The Design of a Resourceable and Retargetable Binary Translator*

Cristina Cifuentes   Mike Van Emmerik   Norman Ramsey†

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

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

Abstract

Binary translation, the automatic translation of executable programs from one machine to another, requires analyses and transformations that could be used in a wide variety of tools intended to reverse engineer binary codes. Our approach to binary translation, which is designed to allow both source and target machines to be changed at low cost, is based on a combination of machine descriptions, binary-interface descriptions, and machine-independent analyses. This approach is producing components and component generators that should be usable in a variety of tools for reverse engineering binary codes.

This paper presents an overview of the full design and gives excerpts from descriptions used in component generators, and presents preliminary results of four static translators instantiated from the UQBT framework described in this paper.

1 Introduction

A binary translator takes an executable binary compiled for source machine M_S and translates it to run on target machine M_T, at speeds approximating native M_T code. Binary translation makes it possible to run existing software on newly released machines, making such machines more appealing to potential customers. It also makes it possible to run legacy binaries on today's machines.

*This research is supported by Australian Research Council grant A49702762 and Sun Microsystems.

†The third author has further support from NSF grants ASC-9612756 and CCR-9733974, and DARPA contract MDA904-97-C-0247.

Binary translation is an instance of the more general problem of analyzing and transforming binary codes. Because binary translation exposes most of the hard problems that come up in *any* effort to analyze and transform binary codes, it is especially worthy of study. The elements of a successful binary translator may be reusable in many different tools for reverse engineering from binaries.

The major limitation of tools that work on binary codes is that they depend on many details of the underlying platform. The University of Queensland Binary Translator, UQBT, strives to adapt easily to changes in both source and target machines. Compilers and other tools have traditionally been called *retargetable* when they can support multiple target machines at low cost. By extension, we call a binary-analysis tool *resourceable* if it can analyze binaries for multiple source machines, also at low cost. This paper presents an overview of UQBT, focusing on the design and implementation properties that make it resourceable and retargetable: careful division into machine-dependent and machine-independent components, and generation of machine-dependent components from specialized descriptions. UQBT's design addresses differences in instruction sets, endianness, calling conventions, and binary-file formats, but not in operating systems—our design is currently restricted to multiplatform operating systems like Solaris, Linux, and Windows NT.

1.1 Problems of binary translation

Although it is not hard to translate some sequences of instructions from one machine to another, other parts of the task make binary translation difficult to implement. It is not always possible to find all the code in a binary program. Existing binary-file formats do not always identify code or distinguish it from data, and static analysis cannot always find the targets of jumps to addresses computed at run time [13, 14]. *Static* binary translators translate as much code as can be found before the program is run, and they fall back to interpretation if a computed jump or call goes to an untranslated instruction. *Dynamic* translators delay some or all of the translation until run time, when branch-target addresses are known.

Although a binary translator's output can approach the speed of native code compiled from source, it most often runs more slowly, when low-level properties of machine M_S are modeled on machine M_T. For example, the Digital Freeport Express translator [11] simulates the byte order of SPARC, and the FX!32 translator [12, 23] simulates the calling sequence of the source $x86$ machine, even though neither of these is native to the target Alpha machine.

1.2 Previous Work

In the late 1980s, binary translation began to replace less efficient techniques for emulation. Projects like the HP3000 emulation on HP Precision Architecture computers [5] and MIMIC, an IBM System/370 simulator on an IBM RT (RISC) PC [15], were catalysts. These and other translators were developed by hardware manufacturers to provide migration paths from old CISC machines to new RISC machines. Projects like Tandem's Accelerator [1], which translated TNS CISC binaries to TNS/R, and Digital's VEST and mx [20, 21], which translated OpenVMS VAX and Ultrix MIPS binaries to the Alpha, are other examples. Binary translation enabled all these manufacturers to move to new hardware while still supporting existing customers.

Around 1992, hardware companies started to use binary translation to run competitors' binaries. AT&T Bell Laboratories set up FlashPort to market binary translation services from PDP-11, 680x0, and IBM 360 machines to MIPS, RS/6000, PowerPC and SPARC machines [22]. Differences between source and target operating systems and platforms required manual intervention, so translations took from 1 week to 6 months. When Sun was migrating its customers to Solaris and discontinuing support for SunOS, Digital created FreePort Express, which translated SunOS binaries to run on Alphas under Digital Unix [11]. To help make Alpha an alternative to Intel PCs, Digital developed FX!32, which translates $x86$ Windows NT binaries to run on Alphas [12, 23]. This translator uses an unusual hybrid of static and dynamic techniques; it initially interprets $x86$ code, then translates and caches executed paths *after* interpretation. Because the cached translations are used in future executions, applications silently speed up with subsequent executions.

2 Goals and Objectives

The literature suggests that each new binary translator has been hand-crafted from scratch—reuse is difficult because binary translators are highly machine-dependent. Another impediment to reuse is that many details of existing translators remain proprietary and unpublished. The goal of the UQBT project is to develop a binary translator that will be constructed from well-specified components. Components should either be reusable across machines or should be generated from specialized specifications, dramatically reducing the cost of coping with machine dependence.

To concentrate on core problems of translation, we have limited the scope of our project in two ways. UQBT translates only user code (applications programs), not kernel code or dynamically linked libraries, and it requires the same operating system to run on both source and target platforms. Our current work supports Solaris, which runs on SPARC and $x86$; we anticipate that future versions will also support Linux and Windows NT. Similar limitations apply to commercial translators like FX!32.

3 Resourceable and Retargetable Binary Translation

Binary translation requires several machine-level analyses:

- Finding procedures and their code,
- Finding targets for indirect transfers of control, and
- Identifying calling conventions and sequences, and recovering (machine-level) types of procedure parameters.

Other analyses may make it possible to improve translated code, e.g., by eliminating byte swapping across machines of different endianness or by promoting local variables to registers.

We simplify the implementations of all analyses by having each operate on one of two levels of abstraction. *Register-transfer lists* (RTLs) are machine-level descriptions of the effects of instructions. They expose all the machine-dependent details of instruction behavior. *Higher register-transfer language* ($\mathcal{HRTL}$) is a higher-level, machine-independent representation of programs.

$\mathcal{HRTL}$ hides the machine-dependent details of implementing both intraprocedural and interprocedural control flow; it is analogous to a compiler's intermediate code.

Like a compiler, a binary translator can be loosely divided into front end, analyzer and optimizer, and back end. The front end decodes a source-machine binary file, produces RTLs, and then lifts the level of abstraction to $\mathcal{HRTL}$ (the intermediate representation) by using knowledge of the source-machine calling conventions and instruction set. The analyzer and optimizer map from source-machine locations to target-machine locations, and it may apply other machine-specific optimizations to prepare for the back end. The back end translates the intermediate $\mathcal{HRTL}$ to target-machine instructions, and it writes a binary file in the required format.

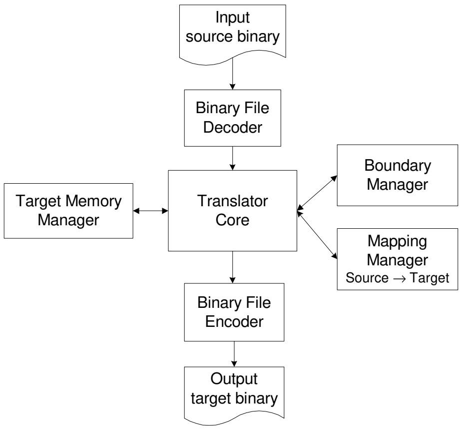

Figure 1: Functional Components of a Binary Translator.

To design for reuse and retargetability, it is more helpful to think of a functional division into components instead of a sequential division into phases. Some components will be machine-independent; others may be generated from machine descriptions. Figure 1 shows a functional decomposition of UQBT.

3.1 Components

The source **binary-file decoder** provides an abstraction representing the contents of the executable binary on the source machine. Unlike the GNU BFD library [6], this abstraction hides information about the source binary file that one might otherwise be tempted to exploit. The binary-file decoder exposes:

1. The initial state of the M_S processor that would apply when about to run this binary on a native M_S processor, including at least the initial value of the program counter,
2. A list of potential entry points for procedures, including at least the initial program counter as an entry point,
3. The contents of the program's code and data segments, by the address those contents would occupy in a running M_S executable, and
4. A list of procedures that are to be linked with the binary dynamically, and the names of the libraries containing them.

This module includes much of the functionality of a dynamic linker/loader. To improve resourceability, a binary-file decoder can be generated from a formal description of a binary-file format. SRL, a simple resourceable loader, implements BFF (a language for describing binary-file formats) and generates C code to load binary files [24]. We have currently experimented with SRL using the DOS EXE, SPARC Elf and Windows PE formats, however, our current prototype works well only on the EXE format.

One of the crucial decisions made by a translator is which locations on machine M_T hold what data from machine M_S. We encapsulate these decisions in a **mapping manager**, which maps locations in code space, locations in data space, and locations referring to registers or other processor state. Most mappings will be determined automatically at translation time, but some mappings may be specified by hand for each pair of platforms, e.g., what registers of machine M_T should be used to represent the contents of registers of machine M_S.

The mapping manager relies on the M_T **memory manager** to allocate locations in the target machine's storage space, e.g., to store translated code.

Because it is impossible in general to identify and translate all code before a program runs, a running image on machine M_T will have a mix of translated and untranslated code. The **boundary manager** tracks the boundary between translated and untranslated code and handles flow of control across the boundary. For example, a branch from translated to untranslated code might go to an interpreter or to a dynamic translator. If the untranslated code is subsequently translated, the boundary moves, and the boundary manager might backpatch the branch.

The **core translator** translates groups of machine instructions. A group may be as small as a basic block or as large as an entire program, and different translation strategies (e.g., full static, partial static, dynamic) may use different group sizes. In general, this component translates groups repeatedly until some termination condition is met; the exact termination condition may depend on the group size and timing of translation. If the group size is sufficiently large, the core translator may do some global analysis and optimization. Because the core translator coordinates the action of all the other components and performs the main translation analyses, we discuss it at length in Section 4.

Finally, the **binary-file encoder**, which is the dual of the binary-file decoder, exports the ability to create an executable binary on machine M_T. It can

1. Specify the contents of the M_T address space at the start of execution,
2. Establish the state of the M_T processor at the start of execution,
3. Write an executable file in the M_T native format, and
4. Set up dynamic linking information for translated or native libraries.

In a static translator, a binary-file encoder might be generated from a description in the BFF language. In a dynamic translator, the encoder would simply write into a running process image and flush the I-cache.

3.2 Intermediate Representations

As discussed above, UQBT uses two intermediate representations. The low-level RTL form is tuned to be mapped directly to and from machine instructions. The higher-level $\mathcal{HRTL}$ form is more like a compiler's intermediate code, because it uses high-level abstractions of control flow. It facilitates standard compiler analyses and translation techniques. A key component of the core translator, which could be used in other binary-analysis tools, is the "lifting of abstraction" from the machine-dependent RTLs to the machine-independent $\mathcal{HRTL}$.

Low-level representation

UQBT uses a simple, low-level register-transfer representation for the effects of machine instructions. A single instruction corresponds to a register-transfer list or RTL, which in UQBT is a sequential composition of effects. Each effect assigns an expression to a location. All side effects are explicit at the top level; expressions are evaluated without side effects, using purely functional *RTL operators*. For example, the effects of the SPARC `call` instruction are represented by the following RTL:

```
r[15] := %pc
%pc   := %npc
%npc  := r[15] + (4 * disp30)
```

This sequence of effect puts the program counter `%pc` in register 15, copies the "next program counter" `%npc` into the program counter, and puts the target address into `%npc`. Because the target address is computed relative to the *original* program counter, the target-address computation uses register 15, which holds the original value of `%pc`. Because the SPARC uses delayed branches, the target address is placed into `%npc`, not directly into `%pc`.

An 'RTL language' is defined by a collection of locations and operators. UQBT uses an RTL language defined by taking the union of locations on machines M_S and M_T and the union of the operators used in the descriptions of machine M_S and M_T. The 'machine X invariant' defines a sub-language of RTLs called the X-RTLs; an RTL is an X-RTL if and only if it can be represented as a single instruction on machine X.

High-level representation

$\mathcal{HRTL}$ is a higher-level language that abstracts away from the machine-dependent details of procedure calls, intraprocedural control flow, and relational expressions. $\mathcal{HRTL}$ supports the following locations:

- An infinite number of registers $\mathrm{r}[x]$,
- An infinite number of variables $\mathrm{v}x$, and
- Memory $\mathrm{m}[x]$.

$\mathcal{HRTL}$ supports a machine's minimal set of control transfer instructions and the assignment to locations:

- Unconditional jump to a computed location,
- Conditional jump,
- Call to a procedure, passing parameters,
- Return from a procedure, and
- Assignment (of locations or expressions).

Assignments and expressions are modeled by RTL effects and RTL expressions. The key property of $\mathcal{HRTL}$ is that it abstracts away from machine-dependent details of control-transfer instructions (e.g., condition codes, delayed branches), from machine-dependent calling conventions, from machine-dependent accesses to local variables, and from machine-dependent relational expressions.

For example, the previous SPARC-RTL for a call is translated to the $\mathcal{HRTL}$ instruction `Call addr`, where `addr` is `%pc + (4 * disp30)` and `%pc` has been instantiated with the source machine address of the instruction.

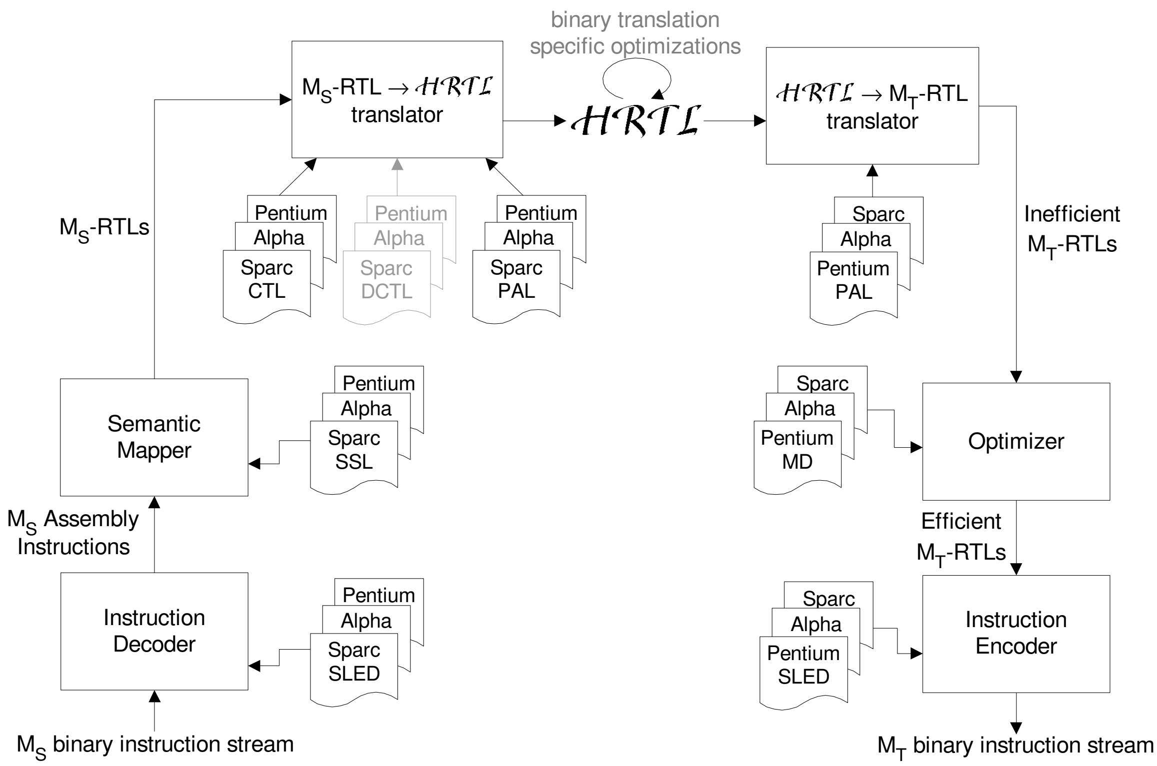

Figure 2: Core Translation Process for a Resourceable and Retargetable Binary Translator.

4 Core Translator

The core translator coordinates the action of the other components of UQBT. It translates a group of M_S instructions as follows:

1. Asks the memory manager for a location in M_T to hold the translated code, and informs the mapping manager of the new mapping,
2. Translates M_S instructions to M_T instructions
   - Uses the mapping manager to help translate access to machine M_S's data,
   - Uses the boundary manager to help translate branches outside the current group, and
3. Informs the boundary manager of the translation of the current group.

The core translator itself is built of reusable components that manipulate the source binary in terms of RTL and $\mathcal{HRTL}$ representations. Most of these components are generated from formal specifications that describe different aspects of the source and target machines and the operating system's ABI (application binary interface). As shown in Figure 2, the main steps in the translation process are decoding the machine instructions, translating the instructions into M_S-RTLs, translating M_S-RTLs to $\mathcal{HRTL}$, analyzing $\mathcal{HRTL}$ for optimizations specific to binary translation, translating $\mathcal{HRTL}$ to M_T-RTLs, optimizing M_T-RTLs, and encoding the output binary file. We explain each step in greater detail below. In Figure 2, greyed-out areas represent specifications or transformations not currently implemented or supported by the system.

4.1 Decoding binary code to M_S-RTLs

Binary code is translated to M_S-RTLs in two steps. The first step uses a decoder to identify each instruction and its operands. Most of this decoder is generated automatically from a SLED (Specification Language for Encoding and Decoding) specification, which describes the binary representation of each instruction [19]. The second step maps each instruction to an M_S-RTL. The semantic mapper is table-driven from an SSL (Semantic Specification Language) specification, which associates an RTL with each instruction [8].

Figure 3 shows tiny excerpts from SLED and SSL specifications for the SPARC. In SLED, `fields of instruction`, `patterns`, and `constructors` introduce the fields of a machine instruction, the binary representations of instruction opcodes in terms of fields, and the construction of a full instruction from opcodes and

```
# SLED specs for immediate call instruction
  fields of instruction (32)
  inst 0:31 op 30:31 disp30 0:29 rd 25:29 op2 22:24 imm22 0:21 a 29:29 cond 25:28 disp22 0:21 op3 19:24
  rs1 14:18 i 13:13 asi 5:12 rs2 0:4 simm13 0:12 opf 5:13 fd 25:29 cd 25:29 fs1 14:18 fs2 0:4

  patterns
   [ TABLE_F2 CALL TABLE_F3 TABLE_F4 ]       is op  = {0 to 3}

  constructors
    call__  addr { addr = L + 4 * disp30! }  is L: CALL & disp30

# SSL specs for immediate call instruction
  call__ disp30                             *32* r[15] := %pc
                                            *32* %pc := %npc
                                            *32* %npc := r[15] + (4 * disp30);
```

Figure 3: Sample SLED and SSL Code for a SPARC Call Instruction.

operands. The example shows a constructor for an immediate call instruction. The fragment of SSL shows the semantics of the call instruction, which also appeared in the example above. The `*32*` notation indicates the number of bits assigned in each effect.

4.2 Translating M_S-RTLs up to $\mathcal{HRTL}$

The core translator lifts the level of abstraction of the M_S-RTLs by a series of analyses and transformations that rewrite machine-dependent RTLs into $\mathcal{HRTL}$. The analyses assume that all machines can perform conditional jumps, unconditional jumps, calls, and returns. The key problem is to identify machine-specific constructs, like delayed branches, the use of the `%npc` register and relational conditions based on set instructions, and to transform them into machine-independent $\mathcal{HRTL}$.

CTL, or Control Transfer Language, describes which machine instructions map into the high-level control transfer instructions of $\mathcal{HRTL}$, namely, conditional and unconditional jumps, calls, and returns. The call mapping identifies only the call instruction itself; parameters are inferred through analysis of the M_S-RTLs. Figure 4 shows an example CTL specification for the SPARC, which uses SLED constructors, not SSL semantics, to identify two call instructions: the immediate call (`call__`) and the indirect call (represented by the 4 cases of the jump and link to register `%o7`). In the example, `call mapping` states those instructions that perform transfers of control which are equivalent to a procedure invocation. For each of the entries, the target address is specified through the keyword `address`. We use CTL to map simple control-flow instructions to $\mathcal{HRTL}$.

```
call mapping
  call__ addr                    & address = addr  |
  JMPL dispA(rs1, i), %o7        & address = rs1+i |
  JMPL absoluteA(i), %o7         & address = i     |
  JMPL indirectA(rs1), %o7       & address = rs1   |
  JMPL indexA (rs1, rs2), %o7  & address = rs1+rs2.
```

Figure 4: Control Transfer Language Specification for SPARC Call Instruction.

Like typical intermediate codes found in compilers, $\mathcal{HRTL}$ does not have delayed branches. We currently use a hand-written transformation pass to remove branch delays and to eliminate references to the SPARC `%npc` register. We have used program-transformation and partial-evaluation techniques to derive this pass from a formal description of the SPARC processor [17], but the code remains voluminous. In a future implementation, we hope to be able to generate this pass automatically from a short description of the source machine's delayed control-transfer instructions (i.e., the DCTL specifications in Figure 2). For architectures that do not support delayed branches this pass will be empty.

We have introduced a specialized, idiom-based analysis, for transforming $x86$ floating point instructions to $\mathcal{HRTL}$ given that such instructions are not register-based but stack-based. Further, because there are no floating point branch instructions on $x86$, sequences of instructions transfer the floating point condition codes to the integer condition codes, so that an integer branch instruction can be used to perform the transfer of control. This transformation is not automated and will require careful analysis in order to support it in a resourceable way.

One of the common problems of decoding machine codes is the fact that indirect transfers of control prevent the decoder from statically decoding extra code along that path. A machine-independent jump recovery technique was developed to recover the targets of an indirect jump via a table [7]. Jump tables are commonly used by compiler writers to implement `case` or `switch` statements. The effect of this technique is an increase on the amount of code statically decoded and hence analyzed, minimizing the need for an interpreter at run time.

The most challenging analysis is the recovery of parameters, local variables, and return values from procedures and call sites. PAL, or Procedural Abstraction Language, describes how a stack frame is set up when a procedure is invoked and how values are accessed on the stack frame, according to the machine's calling convention [9]. PAL allows for the description of valid prologues and epilogues for callers and callees, the locations that can be used for passing parameters, the location(s) used for returning values from a function, and the location block used by locals.

Figure 5 illustrates sample caller and callee prologues and epilogues for the System V SPARC calling convention. PAL extends the SLED language by adding regular expressions to it, in order to use patterns to describe caller and callee prologues and epilogues. Further, PAL describes parameter locations from a callee's (i.e. incoming) and caller's (i.e. outgoing) point of view, as well as return value locations. Space limitations prevent us from including the specifications for local variable locations or to explain the "abstract frame pointer" abstraction `%afp`. Refer to [9] for more information.

UQBT uses PAL to identify parameters and return locations, which are used in the $\mathcal{HRTL}$ representation of the source program. A liveness analysis finds actual parameters and return values. For example, in Figure 6, the left-hand side shows the SPARC-RTLs for a call to the procedure `gcd`, and the right-hand side shows the equivalent $\mathcal{HRTL}$ after analysis has been performed on the parameters of this call. From this $\mathcal{HRTL}$, UQBT can generate Pentium code that uses the native Pentium calling convention, which passes parameters on the stack, not in registers.

Although Figure 6 shows only integer parameters and result value, UQBT's low-level type analysis distinguishes four types of values: integers, floating-point values, pointers to data, and pointers to code, as well as their sizes (in bits). These 4 basic types, which are recovered by our analysis, are all that is needed to identify procedure parameters and results. To translate calls to library functions, we summarize their low-level type signatures in a text file. Some architectures use helper routines for arithmetics that can be performed more efficiently with hand optimized routines than with the machine's standard instructions. For example, SPARC V8 compilers call `.rem` to perform an integer remainder operation. These helper functions are non standard, as they do not necessarily exist on other platforms. They are also not part of library header files. We replace calls to helper routines with the equivalent semantics; e.g. `v0=v0%!v1` for calls to `.urem`.

```
# Prologues and epilogues for callees and callers
    CALLER_PROLOGUE std_call addr IS
        call__ (addr)
    CALLEE_PROLOGUE new_reg_win simm13 IS
        SAVE ($SP, imode(simm13), $SP)
    CALLEE_EPILOGUE std_ret IS
        ret();
        restore_()

# Locations used for parameter passing
INCOMING PARAMETERS
    new_reg_win
    {
        AGGREGATE -> [%afp - simm13 + 64]
        REGISTERS -> %i0 %i1 %i2 %i3 %i4 %i5
        STACK     -> BASE = [%afp - simm13 + 92]
                     OFFSET = 4
    }

OUTGOING PARAMETERS
    AGGREGATE -> [%afp + 64]
    REGISTERS -> %o0 %o1 %o2 %o3 %o4 %o5
    STACK     -> BASE = [%afp + 92]
                 OFFSET = 4

# Location(s) used for returning values
RETURNS
    std_ret
    RECEIVER
    {
        INTEGER   IN %o0
        ADDRESS   IN %o0
        FLOAT     IN %f0
        DOUBLE    IN %f0to1
    }
```

Figure 5: Procedural Abstraction Language Specification for Common SPARC Call and Return Instructions, Parameter Locations, and Return Locations.

```
*32* r[8] := 10             v0 := 10
*32* r[9] := 5              v1 := 5
*32* r[16] := 69 << 10
*32* r[15] := %pc
*32* %pc := %npc            v0 := CALL gcd (v0, v1)
*32* %npc := 0x10a9c
*32* r[11] := r[8]          v3 := v0
```

Figure 6: Example Showing the Effects of Parameter Analysis in SPARC-RTL to $\mathcal{HRTL}$ Translation.

4.3 Manipulating $\mathcal{HRTL}$

The advantage of lifting M_S code to $\mathcal{HRTL}$ is that the code becomes machine-independent, which allows us to decouple the translator's back end from the source machine. The high-level call construct of $\mathcal{HRTL}$ enables the back end to use the target's native calling conventions.

After translation, a running target program maintains an image of the data that would have been stored in an execution on the source machine. If source and target machines have different endianness, translation to M_T-RTLs may require swapping bytes at each M_T load and store instruction. We hope that a special analysis will identify M_T words for which every load and store can be identified at translation time, and so for which byte swapping will be unnecessary. This type of optimization is not provided by general-purpose optimizers, as compiler front ends seldom emit code that uses a non-native byte order.

4.4 Translating $\mathcal{HRTL}$ down to M_T-RTLs

The last major step is translation of $\mathcal{HRTL}$ down to M_T-RTLs, using M_T's native calling conventions and instructions. Generating good native code is a non-trivial exercise involving considerable machine dependence, but we simplify the task greatly by reusing the Davidson-Fraser technique of instruction selection by peephole optimization [10]. UQBT uses PAL to emit native instruction sequences for $\mathcal{HRTL}$ calls and returns. For other $\mathcal{HRTL}$ constructs, UQBT emits very naïve target-machine code, which is then improved using an optimizer. We have experimented with VPO, the Very Portable Optimizer [4], and with C optimizing compilers (both, gcc and cc).

In addition to instruction selection, VPO performs classical optimizations, as well as machine-specific optimizations based on costs of instructions. VPO is retargeted using machine-description files that specify recognizers for the set of target-machine instructions to be used. Instruction costs and machine-specific optimizations are hand-coded.

VPO emits assembly code for the target machine. Encoding of the M_T assembly code into binary can be done in a retargetable way by using a SLED specification. The New Jersey Machine-Code Toolkit, which implements SLED, can generate encoding functions automatically. As a practical matter, however, we have re-used the target system's assembler and linker.

VPO enables UQBT to translate $\mathcal{HRTL}$ to a minimal set of M_T-RTLs containing these locations:

- An infinite number of registers r[x], and
- Memory m[x]

and these operations:

- Setting registers to constant values,
- Loading registers from memory,
- Storing registers to memory, and
- Elementary unary and binary operations on registers, in 2-address form.

Each $\mathcal{HRTL}$ statement is translated into a series (typically three) of minimal M_T-RTLs. VPO transforms these RTLs into efficient code.

For example, the following $\mathcal{HRTL}$ sequence (which comes from SPARC-RTLs):

```
*32* r[9] := 69 << 10
*32* r[8] := r[9] | 464    ! r[8] = 0x115d0
```

is translated to the following 9 Pentium-RTLs:

```
r[100]=69
r[101]=10
r[100]=r[100] << r[101]
r[209]=r[100]              ! r[209] represents r[9]
r[100]=r[209]
r[101]=464
r[100]=r[100] | r[101]
r[208]=r[100]
p[0]=r[208]                ! Copy to parameter 0
```

VPO then transforms this sequence of 9 primitive instructions into a single Pentium assembly instruction:

```
push 0x115d0
```

The address 0x115d0 is pushed as a parameter to a later `printf` function call. In this case, 0x115d0 is the (source) address of the string `"hello world"`.

For about the same effort, $\mathcal{HRTL}$ can also be translated to low level C code, so that standard C compilers can be used as the optimizer and encoder. For example, the above sequence of RTLs can be translated to the following C code:

```
v1=70656;          /* 69 << 10 */
v0=(v1)|(464);
```

5 Implementation and Preliminary Results

UQBT is written in C++ and compiles using `gcc` on Solaris and Linux systems. It uses SLED, SSL and CTL descriptions for SPARC and Pentium, and PAL descriptions for SPARC and Pentium Unix System

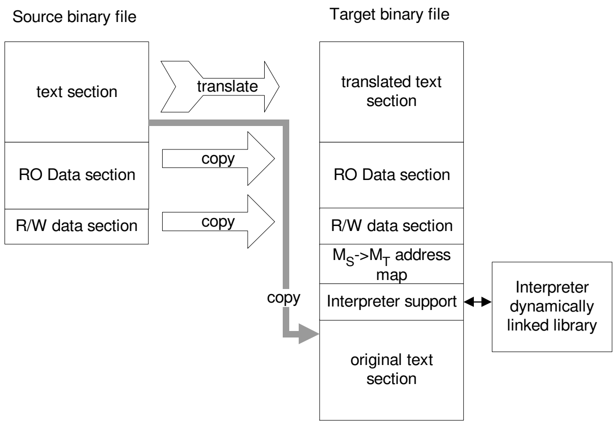

Figure 7: Generation of Target Binary File from UQBT.

V ABI calling conventions [16], which are used under Solaris.

Figure 7 describes the format of the target binary file. The source binary file can be viewed as a combination of text, read-only data, and read-write data sections. The text section is translated as described in Section 4, and the assembly output from the backend is converted to machine code using the target machine's native assembler or compiler. Since translation leaves data addresses unchanged, the data sections are copied as is, and the target binary file is configured to load those sections at the same virtual memory addresses at which they would have been loaded on the source machine. Linker mapfiles are used to set the address for the data sections.

For static translators, UQBT is designed to use an interpreter to handle untranslated code that is discovered at run time. The interpreter will use the M_S-to-M_T mapping computed by the mapping and boundary managers to determine when it can return to translated code, and so this mapping is stored in the target binary. Because the interpreter will use the original source text section, this section is also copied to the target binary. The interpreter itself is designed to be linked dynamically. We are currently building a resourceable interpreter, which uses the same SLED and SSL specifications used by the core translator. It will emulate M_S-RTLs, and it will use PAL specifications to determine how to pass parameters to library routines.

Preliminary Results

We present preliminary results obtained by four of our static translators in SPARC and Pentium systems, i.e. across CISC and RISC register-based machines; they are:

- uqbtss: static SPARC to SPARC
- uqbtsp: static SPARC to Pentium
- uqbtps: static Pentium to SPARC
- uqbtpp: static Pentium to Pentium

A SPARC to SPARC and a Pentium to Pentium translators are useful to test the adequacy of the internal representation, as translated programs should not slow down when translating from machine M to machine M using the same optimizer compiler. Further, as seen in the results in this section, a binary translator can be used as an optimizer of binary code.

The UQBT framework currently decodes and partially analyzes SPEC95 benchmark programs, such as compress, ijpeg, and gcc. The largest of such programs is gcc with 1Mb of binary code (1.6Mb executable on SPARC and 1.2Mb on Pentium). Our type analysis implementation is not complete, therefore we currently only translate programs that take integer and pointer to data parameters. This limits the size of the programs that can currently be translated to the ones presented in Figure 8. Further, our resourceable interpreter is not fully implemented yet, hence we do not support programs that require runtime interpretation. This has not been a problem for the programs presented herein, but will be required for larger programs.

Figure 8 presents results for 4 different instantiations of the UQBT framework. The test programs are: Fibo(40), which calculates the fibonacci of 40 and has 19 lines of C code, Sieve(3000), which calculates the first 3000 primes and has 13 lines of C code, and Mbanner(500K), a modified version of banner(1), which loops 500,000 times to display argv[1] ("ELF" in this case) and has 135 lines of C code. For all programs, we measured the time in seconds to execute the program on the target machine and compared that to the time measurement produced by a native compiler on that target machine; this allows us to see the quality of the translation. Each test program also lists on the second row the size in bytes of the executable file for comparison purposes. SPARC results were obtained on an UltraSPARC II, 250MHz machine with 320Mb RAM running Solaris 2.6. Pentium results were obtained on a Pentium MMX, 250 MHz machine with 128Mb RAM running Solaris 2.6. Source binary programs were compiled using gcc 2.8.1 -O4, except for Fibo which was compiled with gcc 2.8.1 -O0. Translated code versions used two different optimizing C compilers; gcc 2.8.1 and cc 4.2, on both SPARC and Pentium machines. Native code for the target machine was compiled using gcc 2.8.1 with -O0 and -O4 options, on both SPARC and Pentium.

	Translated Code		Native Code	
Program	gcc opt	cc opt	-O0	-O4
Fibo(40) sec	18.2	21.3	41.0	23.0
bytes	24,924	6,700	24,628	24,564
Sieve(3000) sec	23.7	24.1	29.3	24.5
bytes	24,732	6,316	24,552	24,452
Mbanner(500K) sec	25.8	22.2	63.7	26.6
bytes	30,500	12,248	30,652	30,268

Static SPARC to SPARC Translation

	Translated Code		Native Code	
Program	gcc opt	cc opt	-O0	-O4
Fibo(40) sec	23.0	24.3	41.0	23.0
bytes	24,916	6,680	24,628	24,564
Sieve(3000) sec	26.9	23.9	29.3	24.5
bytes	24,776	6,312	24,552	24,452
Mbanner(500K) sec	53.3	36.9	63.7	26.6
bytes	34,188	21,448	30,652	30,268

Static Pentium to SPARC Translation

	Translated Code		Native Code	
Program	gcc opt	cc opt	-O0	-O4
Fibo(40) sec	27.7	28.5	28.6	25.9
bytes	16,512	7,292	16,144	16,152
Sieve(3000) sec	17.8	17.4	18.9	18.6
bytes	16,244	6,548	15,964	15,944
Mbanner(500K) sec	42.5	n/a	80.5	44.8
bytes	22,240		21,524	25,436 [a]

Static SPARC to Pentium Translation

[a] 4,096 bytes added to executable to adjust data page size

	Translated Code		Native Code	
Program	gcc opt	cc opt	-O0	-O4
Fibo(40) sec	25.8	24.5	28.6	25.9
bytes	16,496	7,268	16,144	16,152
Sieve(3000) sec	18.6	17.1	18.9	18.6
bytes	16,228	6,536	15,964	15,944
Mbanner(500K) sec	48.7	46.5	80.5	44.8
bytes	25,664	16,016	21,524	25,436 [a]

Static Pentium to Pentium Translation

[a] 4,096 bytes added to executable to adjust data page size

Figure 8: Running Times and Code Sizes for Static UQBT Translators

As can be seen from the results, the translated code is just as efficient as native code on translations across the same architecture (i.e. SPARC to SPARC and Pentium to Pentium), and small or negligible overhead is created on translations across different architecture machines. This is due to the abstraction of code into $\mathcal{HRTL}$ code and perhaps the small size of the test programs, which do not necessarily test all the features of large programs (such as differences in types or the need for interpretation).

The version of Sieve that is translated from the Pentium to the SPARC runs 9% slower than the version compiled from C source code by the native SPARC compiler, when using gcc as the optimizer with UQBT. Because the Pentium has fewer registers than SPARC, the Pentium compiler did not put all variables in registers. In the translation from the Pentium binary, those variables remain in memory, but when the native SPARC compiler translates the same source code, it puts all variables in registers, so the natively compiled version is 9% faster. In contrast, the cc optimizer does perform this optimization and the result is a generated binary that runs at the same speed as native code.

Translations between machines of different endianness, such as SPARC and Pentium, may require the use of byte swapping at each load and store in order to access initialized data. This is the case of the Mbanner Pentium to SPARC translation, where a 100% overhead is seen. This is due to two main factors: memory locations are not promoted to registers wherever possible, and there are redundant byte swaps due to endianness differences. In our translation to SPARC, the machine has to perform costly byte swapping for one 32-bit load instruction, which results in 10 SPARC instructions. Two redundant 32-bit byte swaps result in 20 SPARC instructions which the optimizer cannot remove. This problem is not seen in SPARC to Pentium translations however because the Pentium has a single instruction to perform 32-bit byte swapping. The two shortcomings identified in the generated code can be rectified by implementing binary translation-specific optimizations at the $\mathcal{HRTL}$ level, before emitting machine-dependent code.

6 Discussion

Our goal for UQBT was not simply to write a binary translator (a daunting task in itself), but to improve understanding of binary translators, and by extension, to improve understanding of other tools for reverse engineering binary codes. In particular, we wish:

- to understand what knowledge of instruction representation, of instruction semantics, of calling conventions, and of binary-file formats is needed to perform binary translation,

- to formalize that knowledge in appropriate description languages,
- to derive components from the descriptions,
- to understand how to implement machine-dependent analyses on the RTL and $\mathcal{HRTL}$ representations, and
- to understand which analyses can be made machine-independent, and how.

Our strategy has been to build a complete translator as quickly as possible, and we have frequently sacrificed generality in favor of expediency. For example, because the RTLs used in M_S-RTL, $\mathcal{HRTL}$, and M_T-RTL support only sequential composition of effects, they are strictly less general than the RTLs used in VPO or in CSDL [18], which support simultaneous composition of effects. We made this choice because eliminating simultaneous composition simplifies both the recovery of control for $\mathcal{HRTL}$ and also the analysis of $\mathcal{HRTL}$. This is a temporary solution which helps to identify experimentally what information must be known. By the end of 1999, we expect to be using reusable λ-RTL specifications, which support simultaneous composition of effects, for semantics mapping and optimizations.

Although we have successfully reused existing SLED descriptions, we have frequently defined new, special-purpose description languages rather than reuse other kinds of existing descriptions. Creating customized descriptions has been easier than writing the analyses needed to extract the right information from existing descriptions. This is especially true of Bailey and Davidson's CCL, or Calling Conventions Language [3]. Analyzing CCL descriptions is difficult, in part because CCL descriptions are not ASCII; to parse them requires mastery of the Frame developers' toolkit. CCL is complex, and it is not obvious how to extract the information that one would find in a PAL description. CCL's authors have used it only to generate an automaton that finds locations for procedures' parameters based on the types of those parameters. This automaton can help a compiler emit suitable calling sequences, and it can be used to emit test cases that have found bugs in several existing compilers [2].

Effort

The development of the UQBT framework has been an order of magnitude larger than writing a hand-crafted binary translator from scratch. This complexity was introduced by the need to make the framework resourceable in particular, and the need to separate machine-dependent concerns from machine-independent analyses. UQBT has been developed over a period of 3.5 years and we expect half a year extra to fully complete the implementation and testing of such framework.

The advantage of the use of specifications is that a new binary translator can be instantiated from the UQBT framework quickly, by specifying the source and target machine, by reusing most of the components in the system or generating the code to support the component, and by, if needed, extending the system to support peculiarities of particular machines, whether through extra constructs in the specification languages or through extra analyses.

Our preliminary results show that the creation of binary translators from the one framework, by using different specifications, is possible. In our 4 example translators, we did not have to change any of the UQBT code, i.e. we reused the components and just instantiated the specifications for the particular machine of interest. Once our framework is complete, we will be able to experimentally test the developer effort in building a binary translator by hand or by reusing this framework.

The size of the specifications gives an estimation of the amount of code a developer would need to write in getting the basic translation system to work, or in determining how much it supports two given pairs of machines. Current specification sizes for SLED, SSL and PAL files are given in Figure 9. This is in contrast to 24,100 lines of source code (.cc, .y and .l files), 6,100 lines of definitions in header files, and 5,800 lines of specification files and partially generated instruction decoders for SPARC and Pentium (supported by the NJMC toolkit; which implements the SLED language). Lines of code of components that support the different specification languages, other than SLED, are included in the cited lines of code.

Machine	SLED	SSL	PAL
SPARC	306	689	173
Pentium	746	1626	172

Figure 9: Number of Lines of Code for SPARC and Pentium Specs.

Our hope is that a deeper understanding of description requirements, as outlined above, will eventually enable tool writers to share and reuse existing descriptions. In the meantime, the University of Queensland Binary Translator demonstrates that formal descriptions minimize the work of resourcing and retargeting a binary translator. The work of writing the specifications is an order of magnitude smaller than writing source code for the machine-dependent components.

More information is available on the UQBT home page at `http://www.csee.uq.edu.au/csm/uqbt.html`.

Acknowledgments

Members of the UQ Binary group, including Doug Simon, David Ung, Shane Sendall, Ian Walters and Trent Waddington, both discussed and helped implement parts of this project. Jack Davidson provided his VPO optimizer and answered questions about its inner workings.

References

[1] K. Andrews and D. Sand. Migrating a CISC computer family onto RISC via object code translation. In *Proceedings ASPLOS V*, pages 213–222, October 1992.

[2] Mark W. Bailey and Jack W. Davidson. Target-sensitive construction of diagnostic programs for procedure calling sequence generators. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 249–257, May 1996.

[3] M.W. Bailey and J.W. Davidson. A formal model and specification language for procedure calling conventions. In *ACM Symposium on Principles of Programming Languages*, pages 298–310, San Francisco, CA, January 1995.

[4] M.E. Benitez and J.W. Davidson. A portable global optimizer and linker. In *Proceedings of the Conference on Programming Languages, Design and Implementation*, pages 329–338. ACM Press, July 1988.

[5] A.B. Bergh, K. Keilman, D.J. Magenheimer, and J.A. Miller. HP3000 emulation on HP precision architecture computers. *Hewlett-Packard Journal*, pages 87 – 89, December 1987.

[6] S. Chamberlain. *libbfd – The Binary File Descriptor Library*, first edition – BFD version 3.0 edition, April 1991.

[7] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *Proceedings of the International Workshop on Program Comprehension*, pages 192–199, Pittsburgh, USA, May 1999. IEEE CS Press.

[8] C. Cifuentes and S. Sendall. Specifying the semantics of machine instructions. In *Proceedings of the International Workshop on Program Comprehension*, pages 126–133, Ischia, Italy, 24–26 June 1998. IEEE CS Press.

[9] C. Cifuentes and D. Simon. Recovery of parameters and return values from binary code. Technical Report in writing, University of Queensland, Department of Computer Science and Electrical Engineering, 1999.

[10] J.W. Davidson and C.W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, October 1984.

[11] Digital. Freeport express. `http://www.novalink.com/freeport-express`, 1995.

[12] R.J. Hookway and M.A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[13] R.N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1979.

[14] J.R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software – Practice and Experience*, 24(2):197–218, February 1994.

[15] C. May. MIMIC: A fast System/370 simulator. In *Proceedings SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 1–13, June 1987.

[16] Prentice Hall, Englewood Cliffs, NJ. *System V Application Binary Interface*, third edition, 1993. Unix Press.

[17] N. Ramsey and C. Cifuentes. A transformational approach to binary translation of delayed branches. Technical Report 440, The University of Queensland, Department of Computer Science and Electrical Engineering, December 1998.

[18] N. Ramsey and JW. 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.

[19] N. Ramsey and M. Fernández. Specifying representations of machine instructions. *ACM Transactions of Programming Languages and Systems*, 19(3):492–524, 1997.

[20] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. Binary translation. *Digital Technical Journal*, 4(4):1–16, 1992.

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

[22] AT T. Flashport. `http://www.att.com/FlashPort`, 1994. AT&T Bell Labs.

[23] T. Thompson. An Alpha in PC clothing. *Byte*, pages 195–196, February 1996.

[24] D. Ung and C. Cifuentes. SRL - a simple retargetable loader. In *Proceedings of the Australian Software Engineering Conference*, pages 60–69, Sydney, Australia, 28 September - 2 October 1997. IEEE CS Press.