A Transformational Approach to Binary Translation of Delayed Branches with Applications to SPARC® and PA-RISC Instructions Sets

Cristina Cifuentes and Norman Ramsey

A Transformational Approach to Binary Translation of Delayed Branches with Applications to SPARC® and PA-RISC Instructions Sets*

Cristina Cifuentes and Norman Ramsey

SMLI TR-2002-104 January 2002

Abstract:

A binary translator examines binary code for a source machine, optionally builds an intermediate representation, and generates code for a target machine. Understanding what to do with delayed branches in binary code can involve tricky case analyses, e.g., if there is a branch instruction in a delay slot. Correctness of a translation is of utmost importance. 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 supports such common architectures as SPARC®, MIPS, and PA-RISC.

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. We show the translation is correct by reasoning about corresponding states on source and target machines.

Instantiation of this algorithm to the SPARC V8 and PA-RISC V1.1 architectures is shown. Of interest, these two machines share seven of 11 classes of delayed branching semantics; the PA-RISC has three classes which are not available in the SPARC architecture, and the SPARC architecture has one class which is not available in the PA-RISC architecture.

Although the delayed branch is an architectural idea whose time has come and gone, the method is significant to anyone who must write tools that deal with legacy binaries. For example, translators using this method could run PA-RISC on the new IA-64 architecture, or they may enable architects to eliminate delayed branches from a future version of the SPARC architecture.

*****This report is a very extended version of TR 440, Department of Computer Science and Electrical Engineering, The University of Queensland, Dec 1998, and describes applications of the technique to translations of SPARC® and PA-RISC codes. This report fully documents the translation algorithms for these machines.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email addresses:
cristina.cifuentes@sun.com
nr@eecs.harvard.edu

A Transformational Approach to Binary Translation of Delayed Branches with Applications to SPARC and PA-RISC Instruction Sets[*]

Cristina Cifuentes
Sun Microsystems Laboratories
Palo Alto, CA 94303, USA
cristina.cifuentes@sun.com

Norman Ramsey
Division of Engineering and Applied Sciences
Harvard University, Cambridge, MA 01238, USA
nr@eecs.harvard.edu

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 [BKMM87, May87, AS92, SCK$^+$93, EA97]. Hardware buyers can use binary translation to run old code on new machines. 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 [CK94, WR96].

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 [SCK$^+$93, LB94]. This paper focuses on translation: the problem of distinguishing code from data is difficult, but solutions are well known.

When a mapping from source locations to target 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 or call instruction appears in a delay slot. Although the delayed

[*]This report is a very extended version of TR 440, Department of Computer Science and Electrical Engineering, The University of Queensland, Dec 1998, and describes applications of the technique to translations of SPARC(R) and PA-RISC codes. This report fully documents the translation algorithms for these machines.

branch is an architectural idea whose time has come and gone, the method is significant to anyone who must write tools that deal with legacy binaries. For example, translators using this method could run PA-RISC code on the new IA-64 architecture. They might also enable architects to eliminate delayed branches from a future version of the SPARC architecture.

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 and problematic cases, and gives confidence that all cases are considered. We have applied the results of the method to the University of Queensland binary translator [CVR99, CV00], and these results could profitably be applied not only to other binary translators [ZT00], but also to any tools that analyze machine instructions, including optimizers [SW93, BDB00], code instrumentors [Wal92, LB94, LS95], fault isolators [WLAG93], and decompilers [CG95, Hof97, CSF98].

Our method uses register transfer lists (RTLs) as a semantic framework in which to reason about instructions on both the source and target machines [RD98]. 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. The source machine has delayed branching semantics, the target does not. 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 successor 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, call, or other transfer of control. 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 [SPA92, §B.26].

The building blocks of this paper are not new. Register-transfer languages have been used to describe instructions for years [BN71, BS82]. Our program transformations draw from standard techniques in compiler optimization [ASU86] and partial evaluation [JGS93].

The contribution of this paper is the idea of applying these well-known techniques to a new problem domain, where they have been used to build a SPARC and PA-RISC architectures front end for the University of Queensland binary translator [CVR99, CV00].

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.

We write R for the set of all possible RTLs. We assume that the effect of any machine instruction can be described by a suitable $r \in R$, as appears to be the case for many real microprocessors. This paper describes a translation between subsets $L_s \subset R$ and $L_t \subset R$. The source-machine language, L_s, has delayed branches; the target-machine language, L_t, does not.

2.1 Register transfer lists

Our RTL formalism is designed for use in tools and component generators, and it makes machine-dependent computation explicit [RD98]. For this paper, 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
$\mid$ *location*	Fetch from a location
$\mid$ *exp binop exp*	Binary RTL operator
$\mid$ *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] 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. Appendix A makes this notion precise by giving a denotation function $\mathcal{R}[\![rtl]\!] : \Sigma \rightarrow \Sigma$.

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. Expressions have their own denotation function $\mathcal{E}[\![e]\!] : \Sigma \rightarrow V$, where V is the domain of values.

In this paper, we assume that locations are single cells in a mutable store, although the full 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
```

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

⟨*RTL for sample instruction*⟩≡
$\$r[24] := \$m[\$r[14] + sx(-12)]$

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

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 those of the SPARC processor, the instruction I can "annul" its successor, in which case the successor is *not* executed; instead the processor stalls for one or more cycles

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:

⟨*RTL for call*⟩≡
  $\$r[15] := PC \mid PC := nPC \mid nPC := target$

The $target$ address in the RTL is distinct from the `target` field in the binary representation of the call instruction. In the case of the SPARC architecture, 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:

⟨*rtl for conditional branch BNE*⟩≡
  $\neg Z \to nPC := target$

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

[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 with each instruction I a Boolean expression a_I (not necessarily a single bit) that tells the processor whether to annul the instruction's successor.

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:

⟨*RTL for generic instruction I*⟩≡
  $b_I \rightarrow nPC := target_I \mid annul := a_I \mid I_c$

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 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 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 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 $\texttt{a} \wedge Z$.

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 $target$.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). We therefore define L_s as follows:

$$L_s \overset{\text{def}}{=} \{I \in R \mid \exists\, b_I, target_I, a_I, I_c : \mathcal{R}[\![b_I \rightarrow nPC := target_I \mid annul := a_I \mid I_c]\!] = \mathcal{R}[\![I]\!]\}.$$

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.

`add rs1, rs2, rd`
  false $\rightarrow nPC := any \mid annul :=$ **false** $\mid \$r[\texttt{rd}] := \$r[\texttt{rs1}] + \$r[\texttt{rs2}]$
`ba,a addr` **true** $\rightarrow nPC := \texttt{addr} \mid annul :=$ **true** $\mid$ **skip**
`call addr` **true** $\rightarrow nPC := \texttt{addr} \mid annul :=$ **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 $(b_I \to nPC := target_I \mid annul := a_I \mid I_c) \equiv src[PC]$
  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.

We have chosen a formalism in which an RTL alone is not sufficient 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:

```
⟨Sparc execution loop⟩≡
  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 execution loop is written using a simple imperative metalanguage, the semantics of which are given in Appendix A. In the body of the paper, 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 as specified by $\mathcal{R}[\![r]\!]$.

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. We define the target language L_t by its canonical form:

$$L_t \stackrel{\text{def}}{=} \{I \in R \mid \exists\, b_I, \mathit{target}_I, I_c : \mathcal{R}[\![b_I \to PC := \mathit{target}_I \mid I_c]\!] = \mathcal{R}[\![I]\!]\}.$$

This is the target execution loop:

```
⟨Pentium execution loop⟩≡
  fun $simple()$ ≡
    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

The problem we are trying to solve is to take a source-machine program whose instructions are in L_s, and to translate it into a target-machine program whose instructions are in L_t, such that when the two programs are executed by their respective execution loops, the target program simulates the source program in a way made precise in §3.1. Informally, a program is said to simulate another program if it reproduces the state of the program being simulated after execution of each source instruction.

Both our formalism and the SPARC architecture manual give a clear semantics of delayed branches in terms of PC, nPC, and $annul$. It would therefore 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 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. This 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

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 may not 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,

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

We call a state *stable* if it satisfies this predicate. The processor ABI (application binary interface) guarantees that the processor will be in a stable state at a program's start location [Pre93], and procedure calling conventions guarantee that the processor will be in a stable state at procedure entry points.

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

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

Appendix B lists the transformations used to get from this definition to something much like $simple$.

We do not show every step in the transformation of $stable$. We perform distribution of sequential composition over **let**, forward substitute assignments to $annul$ and nPC, distribute sequential composition ($loop()$) over conditional, 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
          $[\![PC := succ_s(PC) \mid nPC := succ_s(succ_s(PC)) \mid I_c \mid annul := \textbf{true}]\!]$
          ; $loop()$
    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

$\langle$*stable execution loop*$\rangle{+}{\equiv}$

```
 fun $stable() \equiv$
  let $(b_I \to nPC := target_I \mid annul := a_I \mid I_c) \equiv src[PC]$
  in  if $[\![b_I]\!]$ then
      $[\![PC := succ_s(PC) \mid nPC := target_I \mid I_c \mid annul := a_I]\!];$
      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
    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
```

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 [DMP96]: 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 the following **translation algorithm**

```
⟨stable execution loop⟩+≡
  fun $stable() \equiv$
    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
          $[\![I_c]\!]$;
          $[\![PC := target_I \mid nPC := target_{I'} \mid I'_c \mid annul := \textbf{false}]\!]$
          ; $loop()$
      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 annulling branch in its delay slot, acts as if the first branch never happened, and the second is a non-delaying branch.
- A non-annulling branch with another non-annulling 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 [Kan88, Appendix A]. This case requires potentially unbounded unfolding of $loop$, which we discuss in §7.
- A non-annulling branch with an annulling non-branch in its delay slot acts as a branch to the successor of the target instruction. (Note that the SPARC architecture has an annulling non-branch, viz, `BN,A`.)
- A non-annulling branch with a non-annulling non-branch in its delay slot has the effect of delaying the branch by one or more cycles. This is the common case.
- An annulling non-branch skips over its successor.
- A non-annulling non-branch (i.e., an ordinary computational instruction) simply executes and advances the program counter to its successor.

3.1 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. Recall that 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 $\sigma' = \mathcal{R}[\![I]\!]\sigma$.

A translation is correct if execution on the target machine simulates execution on the source machine. The translator builds a map $\bar{\cdot}$ from source-machine states to target-machine states.[3] In a way made precise below, this map respects the operation of the machine. In our design, $\bar{\cdot}$ 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 $\mathcal{E}[\![\neg annul \wedge nPC = succ_s(PC)]\!]\sigma$.

When referring to states, we use a left superscript of s or t to designate a state on the source or target machine. We use subscripts to number states in sequence.

[3]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 function $\bar{\cdot}$ notation for its readability. 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." Writing the existential quantifiers and relations explicitly would obscure the ideas.

We say the target machine *simulates* the source machine if the following condition holds: if we start the source machine in a state ${}^s\sigma_0$, and the $loop$ function takes it through a sequence of states ${}^s\sigma_0, {}^s\sigma_1, \ldots$, then there is a subsequence of such states ${}^s\sigma_{k_0}, {}^s\sigma_{k_1}, \ldots$ such that $\overline{{}^s\sigma_{k_0}}, \overline{{}^s\sigma_{k_1}}, \ldots$ is a subsequence of the states that the target machine goes through when started in state ${}^t\sigma_0 = \overline{{}^s\sigma_0}$. Further, each state ${}^s\sigma_I, I > 0$ is the state of the source machine after execution of instruction I of the program. ${}^s\sigma_0$ is the initial state of the program. 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 corresponds one to one.[4] We sketch a proof in §6.

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, $\bar{\cdot}$ can mostly map locations to locations, without changing values. The exception is the program counter; its translation must use $codemap$, so we require $\mathcal{E}[\![PC]\!]\overline{\sigma} = codemap(\mathcal{E}[\![PC]\!]\sigma)$. Given a map $\bar{\cdot}$ 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 $annul$. Given an effect I_c, we write its translation $\overline{I_c}$; in general, $\overline{I_c}$ is a sequence of RTLs, not exactly one RTL. Any translation is acceptable as long as for any σ, $\mathcal{R}[\![\overline{I_c}]\!]\overline{\sigma} = \overline{\mathcal{R}[\![I_c]\!]\sigma}$.[5] We also assume that, given any condition b and address $target$, we can construct an instruction sequence implementing $b \rightarrow PC := target$ 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.

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:

[4]In the terminology of [Mil90], the transitions to these intermediate states are "silent."

[5]We extend $\mathcal{R}$ to sequences of RTLs using the standard rule for sequential composition: $\mathcal{R}[\![r_1; r_2]\!] = \mathcal{R}[\![r_2]\!] \circ \mathcal{R}[\![r_1]\!]$.

$queueForTranslation(pc_s, pc_t)$	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

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

4 Application to the SPARC instruction set

To this point, the development of our idea is abstract; it could apply to any machines with and without delayed branches. The value of our work, however, is that it can be applied to real translators for real instruction sets. This section applies our formal analysis to the SPARC V8 instruction set, and it sketches the derivation of a translator. We used this derivation to build the SPARC front end of the University of Queensland binary translator [CVR99, CV00].

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 architecture, only 9 combinations are used:

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

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*

These combinations enable us to classify instructions. 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 $PC + 4$ [SPA92, §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:

$a_I \equiv$ **false**	Never annul.
$a_I \equiv$ **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 Translations of SPARC instructions

Deriving a translation function is tedious but straightforward. We start from the $trans$ algorithm in §3.1, pg 13, and expand it into the following algorithm, parameterized by the class of instructions:

```
⟨sparc translator⟩≡
 fun $trans(pc_s, pc_t) =$
   $codemap(pc_s) \equiv pc_t$
   let $I$ as $(b_I - > nPC := target_I \mid annul := a_I \mid I_c) = src[pc_s]$
   in case $class(I)$ of
   $\mid NCT \Longrightarrow nonBranching, nonAnnulling$
   $\mid SKIP \Longrightarrow nonBranching, annulling$
   $\mid SU \Longrightarrow branching, annulling$
   $\mid SD \Longrightarrow branching, nonAnnulling$
   $\mid DD \Longrightarrow branching, nonAnnulling$
   $\mid SCD \Longrightarrow$ if $(b_I)$ then
           $branching, nonAnnulling$
         else
           $nonBranching, nonAnnulling$
         fi
   $\mid SCDA \Longrightarrow$ if $(b_I)$ then
           $branching, nonAnnulling$
         else
           $nonBranching, annulling$
         fi
   end
```

Before we look at individual translations of some of the classes of instructions, we give some examples in Table 1 of translations of SPARC assembly code into Pentium assembly code, i.e., translations from a machine that has delayed branches to one that does not have delayed branches. 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 be best left to a general-purpose optimizer.

We show only a few representative cases of the transformations applied to the skeleton $trans$ algorithm, the complete algorithm is documented in Appendix C. We apply the transformations listed in Appendix B throughout this process.

Translation of non control transfer instructions

For non-control-transfer instructions, $b_I \equiv$ **false** and $a_I \equiv$ **false**, i.e., these are non branching, non annulling instructions, which correspond to the last arm of the $stable$ algorithm (pg 10). The translation is

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

$class(I)$	$class(I')$	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`

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

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

Table 1: Example translations from SPARC architecture to Pentium architecture

Translation of static unconditional branch with annul intructions

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

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

Translation of static delayed instructions

The next simplest cases are the static delayed (SD) class, with $b_I \equiv$ **true** and $a_I \equiv$ **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 $(b_{I'} \rightarrow nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) \equiv src[succ_s(pc_s)]$
    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 $b_{I'} \equiv$ **false** and $a_{I'} \equiv$ **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 architecture, if I is a call instruction, convention says that translation should resume with $trans(succ_s(succ_s(pc_s)), pc_t)$, or if the call returns a structure, with $trans(succ_s(succ_s(succ_s(pc_s))), pc_t)$.

Translation of dynamic delayed instructions

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.

Translation of static conditional delayed instructions

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⟩+≡
  | $SCD \Longrightarrow$
   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
      ⟨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 [Wal92]. 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

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

which we translate using an ordinary branch instruction:

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

Translation of static conditional delayed annulling instructions

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:

```
⟨specialization of stable for SCDA with NCT in the delay slot⟩≡
  $[\![I_c]\!]$;
  **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.

The most difficult cases arise when the instruction in the delay slot of a branch I is another delayed branch instruction I'. These cases multiply like rabbits. We show just one, but it is almost useful; putting an unconditional SD branch in the delay slot of another unconditional SD branch makes it possible to execute a single non-branching 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.

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

We unfold the call to $loop$ and substitute forward for PC, nPC, and $annul$. Removing dead assignments leaves

```
⟨transformed specialization of stable for SD with SD in the delay slot⟩≡
  $[\![I_c]\!];$
  $[\![I'_c]\!];$
  let $(b_{I''} \rightarrow nPC := target_{I''} \mid annul := a_{I''} \mid I''_c) \equiv src[[\![target_I]\!]]$
  in  if $[\![b_{I''}]\!]$ then
      $[\![PC := target_{I'} \mid nPC := target_{I''} \mid I''_c \mid annul := a_{I''}]\!]$
    else
      $[\![PC := target_{I'} \mid nPC := succ_s(target_{I'}) \mid I''_c \mid annul := a_{I''}]\!]$
    fi
    $; loop()$
  end
```

We can now move $loop()$ inside the conditional, convert it to $stable()$ in one branch, unfold, etc. We wind up with 4 cases based on the values of $[\![a_{I''}]\!]$ and $[\![b_{I''}]\!]$. 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), i.e., where $b_{I''} =$ **false** and $a_{I''} =$ **false**. This case reduces to

```
⟨further specialization of stable for SD with SD in the delay slot (class(I")=NCT)⟩≡
  $[\![I_c]\!];$
  $[\![I'_c]\!];$
  $[\![PC := target_{I'} \mid I''_c]\!]$
  $; stable()$
```

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:

```
⟨translation cases for $class(I'')$, where class(I) = SD and class(I') = SD⟩≡
  $\mid NCT \Longrightarrow$
    $pc_t := emit(pc_t, \overline{I_c});$
```

```
$pc_t := emit(pc_t, \overline{I'_c});$
$pc_t := emit(pc_t, \overline{I''_c});$
$pc_t := emit(pc_t, PC := codemap(target_{I'}));$
$queueForTranslation(target_{I'}, codemap(target_{I'}));$
```

Since on the SPARC architecture 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 1. The instruction in the delay slot of I' (`mov 3, %o0` in Table 1) is not executed.

Refer to Appendix C for a complete description of the SPARC V8 translation algorithm.

4.3 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.
$a_I \equiv$ **false**	Push the delay instruction to both addresses.
$a_I \equiv$ **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 third, fourth, and fifth examples in Table 1 show how this strategy is applied to the unconditional (*SD*), conditional (*SCD*), and conditional annulled (*SCDA*) branches on the SPARC architecture. On the MIPS, programmers may not put branches in delay slots [Kan88, Appendix A], and $a_I \equiv$ **false** always, so a single instance of this strategy applies to every branch instruction [SW93].

5 Application to the PA-RISC instruction set

The PA-RISC architecture’s concept of delayed branches is the same as that of the SPARC architecture; however, the notation used in the architectural manual is fairly different at first glance. Any transfer of control instruction is delayed, the instruction following the control transfer instruction (the *delay slot* instruction) is executed before control reaches the target of the branching instruction.

Execution of the delay slot instruction is optional; this is determined by the “nullify” bit in the branch instruction [Pac94]. The concept of the nullify bit is modelled in our system by the annul state.

5.1 Modelling PA-RISC instruction address queues

The PA-RISC processor’s documentation describes an instruction address (IA) queue for handling of the address of the instruction to be executed next. There are three elements in the IA queue; the _Front, _Back and the _Next elements. The _Front element is the address of the instruction to be executed, i.e., this is equivalent to the Program Counter (PC) in our model. The _Back element holds the address of the next instruction (after the one at _Front). This element is equivalent to our nPC state, before the nPC is modified by the execution of the instruction at PC. The _Next element is only used in control transfer instructions and refers to the address of the next PC after execution of the control transfer instruction, i.e., it is the new value of nPC after _Back is updated.

There are two address queues: one for offsets within a given space, and one for different spaces. The former is called the Instruction Address Offset Queue (IAOQ) and the latter the Instruction Address Space Queue (IASQ). An address in the PA-RISC is determined by the combination of a given space address and the offset address. The combined IASQ_Front and IAOQ_Front elements provide the virtual address of the current executing instruction, whereas the IASQ_Back and IAOQ_Back provides the virtual address of the following instruction that will be executed (see PA-RISC architecture manual [Pac94] at page 4-11).

```
Instruction Address Offset Queue (IAOQ)
IAOQ_Front < −− IAOQ_Back;
IAOQ_Back < −− IAOQ_Next;
if (taken branch)
        IAOQ_Next < −− Branch target offset;
else
        IAOQ_Next < −− IAOQ_Back + 4;
```

```
Instruction Address Space Queue (IASQ)
IASQ_Front < −− IASQ_Back;
IASQ_Back < −− IASQ_Next;
if (BE or BLE)
        IASQ_Next < −− Branch target space ID;
else
        IASQ_Next < −− IASQ_Back;
```

Figure 1: Updating Instruction Address Queues in the PA-RISC Architecture

For illustration purposes, we have reproduced Figure 4-3 of the PA-RISC architecture manual [Pac94], pg 4-12, in Figure 1. This figure describes the changes of state of the IA queues during control transfer instruction execution. Conceptually, the offset and space queues provide one address, and the 3 different addresses being stored in the IA queues expose a temporary address that is stored in the physical queue. In our system, we model two states; the address of the current instruction (PC) and that of the next instruction to be executed (nPC). These two states are equivalent to storing the three different values in the following algorithm:

```
if (taken branch)
   nPC <-- Branch target offset;
else
   nPC <-- nPC + 4;
```

In other words, there are two values for nPC in the previous algorithm; the initial nPC value and the new nPC value; these are equivalent to the _Next and _Back values in the PA-RISC notation. We therefore model these IA queues using our PC and nPC state variables.

5.2 Classification of PA-RISC instructions

PA-RISC has several types of control transfer instructions, commonly referred to as branching instructions in the PA-RISC architecture documentation [Pac94]. Branching instructions are either unconditional or conditional. Unconditional instructions are local or external. Conditional instructions can only be local. Figure 2 enumerates the different types of branching instructions.

Branching Instructions		
Unconditional		
Local	External	Conditional
MOVB, MOVIB COMPBx, COMPIBx ADDBx, ADDIBx BB, BBV	BL GATE BLR BV	BE BLE

Figure 2: Classification of PA-RISC Control Transfer Instructions

We briefly describe the instructions of Figure 2. The unconditional branching instructions are calls or jumps; these are: `BL` is the branch and link (call) instruction, `GATE` is the gateway instruction (a call that changes priviledge level), `BLR` is the branch and link register instruction, `BV` is the branch vectored instruction, `BE` is the branch external instruction, and `BLE` is the branch and link external instruction. Conditional branching instructions are the ones denoted by an `x` as the prefix to the instruction name (e.g., `ADDBx`). Such instructions take two forms, and branch on the result of the operation being true or branch on false. The operation per se is part of the instruction, and are as follows: `COMPBx` does a compare instruction and then branches on true or false, `COMPIBx` does a compare immediate and branches on true or false, `ADDBx` does an add and branches on the result of that add being true or false, and `ADDIBx` does and add immediate and branches on true or false. The `BB` instruction branches on bit and the `BBV` branches on variable bit.

Conditional branching instructions effectively perform the semantics of a non-control transfer instruction (move, compare, or add) and then perform a branch based on a condition sometimes determined by the instruction being executed (move, compare, or add). We model these conditional instructions in the following way:

⟨*conditional branching model*⟩≡
    $src[PC] = (I_c; Bcondx)$

where x can be true or false. That is, a conditional branching instruction is a compound instruction which can be modelled as two separate instructions, one performing computation and the next performing the branch. We use the `Bcondx` notation to refer to any such conditional branch (on true or false) derived from a conditional branching instruction.

We now classify the PA-RISC V1.1 architecture instructions based on our variables: 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). This classification leads to 9 combinations on the PA-RISC:

Instruction	b_I	a_I	$target_I$	I_c	*Class*
`BL`	**true**	**false**	static	$\$r[t] := nPC + 4$	*SD*
`GATE`	**true**	**false**	static	⟨*change priviledge level*⟩	*SD*
`BLR`	**true**	**false**	dynamic	$\$r[t] := nPC + 4$	*DD*
`BV`	**true**	**false**	dynamic	**skip**	*DD*
`BL,n`	**true**	**true**	static	$\$r[t] := nPC + 4$	*SU*
`GATE,n`	**true**	**true**	static	⟨*change priviledge level*⟩	*SU*
`BLR,n`	**true**	**true**	dynamic	$\$r[t] := nPC + 4$	*DU*
`BV,n`	**true**	**true**	dynamic	**skip**	*DU*
`BE`	**true**	**false**	static	⟨*change space*⟩	*SD*
`BE,n`	**true**	**true**	static	⟨*change space*⟩	*SU*
`BLE`	**true**	**false**	static	$\$r[31] := nPC + 4$ ⟨*change space*⟩	*SD*
`BLE,n`	**true**	**true**	static	$\$r[31] := nPC + 4$ ⟨*change space*⟩	*SU*
BcondT	$test(cond)$	**false**	static	**skip**	*SCD*
BcondT$_>$,n	$test(cond)$	$test(cond)$	static	**skip**	$SCDA_>$
BcondT$_<$,n	$test(cond)$	$\neg test(cond)$	static	**skip**	*SCDA*
BcondF	$\neg test(cond)$	**false**	static	**skip**	*SCD*
BcondF$_>$,n	$\neg test(cond)$	$\neg test(cond)$	static	**skip**	$SCDA_>$
BcondF$_<$,n	$\neg test(cond)$	$test(cond)$	static	**skip**	*SCDA*
NCT	**false**	**false**	N/A	⟨*varies*⟩	*NCT*
NCTcond	**false**	$test(cond)$	N/A	⟨*varies*⟩	*NCTA*

The 9 classes are as follows:

SD	Static delayed (unconditional)
DD	Dynamic delayed (unconditional)
SU	Static unconditional (not delayed)
DU	Dynamic unconditional (not delayed)
SCD	Static conditional delayed
SCDA	Static conditional delayed, annulling
$SCDA_>$	Static conditional delayed, annulling on $>$ displacement
NCT	Non-control-transfer instructions (arithmetic, etc.)
NCTA	Non-control-transfer, annulling

Note that compound trap instructions such as “add and trap on overflow” were not modelled in this study. In practice, these instructions behave in a similar way to the SPARC V8 TRAP class. Therefore, in the PA-RISC architecture, the SPARC SKIP class is not present, and in the SPARC architecture, the DU, $SCDA_>$ and NCTA classes are not present.

5.3 Derivation of a translator

We derived a translator in the same way that the SPARC translator was derived (§4.2). The complete set of transformations and final algorithm are documented in Appendix D.

6 Proving correctness

We prove correctness of translation by reasoning about transitions from states to states. As noted in §3.1, we want to show that running the translated code results in an execution on the target machine that simulates the original execution on the source machine. Formally, if we start the source machine in a state ${}^s\sigma_0$, and the $loop$ function takes it through a sequence of states ${}^s\sigma_0, {}^s\sigma_1, \ldots$, then there is a subsequence of such states ${}^s\sigma_{k_0}, {}^s\sigma_{k_1}, \ldots$ such that $\overline{{}^s\sigma_{k_0}}, \overline{{}^s\sigma_{k_1}}, \ldots$ is a subsequence of the states that the target machine goes through when started in state ${}^t\sigma_0 = \overline{{}^s\sigma_0}$. Each state ${}^s\sigma_I, I > 0$ is the state of the source machine after execution of instruction I of the program.

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

1. $\mathcal{E}[\![annul = \textbf{false} \wedge nPC = succ_s(PC)]\!]{}^s\sigma_m$,
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 $\mathcal{E}[\![annul = \textbf{false} \wedge nPC = succ_s(PC)]\!]{}^s\sigma_{m+i}$. 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 assume that the translations of expressions and computational effects, whatever they are, satisfy the following identities:

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

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 $\overline{\cdot}$ use *codemap* to map the source program counter to the target program counter. To translate a branch, we therefore write

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

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

$$\textbf{false} \rightarrow nPC := any \mid annul := \textbf{false} \mid I_c.$$

The action of $stable$ on this form is $\mathcal{R}[\![PC := succ_s(PC) \mid I_c]\!]$. I_c leaves the program counter unchanged, so we rewrite this as $\mathcal{R}[\![I_c; PC := succ_s(PC)]\!]$. 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 $\mathcal{R}[\![\overline{I_c}; PC := succ_t^{(j)}(PC)]\!]$. Given ${}^s\sigma_m$ and ${}^t\sigma_n$ satisfying the hypotheses of the transition theorem, after one step, the source machine reaches the state

$$ {}^s\sigma_{m+1} = \mathcal{R}[\![PC := succ_s(PC)]\!](\mathcal{R}[\![I_c]\!]{}^s\sigma_m). $$

After j steps, the target machine reaches a state

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

From $trans$, $codemap(succ_s(pc_s)) = succ_t^{(j)}(pc_t)$, so by $(*)$

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

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

$$\begin{aligned} {}^s\sigma' = {} & \textbf{if } \mathcal{E}[\![b_I]\!](\mathcal{R}[\![I_c]\!]{}^s\sigma) \textbf{ then } (\mathcal{R}[\![PC := target_I]\!] \circ \mathcal{R}[\![I'_c]\!] \circ \mathcal{R}[\![I_c]\!]){}^s\sigma \\ & \textbf{else } (\mathcal{R}[\![PC := succ_s(PC)]\!] \circ \mathcal{R}[\![I_c]\!]){}^s\sigma \textbf{ fi} \end{aligned}$$

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

$$\begin{aligned} {}^t\sigma' = {} & \textbf{if } \mathcal{E}[\![\overline{b_I}]\!](\mathcal{R}[\![\overline{I_c}]\!]\overline{{}^s\sigma}) \textbf{ then} \\ & \quad (\mathcal{R}[\![PC := codemap(target_I)]\!] \circ \mathcal{R}[\![\overline{I'_c}]\!] \circ \mathcal{R}[\![PC := bb]\!] \circ \mathcal{R}[\![\overline{I_c}]\!])\overline{{}^s\sigma} \\ & \textbf{else} \\ & \quad (\mathcal{R}[\![PC := succ_t(PC)]\!] \circ \mathcal{R}[\![\overline{I_c}]\!])\overline{{}^s\sigma} \textbf{ fi} \end{aligned}$$

Because $\mathcal{R}[\![PC := t_1]\!] \circ \mathcal{R}[\![PC := t_2]\!] = \mathcal{R}[\![PC := t_1]\!]$, and because $\mathcal{R}[\![\overline{I'_c}]\!]$ 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.

7 Experience

We have used translators for delayed branches in two tools: a binary translator [CVR99, CV00] and a decompiler [CSF98]. In both tools, we translate machine instructions into a 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.

Many problems of binary translation 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.
- Different machines use different representations of condition codes, and a naïve translation would emulate 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. This has been implemented in the binary translator system.
- 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 first attempt at translating delayed branches 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 replaced our first attempt with a new implementation 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 formal methods tools that might have helped us transform $stable$, but we were left with the impression that this is still a research problem [Sha96], 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 manually checked that the intermediate forms and control-flow graphs derived from the translation were correct at each relevant basic block. We also executed test cases on both source and target machines, to make sure the proper effects were executed in both source and target codes.

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 architecture 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 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. A fall back interpreter or a fallback translation algorithm that takes both nPC and PC as parameters is needed for completeness.

Acknowledgments

We thank Jack Davidson for helpful discussions, Mike Van Emmerik for the implementation of the translator, and Mitch Wand for help with semantics. This work has been supported by grant A49702762 from the Australian Research Council. The second author had additional support from National Science Foundation grants ASC-9612756 and CCR-9733974 and from DARPA contract MDA904-97-C-0247.

References

[AS92] Kristy Andrews and Duane Sand. Migrating a CISC computer family onto RISC via object code translation. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, in *SIGPLAN Notices*, 27(9):213–222, October 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 35(5):1–12, May 2000.

[BKMM87] Arndt Bergh, Keith Keilman, Daniel Magenheimer, and James Miller. HP3000 emulation on HP Precision Architecture computers. *Hewlett-Packward Journal*, December 1987.

[BN71] C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples*. McGraw-Hill, New York, 1971.

[BS82] Mario R. Barbacci and Daniel P. Siewiorek. *The Design and Analysis of Instruction Set Processors*. McGraw-Hill, New York, NY, 1982.

[CG95] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software—Practice & Experience*, 25(7):811–829, July 1995.

[CK94] Bob Cmelik and David Keppel. 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, May 1994.

[CSF98] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *Proceedings of the International Conference on Software Maintenance*, pages 228–237. IEEE-CS Press, November 1998.

[CV00] Cristina Cifuentes and Mike Van Emmerik. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, March 2000.

[CVR99] Cristina Cifuentes, Mike Van Emmerik, and Norman Ramsey. The design of a resourceable and retargetable binary translator. In *Working Conference on Reverse Engineering (WCRE'99)*, pages 280–291, October 1999.

[DMP96] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 18(6):730–751, November 1996.

[EA97] Kemal Ebcioğlu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In 24^{th} *Annual International Symposium on Computer Architecture*, pages 26–37, 1997.

[Hof97] Thomas Hoffman. Recovery firm hot on heels of missing source code. *Computer World*, March 24 1997.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[Kan88] Gerry Kane. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[LB94] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software—Practice & Experience*, 24(2):197–218, February 1994.

[LS95] James R. Larus and Eric Schnarr. 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, June 1995.

[May87] Cathy May. MIMIC: A fast System/370 simulator. *Proceedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, in *SIGPLAN Notices*, 22(7):1–13, June 1987.

[Mil90] Robin Milner. Operational and algebraic semantics of concurrent processes. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1201–1242. The MIT Press, New York, N.Y., 1990.

[Pac94] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett Packard, third edition, 1994. HP Part Number: 09740-90039.

[Pre93] Prentice Hall, Englewood Cliffs, NJ. *System V Application Binary Interface, SPARC Architecture Processor Supplement*, third edition, 1993. Unix Press.

[RD98] Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474 of *LNCS*, pages 172–188. Springer Verlag, June 1998.

[SCK$^+$93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[Sha96] Natarajan Shankar. Steps towards mechanizing program transformations using PVS. *Science of Computer Programming*, 26(1–3):33–57, May 1996.

[SPA92] SPARC International, Englewood Cliffs, NJ. *The SPARC Architecture Manual, Version 8*, 1992.

[SW93] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization. *Journal of Programming Languages*, 1:1–18, March 1993. Also available as WRL Research Report 92/6, December 1992.

[Wal92] David W. Wall. Systems for late code modification. In Robert Giegerich and Susan L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, pages 275–293. Springer-Verlag, 1992.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, December 1993.

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume 24,1 of *ACM SIGMETRICS Performance Evaluation Review*, pages 68–79, New York, May23–26 1996. ACM Press.

[ZT00] Cindy Zheng and Carol Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *Computer*, 33(3):47–52, March 2000.

A Formal models

This appendix provides formal semantics for the RTLs and the metalanguage used in the body of the paper. We use a "core" metalanguage that is simpler than what is used in the body. Here is its grammar; the "{braces}" are EBNF meta-notation for sequences. statement

$$\begin{aligned}
\textit{program} &\Rightarrow \{\textbf{fun}\ \textit{function-name}\ () \equiv \textit{statement}\} \\[1em]
\textit{statement} &\Rightarrow \textit{statement};\ \textit{statement} \\
&\mid\ eval_r(rtl) \\
&\mid\ \textbf{if}\ \textit{meta-exp}\ \textbf{then}\ \textit{statement}\ \textbf{else}\ \textit{statement}\ \textbf{fi} \\
&\mid\ \textbf{let}\ (\textit{rtl}) \equiv memory[exp]\ \textbf{in}\ \textit{statement}\ \textbf{end} \\
&\mid\ \textit{function-name}() \\[1em]
\textit{meta-exp} &\Rightarrow eval_e(\textit{exp}) \\
&\mid\ \neg\ \textit{meta-exp} \\
&\mid\ \textit{meta-exp} \wedge \textit{meta-exp}
\end{aligned}$$

The program fragments used in the paper should be rewritten into the simpler core using the following rules:

- The expression $[\![e]\!]$ is short for $eval_e(e)$.
- The statement $[\![r]\!]$ is short for $eval_r(r)$.
- An **if** statement using $elseif$ is short for nested **if** statements.
- A **let** binding multiple RTLs is short for nested **let** bindings (as in ML).

We present the semantic rules in three layers: expressions, which map states to values; RTLs, which map states to states; and statements, which map continuations to continuations. To simplify the semantic equations, we use a revised grammar for RTLs. explist

$$\begin{aligned}
\textit{rtl} &\Rightarrow exp_g \rightarrow \textit{location} := exp_r \mid \textit{rtl} \\
\textit{rtl} &\Rightarrow \textit{location} := exp_r \mid \textit{rtl} \\
\textit{rtl} &\Rightarrow \\
\textit{rtl} &\Rightarrow \textit{name} \\[1em]
\textit{exp} &\Rightarrow \textit{constant} \\
&\mid\ \textit{location} \\
&\mid\ \textit{exp operator exp} \\
&\mid\ \textit{operator}\ \texttt{(}\ \textit{explist}\ \texttt{)} \\
&\mid\ \textit{name} \\[1em]
\textit{explist} &\Rightarrow \textit{exp}\ \texttt{,}\ \textit{explist} \\
&\mid
\end{aligned}$$

In the right-hand sides for *rtl*, the vertical bar $\mid$ is the RTL parallel-composition operator; elsewhere, it is the EBNF metasymbol for alternatives.

To define denotations, we begin with domains $Bits$ (bit vectors), $Bool$ (Booleans), and an unspecified domain L of locations. We construct domains for values and states.

$$\begin{array}{rcll} V & = & Bits + Bool & \text{(values)} \\ L & & & \text{(locations)} \\ \sigma : \Sigma & = & L \to V & \text{(states)} \end{array}$$

The **let** construct in our metalanguage binds names to RTLs and expressions. The body of the paper does not say how to give meanings to such names; this appendix uses environments. Making environments explicit adds precision but also detail; the detail would be distracting in the body of the paper.

$$\rho : Env_r \quad = \quad \textit{name} \to exp + rtl \qquad \text{(RTL environments)}$$

The environment ρ is unusual because the denotations of names are *syntax*, not values.

We assume that the denotations of constants, operators, and locations are given by functions $\mathcal{N}$, $\mathcal{O}$, and $\mathcal{L}$. We provide semantic equations for expressions. We use "cons" and "nil," written "::" and "[]", to construct lists. The denotation functions have the following types:

$$\begin{array}{rl} \mathcal{N}: & \textit{constant} \to V \\ \mathcal{O}: & \textit{operator} \to V\ list \to V \\ \mathcal{L}: & location \to Env_r \to \Sigma \to L \\ \mathcal{E}: & exp \to Env_r \to \Sigma \to V \\ \mathcal{EL}: & exp\ list \to Env_r \to \Sigma \to V\ list \end{array}$$

Here are the semantic equations defining $\mathcal{E}$ on expressions and $\mathcal{EL}$ on lists of expressions.

$$\begin{array}{lcl} \mathcal{E}[\![\textit{constant}]\!]\rho\sigma & = & \mathcal{N}[\![\textit{constant}]\!] \\ \mathcal{E}[\![location]\!]\rho\sigma & = & \sigma(\mathcal{L}[\![location]\!]\rho\sigma) \\ \mathcal{E}[\![exp_1\ \textit{operator}\ exp_2]\!]\rho\sigma & = & \mathcal{O}[\![\textit{operator}]\!](\mathcal{E}[\![exp_1]\!]\rho\sigma :: \mathcal{E}[\![exp_2]\!]\rho\sigma :: []) \\ \mathcal{E}[\![\textit{operator}(explist)]\!]\rho\sigma & = & \mathcal{O}[\![\textit{operator}]\!](\mathcal{EL}[\![explist]\!]\rho\sigma) \\ \mathcal{E}[\![\textit{name}]\!]\rho\sigma & = & \mathcal{E}(\rho(\textit{name}))\rho\sigma \\ \\ \mathcal{EL}[\![exp, explist]\!]\rho\sigma & = & \mathcal{E}[\![exp]\!]\rho\sigma :: \mathcal{EL}[\![explist]\!]\rho\sigma \\ \mathcal{EL}[\![\,]\!]\rho\sigma & = & [] \end{array}$$

As noted above, names denote syntax, which is unusual.

The semantics of RTLs are slightly more complicated than one might expect, because an RTL represents a multiple, simultaneous assignment. The key is the auxiliary function $\mathcal{U}$ ("update"), which computes the values of all the guards, locations and right-hand sides. $\mathcal{U}$ returns a function, which performs those assignments that are called for by the values of the guards. The denotation function $\mathcal{R}$ uses $\mathcal{U}$ to compute this function, then applies the function to the current state σ. We use "upds" to update a state and polymorphic "if" to choose between states. Here are the types of the relevant functions.

$$\begin{aligned}
\mathcal{R}: &\quad rtl \to Env_r \to \Sigma \to \Sigma \\
\mathcal{U}: &\quad rtl \to Env_r \to \Sigma \to \Sigma \to \Sigma \\
\mathrm{upds}: &\quad \Sigma \to L \to V \to \Sigma \\
\mathrm{if}: &\quad \forall \alpha . V \to \Sigma \to \alpha \to \alpha
\end{aligned}$$

The following definition of $\mathcal{U}$ ignores a subtlety; properly speaking, an RTL should not have a denotation if the same location appears on more than one left-hand side. In our definition, the leftmost assignment takes priority. This simplification is warranted because it enables us to avoid putting error values in our domains, but readers should take care not to use transformations whose correctness depends on the order in which "simultaneous" assignments take place!

$$\begin{aligned}
&\mathcal{U}[\![location := exp \mid rtl]\!]\rho\sigma \quad = \quad \lambda\sigma'.\mathrm{upds}(\mathcal{U}[\![rtl]\!]\rho\sigma\sigma')(\mathcal{L}[\![location]\!]\rho\sigma)(\mathcal{E}[\![exp]\!]\rho\sigma) \\
&\mathcal{U}[\![exp_g \to location := exp_r \mid rtl]\!]\rho\sigma = \\
&\qquad \mathrm{if}(\mathcal{E}[\![exp_g]\!]\rho\sigma)(\lambda\sigma'.\mathrm{upds}(\mathcal{U}[\![rtl]\!]\rho\sigma\sigma')(\mathcal{L}[\![location]\!]\rho\sigma)(\mathcal{E}[\![exp_r]\!]\rho\sigma))(\mathcal{U}[\![rtl]\!]\rho\sigma) \\
&\mathcal{U}[\![\,]\!]\rho\sigma \quad = \quad \lambda\sigma'.\sigma' \\
&\mathcal{U}[\![\mathsf{name}]\!]\rho\sigma \quad = \quad \mathcal{U}(\rho(\mathsf{name}))\rho\sigma \\
&\mathcal{R}[\![rtl]\!]\rho\sigma \quad = \quad \mathcal{U}[\![rtl]\!]\rho\sigma\sigma
\end{aligned}$$

Next, we present the semantics of statements. We use a continuation semantics in which the answers are sequences of states; these are the sequences used in §6. Statements also require an additional environment ρ_f, which is used to give meanings to function calls. Our functions are parameterless, and the denotation of a function is the same as the denotation of a statement: a function from continuations to continuations.

$$\begin{aligned}
C &= \Sigma \to A && \text{(continuations)} \\
A &= \Sigma\ list && \text{(answers)} \\
\rho_f : Env_f &= \mathsf{name} \to C \to C && \text{(function environments)} \\
\mathcal{S} &: statement \to Env_f \to Env_r \to C \to C
\end{aligned}$$

The most important semantic equation is the one for $eval_r$; evaluating an RTL adds a state to the answer. Since our interpreters don't halt, it doesn't really matter if we add the pre-state or the post-state; we've chosen the pre-state because it is simpler. The other semantic equations are more or less standard.

$$\begin{aligned}
&\mathcal{S}[\![eval_r(rtl)]\!]\rho_f\rho_r\theta &&= \lambda\sigma.\sigma :: \theta(\mathcal{R}[\![rtl]\!]\rho_r\sigma)\\
&\mathcal{S}[\![statement_1; statement_2]\!]\rho_f\rho_r\theta &&= \mathcal{S}[\![statement_1]\!]\rho_f\rho_r(\mathcal{S}[\![statement_2]\!]\rho_f\rho_r\theta)\\
&\mathcal{S}[\![\textbf{if } exp \textbf{ then } statement_1 \textbf{ else } statement_2 \textbf{ fi}]\!]\rho\theta = \\
&\qquad \lambda\sigma.\text{if}(\mathcal{E}[\![exp]\!]\rho_r\sigma)(\mathcal{S}[\![statement_1]\!]\rho_f\rho_r\theta\sigma)(\mathcal{S}[\![statement_2]\!]\rho_f\rho_r\theta\sigma)\\
&\mathcal{S}[\![\textbf{let } (rtl) \equiv memory[exp] \textbf{ in } statement \textbf{ end}]\!]\rho_f\rho_r\theta = \\
&\qquad \lambda\sigma.\mathcal{S}[\![statement]\!]\rho_f(\text{decode}(\mathcal{E}[\![exp]\!]\rho_r\sigma)\sigma rtl\ \rho_r)\theta\sigma\\
&\mathcal{S}[\![\textit{function-name}()]\!]\rho_f\rho_r\theta &&= \rho_f\,\textit{function-name}\,\theta
\end{aligned}$$

The decode function in the rule for **let** binding hides substantial work. When applied as in

$$\text{decode}\ addr\ \sigma\ rtl\ \rho_r,$$

the decode function performs the following computations:

1. It examines the machine state σ, looking at the contents of memory at address $addr$ in that state. Using the machine-dependent rules for binary representations of machine instructions, and the machine-dependent meanings of those instructions, it computes $\mathbf{I} : \Sigma \to \Sigma$, a representation of the semantics of the machine instruction located at address $addr$.

2. It identifies the free variables of *rtl*, and it chooses bindings for those variables such that given those bindings, the denotation of *rtl* is **I**.

3. Finally, it adds those bindings to environment ρ_r, and returns the new environment.

The only remaining semantic equation is the equation for function definitions. A function definition adds another function to the environment ρ_f. This equation requires a fixed-point computation to compute the new environment ρ', because functions may call themselves recursively.

$$\begin{aligned}
&\mathcal{D}[\![\textbf{fun } name() \equiv statement]\!]\rho_f \quad = \quad \rho'\\
&\quad \text{where } \rho' = \text{upde}(\rho_f, name, \mathcal{S}[\![statement]\!]\rho'(\lambda x.\bot))
\end{aligned}$$

Here, $(\lambda x.\bot)$ is the empty RTL environment, and upde adds a binding to an environment.

Given a machine state σ_0, which determines a binary program and a program counter, the sequence of states produced by the execution of that program is $\rho_f\, i\, (\lambda\sigma.\bot)\sigma_0$, where i is the name of the interpreter ($loop$ or $simple$), and ρ_f is the environment produced by processing the definitions of the interpreters. If the interpreter runs forever without halting, the usual least-fixed-point calculation produces an infinite list of states.

B Transformations

This appendix lists the transformations used in the body of the paper. This list should make it easier for readers to verify the soundness of our transformations. It may also help readers judge how difficult it would be to automate the transformations. Our purpose is to build readers' intuitions; hence, we do not attempt to be completely precise and formal.

- *Commutativity of simultaneous composition.*

  $$S \mid T \equiv T \mid S,$$

  provided S and T have disjoint locations on their left-hand sides. This should be the case for all RTLs of interest.

- *Associativity of sequential composition.*

  $$S; (T; V) \equiv (S; T); V \equiv S; T; V$$

- *Converting simultaneous composition to sequential composition.*

  $$S \mid T \equiv S; T,$$

  provided no location on any left-hand side of S appears in a guard or a right-hand-side of T. In other words, if the locations changed by S don't affect the values computed by T, S and T can be performed in sequence, rather than in parallel.

  This transformation changes the answer produced by a program; it introduces a new intermediate state. Introducing a finite number of such states is harmless, as it doesn't prevent us from proving the transition lemma. Introducing such new states may be necessary when the target machine's instruction set does not contain the source machine's instruction set; this is the rule that enables us to use a *sequence* of target-machine instructions to translate a *single* source-machine instruction.

- *Distribution of sequential composition over conditional.*

  $$\textbf{if } P \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}; S_3 \equiv \textbf{if } P \textbf{ then } S_1; S_3 \textbf{ else } S_2; S_3 \textbf{ fi}$$

  We can move a trailing statement inside a conditional, provided we replicate it on both branches. There is a slightly more restrictive rule for moving leading statements inside conditionals.

  $$S_1; \textbf{if } P \textbf{ then } S_2 \textbf{ else } S_3 \textbf{ fi} \equiv \textbf{if } P \textbf{ then } S_1; S_2 \textbf{ else } S_1; S_3 \textbf{ fi},$$

  provided no location modified by S_1 is read by P.

- *Distribution of sequential composition over* **let**. We can move a leading statement inside **let**, provided there's no clash of bound variables.

  $$S_1; \textbf{let } B \textbf{ in } S_2 \textbf{ end} \equiv \textbf{let } B \textbf{ in } S_1; S_2 \textbf{ end},$$

  provided no name free in S_1 is bound in B. There is a similar rule for trailing statements.

- *Alpha-conversion of* **let**. Given **let** B **in** S **end**, we can rename bound variables in B and S.
- *"The Trick" on Booleans.*

$$S \equiv \textbf{if } P \textbf{ then } S \textbf{ else } S$$

- *Constant folding.* We partially evaluate with respect to constants as needed. For example, **if** $[\![\textbf{true}]\!]$ **then** S_1 **else** S_2 **fi** $\equiv S_1$.
- *Dead-code elimination.* Assignments to locations that are never read can be eliminated.
- *Forward substitution (assignment).* If exactly one write to a location reaches a read of that location, we can substitute the expression written for the read from the location.
- *Forward substitution (conditional).* Given a statement **if** P **then** S_1 **else** S_2 **fi**, we may substitute $[\![\textbf{true}]\!]$ for P in S_1, and $[\![\textbf{false}]\!]$ for P in S_2, provided no conflicting change of state intervenes.
- *Introduce assignment.* If we know the value of a particular location at a particular program point, we can introduce a redundant assignment to that location. For example, we can introduce $annul := annul$.
- *Function inlining.*

$$name() \equiv S, \text{ provided } \rho_f(name) = S$$

  This rule, together with the preceding two rules, enables us to replace some calls to $loop$ with calls to $stable$.

In deriving a translator, we introduce a **case** statement, to which we have applied the following transformation.

- *Interchanging the order of tests in an* **if** *statement and a* **case** *statement.* The order of tests of an **if** statement and a **case** statement that is nested within the **if** statement can be interchanged by distributing one over all of the arms of the other.

```
if $P$ then
  case $F$ of
  | $E_1 \Longrightarrow S_1$          case $F$ of
  . . .                      | $E_1 \Longrightarrow$ if $P$ then $S_1$ else $S_e$ fi
  | $E_n \Longrightarrow S_n$      $\equiv$    . . .
  end                        | $E_n \Longrightarrow$ if $P$ then $S_n$ else $S_e$ fi
else                         end
  $S_e$
fi
```

C Complete SPARC V8 translation algorithm

We document the full SPARC V8 algorithm after applying transformations as described in §4.2. Note that for the cases where $loop()$ is derived, we emit an `unroll loop(pc,nPC)` call as per suggested in §7.

```
⟨final sparc translator⟩≡
  **fun** $trans(pc_s, pc_t) =$
    $codemap(pc_s) \equiv pc_t$
    **let** $I$ **as** $(b_I - > nPC := target_I \mid annul := a_I \mid I_c) = src[pc_s]$
    **in case** $class(I)$ **of**
    $\mid NCT \Longrightarrow /* \neg b_I and \neg a_I */$
            $pc_t := emit(pc_t, \overline{I_c});$
            $trans(succ_s(pc_s), pc_t)$

    $\mid SKIP \Longrightarrow /* \neg b_I and a_I */$
            $pc_t := emit(pc_t, \overline{I_c});$
            $trans(succ_s(succ_s(pc_s)), pc_t)$

    $\mid SU \Longrightarrow /* b_I and a_I */$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, PC := codemap(target_I));$
            $queueForTranslation(target_I, codemap(target_I))$

    $\mid SD \Longrightarrow /* b_I and \neg a_I */$
    $\mid DD \Longrightarrow /* b_I and \neg a_I */$
            **let** $I'$ **as** $(b_{I'} - > nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) = src[succ_s(pc_s)]$
            **in case** $class(I')$ **of**
            $\mid NCT \Longrightarrow /* \neg b_{I'} and \neg a_{I'} */$
                    $pc_t := emit(pc_t, \overline{I_c});$
                    $pc_t := emit(pc_t, \overline{I'_c});$
                    $pc_t := emit(pc_t, PC := codemap(target_I));$
                    $queueForTranslation(target_I, codemap(target_I));$

            $\mid SKIP \Longrightarrow /* \neg b_{I'} and a_{I'} */$
                    $pc_t := emit(pc_t, \overline{I_c});$
                    $pc_t := emit(pc_t, \overline{I'_c});$
                    $pc_t := emit(pc_t, PC := codemap(succ_s(target_I)));$
                    $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)));$

            $\mid SU \Longrightarrow /* b_{I'} and a_{I'} */$
                    $pc_t := emit(pc_t, \overline{I_c});$
                    $pc_t := emit(pc_t, \overline{I'_c});$
                    $pc_t := emit(pc_t, PC := codemap(target_{I'}));$
            $queueForTranslation(target_{I'}, codemap(target_{I'}));$

        $| \ SD \Longrightarrow$
        $| \ DD \Longrightarrow / * \, b_{I'} and \neg a_{I'} * /$
            $pc_t := emit(pc_t, \overline{I_c});$
            $unrollloop(pc_s, nPC) / * I'' * /$

        $| \ SCD \Longrightarrow / * \, b_{I'} and \neg a_{I'} * // * \neg b_{I'} and \neg a_{I'} * /$
            **local** $bb := newBlock();$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{b_{I'}} - > PC := bb);$
            $bb := emit(bb,$ "unroll loop(pc_s, nPC)"$);$
            $pc_t := emit(pc_t, \overline{I'_c});$
            $pc_t := emit(pc_t, PC := codemap(target_I));$
            $queueForTranslation(target_I, codemap(target_I));$

        $| \ SCDA \Longrightarrow / * \, b_{I'} and \neg a_{I'} * // * \neg b_{I'} and a_{I'} * /$
            **local** $bb := newBlock();$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{b_{I'}} - > PC := bb);$
            $bb := emit(bb,$ "unroll loop(pc_s, nPC)"$);$
            $pc_t := emit(pc_t, \overline{I'_c});$
            $pc_t := emit(pc_t, PC := codemap(succ_s(target_I)));$
            $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)));$

    $| \ SCD \Longrightarrow / * \, b_I and \neg a_I * /$
        **let** $I'$ **as** $(b_{I'} - > nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) = src[succ_s(pc_s)]$
        **in case** $class(I')$ **of**
        $| \ NCT \Longrightarrow / * \, (\neg b_{I'} and \neg a_{I'}) or (\neg b_I and \neg a_I) * /$
            **local** $bb := newBlock();$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{b_I} - > PC := bb);$
            $bb := emit(bb, \overline{I'_c});$
            $bb := emit(bb, PC := codemap(target_I));$
            $queueForTranslation(target_I, codemap(target_I))$
            $trans(succ_s(pc_s), pc_t)$

        $| \ SKIP \Longrightarrow / * \neg b_{I'} and a_{I'} or \neg b_I and \neg a_I * /$
            **local** $bb := newBlock();$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{b_I} - > PC := bb);$
            $bb := emit(bb, \overline{I'_c});$
            $bb := emit(bb, PC := codemap(succ_s(target_I)));$
        $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)))$
        $trans(succ_s(pc_s), pc_t)$

$|\ SU \Longrightarrow /*\ b_{I'}\, and\, a_{I'}\, or \neg b_I\, and \neg a_I\ */$
        local $bb := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
        $bb := emit(bb, \overline{I'_c});$
        $bb := emit(bb, PC := codemap(target_{I'}));$
        $queueForTranslation(target_{I'}, codemap(target_{I'}))$
        $trans(succ_s(pc_s), pc_t)$

$|\ SD \Longrightarrow$
$|\ DD \Longrightarrow /*\ b_{I'}\, and \neg a_{I'}\, or \neg b_I\, and \neg a_I\ */$
        local $bb := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
        $bb := emit(bb,$ "unroll loop(pc_s, nPC)"$); /* I'' */$
        $trans(succ_s(pc_s), pc_t)$

$|\ SCD \Longrightarrow /*\ b_{I'}\, and \neg a_{I'}\, or \neg b_I\, and \neg a_I\ */$
        $/* \neg b_{I'}\, and \neg a_{I'}\ */$
        local $bb := newBlock();$
        local $bb' := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
        $bb := emit(bb, \overline{b_{I'}} -> PC := bb');$
        $bb' := emit(bb',$ "unroll loop(pc_s, nPC)"$);$
        $bb := emit(bb, \overline{I'_c});$
        $bb := emit(bb, PC := codemap(target_I));$
        $queueForTranslation(target_I, codemap(target_I))$
        $trans(succ_s(pc_s), pc_t)$

$|\ SCDA \Longrightarrow /*\ b_{I'}\, and \neg a_{I'}\ */$
        $/* \neg b_{I'}\, and\, a_{I'}\ */$
        $/* \neg b_I\, and \neg a_I\ */$
        local $bb := newBlock();$
        local $bb' := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
        $pc_t := emit(pc_t, \overline{b_{I'}} -> PC := bb');$
        $bb' := emit(bb',$ "unroll loop(pc_s, nPC)"$);$
        $bb := emit(bb, \overline{I'_c});$
        $bb := emit(bb, PC := codemap(succ_s(target_I)));$
        $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)));$
        $trans(succ_s(pc_s), pc_t)$
    **end**

$| SCDA \Longrightarrow /* b_I and \neg a_I */$
    **let** $I'$ **as** $(b_{I'} -> nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) = src[succ_s(pc_s)]$
    **in case** $class(I')$ **of**
    $| NCT \Longrightarrow /* \neg b_{I'} and \neg a_{I'} *//* \neg b_I and a_I */$
        **local** $bb := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
        $bb := emit(bb, \overline{I'_c});$
        $bb := emit(bb, PC := codemap(target_I));$
        $queueForTranslation(target_I, codemap(target_I));$
        $trans(succ_s(succ_s(pc_s)), pc_t)$

    $| SKIP \Longrightarrow /* \neg b_{I'} and a_{I'} *//* \neg b_I and a_I */$
        **local** $bb := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(\overline{b_I} -> PC := bb);$
        $bb := emit(bb, \overline{I'_c});$
        $bb := emit(bb, PC := codemap(succ_s(target_I)));$
        $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)));$
        $trans(succ_s(succ_s(pc_s)), pc_t)$

    $| SU \Longrightarrow /* b_{I'} and a_{I'} *//* \neg b_I and a_I */$
        **local** $bb := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, bb);$
        $bb := emit(bb, \overline{I'_c});$
        $bb := emit(bb, PC := codemap(target_{I'}));$
        $queueForTranslation(target_{I'}, codemap(target_{I'}));$
        $trans(succ_s(succ_s(pc_s)), pc_t)$

    $| SD \Longrightarrow$
    $| DD \Longrightarrow /* b_{I'} and \neg a_{I'} *//* \neg b_I and a_I */$
        **local** $bb := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
        $bb := emit(bb,$ "unroll loop(pc_s, nPC)"$); /* I'' */$
        $trans(succ_s(succ_s(pc_s)), pc_t)$
    $| \; SCD \Longrightarrow /* \, b_{I'} and \neg a_{I'} */ /* \neg b_{I'} and \neg a_{I'} */$
        $/* \neg b_I and a_I */$
        **local** $bb := newBlock()$;
        **local** $bb' := newBlock()$;
        $pc_t := emit(pc_t, \overline{I_c})$;
        $pc_t := emit(pc_t, bb)$;
        $bb := emit(bb, \overline{b_{I'}} -> PC := bb')$;
        $bb' := emit(bb',$ "unroll loop(pc_s, nPC)"$)$;
        $bb := emit(bb, \overline{I'_c})$;
        $bb := emit(bb, PC := codemap(target_I))$;
        $queueForTranslation(target_I, codemap(target_I))$;
        $trans(succ_s(succ_s(pc_s)), pc_t)$

    $| \; SCDA \Longrightarrow /* \, b_{I'} and \neg a_{I'} */ /* \neg b_{I'} and a_{I'} */$
        $/* \neg b_I and a_I */$
        **local** $bb := newBlock()$;
        **local** $bb' := newBlock()$;
        $pc_t := emit(pc_t, \overline{I_c})$;
        $pc_t := emit(pc_t, \overline{b_I} -> PC := bb)$;
        $bb := emit(bb, \overline{b_{I'}} -> PC := bb')$;
        $bb' := emit(bb',$ "unroll loop(pc_s, nPC)"$)$;
        $bb := emit(bb, \overline{I'_c})$;
        $bb := emit(bb, PC := codemap(succ_s(target_I)))$;
        $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)))$;
        $trans(succ_s(succ_s(pc_s)), pc_t)$
    **end**
**end**
```

D Complete PA-RISC translation algorithm

We sketch the transformation steps followed in order to derive a translator for the removal of PA-RISC delayed branches. Starting from the original translation algorithm $trans$ in §3.1, we instantiate it to the classes of instructions of the PA-RISC V1.1 architecture, as documented in §5.2, leading to the following initial algorithm.

```
⟨pa-risc translator⟩≡
  fun $trans(pc_s, pc_t) =$
    $codemap(pc_s) \equiv pc_t$
    let $I$ as $(b_I -> nPC := target_I \mid annul := a_I \mid I_c) = src[pc_s]$
    in case $class(I)$ of
    $\mid NCT \Longrightarrow nonBranching, nonAnnulling$
    $\mid NCTA \Longrightarrow$ if $(cond)$ then
              $nonBranching, annulling$
           else
              $nonBranching, nonAnnulling$
    $\mid SU \Longrightarrow branching, annulling$
    $\mid DU \Longrightarrow branching, annulling$
    $\mid SD \Longrightarrow branching, nonAnnulling$
    $\mid DD \Longrightarrow branching, nonAnnulling$
    $\mid SCD \Longrightarrow$ if $(b_I)$ then
              $branching, nonAnnulling$
           else
              $nonBranching, nonAnnulling$
    $\mid SCDA \Longrightarrow$ if $(b_I)$ then
              $branching, nonAnnulling$
           else
              $nonBranching, annulling$
    $\mid SCDA_> \Longrightarrow$ if $(b_I)$ then
              $branching, annulling$
           else
              $nonBranching, nonAnnulling$
    end
```

expanding each of the arms of the **case** statement, we get:

```
⟨pa-risc translator⟩+≡
  fun $trans(pc_s, pc_t) =$
    $codemap(pc_s) \equiv pc_t$
    let $I$ as $(b_I - > nPC := target_I \mid annul := a_I \mid I_c) = src[pc_s]$
    in case $class(I)$ of
    $\mid NCT \Longrightarrow /* \neg b_I and \neg a_I */$
          $pc_t := emit(pc_t, \overline{I_c});$
          $trans(succ_s(pc_s), pc_t)$

    $\mid NCTA \Longrightarrow /* \neg b_I and a_I */$
          if $(cond)$ then
            $pc_t := emit(pc_t, \overline{I_c});$
            $trans(succ_s(succ_s(pc_s)), pc_t)$

          $/* \neg b_I and \neg a_I */$
          else
            $pc_t := emit(pc_t, \overline{I_c});$
            $trans(succ_s(pc_s), pc_t)$
          fi

    $\mid SU \Longrightarrow /* b_I and a_I */$
          $pc_t := emit(pc_t, \overline{I_c});$
          $pc_t := emit(pc_t, PC := codemap(target_I));$
          $queueForTranslation(target_I, codemap(target_I))$

    $\mid DU \Longrightarrow /* b_I and a_I */$
          $/* Same$ as $SU only that the target address target_I is known$
            $dynamically */$

    $\mid SD \Longrightarrow /* b_I and \neg a_I */$
          let $I'$ as $(b_{I'} - > nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) = src[succ_s(pc_s)]$
          in case $class(I')$ of
          $\mid NCT \Longrightarrow /* \neg b_{I'} and \neg a_{I'} */$
                $pc_t := emit(pc_t, \overline{I_c});$
                $pc_t := emit(pc_t, \overline{I'_c});$
                $pc_t := emit(pc_t, PC := codemap(target_I));$
                $queueForTranslation(target_I, codemap(target_I))$

          $\mid NCTA \Longrightarrow /* \neg b_{I'} and a_{I'} */$
                if $(cond)$ then
                  $pc_t := emit(pc_t, \overline{I_c});$
                  $pc_t := emit(pc_t, \overline{I'_c});$
```

```
        $pc_t := emit(pc_t, PC := codemap(succ_s(target_I)))$;
        $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)))$
      $/ * \neg b_{I'} and \neg a_{I'} * /$
      **else**
        $pc_t := emit(pc_t, \overline{I_c})$;
        $pc_t := emit(pc_t, \overline{I'_c})$;
        $pc_t := emit(pc_t, PC := codemap(target_I))$;
        $queueForTranslation(target_I, codemap(target_I))$
      **fi**

  $| SU \Longrightarrow$
  $| DU \Longrightarrow / * b_{I'} and a_{I'} * /$
      $pc_t := emit(pc_t, \overline{I_c})$;
      $pc_t := emit(pc_t, \overline{I'_c})$;
      $pc_t := emit(pc_t, PC := codemap(target_{I'}))$;
      $queueForTranslation(target_{I'}, codemap(target_{I'}))$

  $| SD \Longrightarrow$
  $| DD \Longrightarrow / * b_{I'} and \neg a_{I'} * /$
      $pc_t := emit(pc_t, \overline{I_c})$;
      $unrollloop(pc_s, nPC) / * I'' * /$

  $| SCD \Longrightarrow / * b_{I'} and \neg a_{I'} * /$
      **if** $(b_{I'})$ **then**
        $pc_t := emit(pc_t, \overline{I_c})$;
        $unrollloop(pc_s, nPC)$

      $/ * \neg b_{I'} and \neg a_{I'} * /$
      **else**
        $pc_t := emit(pc_t, \overline{I_c})$;
        $pc_t := emit(pc_t, \overline{I'_c})$;
        $pc_t := emit(pc_t, PC := codemap(target_I))$;
        $queueForTranslation(target_I, codemap(target_I))$
      **fi**

  $| SCDA \Longrightarrow / * b_{I'} and \neg a_{I'} * /$
      **if** $(b_{I'})$ **then**
        $pc_t := emit(pc_t, \overline{I_c})$;
        $unrollloop(pc_s, nPC)$

      $/ * \neg b_{I'} and a_{I'} * /$
      **else**
        $pc_t := emit(pc_t, \overline{I_c})$;
```

```
            $pc_t := emit(pc_t, \overline{I'_c});$
            $pc_t := emit(pc_t, PC := codemap(succ_s(target_I)));$
            $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)));$
          **fi**

    $| \; SCDA_> => /* b_{I'} and a_{I'} */$
          **if** $(b_{I'})$ **then**
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{I'_c});$
            $pc_t := emit(pc_t, PC := codemap(target_{I'}));$
            $queueForTranslation(target_{I'}, codemap(target_{I'}))$

          $/* \neg b_{I'} and \neg a_{I'} */$
          **else**
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{I'_c});$
            $pc_t := emit(pc_t, PC := codemap(target_I));$
            $queueForTranslation(target_I, codemap(target_I))$
          **fi**
    **end**

$| \; DD \Longrightarrow /* b_I and \neg a_I */$
    $/* Same$ **as** $SD$ **case** $only that the target address target_I is$
      $known dynamically */$

$| \; SCD \Longrightarrow /* b_I and \neg a_I */$
    **if** $(b_I)$ **then**
      **let** $I'$ **as** $(b_{I'} -> nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) = src[succ_s(pc_s)]$
      **in case** $class(I')$ **of**
      $| \; NCT \Longrightarrow /* \neg b_{I'} and \neg a_{I'} */$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{I'_c});$
            $pc_t := emit(pc_t, PC := codemap(target_I));$
            $queueForTranslation(target_I, codemap(target_I))$

      $| \; NCTA \Longrightarrow /* \neg b_{I'} and a_{I'} */$
            **if** $(cond)$ **then**
              $pc_t := emit(pc_t, \overline{I_c});$
              $pc_t := emit(pc_t, \overline{I'_c});$
              $pc_t := emit(pc_t, PC := codemap(succ_s(target_I)));$
              $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)))$
            $/* \neg b_{I'} and \neg a_{I'} */$
            **else**
```

```
        $pc_t := emit(pc_t, \overline{I_c})$;
        $pc_t := emit(pc_t, \overline{I'_c})$;
        $pc_t := emit(pc_t, PC := codemap(target_I))$;
        $queueForTranslation(target_I, codemap(target_I))$
      fi

$| SU \Longrightarrow$
$| DU \Longrightarrow /* b_{I'} and a_{I'} */$
      $pc_t := emit(pc_t, \overline{I_c})$;
      $pc_t := emit(pc_t, \overline{I'_c})$;
      $pc_t := emit(pc_t, PC := codemap(target_{I'}))$;
      $queueForTranslation(target_{I'}, codemap(target_{I'}))$

$| SD \Longrightarrow$
$| DD \Longrightarrow /* b_{I'} and \neg a_{I'} */$
      $pc_t := emit(pc_t, \overline{I_c})$;
      $unrollloop(pc_s, nPC) /* I'' */$

$| SCD \Longrightarrow /* b_{I'} and \neg a_{I'} */$
      if $(b_{I'})$ then
        $pc_t := emit(pc_t, \overline{I_c})$;
        $unrollloop(pc_s, nPC)$

      $/* \neg b_{I'} and \neg a_{I'} */$
      else
        $pc_t := emit(pc_t, \overline{I_c})$;
        $pc_t := emit(pc_t, \overline{I'_c})$;
        $pc_t := emit(pc_t, PC := codemap(target_I))$;
        $queueForTranslation(target_I, codemap(target_I))$
      fi

$| SCDA \Longrightarrow /* b_{I'} and \neg a_{I'} */$
      if $(b_{I'})$ then
        $pc_t := emit(pc_t, \overline{I_c})$;
        $unrollloop(pc_s, nPC)$

      $/* \neg b_{I'} and a_{I'} */$
      else
        $pc_t := emit(pc_t, \overline{I_c})$;
        $pc_t := emit(pc_t, \overline{I'_c})$;
        $pc_t := emit(pc_t, PC := codemap(succ_s(target_I)))$;
        $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)))$;
      fi
```

```
        | $SCDA_{>} => /* b_{I'} and a_{I'} */$
            **if** $(b_{I'})$ **then**
              $pc_t := emit(pc_t, \overline{I_c});$
              $pc_t := emit(pc_t, \overline{I'_c});$
              $pc_t := emit(pc_t, PC := codemap(target_{I'}));$
              $queueForTranslation(target_{I'}, codemap(target_{I'}))$

            $/* \neg b_{I'} and \neg a_{I'} */$
            **else**
              $pc_t := emit(pc_t, \overline{I_c});$
              $pc_t := emit(pc_t, \overline{I'_c});$
              $pc_t := emit(pc_t, PC := codemap(target_I));$
              $queueForTranslation(target_I, codemap(target_I))$
            **fi**
        **end**

      $/* \neg b_I and \neg a_I */$
      **else**
        $pc_t := emit(pc_t, \overline{I_c});$
        $trans(succ_s(pc_s), pc_t)$
      **fi**

  | $SCDA \Longrightarrow /* b_I and \neg a_I */$
      **if** $(b_I)$ **then**
        **let** $I'$ **as** $(b_{I'} -> nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) = src[succ_s(pc_s)]$
        **in case** $class(I')$ **of**
        | $NCT \Longrightarrow /* \neg b_{I'} and \neg a_{I'} */$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{I'_c});$
            $pc_t := emit(pc_t, PC := codemap(target_I));$
            $queueForTranslation(target_I, codemap(target_I))$

        | $NCTA \Longrightarrow /* \neg b_{I'} and a_{I'} */$
            **if** $(cond)$ **then**
              $pc_t := emit(pc_t, \overline{I_c});$
              $pc_t := emit(pc_t, \overline{I'_c});$
              $pc_t := emit(pc_t, PC := codemap(succ_s(target_I)));$
              $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)))$
            $/* \neg b_{I'} and \neg a_{I'} */$
            **else**
              $pc_t := emit(pc_t, \overline{I_c});$
              $pc_t := emit(pc_t, \overline{I'_c});$
      $pc_t := emit(pc_t, PC := codemap(target_I));$
      $queueForTranslation(target_I, codemap(target_I))$
    **fi**

$| SU \Longrightarrow$
$| DU \Longrightarrow /* b_{I'} and a_{I'} */$
    $pc_t := emit(pc_t, \overline{I_c});$
    $pc_t := emit(pc_t, \overline{I'_c});$
    $pc_t := emit(pc_t, PC := codemap(target_{I'}));$
    $queueForTranslation(target_{I'}, codemap(target_{I'}))$

$| SD \Longrightarrow$
$| DD \Longrightarrow /* b_{I'} and \neg a_{I'} */$
    $pc_t := emit(pc_t, \overline{I_c});$
    $unrollloop(pc_s, nPC) /* I'' */$

$| SCD \Longrightarrow /* b_{I'} and \neg a_{I'} */$
    **if** $(b_{I'})$ **then**
      $pc_t := emit(pc_t, \overline{I_c});$
      $unrollloop(pc_s, nPC)$

    $/* \neg b_{I'} and \neg a_{I'} */$
    **else**
      $pc_t := emit(pc_t, \overline{I_c});$
      $pc_t := emit(pc_t, \overline{I'_c});$
      $pc_t := emit(pc_t, PC := codemap(target_I));$
      $queueForTranslation(target_I, codemap(target_I))$
    **fi**

$| SCDA \Longrightarrow /* b_{I'} and \neg a_{I'} */$
    **if** $(b_{I'})$ **then**
      $pc_t := emit(pc_t, \overline{I_c});$
      $unrollloop(pc_s, nPC)$

    $/* \neg b_{I'} and a_{I'} */$
    **else**
      $pc_t := emit(pc_t, \overline{I_c});$
      $pc_t := emit(pc_t, \overline{I'_c});$
      $pc_t := emit(pc_t, PC := codemap(succ_s(target_I)));$
      $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)));$
    **fi**

$| SCDA_> => /* b_{I'} and a_{I'} */$
```

```
            if ($b_{I'}$) then
              $pc_t := emit(pc_t, \overline{I_c});$
              $pc_t := emit(pc_t, \overline{I'_c});$
              $pc_t := emit(pc_t, PC := codemap(target_{I'}));$
              $queueForTranslation(target_{I'}, codemap(target_{I'}))$

            $/ * \neg b_{I'} and \neg a_{I'} * /$
            else
              $pc_t := emit(pc_t, \overline{I_c});$
              $pc_t := emit(pc_t, \overline{I'_c});$
              $pc_t := emit(pc_t, PC := codemap(target_I));$
              $queueForTranslation(target_I, codemap(target_I))$
            fi
      end

      $/ * \neg b_I and a_I * /$
      else
        $pc_t := emit(pc_t, \overline{I_c});$
        $trans(succ_s(succ_s(pc_s)), pc_t)$
      fi

$| \ SCDA_> => / * b_{I'} and a_{I'} * /$
      if ($b_{I'}$) then
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{I'_c});$
        $pc_t := emit(pc_t, PC := codemap(target_{I'}));$
        $queueForTranslation(target_{I'}, codemap(target_{I'}))$

      $/ * \neg b_{I'} and \neg a_{I'} * /$
      else
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{I'_c});$
        $pc_t := emit(pc_t, PC := codemap(target_I));$
        $queueForTranslation(target_I, codemap(target_I))$
      fi
end
```

We perform the following transformations on the code:

- For the (SD,SCD), (SD,SCDA) and (SD,SCDAgt) cases, we apply the distribution of sequential composition over `let` to the statement $pc_t := emit(pc_t, \overline{I_c})$;
- For the (SD,SCD), (SD,SCDA) and (SD,SCDAgt) cases, we create a new memory address (denoted `bb`) to hold the code on one side of the branch and generate the code to support the branch,
- For the NCTA, SCD, SCDA and SCDAgt cases, we apply the distribution of sequential composition over conditional,
- For the SCD, SCDA and SCDAgt cases, we interchange the order of test in an `if` and `case` statement and we apply the inverse of the distribution of sequential composition over `let` to the statement $pc_t := emit(pc_t, \overline{I_c})$;, and
- For the SCD, SCDA and SCDAgt cases, we create a new memory address (denoted `bb` and `bb'`) and replace the conditional branch on b_I with appropriate code for both branches.

obtaining the following final version of the algorithm:

```
⟨final pa-risc translator⟩≡
 fun $trans(pc_s, pc_t) =$
   $codemap(pc_s) \equiv pc_t$
   let $I$ as $(b_I - > nPC := target_I \mid annul := a_I \mid I_c) = src[pc_s]$
   in case $class(I)$ of
   $\mid NCT \Longrightarrow /* \neg b_I and \neg a_I */$
         $pc_t := emit(pc_t, \overline{I_c});$
         $trans(succ_s(pc_s), pc_t)$

   $\mid NCTA \Longrightarrow /* \neg b_I and a_I * // * \neg b_I and \neg a_I */$
         $pc_t := emit(pc_t, \overline{I_c});$
         if $(cond)$ then
           $trans(succ_s(succ_s(pc_s)), pc_t)$
         else
           $trans(succ_s(pc_s), pc_t)$

   $\mid SU \Longrightarrow /* b_I and a_I */$
         $pc_t := emit(pc_t, \overline{I_c});$
         $pc_t := emit(pc_t, PC := codemap(target_I));$
         $queueForTranslation(target_I, codemap(target_I))$

   $\mid DU \Longrightarrow /* b_I and a_I */$
         $/* Same$ as $SU only that the target address target_I is known$
           $dynamically */$

   $\mid SD \Longrightarrow /* b_I and \neg a_I */$
```

```
**let** $I'$ **as** $(b_{I'} - > nPC := target_{I'} \mid annul := a_{I'} \mid {I'}_c) = src[succ_s(pc_s)]$
**in case** $class(I')$ **of**
$\mid NCT \Longrightarrow /* \neg b_{I'}\, and \neg a_{I'} */$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{{I'}_c});$
        $pc_t := emit(pc_t, PC := codemap(target_I));$
        $queueForTranslation(target_I, codemap(target_I))$

$\mid NCTA \Longrightarrow /* \neg b_{I'}\, and\, a_{I'} */$
        **if** $(cond)$ **then**
          $pc_t := emit(pc_t, \overline{I_c});$
          $pc_t := emit(pc_t, \overline{{I'}_c});$
          $pc_t := emit(pc_t, PC := codemap(succ_s(target_I)));$
          $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)))$
        $/* \neg b_{I'}\, and \neg a_{I'} */$
        **else**
          $pc_t := emit(pc_t, \overline{I_c});$
          $pc_t := emit(pc_t, \overline{{I'}_c});$
          $pc_t := emit(pc_t, PC := codemap(target_I));$
          $queueForTranslation(target_I, codemap(target_I))$
        **fi**

$\mid SU \Longrightarrow$
$\mid DU \Longrightarrow /* b_{I'}\, and\, a_{I'} */$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{{I'}_c});$
        $pc_t := emit(pc_t, PC := codemap(target_{I'}));$
        $queueForTranslation(target_{I'}, codemap(target_{I'}))$

$\mid SD \Longrightarrow$
$\mid DD \Longrightarrow /* b_{I'}\, and \neg a_{I'} */$
        $pc_t := emit(pc_t, \overline{I_c});$
        $unrollloop(pc_s, nPC) /* I'' */$

$\mid SCD \Longrightarrow /* b_{I'}\, and \neg a_{I'} * // * \neg b_{I'}\, and \neg a_{I'} */$
          **local** $bb := newBlock();$
          $pc_t := emit(pc_t, \overline{I_c});$
          $pc_t := emit(pc_t, \overline{b_{I'}} - > PC := bb);$
          $bb := emit(bb,$ "unroll loop(pc_s, nPC)"$);$
          $pc_t := emit(pc_t, \overline{{I'}_c});$
          $pc_t := emit(pc_t, PC := codemap(target_I));$
          $queueForTranslation(target_I, codemap(target_I))$
        $\mid SCDA \Longrightarrow / * b_{I'} and \neg a_{I'} * // * \neg b_{I'} and a_{I'} * /$
                **local** $bb := newBlock()$;
                $pc_t := emit(pc_t, \overline{I_c})$;
                $pc_t := emit(pc_t, \overline{b_{I'}} -> PC := bb)$;
                $bb := emit(bb,$ "unroll loop(pc_s, nPC)"$)$;
                $pc_t := emit(pc_t, \overline{I'_c})$;
                $pc_t := emit(pc_t, PC := codemap(succ_s(target_I)))$;
                $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)))$;

        $\mid SCDA_> => / * b_{I'} and a_{I'} * // * \neg b_{I'} and \neg a_{I'} * /$
                **local** $bb := newBlock()$;
                $pc_t := emit(pc_t, \overline{I_c})$;
                $pc_t := emit(pc_t, \overline{b_{I'}} -> PC := bb)$;
                $bb := emit(bb, \overline{I'_c})$;
                $bb := emit(bb, PC := codemap(target_{I'}))$;
                $queueForTranslation(target_{I'}, codemap(target_{I'}))$
                $pc_t := emit(pc_t, \overline{I'_c})$;
                $pc_t := emit(pc_t, PC := codemap(target_I))$;
                $queueForTranslation(target_I, codemap(target_I))$
        **end**

    $\mid DD \Longrightarrow / * b_I and \neg a_I * /$
        $/ * Same$ **as** $SD$ **case** $only that the target address target_I is$
          $known dynamically * /$

    $\mid SCD \Longrightarrow / * b_I and \neg a_I * /$
        **let** $I'$ **as** $(b_{I'} -> nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) = src[succ_s(pc_s)]$
        **in case** $class(I')$ **of**
          $\mid NCT \Longrightarrow / * (\neg b_{I'} and \neg a_{I'}) or (\neg b_I and \neg a_I) * /$
                **local** $bb := newBlock()$;
                $pc_t := emit(pc_t, \overline{I_c})$;
                $pc_t := emit(pc_t, \overline{b_I} -> PC := bb)$;
                $bb := emit(bb, \overline{I'_c})$;
                $bb := emit(bb, PC := codemap(target_I))$;
                $queueForTranslation(target_I, codemap(target_I))$;
                $trans(succ_s(pc_s), pc_t)$

          $\mid NCTA \Longrightarrow / * (\neg b_{I'} and a_{I'}) or (\neg b_{I'} and \neg a_{I'}) or (\neg b_I and \neg a_I) * /$
                **local** $bb := newBlock()$;
                $pc_t := emit(pc_t, \overline{I_c})$;
                $pc_t := emit(pc_t, \overline{b_I} -> PC := bb)$;
                $bb := emit(bb, \overline{I'_c})$;
                **if** $(cond)$ **then**
        $bb := emit(bb, PC := codemap(succ_s(target_I)));$
        $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)))$
      **else**
        $bb := emit(bb, PC := codemap(target_I));$
        $queueForTranslation(target_I, codemap(target_I))$
      **fi**
      $trans(succ_s(pc_s), pc_t)$

| $SU \Longrightarrow$
| $DU \Longrightarrow /*(b_{I'} and a_{I'}) or (\neg b_I and \neg a_I)*/$
      **local** $bb := newBlock();$
      $pc_t := emit(pc_t, \overline{I_c});$
      $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
      $bb := emit(bb, \overline{I'_c});$
      $bb := emit(bb, PC := codemap(target_{I'}));$
      $queueForTranslation(target_{I'}, codemap(target_{I'}))$
      $trans(succ_s(pc_s), pc_t)$

| $SD \Longrightarrow$
| $DD \Longrightarrow /*(b_{I'} and \neg a_{I'}) or (\neg b_I and \neg a_I)*/$
      **local** $bb := newBlock();$
      $pc_t := emit(pc_t, \overline{I_c});$
      $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
      $unrollloop(pc_s, nPC) /* I'' */$
      $trans(succ_s(pc_s), pc_t)$

| $SCD \Longrightarrow /*(b_{I'} and \neg a_{I'}) or (\neg b_{I'} and \neg a_{I'}) or (\neg b_I and \neg a_I)*/$
      **local** $bb := newBlock();$
      **local** $bb' := newBlock();$
      $pc_t := emit(pc_t, \overline{I_c});$
      $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
      $bb := emit(bb, \overline{b_{I'}} -> PC := bb');$
      $bb' := emit(bb',$ "unroll loop(pc_s, nPC)"$);$
      $bb := emit(bb, \overline{I'_c});$
      $bb := emit(bb, PC := codemap(target_I));$
      $queueForTranslation(target_I, codemap(target_I))$
      $trans(succ_s(pc_s), pc_t)$

| $SCDA \Longrightarrow /*(b_{I'} and \neg a_{I'}) or (\neg b_{I'} and a_{I'}) or (\neg b_I and \neg a_I)*/$
      **local** $bb := newBlock();$
      **local** $bb' := newBlock();$
      $pc_t := emit(pc_t, \overline{I_c});$
      $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
```

```
            $bb := emit(bb, \overline{b_{I'}} -> PC := bb');$
            $bb' := emit(bb',$ "unroll loop(pc_s, nPC)"$);$
            $bb := emit(bb, \overline{I'_c});$
            $bb := emit(bb, PC := codemap(succ_s(target_I)));$
            $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)));$
            $trans(succ_s(pc_s), pc_t)$

        $| SCDA_> => /*(b_{I'} and a_{I'}) or (\neg b_{I'} and \neg a_{I'}) or (\neg b_I and \neg a_I)*/$
            **local** $bb := newBlock();$
            **local** $bb' := newBlock();$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
            $bb := emit(bb, \overline{I'_c});$
            $bb := emit(bb, \overline{b_{I'}} -> PC := bb');$
            $bb' := emit(bb', PC := codemap(target_{I'}));$
            $queueForTranslation(target_{I'}, codemap(target_{I'}))$
            $bb := emit(bb, PC := codemap(target_I));$
            $queueForTranslation(target_I, codemap(target_I))$
            $trans(succ_s(pc_s), pc_t)$
        **end**

    $| SCDA \Longrightarrow /*b_I and \neg a_I*/$
      **let** $I'$ **as** $(b_{I'} -> nPC := target_{I'} \mid annul := a_{I'} \mid I'_c) = src[succ_s(pc_s)]$
        **in case** $class(I')$ **of**
        $| NCT \Longrightarrow /*(\neg b_{I'} and \neg a_{I'}) or (\neg b_I and a_I)*/$
            **local** $bb := newBlock();$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
            $bb := emit(bb, \overline{I'_c});$
            $bb := emit(bb, PC := codemap(target_I));$
            $queueForTranslation(target_I, codemap(target_I));$
            $trans(succ_s(succ_s(pc_s)), pc_t)$

        $| NCTA \Longrightarrow /*(\neg b_{I'} and a_{I'}) or (\neg b_{I'} and \neg a_{I'}) or (\neg b_I and a_I)*/$
            **local** $bb := newBlock();$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{b_I} -> PC := bb);$
            $bb := emit(bb, \overline{I'_c});$
            **if** $(cond)$ **then**
              $bb := emit(bb, PC := codemap(succ_s(target_I)));$
              $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)))$
            **else**
              $bb := emit(bb, PC := codemap(target_I));$
```

```
          $queueForTranslation(target_I, codemap(target_I))$
        **fi**
        $trans(succ_s(succ_s(pc_s)), pc_t)$

| $SU \Longrightarrow$
| $DU \Longrightarrow /*(b_{I'}\,and\,a_{I'})\,or(\neg b_I\,and\,a_I)*/$
        **local** $bb := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{b_I} - > PC := bb);$
        $bb := emit(bb, \overline{I'_c});$
        $bb := emit(bb, PC := codemap(target_{I'}));$
        $queueForTranslation(target_{I'}, codemap(target_{I'}));$
        $trans(succ_s(succ_s(pc_s)), pc_t)$

| $SD \Longrightarrow$
| $DD \Longrightarrow /*(b_{I'}\,and\neg a_{I'})\,or(\neg b_I\,and\,a_I)*/$
        **local** $bb := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{b_I} - > PC := bb);$
        $bb := emit(bb,$ "unroll loop(pc_s, nPC)"$); /*I''*/$
        $trans(succ_s(succ_s(pc_s)), pc_t)$

| $SCD \Longrightarrow /*(b_{I'}\,and\neg a_{I'})\,or(\neg b_{I'}\,and\neg a_{I'})\,or(\neg b_I\,and\,a_I)*/$
        **local** $bb := newBlock();$
        **local** $bb' := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{b_I} - > PC := bb);$
        $bb := emit(bb, \overline{b_{I'}} - > PC := bb');$
        $bb' := emit(bb',$ "unroll loop(pc_s, nPC)"$);$
        $bb := emit(bb, \overline{I'_c});$
        $bb := emit(bb, PC := codemap(target_I));$
        $queueForTranslation(target_I, codemap(target_I));$
        $trans(succ_s(succ_s(pc_s)), pc_t)$

| $SCDA \Longrightarrow /*(b_{I'}\,and\neg a_{I'})\,or(\neg b_{I'}\,and\,a_{I'})\,or(\neg b_I\,and\,a_I)*/$
        **local** $bb := newBlock();$
        **local** $bb' := newBlock();$
        $pc_t := emit(pc_t, \overline{I_c});$
        $pc_t := emit(pc_t, \overline{b_I} - > PC := bb);$
        $bb := emit(bb, \overline{b_{I'}} - > PC := bb');$
        $bb' := emit(bb',$ "unroll loop(pc_s, nPC)"$);$
        $bb := emit(bb, \overline{I'_c});$
        $bb := emit(bb, PC := codemap(succ_s(target_I)));$
```

```
                $queueForTranslation(succ_s(target_I), codemap(succ_s(target_I)));$
                $trans(succ_s(succ_s(pc_s)), pc_t)$

            $| \; SCDA_> => /*(b_{I'} and a_{I'}) or (\neg b_{I'} and \neg a_{I'}) or (\neg b_I and a_I) */$
                **local** $bb := newBlock();$
                **local** $bb' := newBlock();$
                $pc_t := emit(pc_t, \overline{I_c});$
                $bb := emit(bb, \overline{b_{I'}} -> PC := bb');$
                $bb := emit(bb, \overline{I'_c});$
                $bb := emit(bb, \overline{b_{I'}} -> PC := bb');$
                $bb' := emit(bb', PC := codemap(target_{I'}));$
                $queueForTranslation(target_{I'}, codemap(target_{I'}))$
                $bb := emit(bb, PC := codemap(target_I));$
                $queueForTranslation(target_I, codemap(target_I))$
                $trans(succ_s(succ_s(pc_s)), pc_t)$
            **end**

        $| \; SCDA_> => /*(b_{I'} and a_{I'}) or (\neg b_{I'} and \neg a_{I'}) */$
            **local** $bb := newBlock();$
            $pc_t := emit(pc_t, \overline{I_c});$
            $pc_t := emit(pc_t, \overline{I'_c});$
            $pc_t := emit(pc_t, \overline{b_{I'}} -> PC := bb);$
            $bb := emit(bb, PC := codemap(target_{I'}));$
            $queueForTranslation(target_{I'}, codemap(target_{I'}))$
            $pc_t := emit(pc_t, PC := codemap(target_I));$
            $queueForTranslation(target_I, codemap(target_I))$
        **end**
```

About the Authors

Cristina Cifuentes is a Senior Staff Engineer at Sun Microsytems Laboratories in Mountain View, California, where she investigates techniques and applications of binary translation. Cristina has published in the areas of binary translation, program comprehension, software maintenance, compiler construction, reverse engineering, decompilation, copyright and legal aspects of computing. She has co-edited two books, given invited lectures worldwide on various topics, and has served in the program committee of numerous conferences and workshops. Cristina was principal investigator of the Walkabout, UQBT, and the dcc projects. Prior to joining Sun Microsystems Laboratories in July 2000, she held academic positions at The University of Queensland and The University of Tasmania, Australia. Cristina obtained a Ph.D. from the Queensland University of Technology, Australia, in 1995.

Norman Ramsey began his research career in physics, a field in which he spent several years before deciding that engineering was more fun than science. After earning his PhD from Princeton in 1993, he spent several years in "industrial research" before returning to academia. He is currently Assistant Professor of Computer Science at Harvard University. His research interests lie in compilers, languages, and tools for programmers. He has worked on a significant number of software tools, most recently the Quick C-- compiler, but he may be best known for his literate-programming tool "Noweb."