target_I$. The translation is then

```
⟨translation cases for class(I'), where class(I) = SD⟩≡
  | $NCT \Longrightarrow$
      $pc_t := emit(pc_t, \overline{I_c});$
      $pc_t := emit(pc_t, \overline{I'_c});$
      $pc_t := emit(pc_t, PC := codemap(target_I));$
      $queueForTranslation(target_I, codemap(target_I));$
```

This translation is not sufficient for call instructions, because a called procedure may use the program counter captured by I_c, and its use of that program counter is determined by software convention, not by the semantics of the hardware. On the SPARC, if I is a call instruction, translation should resume with $trans(succ_s(succ_s(pc_s)), pc_t)$, or if the call returns a structure, with $trans(succ_s(succ_s(succ_s(pc_s))), pc_t)$.

The treatment of class *DD* (dynamic delayed) branches is similar to that of class *SD*, except that the target addresses are computed dynamically. This means that it is not possible to use *codemap* at translation time; the translated code might use *codemap* at run time, or it might call an interpreter or a dynamic translator.

The most common class involving dynamic conditions is the *SCD* (static conditional delayed) class, in which b_I is dynamic and a_I is **false**. Again, the translation depends on what is in the delay slot.

```
⟨cases for translation of I⟩+≡
 | SCD ⟹
  let $(b_{I'} \to nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) \equiv src[succ_s(pc_s)]$
  in  case $class(I')$ of
        ⟨translation cases for class(I'), where class(I) = SCD⟩
  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

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

Because I_c does not affect PC, we transform *stable* as follows:[7]

```
⟨transformed specialization of stable for SCD with NCT in the delay slot⟩≡
  $[\![I_c]\!]$;
  if $[\![b_I]\!]$ then
    $PC := [\![target_I]\!] \mid [\![I'_c]\!]$
  else
    $PC := succ_s(PC);$
  fi
  ; $stable()$
```

[7] We have the alternative of unfolding the call to *stable* in the **else** branch and moving both I_c and I'_c ahead of the **if**. This transformation leads to a translation in which I'_c moves ahead of the branch, and I'_c's successor follows the branch. Epoxie and Noxie use this translation (Wall 1992). The problem is that, if the branch condition b_I tests condition codes, and I'_c sets condition codes, it will be necessary to save and restore the condition codes in order to get the correct branch instruction. It is much simpler to move I'_c into a new block, which the optimizer can sometimes eliminate.

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. *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 cases for class *SCDA* (static delayed branches that annul when not taken) are similar to those of class *SCD*. For example, when *SCDA* is followed by *NCT*, b_I is dynamic, $a_I \equiv \neg b_I$, and $b_{I'} \equiv a_{I'} \equiv$ **false**. *stable* reduces to:

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

The translation is like that of class *SCD*, creating a new basic block, but the recursive call is $trans(succ_s(succ_s(pc_s)), pc_t)$, so translation resumes *after* the delay slot instead of *at* the delay slot.

Simplified translation of many branch instructions

When translating a branch with a non-branch in the delay slot, our method can be reduced to a simple strategy: rewrite the branch as a non-delayed branch, and push the delay instruction to the destination address, the fall-through address, neither, or both, according to the table below.

$a_I \equiv b_I$	Push the delay instruction to the fall-through address.
$a_I \equiv \neg b_I$	Push the delay instruction to the destination address.
$a_I \equiv \mathbf{false}$	Push the delay instruction to both addresses.
$a_I \equiv \mathbf{true}$	Discard the delay instruction.

To push the delay instruction to the destination address, we create a new "trampoline" basic block, which avoids problems in case other branches also flow to the same address.

The last three entries in Table 1 show how this strategy is applied to the unconditional (*SD*), conditional (*SCD*), and conditional annuled (*SCDA*) branches on the SPARC. On the MIPS, programmers may not put branches in delay slots (Kane 1988, Appendix A), and $a_I \equiv \mathbf{false}$ always, so a single instance of this strategy applies to every branch instruction (Srivastava and Wall 1993).

5 Proving Correctness

We prove correctness of translation by reasoning about transitions from states to states. As noted in Section 4, we want to show that running the translated code results in an execution on the target machine that simulates the original execution on the soure machine. Formally, if we start the source machine in a state ${}^s\sigma_1$, and the *loop* function takes it through a sequence of states ${}^s\sigma_1, {}^s\sigma_2, \ldots$, then there is a subsequence of such states ${}^s\sigma_{k_1}, {}^s\sigma_{k_2}, \ldots$ such that $\overline{{}^s\sigma_{k_1}}, \overline{{}^s\sigma_{k_2}}, \ldots$ is a subsequence of the states that the target machine goes through when started in state ${}^t\sigma_1 = \overline{{}^s\sigma_1}$.

The result desired follows directly from this *transition theorem*: If ${}^s\sigma_m$ is a source-machine state such that

(1) $\mathcal{E}[\![annul = \mathbf{false} \wedge nPC = succ_s(PC)]\!]{}^s\sigma_m$,
(2) there is a corresponding target-machine state ${}^t\sigma_n = \overline{{}^s\sigma_m}$, and
(3) *trans* has been called with arguments $(\mathcal{E}[\![PC]\!]{}^s\sigma_m, \mathcal{E}[\![PC]\!]{}^t\sigma_n)$,

then there is an i such that in i steps, the source machine reaches a state ${}^s\sigma_{m+i}$ that also satisfies $\mathcal{E}[\![annul = \mathbf{false} \wedge nPC = succ_s(PC)]\!]{}^s\sigma_{m+i}$. Also, there is a j such that in j steps, the target machine reaches a state ${}^t\sigma_{n+j} = \overline{{}^s\sigma_{m+i}}$, and furthermore (a) $i > 0$ or $j > 0$ and (b) *trans* has been called with arguments $(\mathcal{E}[\![PC]\!]{}^s\sigma_{m+i}, \mathcal{E}[\![PC]\!]{}^t\sigma_{n+j})$.

We prove the transition theorem by case analysis on the classes of the instructions located at $src[PC]$. We use the standard rule for sequential composition ($\mathcal{C}[\![R_1; R_2]\!] = \mathcal{C}[\![R_2]\!] \circ \mathcal{C}[\![R_1]\!]$) as well as the identities for the translation of expressions and computational effects:

$$\mathcal{E}[\![\overline{e}]\!]\overline{\sigma} = \mathcal{E}[\![e]\!]\sigma$$
$$\overline{\mathcal{C}[\![I_c]\!]\sigma} = \mathcal{C}[\![\overline{I_c}]\!]\overline{\sigma}$$

Because of condition 1, we can substitute *stable* for *loop*, so we can apply our transformed version of *stable*, which assigns directly to PC. We assume that all mappings $\overline{\bullet}$ use *codemap* to map the source program counter to the target program counter. To translate a branch, we therefore write

$$\begin{aligned}\overline{\mathcal{C}[\![PC := \mathit{target}]\!]\sigma} &= \overline{\mathrm{subst}_{PC}^{\mathit{target}}\sigma} \\ &= \mathrm{subst}_{\overline{PC}}^{\mathit{codemap}(\mathit{target})}\overline{\sigma} \\ &= \mathrm{subst}_{PC}^{\mathit{codemap}(\mathit{target})}\overline{\sigma} \\ &= \mathcal{C}[\![PC := \mathit{codemap}(\mathit{target})]\!]\overline{\sigma} \qquad (*)\end{aligned}$$

The simplest case in the proof of the transition theorem is a non-control-transfer instruction (NCT). The canonical form of such an instruction is

$$\mathbf{false} \to nPC := \mathit{any} \mid \mathit{annul} := \mathbf{false} \mid I_c.$$

The action of *stable* on this form is $\mathcal{C}[\![PC := \mathit{succ}_s(PC) \mid I_c]\!]$. I_c leaves the program counter unchanged, so we rewrite this as $\mathcal{C}[\![I_c; PC := \mathit{succ}_s(PC)]\!]$. The binary translation has the form $\overline{I_c}$, which may be a sequence of j instructions. Therefore j applications of *simple*, or equivalently, j state transitions on the target machine, have the effect of $\mathcal{C}[\![\overline{I_c}; PC := \mathit{succ}_t^{(j)}(PC)]\!]$. Given ${}^s\sigma_m$ and ${}^t\sigma_n$ satisfying the hypotheses of the transition theorem, after one step, the source machine reaches the state

$${}^s\sigma_{m+1} = \mathcal{C}[\![PC := \mathit{succ}_s(PC)]\!](\mathcal{C}[\![I_c]\!]{}^s\sigma_m).$$

After j steps, the target machine reaches a state

$$\begin{aligned}{}^t\sigma_{n+j} &= \mathcal{C}[\![PC := \mathit{succ}_t^{(j)}(PC)]\!](\mathcal{C}[\![\overline{I_c}]\!]{}^t\sigma_n) \\ &= \mathcal{C}[\![PC := \mathit{succ}_t^{(j)}(PC)]\!](\mathcal{C}[\![\overline{I_c}]\!]\overline{{}^s\sigma_m}) \\ &= \mathcal{C}[\![PC := \mathit{succ}_t^{(j)}(PC)]\!](\overline{\mathcal{C}[\![I_c]\!]{}^s\sigma_m})\end{aligned}$$

From *trans*, $\mathit{codemap}(\mathit{succ}_s(pc_s)) = \mathit{succ}_t^{(j)}(pc_t)$, so by $(*)$

$$\begin{aligned} {}^t\sigma_{n+j} &= \overline{\mathcal{C}[\![PC := \mathit{succ}_s(PC)]\!](\mathcal{C}[\![I_c]\!]{}^s\sigma_m)} \\ &= \overline{{}^s\sigma_{m+1}} \end{aligned}$$

Thus, after one step on the source and j steps on the target, we again reach a pair of states satisfying the conditions of the transition theorem.

As another example, consider an instruction of class *SCD* with an instruction of class *NCT* in the delay slot. If the source machine begins in state ${}^s\sigma$, after 1 or 2 steps it reaches state ${}^s\sigma'$, where

$$\begin{aligned} {}^s\sigma' = &\ \mathbf{if}\ \mathcal{E}[\![b_I]\!](\mathcal{C}[\![I_c]\!]{}^s\sigma)\ \mathbf{then}\ (\mathcal{C}[\![PC := \mathit{target}_I]\!] \circ \mathcal{C}[\![I'_c]\!] \circ \mathcal{C}[\![I_c]\!]){}^s\sigma \\ &\quad \mathbf{else}\ (\mathcal{C}[\![PC := \mathit{succ}_s(PC)]\!] \circ \mathcal{C}[\![I_c]\!]){}^s\sigma\ \mathbf{fi} \end{aligned}$$

If the target machine begins in state ${}^t\sigma = \overline{{}^s\sigma}$, it reaches state ${}^t\sigma'$, where

$$\begin{aligned} {}^t\sigma' = &\ \mathbf{if}\ \mathcal{E}[\![\overline{b_I}]\!](\mathcal{C}[\![\overline{I_c}]\!]\overline{{}^s\sigma})\ \mathbf{then} \\ &\qquad (\mathcal{C}[\![PC := \mathit{codemap}(\mathit{target}_I)]\!] \circ \mathcal{C}[\![\overline{I'_c}]\!] \circ \mathcal{C}[\![PC := bb]\!] \circ \mathcal{C}[\![\overline{I_c}]\!])\overline{{}^s\sigma} \\ &\quad \mathbf{else} \\ &\qquad (\mathcal{C}[\![PC := \mathit{succ}_t(PC)]\!] \circ \mathcal{C}[\![\overline{I_c}]\!])\overline{{}^s\sigma}\ \mathbf{fi} \end{aligned}$$

Because $\mathcal{C}[\![PC := t_1]\!] \circ \mathcal{C}[\![PC := t_2]\!] = \mathcal{C}[\![PC := t_1]\!]$, and because $\mathcal{C}[\![\overline{I'_c}]\!]$ commutes with assignments to PC, it is easy to show that ${}^t\sigma' = \overline{{}^s\sigma'}$.

The other cases for translation can be proved correct in similar fashion.

6 Experience

We have used translators for delayed branches in two tools: a binary translator and a decompiler (Cifuentes, Simon, and Fraboulet 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 like structured control flow.

There are many issues that are relevant to completely general binary translation but which are beyond the scope of this paper.

- Our translator does not guarantee that the source and target codes have the same atomicity properties; providing atomic three-address operations on a two-address machine would be prohibitively expensive.

- Self-modifying code and dynamic code generation can be handled either by resorting to interpretation or by invoking the translator dynamically; we intend to evaluate these alternatives experimentally.
- Different machines use different representations of condition codes, and a naïve translation would emulate the source-machine condition codes in a target-machine register. This emulation may be necessary in some cases (e.g., when a Pentium program depends on the value of the "parity of the least-significant byte" bit), but in common cases, one definition of condition codes reaches one use (in a conditional branch), and the source-machine condition code can be eliminated by forward substitution.
- The CPU model used in this paper models hardware exceptions as assignment to a special "exception location." This model is suitable only for a machine with precise exceptions. It is an open question whether a similar formalism could help derive a translation between machines with precise and imprecise exceptions.

Our original implementation was based on a case analysis of the SPARC's architecture manual. This analysis created an extra basic block for every delayed instruction that needed to be executed along any given path. More seriously, 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.

We have since replaced our original implementation with one based on the method described in this paper. The new implementation is used in both tools. The advantages of the new method are three-fold: it can handle any branch in a delay slot, even if the target is a branch; it generates better intermediate code than before; and we recover control-flow graphs with fewer basic blocks.

All the transformations discussed in this paper 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_I's, b_I's, and I_c's. This work is not intellectually demanding, but it is tedious because there are many cases.

Our implementation includes simple optimizations not mentioned above. For example, we do not create the `nop` instructions shown in Table 1 when I_c is **skip**. There are also many cases in which further transformation of *stable* can show that it is not necessary to create new basic blocks.

To test the correctness of our implementation, we developed a test suite that includes not only standard programs but also artificial programs with different kinds of branches in delay slots. We checked by hand that the intermediate forms and control-flow graphs derived from the translation were correct at each relevant basic block.

As presented in this paper, 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 have cut off the unfolding at one step beyond what is shown in this paper. 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.

Acknowledgements

We thank Jack Davidson for helpful discussions and Mike van Emmerik for the implementation of the algorithm. This work has been supported by grant A49702762 from the Australian Research Council. The first author has additional support from National Science Foundation grants ASC-9612756 and CCR-9733974 and from DARPA contract MDA904-97-C-0247. The second author has additional support from Sun Microsystems.

References

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley.

Barbacci, Mario R. and Daniel P. Siewiorek. 1982.
The Design and Analysis of Instruction Set Processors.
New York, NY: McGraw-Hill.

Bell, C. Gordon and Allen Newell. 1971.
Computer Structures: Readings and Examples.
New York: McGraw-Hill.

Cifuentes, Cristina and K. John Gough. 1995 (July).
Decompilation of binary programs.
Software—Practice & Experience, 25(7):811–829.

Cifuentes, Cristina, Doug Simon, and Antoine Fraboulet. 1998 (November).
Assembly to high-level language translation.
In *Proceedings of the International Conference on Software Maintenance*, pages 228–237. IEEE-CS Press.

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

Danvy, Olivier, Karoline Malmkjær, and Jens Palsberg. 1996 (November).
Eta-expansion does The Trick.
ACM Transactions on Programming Languages and Systems, 18(6):730–751.

Hoffman, Thomas. 1997 (March 24).
Recovery firm hot on heels of missing source code.
Computer World.

Jones, Neil D., Carsten K. Gomard, and Peter Sestoft. 1993 (June).
Partial Evaluation and Automatic Program Generation.
International Series in Computer Science: Prentice Hall International.
ISBN number 0-13-020249-5 (pbk).

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

Larus, James R. and Thomas Ball. 1994 (February).
Rewriting executable files to measure program behavior.
Software—Practice & Experience, 24(2):197–218.

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

Milner, Robin. 1990.
Operational and algebraic semantics of concurrent processes.
In van Leewen, J., editor, *Handbook of Theoretical Computer Science*, Vol. B: Formal Models and Semantics, chapter 19, pages 1201–1242.
New York, N.Y.: The MIT Press.

Ramsey, Norman and Jack W. Davidson. 1998 (June).
Machine descriptions to build tools for embedded systems.
In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, Vol. 1474 of *LNCS*, pages 172–188. Springer Verlag.

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

Shankar, Natarajan. 1996 (May).
Steps towards mechanizing program transformations using PVS.
Science of Computer Programming, 26(1–3):33–57.

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

SPARC International. 1992.
The SPARC Architecture Manual, Version 8.
Englewood Cliffs, NJ: Prentice Hall.

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

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

Wall, David W. 1992.
Systems for late code modification.
In Giegerich, Robert and Susan L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, pages 275–293. Springer-Verlag.

Witchel, Emmett and Mendel Rosenblum. 1996 (May23–26).
Embra: Fast and flexible machine simulation.
In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Vol. 24,1 of *ACM SIGMETRICS Performance Evaluation Review*, pages 68–79, New York.