Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework

Cristina Cifuentes, Mike Van Emmerik,
Norman Ramsey, and Brian Lewis

Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework

Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis

SMLI TR-2002-105 January 2002

Abstract:

Binary translation, the process of translating binary executables, makes it possible to run code compiled for source (input) machine M_s on target (output) machine M_t. Unlike an interpreter or emulator, a binary translator makes it possible to approach the speed of native code on machine M_t. Translated code may still run slower than native code because low-level properties of machine M_s must often be modeled on machine M_t.

The University of Queensland Binary Translation (UQBT) framework is a retargetable framework for experimenting with static binary translation on CISC and RISC machines. The system was built jointly by The University of Queensland and Sun Microsystems Laboratories in order to experiment with translations to and from different machines, to understand how to migrate applications from other UNIX®-based platforms to a (SPARC®, Solaris™) platform, and to experiment with translations from the current SPARC architecture to a future, not yet existing, version of the SPARC architecture.

This paper describes the overall design and architecture of the UQBT framework, the goals for the project, the resulting framework, experiences with translations across different machines, and lessons learned.

Sun microsystems

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

email address:
cristina.cifuentes@sun.com
emmerik@itee.uq.edu.au
nr@eecs.harvard.edu
brian.t.lewis@intel.com

Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework

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

Mike Van Emmerik
The University of Queensland
Brisbane QLD 4072, Australia
emmerik@itee.uq.edu.au

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

Brian Lewis*
Sun Microsystems Laboratories
Palo Alto, CA 94303, USA
brian.lewis@sun.com

1 Introduction

Binary translation, the process of translating binary executables,[1] makes it possible to run code compiled for a source (input) machine M_s on a target (output) machine M_t. Unlike an interpreter or emulator, a binary translator makes it possible to approach the speed of native code on machine M_t. Translated code may run more slowly than native code because low-level properties of machine M_s must often be modeled on machine M_t. For example, the Digital Freeport Express translator [Dig95] simulates the byte order of the SPARC(R) architecture, and the FX!32 translator [Tho96, HH97] simulates the calling sequence of the source x86 machine, even though neither of these is native to the target Alpha architecture.

Binary translation techniques aim to translate the code (i.e., image) of an executable from one machine to equivalent code for another machine. Although it is not difficult to translate most

*Now at Intel Microprocessor Research Labs. Email: brian.t.lewis@intel.com

[1]In this document, the terms *binary executable*, *executable*, and *binary* files are used as synonyms to refer to the binary image file generated by a compiler or assembler to run on a particular computer.

machine instruction sequences from one machine to another, other considerations make the task difficult in practice. For example, binary code often mixes data and instructions in such a way that they cannot be distinguished. This problem is exacerbated by indirect or indexed jumps, where the target address of the jump may be hard to determine statically, even though it will be known at runtime. Further, some older operating systems did not provide systems programmers with an ABI (application binary interface) for low-level system calls, instead inviting application writers to directly access devices and system facilities, bypassing the operating system. These and other problems are common to binary-code manipulation tools such as disassemblers and decompilers. The static parsing of the machine instructions in a binary must be partially incomplete given its equivalence to the halting problem [HM79] and hence it is undecidable in general. Nevertheless, for binary translation purposes, this does not mean that the problem cannot be solved. In fact, given that the translated binary must be executable, information that cannot be determined statically will be available dynamically, and can be used by a runtime translator or an interpreter that operates at runtime to decode the instructions.

Problems that are specific to binary translation are due to its multi-platform nature: there is a need to address the differences between source and target architectures (e.g., CISC vs RISC); the endianness of the machines (e.g., little vs big); machine-dependent issues (e.g., delayed branches and register windows on the SPARC architecture); and compiler-generated idioms. As well, there are differences in operating system services and graphic subsystem calls—these are the hardest to address. Previous work reported in the literature suggested that a new binary translator must be hand-crafted for each pair of supported platforms due to machine dependency constraints.

In 1996 we started developing ideas and a framework for experimenting with such binary translation ideas. The University of Queensland Binary Translation (UQBT) project started at the University of Queensland, Australia, under the direction of Cristina Cifuentes. It was prompted by colleagues at Sun Microsystems Laboratories, who suggested the possibility of working with binaries and transforming them into binaries for another machine. Norman Ramsey and Mike Van Emmerik joined Cristina in shaping the project, Norman from a design point of view and Mike from an implementation point of view. During 2000-2001, the project became joint work with Sun Microsystems Laboratories. Brian Lewis joined this effort and worked on various back ends.

Experience with a prior binary-manipulation project had shown that most users understand little about the manipulation of binaries: they just want tools, and are rarely interested in knowing about the tool's internals or its data representations. Just as compiler technology has matured to the point of having retargetable compilers that can perform code generation for a variety of machines, we thought it was time to start understanding how to make binary-manipulation tools retargetable, for both source and target machines. Our focus was on static translators, hence the UQBT framework is a static binary translation framework.

We therefore set the following goals for the project:

- to understand what aspects of instruction representation and semantics are needed to perform binary translation,
- to write those aspects as formal machine descriptions,
- to derive components of binary translators from those descriptions,

- to understand how to implement existing machine-dependent analyses on a machine-independent RTL (register transfer) representation,
- to understand which of these analyses can be made machine-independent, and how, and
- to develop a framework for experimentation with binary manipulation ideas.

It is clear that an “all purpose” binary translator is very hard to develop, therefore some bounds for the research were established.

To keep things simple, *translation was limited to user code* (i.e., applications programs), not kernel code or system calls (i.e., systems programs), or dynamically linked libraries (as these are sometimes written in assembly code and often closely resemble systems programs). Note that this approach is not limiting; Digital used it for their FX!32 hybrid translator for x86 Windows 32-bit binaries on Alpha [HH97].

We worked within the context of a *multi-platform operating system*, the Solaris(TM) environment in particular, which runs on the SPARC and x86 platforms. Similar ideas work on other multi-platform OSs such as Linux and Windows NT. We also experimented with cross-OS translations where the two OSs were similar in nature; for example, Solaris and Linux environments. We have successfully translated (Solaris,SPARC) binaries onto (Linux,x86) binaries, provided the same runtime libraries were available in both systems.

The UQBT framework is our answer to the above goals. The framework consists of 89 C++ files, 77 header files, 40 specification and matching files, and numerous configuration and regression testing files. Users can instantiate translators out of the UQBT framework by configuring the system for a given pair of source and target machines. We instantiated 8 different translators, with time varying from 3 staff-months to 12 staff-months, as specifications or APIs must be written for each. One of our goals was to reduce the amount of retargetting work needed when supporting a new machine (whether source or target), by supporting machine-independent analyses wherever possible.

Issues	SPARC	Pentium	mc68328	PA-RISC	ARM
Architecture	RISC	CISC	CISC	RISC	RISC
Endianness	both	little	little	big	both
Branching	delayed	non-delayed	non-delayed	delayed	non-delayed
Parameter passing	registers then stack	stack (registers possible)	stack	registers then stack	registers then stack
Stack grows	low memory	low memory	low memory	high memory	low memory
Float ops depend on int instructions	no	yes	no	no	no
Fixed pointers	frame ptr	frame ptr	frame ptr, global data ptr	frame ptr	frame ptr

Figure 1: Main Differences Between Architectures Used

Figure 1 shows some of the differences between the machines our 8 translators supported. Differences between the architectures are summarized in terms of their instruction set (i.e., CISC or RISC), their endianness, the delayed or otherwise branching support, the locations used for parameter passing at procedure call boundaries, the direction the stack grows towards (i.e., low or high memory), whether floating point operations rely on integer instructions to obtain a result, and the types of fixed pointers that are commonly used in programs for those machines.

The UQBT framework is suitable for experimenting with binary manipulation; it is not a production system and was never designed to be one. Several undergraduate and some postgraduate students have worked on the system.

This paper is structured in the following way. §2 describes the UQBT framework and its intermediate code representations. §3 explains how users can instantiate translators out of the framework. §4 gives our experiences with the instantiation of new front ends and back ends for the UQBT framework. §5 discusses our lessons learned: what worked well, what did not, what was missing, and what we would do differently if doing it over again. Lastly, some conclusions close this paper.

2 The UQBT Framework

The UQBT framework was designed with retargetability in mind. We were interested in supporting binaries for different input and output machines, so we made the framework retargetable for both. In our notation, we refer to the input machine as the *source machine* (M_s), and the output machine as the *target machine* (M_t). The framework was designed so that users could instantiate new translators out of the framework for their source and target machines of choice, to run on a *host platform*.

As with a compiler, a binary translator can be viewed as composed of two main parts, a front end and a back end, only here both parts deal with machine-specific information. In a binary translator, the front end translates machine-specific information into an intermediate representation suitable for analysis. The back end translates that intermediate representation down to machine code for a target machine.

To make a compiler retargetable, two approaches are most often used. The most common approach makes use of machine description files that describe machine instructions, including some semantic features at times, to support compilation or optimization of code. Examples of this approach include the GNU compiler suite [Sta93] and the VPO optimizer [BD88]. With both of these systems, some modules are machine-independent, and machine-dependent modules need to be written by the compiler/optimizer writer. The amount of code to be written varies depending on the complexity of the instruction set and the optimization support wanted. The second approach makes use of abstract classes describing the machine-independent information that is needed to analyze code. This object oriented approach relies on subclassing to support other machines, and new features of a machine that do not exist in other machines are added only to the machine's subclass. In the event that a new feature is available in multiple machines, that feature can be supported in a higher-level class. The abstraction does not need to be implemented using an object oriented language per se, the abstraction can be described in non-object oriented programs in the form of an application programmer interface (API). Programs such as the X toolkit have shown that extensibility is possible when using an API approach.

In the UQBT framework, the first phase of the project concentrated on the front end, and we aimed to support retargetability by describing machine and operating system conventions. The second phase of the project concentrated on the back end. We placed less emphasis on the use of machine descriptions (though we still used them at times), and placed more emphasis on using an abstract API to structure the back end translation process so we could experiment with multiple back ends. We first explain the intermediate representation we used, then explain the front and back ends.

2.1 Intermediate Representations

As with compilers, binary translators make use of several intermediate representations of the code to be translated. A program is a collection of procedures.[2] A procedure is represented by its control flow graph and instructions. Instructions are represented by one of two intermediate representations. A control flow graph is a series of basic blocks that describe transfers of control in a procedure. Individual basic blocks contain information about the instructions in that block.

Two main intermediate representations for the code of the program are used in the framework: a low-level representation based on register transfers called RTL, and a high-level, machine-independent representation called HRTL. A high-level view of a binary translator based on these two representations is shown in Figure 2. The decoding stage translates the code in the source binary file into RTLs for that source machine M_s. An analysis stage then translates those M_s-RTLs into HRTLs, by abstracting away information from the underlying machine. Once we have HRTL code, standard compiler back end techniques are used to generate binary code for the target machine M_t. An analysis stage translates the HRTL code into RTLs for the target machine, ensuring the M_t-RTLs represent instructions of the target machine. Finally, the encoding stage translates the machine-specific M_t-RTLs into binary code for the target machine.

2.1.1 RTL

A register transfer list (RTL) is a collection of sequential effects. Each effect has the form 'location := expression', and the expression is always evaluated without side effects, so that all state change is explicit. RTL expressions are represented as trees, the leaves of which refer to constants or to the values contained in locations. Note that although the tree leaves refer to locations, the values themselves are not necessarily calculated, only the location is indicated. The internal nodes of the trees are 'RTL operators'. The locations available are an infinite number of registers `$r[.]` and infinite memory `$m[.]`. Register locations can be named, for example, `$OF` can represent the 1-bit overflow flag. In this paper, we use the symbol $ to denote locations.

[2]We use the term procedure to represent a code chunk that is invoked by a `call` in the program. The code chunk may or may not return to its parent caller, and it may or may not return a value as the result of the call operation.

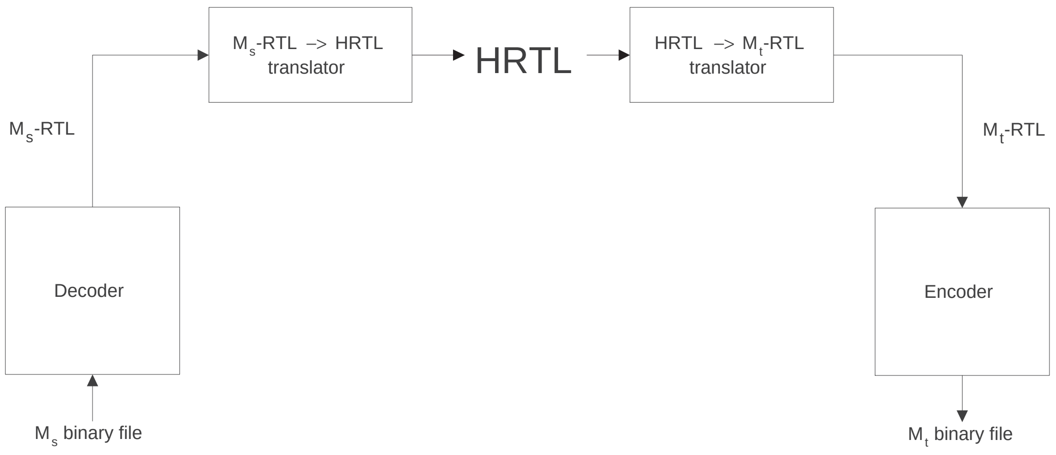

Figure 2: Intermediate Representations and a Block-level View of the UQBT Framework

To illustrate this, the following is an ASCII representation of an RTL representing the effect of the SPARC architecture `andcc` instruction. This instruction has an `i` field which determines whether the instruction takes a register or an signed immediate value as its second operand:

```
$r[rd] <--      and (*$r[rs1], if i = 0 then *$r[rs2] else simm13! fi);
$NF    <-- bit (and (*$r[rs1], if i = 0 then *$r[rs2] else simm13! fi) < 0);
$ZF    <-- bit (and (*$r[rs1], if i = 0 then *$r[rs2] else simm13! fi) = 0);
$OF    <-- 0;
$CF    <-- 0
```

This RTL does a bitwise AND of the contents of register `r[rs1]`, either with the contents of register `r[rs2]` or with a signed immediate value (`simm13`). The symbol * denotes a value and the symbol ! stands for sign extension. This result is stored in register `r[rd]`, and it is also used to set two of the four condition codes. The other two condition codes are set to zero by the instruction.

RTLs are complex and detailed. Machine descriptions make use of the 'superoperator' technique [Pro95] to simplify the description of condition code effects. For example, we define a superoperator LOGICAL for the SPARC architecture such that LOGICAL(X) stands for

```
$NF  <-- bit ((X) < 0);
$ZF  <-- bit ((X) = 0);
$OF  <-- 0;
$CF  <-- 0
```

Register locations carry with them typing information in the form of a (type, size, sign) tuple. The type is one of four low-level, machine-like, types; namely, integer, float, pointer to data or pointer to code. The size is the number of bits of the type, typically from 1 to 64. The sign is a boolean denoting whether the location is signed or not (integer types only).

An 'RTL language' is defined by a collection of locations and operators. For binary translation, a suitable RTL language is 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; i.e., there is a 1:1 correspondence between assembly instructions for machine X and X-RTL instructions.

2.1.2 HRTL

The high-level register transfer language, HRTL, is a collection of instructions that affect the state of locations. The language is defined by a set of operators and instructions. The main aim in the design of the HRTL language was to be able to express code semantics in a way that was machine-independent. As such, translations to the HRTL language require analyses that abstract away the peculiarities of individual machine instruction sets. For example, features such as the next PC register in architectures that support delayed branching semantics, are not exposed in the HRTL representation; such code needs to be transformed into equivalent code that does not make use of the next PC register.

HRTL instructions operate on locations. As per RTL, there are an infinite number of registers (`$r[.]`, which may be named, e.g., `$X`) and an infinite memory (`$m[.]`). Further, HRTL has an infinite number of variables (`$v.`), which are locations that can hold actual and formal parameters of procedure calls. Both register and variable locations carry typing information.

HRTL supports assignments of the RTL form 'location := expression', as well as higher-level instructions such as conditional, unconditional and computed jumps of the form 'jump [cond expression] <list of locations>', procedure call of the form 'call location (<list of variables>)', and returns of the form 'return location'. Assignments also support the form 'location := call location (<list of variables>)'.

As an illustration, the following is an ASCII representation of a HRTL that represents the SPARC architecture `andcc %o3,64,%g1` instruction after transforming the sample SPARC-RTL of §2.1.1. In the example, `%o3` is represented by register `$r[10]` and `%g1` by register `$r[1]`. Note that condition code analysis will either determine that assignments to condition codes are dead (and hence remove them) or move the effect of the condition code assignment to another HRTL that makes use of a conditional expression. In this example, the condition codes were found to be dead.

```
$r[1] <-- and (*$r[10], 64);
```

A more interesting example is the HRTL representation of the SPARC architecture `call` instruction, which stores the return program counter address in register `%o7` (i.e., register `$r[15]`). The SPARC-RTL for this instruction is

```
$r[15] <-- *$pc;
$pc    <-- *$npc;
$npc   <-- *$r[15] + (4 * disp30);
```

The SPARC architecture `call` instruction with displacement `0x4318`, invoked at program counter location 0x00010B20 is transformed into the following HRTL instruction

```
 call 0x00021780;
```

where `0x00021780` is equivalent to the computation of `0x00010B20 + (4 * 0x4318)`.

2.1.3 An Example

In order to give readers an idea of the implications of the HRTL representation, we show, without going into low-level details about the translation process, how a series of SPARC-RTLs and Pentium-RTLs end up being represented in HRTL. Our sample C language statement is from the Fibonacci program, where the following statement invokes the fibonacci procedure `fib` with the argument `number` and stores its result in variable `value`

```
  value = fib (number);
```

The unoptimized SPARC-assembly code for the C language statement is

```
  ld [%fp-20], %o0                       // load parameter
  call 0x10a9c                           // call fib
  nop
  st %o0, [%fp-24]                       // store return value
```

which corresponds to the following SPARC-RTLs. Note that only control transfer instructions make explicit changes to the `$pc` and `$npc` registers.

```
  $r[8]           <-- *$m[*$r[30]-20]; // load parameter
  $r[15]          <-- 0x010b40;        // call fib
  $pc             <-- *$npc;
  $npc            <-- 0x010a9c;
  $m[*$r[30]-24] <-- *$r[8];           // store return value
```

The same C language statement is represented by the following Pentium-assembly instructions

```
  movl 0xfffffffc(%ebp), %eax            // load parameter
  pushl %eax                             // put it on the stack
  call 0x8048960                         // call fib
  addl $0x4, %esp                        // fix up the stack
  movl %eax, 0xfffffff8(%ebp)            // store return value
```

which corresponds to the following Pentium-RTLs

```
$r[24]         <-- *$m[*$r[29]-4]    // load parameter
$r[28]         <-- *$r[28] - 4       // put it on the stack
$m[*$r[28]]    <-- *$r[24]
$r[28]         <-- *$r[28] - 4       // call fib
$m[*$r[28]]    <-- *$pc + 5
$pc            <-- 0x8048960
$r[28]         <-- *$r[28] + 4       // fix stack frame
$r[24]         <-- *$r[24]
$m[*$r[29]-8] <-- *$r[24]           // assign return value
```

After transformational analysis, the HRTL code obtained for both RTL representations follows

```
HRTL (SPARC)                 HRTL (Pentium)

$v0 <-- *$m[*$afp+100]       $v3 <-- *$m[*$afp+4]
$v0 <-- call fib (*$v0)      $v4 <-- *$v3
$m[*$afp+96] <-- *$v0        $v3 <-- call fib (*$v4)
                             $m[*$afp] <-- *$v3
```

The `$afp` named location is the abstract frame pointer, which points into the local memory stack for the procedure. It is clear that in both cases the code fetches a value from the local stack into a variable, then it passes the variable's value to the procedure call `fib`, which returns a result that it then places in another variable. That result value is then stored onto the local memory stack. Simple forward substitution would make both codes have the same number of HRTL instructions; that step is a simple optimization step left to the back end.

This example illustrates the benefits we found from having a high-level, machine-independent representation. It is suitable for generating native code for a target machine in an optimal way, instead of emulating features of the source machine. In this case, we did not have to directly emulate the SPARC architecture `$pc` or `$npc` registers, which would have required us to update the equivalent of their values after every Pentium instruction in, most likely, two of the scarce Pentium registers.

The Appendix shows a complete example for a translation of a Pentium-compiled recursive fibonacci program to the SPARC architecture.

2.2 Front End

The UQBT front end translates a source binary file into HRTL. Figure 3 provides a dataflow view of the transformations of the code to arrive at the HRTL representation. In this figure, boxes represent reusable components of the framework (some require some code to be written), document files represent specifications, notched boxes represent APIs, and the circle represents machine-specific code that needs to be written by the binary translator writer.

In short, the UQBT front end provides a binary file decoder which reads an instruction stream from an input binary file. This instruction stream is parsed into a sequence of source machine

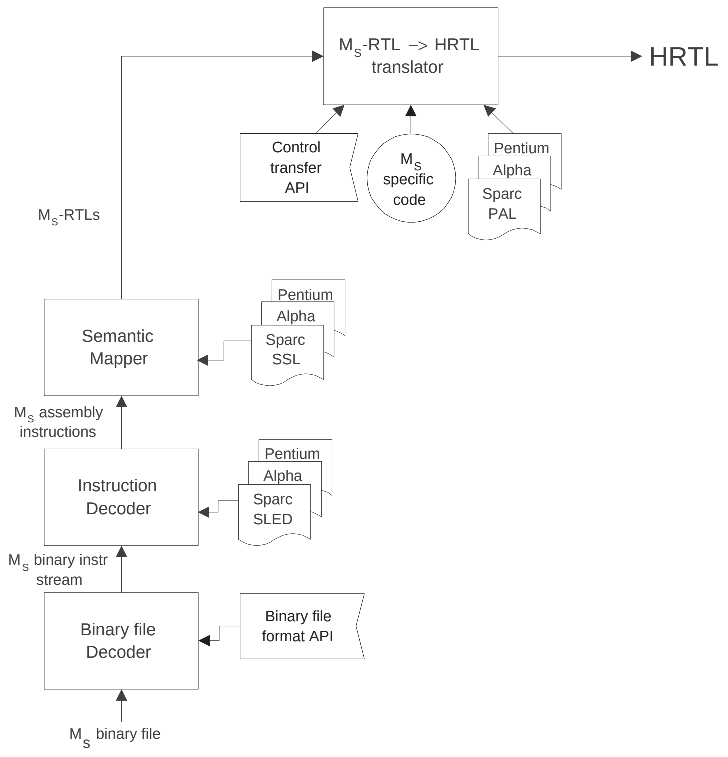

Figure 3: The Retargetable Front End of the UQBT Framework

instructions using a description of the syntax of machine instructions for that machine. A semantic mapper then translates each source machine instruction into a sequence of source machine-specific RTLs (M_s-RTLs) using a specification of the semantics for the source machine's instructions. The resulting M_s-RTL stream is analyzed by an M_s-RTL to HRTL translator using control transfer information, procedural conventions and other information.

The specifications and APIs capture information about the machine or the operating system used by the source binary file. These specifications and APIs support retargetability of the front end, and we briefly describe them below, prior to describing the parts of the front end itself. Note that for historical reasons, these descriptions end up being implemented in different files that use different languages; conceptually, these descriptions could be part of the one machine-description file. The languages we use are described in the literature, and we give references for them below.

- Machine-specific information: a computer architecture's properties are described using three kinds of information. These are 1) the syntax of its instructions (their mnemonics, operands, and instruction bitfields); 2) their semantics; and 3) a description of its control transfer instructions. The following specifications and APIs provide this information:
  - SLED (Specification Language for Encoding and Decoding) specifications allow users to specify the mapping between the binary and the assembly representation of a machine instruction set, as well as the machine's registers and names for those registers [RF97].

    - SSL (Semantic Specification Language) specifications allow users to specify the mapping between assembly instructions and their equivalent RTLs, to name new registers and declare overlaps, to define superoperators in the form of macros for commonly set condition codes, and to specify the fetch-execute cycle for the machine [CS98].
    - Control transfer API allows users to support the identification of instructions that perform transfers of control (unconditional jumps, conditional jumps, calls, and returns).
- OS-specific information: conventions used by the operating system are described using two kinds of information: the procedure calling conventions and the format of the binary file. These conventions are supported using the following API and specification:
    - The binary file format API allows users to load and decode binary files and hides details about the binary's internal representation. This API assumes that there is at least one entry point into the program, that code and data sections exist with known addresses, and that there may be a symbol table (which is used for names if it exists), [CERL01] at Chapter 5.
    - PAL (Procedural Abstraction Language) allows users to describe the conventions dealing with procedures; such as, calling conventions, parameter passing conventions, storage conventions for local variables, and callee stack frame information [CS00].

2.2.1 The Front End Modules

The front end is composed of a series of conceptual modules that transform the source input text stream into a high-level representation. This series of modules is briefly described herein; more details are available in the UQBT documentation [CERL01].

Binary file decoder

The binary file decoder decodes the source binary file into an internal UQBT representation that supports the binary file format API. All text and data sections are copied "as is" into memory and information about the binary file is stored in an internal representation.

The binary file decoder makes the entry point into the text section available, as well as the main entry point into the user-written part of the program (if available and detectable).

Instruction decoder

The instruction decoder disassembles the (text/code) instruction stream starting at the main entry point. The disassembly matches the binary representation of instructions into equivalent assembly instructions, as specified in the machine's SLED file. This support is done using the New Jersey Machine Code toolkit's matching files [RF97].

The algorithm used is the standard reaching algorithm from a given entry point [SCK$^+$93, CG95]: Starting at an entry point, follow the program's path until a transfer of control is met. When a control transfer implies more than one target address, follow one of the addresses and place the others on a queue to be processed (i.e., these are new entry points). When a path ends, follow another path from the queue to be processed, until no addresses remain in the queue.

Note that since the translation is done statically, it is not always possible to know a priori the targets of computed transfers of control such as indirect or indexed jumps or indirect calls. Analysis of such cases is described in §2.2.3, which allows for better coverage when decoding the text section; however, it cannot guarantee complete code coverage.

Semantic mapper

The semantic mapper maps assembly instructions into RTLs for the source machine, effectively generating M_s-RTLs for the source binary file's code section. The RTLs are provided by the SSL specification for that machine. This step is done immediately after an assembly instruction is matched; that instruction is transformed into a list of register transfers and stored in this representation.

M_s-RTL to HRTL translator

The M_s-RTL to HRTL translator is the key module in the framework that allows us to achieve machine independence in representating the program's code. This module transforms RTL instructions into HRTL instructions by supporting an informal control transfer API, performing analyses on procedural information (such as parameters, locals and return locations), and adding any extra hand-written code to support peculiarities of the source instruction set. The latter peculiarities include delayed branches on the SPARC architecture or floating point stack-based instructions on x86. These are explained in §2.2.2.

Control transfer. Support for the control transfer API allows us to transform RTL instructions into very simple HRTL instructions that do not take the required parameters yet. For example, a sequence of RTL instructions can be transformed into a call HRTL instruction even though no analysis has been performed to determine the parameters to the call instruction per se. This intermediate state in our HRTL representation is what we refer to as I-RTL. In a similar way, jumps and return instructions are transformed into I-RTLs and then analyzed for operands.

Procedural abstraction. The procedural abstraction analysis makes use of PAL descriptions to determine the parameters passed to a procedure. This interprocedural analysis is performed based on a liveness analysis on locations that are valid parameter-locations at the caller and callee sites. We also transform the code in each procedure so that references to the frame pointer are transformed into references to an abstract frame pointer (`$afp`), which is a conceptual pointer to the end of the stack frame (usually, where the stack pointer points after the callee prologue). The analysis keeps track of changes to registers relating to `$afp`, usually the stack pointer, and often also a frame pointer register. It knows the relationship between `$afp` and these registers at every instruction, and replaces references to these registers with `$afp` plus or minus a constant.

The advantage of using a conceptual frame pointer such as `$afp` is that we do not need to emulate a moving stack pointer on the target machine, which may not be efficient; for example, RISC machines do not have push instructions, and simulating them is very inefficient. Instead, we generate code that is much more in harmony with the target architecture. An array of bytes is generated to handle these source stack frame locations, with the array name becoming the equivalent location to `$afp` (or array name + size for machines where the stack grows upwards). A complete description of these analyses is available in [CS00].

Note that no changes are made to the data of the source program, that is, the data and all references to them remain unchanged.

Once a HRTL representation of the program has been obtained, binary translation-specific optimizations can be performed on the representation. For example, when translating to a machine with different endianness, the common technique is to perform byte swapping of data after each load and before each store instruction. An analysis at the HRTL level may help reduce the number of byte swaps required at each load and store from memory. This type of optimization, though feasible, was not implemented in the UQBT framework.

2.2.2 Machine-specific Analyses

The M_s-RTL to HRTL translator normally includes code that is written for a particular machine to perform code transformations to remove that machine's peculiarities. In our experience, for every source machine we supported, we had to add special analysis to transform away some feature of that machine. The following lists some of these machine features.

SPARC

The SPARC architecture supports delayed branches: the instruction following a branch—the delay slot instruction—is executed before the transfer of control reaches the branch's target instruction. The processor keeps track of instructions by using an extra program counter register, the next program counter or `$npc`, which is updated by all branching instructions as well as at each iteration of the fetch-execute cycle for non-branching instructions. The `$npc` register is a SPARC-specific register. When doing binary translation, unless assignments to the `$npc` register are transformed away (in effect removing the register from the code), it would be necessary to emulate that register and all assignments to it. For this purpose, we developed a transformational analysis to remove delayed branches [CR02]. The technique is general enough to be used on any processor with delayed branches such as the MIPS, SPARC, and PA-RISC processors.

Pentium

The Pentium architecture supports floating point instructions using a different set of registers than those used by integer instructions. These floating point instructions originated from a time when they were implemented by a separate math coprocessor: the 80x87 series, e.g., the 80387 provides floating point support for the 80386 processor. These separate floating point instructions, registers, and conventions persist despite the fact that from the 80486 processor onwards, floating point support has been integrated onto the processor chip itself.

Because of this, floating point condition codes must be moved into integer condition code registers (via the `$ax` register) in order to perform branching. There are no branch instructions based on floating point condition codes; they would have been difficult to support since the branch processor and floating point condition codes are in separate chips. In fact, later Pentiums (from the Pentium Pro onwards) have a set of compare instructions that affect the integer flags. Most compilers do not take advantage of these instructions, since they are not available on earlier processors.

On the Pentium architecture, floating point instructions make use of an 8-register stack. This register stack can either be accessed as an actual stack or indexed directly as an array relative to the top of stack. Since we do not want to emulate a floating point stack in translated code, we transform the code to use a conventional "flat" register model. Instead of instructions that implicitly use the top and next of stack as operands, the transformed code refers to actual registers such as `$r[39]`. We had to take into consideration all the tricks commonly used to implement floating point compares. For example, some compilers would store the floating point flags to register `$ax`, then move register `$ax` to the integer flags, before performing an integer branch. Others would avoid the movement from register `$ax` to the flags (a slow, "non pairable" instruction) using code like this

```
    FLD [a]            ; Load first Floating Point operand
    FCOMP [b]          ; Compare with second FP operand
    FNSTSW AX          ; Store floating point flags to register AX
    AND AH,41H         ; Isolate the C3 (zero) and C0 (less) bits
    JZ AGreaterThanB   ; Branch if these are both zero (so a > b)
```

Note that the actual branch condition (branch if zero) in the last instruction bears no relationship to the floating point comparison that this code sequence implements (floating point greater than). Hence there must be a good deal of pattern matching in the Pentium front end to convert these idiomatic sequences to high level branches. The above code would be converted to the following HRTL code by a UQBT translator[3]

```
08049f28 *32* v7 := b
         *32* v6 := a
08049f2b  JCOND ((v6 > v7):<32f>) 0x08049f0c
```

[3]UQBT's HRTL ASCII representation does not use $ or * to denote locations or content of a location. The `*32*` notation denotes the size in bits of the assignment attached to that line. The `<32f>` notation denotes a (size,type) tuple for a location or expression. `JCOND` stands for conditional jump instruction.

Note that the x86 details of copying from the floating point flags to `$ax`, the `AND` instruction, the immediate value 41H, and the branch if zero are replaced by a high level comparison between two floating point variables. The generated C code becomes

```
  if (v6 > v7) goto L10;
```

68328

The MC68328 architecture is used in the Palm Pilot device. The PalmOS makes use of a dedicated register (`$a5`) to access global data in the data section. The global data pointer acts much like a frame pointer. In UQBT, instead of emulating the global data pointer register and each of its updates, we extended the frame pointer removal analysis already done by the M_s-RTL to HRTL translator in order to transform the pointer into an abstract global pointer (`$agp`). The `$agp` abstract register is deemed to point to the start of a global array of bytes (compare this with the `$afp` register, which points to the start of the array of bytes that represents a procedure frame). This allows us to generate target code that does not depend on the PalmOS initializing the `$a5` register before the start of main. In some systems, a global data pointer actually points to the middle of the data so that positive and negative offsets can be used to double the register's reach. In this case, the actual register will contain a constant offset from the start of the global data array.

In order to specify the behavior of a global pointer in the PAL language, we introduced a special keyword `GLOBALOFFSET`, which is a constant for any particular input file but can vary from one input file to another. Our binary loader class has a method called `GetGlobalOffsetInfo` that returns the total size of the global data array, and another value that is useful for specifying the global data behavior. In the case of the PalmOS loader, this is the offset from the start of the global data to where the `$a5` register points. Other loaders can return different information if desired; of course, the loader and the PAL file must agree on what meaning is attached to the `GLOBALOFFSET` value. For example, the following appears in the PalmOS PAL specification[4]

```
GLOBAL ABSTRACTION
     %a5 -> %agp + GLOBALOFFSET
```

This specification says that the `$a5` register points a constant distance into the global data array, by an amount `GLOBALOFFSET` that is known to the binary file loader. An example of generated C code that makes use of an `%agp` reference is as follows

```
   *((int8*)((_globals)+(57)))=0;
```

PA-RISC

The PA-RISC architecture supports delayed branches. As a result, the delayed branching transformational analysis developed for the SPARC architecture was reused in the PA-RISC front end by

[4]PAL uses % to denote named registers.

instantiating a new version of the algorithm to support its instruction set. It is interesting that the PA-RISC has three classes of delayed branches different from those in the SPARC architecture. We were able to support these using the algorithm although this was not obvious at first. Details on this transformation are available in a technical report [CR02].

The PA-RISC instruction decoder makes significant use of guarded assignments. In order to make the code more readable (and to allow higher level analyses to recognise certain expressions), it is necessary to do extra simplification of PA-RISC assignments (including guards), and also to do forward substitution within the RTLs of each instruction.

2.2.3 Other Instruction Decoding Analyses

In a static translator, coverage of the source program's code section is important, since translating more code statically minimizes the need for a runtime interpreter to execute untranslated pieces of code. Heavy reliance on runtime interpretation hampers the use of translated programs, as interpretation is slow and users want their programs to execute without perceivable pauses.

In order to improve the coverage of decoding achieved by the standard decoding algorithm, we implemented an analysis to recover the targets of computed jumps when such jumps represent `switch` statements in C and similar languages. We also implemented a speculative decoding algorithm to traverse the remaining sections of the code section that had not been decoded yet, and determine if they represent valid code that could be translated statically. Descriptions of these two techniques follow.

We note that these techniques considerably increase static translation coverage, but they do not guarantee complete coverage. The Digital VEST and mx translators made use of backward symbolic execution to resolve as many computed branch targets as feasible [SCK$^+$93]. In general, this is a hard problem to solve, although extensive testing and taking advantage of commonly-generated compiler idioms increases coverage. However, we also note that code written back in the 1980s and early 1990s was mainly written in the C language and therefore was not object oriented and did not support dynamic dispatching of procedures (methods). Dynamic dispatching is implemented by C++ compilers using a virtual method table and an indirect call based on a register value. That register value is normally only determined at runtime based on the actual class of the method being invoked, so these programs require further analysis to increase their static translation coverage. Similar techniques can be used to recover the targets of virtual method calls [TC02].

Recovery of Computed Jumps

The standard method of decoding machine code involves following all reachable paths from the entry point [SCK$^+$93, CG95]. This method does not give a complete coverage of the code space in the presence of indirect transfers of control such as indexed jumps and indirect calls. A common technique used to overcome this problem is the use of patterns. A pattern is used for a particular compiler to deal with the particular code sequences that compiler, or family of compilers, generates for a table lookup. This technique is used often since most tools deal with only a particular set of compilers; for example, TracePoint just processes Windows binaries generated by the Microsoft C++ compiler [Tra97]. In the presence of optimized code, patterns do not tend to work very effectively, even when the code is generated by a compiler known to the pattern recognizer.

We developed a machine-independent technique to recover computed jumps. This technique was developed after studying code generated by C, C++, Modula, and Pascal compilers, and was tested on both SPARC and Pentium architectures using moderately sized programs (e.g., the SPEC 1995 benchmark suite).

The technique can be summarized as follows. At an indexed jump instruction, perform backwards slicing of the code up until either the start of the procedure containing the indexed jump or when the terminating conditions for a computed jump slice are met. Copy propagate values in the slice's RTL representation and remove any dead instructions from the slice. Transform the remaining instructions to a normal form and compare against the three normal forms documented in [CE01]. If a match exists, the bounds of a jump table can be determined, as well as the size and location of the jump table, and the form of its contents (whether addresses or offsets). In this case, replace the indexed jump by a HRTL computed jump having the appropriate target addresses. Finally, follow each target address to decode more code. This last step significantly increases code coverage.

Speculative Decoding

To get better coverage when decoding the code section, particularly for binaries that were compiled using compilers for object-oriented languages which use virtual tables, we implemented a crude but effective speculative decoding technique that was used after the standard decoding phase finished.

The technique is simple. For every gap in the code section where there are untranslated bytes, we decode the bytes speculatively as if the gap was a procedure in the source binary. If an illegal instruction is encountered, the translated code is discarded, and speculation continues at the next gap.

The source and target code addresses of all translated procedures, including these "speculative procedures," are put into a table. This essentially creates a forest of translated code trees. The table is used at runtime to handle indirect branches through registers and virtual method calls.

To support object oriented programs, register calls are translated into a call to a small runtime procedure. That procedure, in turn, calls a C language function to look up the address in the table and jump to the target address if found. A simple binary search is used. Note that the address looked up for a register call will be for a procedure in the *source* binary; the procedure table is used to find the address of the corresponding procedure in the translated (*target*) binary. The machine code of the source binary is needed, in case the target of the register call is not found in the table of (source, target) address pairs. This would mean that despite the speculative decoding, not all source functions were found and decoded. All live potential parameter locations are passed to the function handling the register call.

This simple technique was found to work reasonably well, although in some cases overlapping procedures were generated, which wasted some space in the target binary.

Helper Functions

The runtime library for the SPARC architecture, as well as that of other architectures, provides some architecture-specific functions to perform simple arithmetic functions efficiently. For example, the `.umul` function performs an unsigned multiplication. We dubbed these functions "helper functions."

Helper functions have a predefined semantics and their parameters are known and fixed in number. We search for these functions during the decoding phase and, when matched, replace them with the helper function's semantics in terms of RTLs. For example, we replace a call to `.umul` by the RTL expression `$r[8]*$r[9]`.

2.3 Back End

We experimented with different types of back ends to try to determine the benefits of one approach over another. In 1999, the UQBT framework initially supported a simple C back end that used the C compiler as a macro assembler. This allowed us to have our first complete binary translator for several different source and target machines, and we instantiated four different translators at that time. Over time, we added additional back ends and Figure 4 shows the different kinds of back ends that we eventually built. We supported commonalities between these back ends using an abstract class that we called the code expander. In its present form, the code expander does not make use of PAL specifications for determining calling conventions, but rather makes use of a predetermined one.

2.3.1 The C Back End

The C back end was the first back end we wrote for the UQBT framework. The idea behind it is simple: translate HRTL code into low-level C code, without attempting to recover any "high-level" control flow structure (no `do`, `while` or `for` loops). All control flow is in the form of gotos. The binary translator does not "understand the data," but merely replicates the bit pattern changes that the source program performed. As a result, the generated C code does much casting of expressions between different types, and all memory references are of the form `*(<type>*)<address expression>`. Automatic (data) type coercion of the C programming language must be avoided. Actual conversion operations (e.g., sign extending of integers; converting from integer to floating point) is often only expressable in the C language using casts. Combined with the often redundant and copious parentheses, the resultant code is difficult for humans to read. In fact, there is an outstanding bug with the GNU `gcc` compiler, where a particularly "ugly" expression is handled incorrectly when optimized. Figure 5 illustrates some of the generated low-level C code.

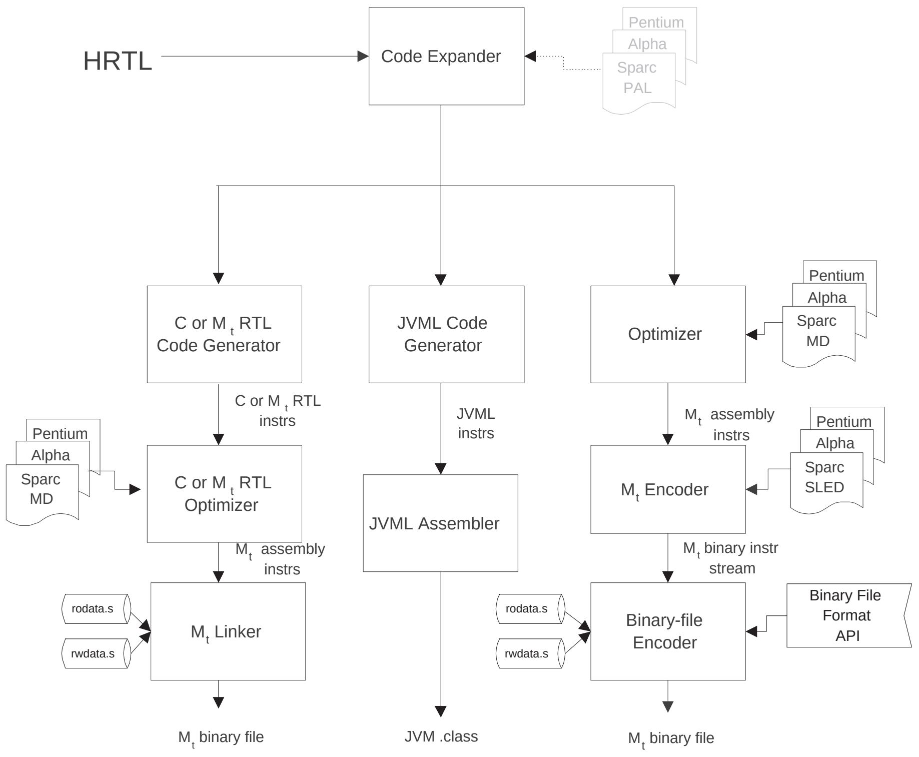

Figure 4: Conceptual View of the UQBT Framework's Back End

For each procedure we generate one C language file, and for the whole program one makefile. Data sections are copied without modification into assembly files (using `.byte` statements) and a map file is generated to force the data sections to be loaded at the same virtual address as in the original source binary.

The generated code is then compiled using a C compiler (normally using GNU `gcc` or Sun Microsystems `cc` compilers) and then linked with the original data sections of the source program. Note, however, that some features such as endianness support require powerful macro support to implement efficiently, which in practice may limit the choice of target compiler to GNU `gcc`. The source binary's code section is optionally also copied to the target binary.

We found it necessary to keep an accurate record of the "current type" of an expression while generating its code. The reason for this is that the C compiler does the same thing, and it bases its decision on whether to emit conversion operations on the current type of operands. In some cases, the only way to tell the C compiler to emit code to perform a certain operation is to correctly cast the operands. An example is the right shift operation; the C operator for both arithmetic and logic right shift is `>>`; which instruction is emitted depends on the type of the operands. Thus, when a sub-expression of a given expression is computed, we take great care to specify the correct type for the sub-expression. For example, the only sub-expression in the floating point to integer expression `ftoi(64,32,m[1000])` is `$m[1000]`, and its type is 64-bit float.

```
#include "uqbt.h"
int  main(int  v0, int  v1) {
    int  v2, v3, v4, v5, v9, v10;
    double  v8;
    union {
        double  d65;
        struct {
            float  f34;
            float  f35;
        } f;
    } d65;

    v3=67584;
    v2=(v3)|(304);
    v4= *((int *)(*(unsigned int *)&v2));
    v5= *((int *)((*(unsigned int *)&v2)+(4)));
    v9=v4;
    v10=v5;
    d65.f.f34=*(float *)&v9;
    d65.f.f35=*(float *)&v10;
    v8=d65.d65;
    d65.f.f34=*(float *)&v8;
    *(int *)&d65.f.f35= *((int *)(((int*)(&v8))+(1)));
    /* ... */
}
```

Figure 5: Sample Generated Low-level C Code.

Another challenge was the handling of overlapped registers. For example, the 32 bit Pentium register `$eax` is overlapped by the 16 bit register `$ax` and the 8 bit register `$al`. In many architectures, single precision floating point registers overlap double precision registers. Our approach to this is to ignore the overlap in the internal representation (RTLs and HRTLs), and give the overlapped registers different register numbers. To handle the overlap, we emit C code that declares the overlapped registers as unions. Since this makes the code even more difficult for humans to understand, this declaration is done only when needed. An extra pass in the back end is used to find all registers used in a procedure.

2.3.2 The JVML Back End

The bytecode for the Java(TM) programming language (i.e., the Java virtual machine language (JVML)) back end was written as an experiment in translating machine code to Java bytecodes.

We translated HRTL code into classfiles for the Java virtual machine (JVM(TM)),[5] using the Jasmin JVM bytecode assembler [Mey97]. The JVML back end operated much like the C back end but was more limited in what it could translate. It was single-pass and produced a bytecode assembly file by translating each HRTL procedure into the bytecodes for a Java language method. A postorder traversal of each HRTL's sub-expressions produced the stack-oriented JVM bytecodes for that HRTL.

The resulting class file made use of a compatibility library class that emulated the source platform's libraries on the target JVM platform. This library also included methods to support a limited subset of C's memory model. Memory was represented using a single large, pre-allocated Java language byte array `memory`. References to a data item at a particular address `x` were read from or written to `memory[x]`. The compatibility class contained a method that was called first when the translated program began execution. That method initialized the `memory` array by copying the program's command line arguments and the source program's bss and data segments into the appropriate array elements. `malloc` was implemented by a method that allocated a block of storage from `memory` then returned that block's offset. Pointer arithmetic done by the source program was implemented using arithmetic operations on `memory` offsets. Function pointers were not supported.

This experimental JVM back end was suitable for translating small numerical SPARC architecture binary programs. It supported 32 and 64 bit floating point and signed 8, 16, 32, and 64 bit integer types, but did not implement the entire `C` memory model. Overlapping registers were not supported. It also did not implement unsigned integers since these are not directly supported by the JVM. It would have been possible to add compatibility methods to support unsigned integers, but this was never done.

2.3.3 The RTL Back End

The RTL back end was an experiment at having more control over the types of optimizations that were applied to the generated code; control that was not available with the C back end as the C compiler would dictate what optimizations to use.

The idea behind the RTL back end was not only to have better control over the types of optimizations applied to the generated code, but it was also our feeling that interfacing at the RTL level would provide us with better target code, as instructions would be more easily expressible at the RTL level than through C language statements that include numerous type casts.

For our experiments, we used the Very Portable Optimizer VPO [BD88, VPO98]. An RTL interface to VPO was made available in 1999. This interface supports the SPARC, x86 and ARM architectures among others. VPO makes use of machine description files to describe the instructions of a machine. VPO implements a series of machine-independent optimizations and allows the user to write machine-dependent optimizations to support a new machine. We reused the existing machine descriptions and optimizations rather than writing our own.

[5]The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java(TM) platform.

Our RTL back end was mainly a translator of HRTL to M_t-RTL code (in VPO's RTL format), which was passed to the VPO retargetable optimizer through its RTL interface. The optimized code emitted by VPO was then linked with the original data sections to generate the final target binary.

2.3.4 The Object Code Back End

The object code back end was written as an experiment in interfacing with an optimizer at the object code level: i.e., it emits a binary executable directly, without any particular intermediate representation. This back end enables the use of any optimizer that can operate on an executable binary.

The object code back end translates HRTL code to M_t-assembly without performing any but the very simplest register allocation. It allocates all variables in memory, in the procedure's local stack, and emits loads and stores to those memory locations. It relies on the optimizer to do efficient register allocation. The back end encodes M_t-assembly instructions into their binary representation using SLED machine instruction descriptions and the New Jersey Machine Code toolkit's encoding routines [RF97]. We experimented with the use of this back end in conjunction with a proprietary SPARC architecture post-link optimizer.

2.4 Runtime Support

The UQBT back ends generate source code for the translated program in either the C, JVML or machine code languages, by providing a series of files (one per translated procedure); the data files in assembly language format; and a makefile to build the final target binary.

Figure 6 illustrates the components of the target binary. Basically, the source text section is translated into an equivalent text section, the read-only and read-write data sections are copied as is to the target binary, and several sections are added for interpretation support.

A static binary translator requires some runtime support in the form of an interpreter or dynamic translator to emulate those parts of the source program that could not be translated statically. In the UQBT framework, we provide for speculative decoding and recovery of jump tables as analyses that aid in achieving a greater coverage of translation of the source text section (see §2.2.3). However, no available static technique can guarantee full coverage of decoding of the text section for all input programs, therefore, interpretation support is needed in the generated target binaries.

A UQBT-generated target binary that requires interpretation support should add the following sections to the target binary: the source to target address mapping (to aid in determining when a piece of code has been translated); some basic interpretation support (to check for translated code that may be part of a pre-translated tree of code resulting from the speculative decoding analysis, or to invoke an interpreter); and the original (source) text section (for the interpreter to interpret instructions from this text section).

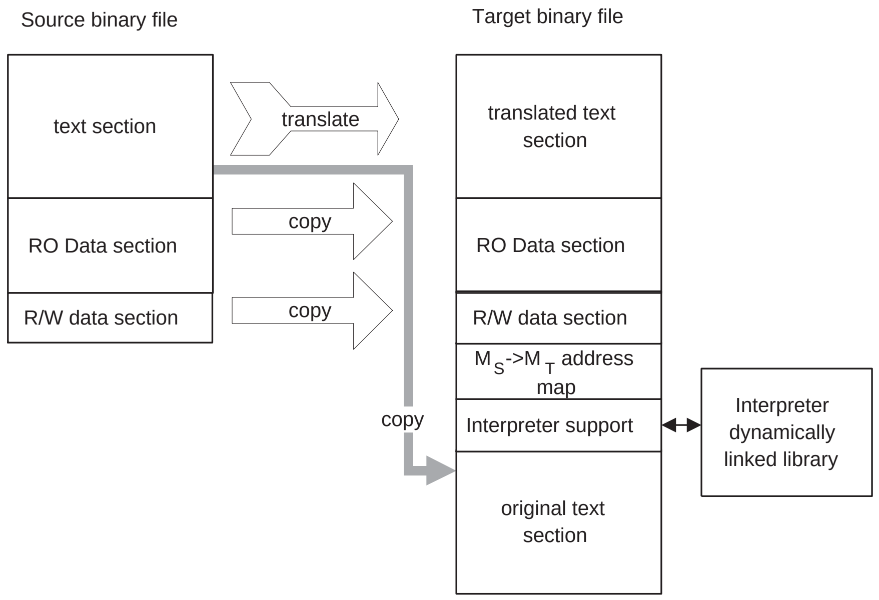

Figure 6: Internal Format for Source and Target (Generated) Binaries

We experimented with two different interpreters: a UQBT-RTL interpreter and a source machine instruction interpreter (emulator). The former interpreter was slow and not fully tested. The latter emulator worked well and was capable, for example, of running all SPEC 95 benchmark programs. Neither interpreter was fully integrated with the UQBT framework since we did not want to become dependent on them: one of our goals was to use analysis to avoid, as much as possible, relying on runtime support.

In principle, integrating an interpreter is relatively simple to do. However, it does require a map of source-to-target register locations. This is a mapping of source registers to the translated virtual (in memory) and physical (in target machine register) locations. Integrating a runtime emulator also requires support for determining when (i.e., at what instructions) it is safe to return to translated code. Safe points normally include the return from a procedure for example. At a safe point, the state of the translated program needs to be mapped based on the state of the emulated machine.

3 Instantiation of Translators

The UQBT framework supports the instantiation of different translators with specifications of the source and target machines. For example, to instantiate a SPARC to SPARC translator, the UQBT framework is configured to use specifications for the SPARC architecture and APIs for the Solaris environment, and any other SPARC-related information needed to generate HRTL code. The C back end is normally used as the back end. This results in a `uqbtss` (SPARC to SPARC) translator. In a similar way, a `uqbtsp` (SPARC to Pentium) translator, a `uqbths` (PA-RISC to SPARC) translator, and other translators can be instantiated.

In order to instantiate a new translator, the APIs and specification files for the source platform must be described. Testing of each step is important in order to ensure correctness of translation; hence, we provide ways to test the output from each step in the translation process.

The instantiation process consists of the following steps:

1. Binary file decoder support
2. Instruction decoding support
3. Instruction semantics support
4. Control transfer support
5. Procedural abstraction support
6. Machine-specific support
7. Code generation support

Each step is described in some detail in the following subsections assuming that the given API or specification is not already available. In those cases where it is available, there is no time overhead to reuse such an API or specification, other than to perform that step's tests.

3.1 Binary file Decoder Support

In order to support different binary file formats, such as ELF, PE or PRC, the UQBT framework exports a loader API called `BinaryFile`, which is an abstract class that makes available functions to:

- construct, load and unload binary files,
- extract information from sections,
- extract information from symbol tables (if any),
- extract information from relocation tables (if any),
- display/dump the contents of all headers, and
- obtain initial program state information, such as entry point(s), and determine whether a given address is a dynamically-linked address or not.

For a given binary file format BFF_i, the `BinaryFile` API is realized by implementing the derived BFF_iBinaryFile class. In this class, some extra functions are used to implement the `BinaryFile` interface, depending on the complexity of implementing the API functionality.

This first step is tested by displaying/dumping onto the standard output stream the contents of all headers and sections read from the file. These contents are then checked manually against the contents produced by a binary file dump tool, commonly distributed with operating systems these days. For example, in the Solaris environment, the `elfdump` tool displays the contents of an ELF's file headers. A typical size of information dumped is around 500 lines of ASCII text. Similar results are achieved by using GNU's `objdump` tool.

The UQBT framework provides a skeleton file `bffDump.cc` that is the basis for creating a binary file dumper. The program displays the raw contents of each of the headers and fields of the given binary file.

3.2 Instruction Decoding Support

This step is the most time consuming one in the instantiation process as the binary translator writer needs to become fully familiar with the M_s instruction set to be supported. This step involves reading through architecture manuals and representing the information for instructions in terms of the SLED language. SLED allows for describing the types of instructions in a given machine, the fields of such instructions, and the values of particular instruction fields. Describing the instructions, in essence, maps binary bit streams to assembly instructions. In our experience, depending on how complex the instruction set is, and how familiar the developer is with the instruction set and the SLED language, this step can take anywhere from 2 weeks to 2 months.

The New Jersey Machine Code toolkit (NJMCTK) not only supports the SLED language, but also supports an abstraction to decode and encode machine instructions. The decoding abstraction provides for a *matching* statement whose syntax resembles that of a C language `switch` statement, and whose semantics are equivalent to matching the series of bits that make up an instruction and returning the values of the variable fields of an instruction. In this way, a decoder (disassembler) of machine instructions can be written, and the time to do this is minimal (less than 1 day); in fact, it is possible to automatically generate such a decoder. This decoder is tested against an existing disassembler for the source machine through a series of test cases. This is possible as most UNIX(R) systems include a `dis` utility in their distribution or one is included as part of the GNU `binutils`.

Sample decoders are provided with the UQBT distribution. A skeleton driver program is provided to build a disassembler for any machine. The function `disassembleOneInstruction` is implemented as part of the matching file to be provided by the binary translator writer.

3.3 Instruction Semantics Support

Writing the SSL specification is routine once the SLED specification has been written, because by then the developer is very familiar with the instruction set. The learning curve for the SSL language is not too demanding. In our experience, writing an SSL specification takes about one third of the time that it takes to write the SLED specification.

We currently test an SSL specification using an extension to the decoder. Instead of dumping raw assembly information, we cause the decoder to dump RTL information. That information is manually checked against the assembly output.

We initially tested this step using an RTL encoder. That is, after generating RTL instructions, we would encode them using a different tool, and compare the execution results of the translated code against the execution of the source M_s binary program. We used the VPO optimizer for this purpose since it operates on RTLs. However, VPO requires knowledge of live register information at each procedure call site, hence, only simple programs can be translated using this testing scheme, as no live register analysis has been done on the RTLs at this point in the translation process.

3.4 Control Transfer Support

The control transfer API is not formally specified, although it is hardcoded in the required code for a translator. Control transfer instructions such as branches and calls are matched against in the instruction decoder to transform control transfer M_s-RTLs into an "in between" HRTL form that we call I-RTL. I-RTL stands for "independent RTL", as they resemble HRTL instructions though their operands have not been obtained yet (these are obtained through procedural abstraction analysis).

This step is almost trivial and can be implemented in one day after looking at skeletons of existing translators.

3.5 Procedural Abstraction Support

The PAL specification is based on the operating system ABI conventions for setting up the call stack frame, parameter passing and calling conventions. Specifying this information takes little time (two days) once the information to be specified has been determined.

Testing of the PAL specification and the resultant HRTL instructions can only be done by compiling to the same machine, then comparing the execution results of the generated target binary against those of the source binary. For example, for a translation from a (SPARC,Solaris) binary, we compile HRTL back into a (SPARC,Solaris) binary and then compare the results of running the two programs.

Programs do not necessarily adhere to the OS ABI calling conventions. Therefore, commonly used non-ABI conventions need also be included in a PAL specification. Such cases are normally detected by running programs compiled by different compilers from the source platform; once a non-ABI convention is reached, the translator will warn the user about it.

3.6 Machine-specific Support

This step varies from machine to machine and involves the removal of machine-specific features that are not supported by the UQBT framework in a machine-independent way. The features are normally too specific to be available in multiple machines, and so the binary translator writer needs to provide some analysis support to transform I-RTL code that exhibits such features into HRTL code. Depending on the level of complexity of the feature at hand, this step can take anywhere from less than one week to one staff-month. Some of the support that we have introduced for different front ends was explained in §2.2.2.

3.7 Code Generation Support

This last step consists of choosing an existing back end and possibly adding specific support for the target machine. The C back end is the most versatile one and the one that takes the least amount of time in instantiating; all that is necessary is to use a C compiler on the target machine to compile the generated code.

The RTL Back End

In order to instantiate a M_t-RTL back end for a machine M_t, two steps are needed: create a HRTL to M_t-RTL translator, and support M_t in the optimizer (VPO in our case). We experimented with generating code for M_t machines that VPO already supported, so we only needed to write the M_t-RTL translator. This step requires writers to satisfy VPO's M_t-RTL invariant: that each M_t-RTL must map to one M_t assembly instruction. This is not technically difficult but does require care, for example, to properly "downshift" complex HRTL expressions to a sequence of simple M_t-RTLs.

Writing an M_t-RTL translator normally takes 3-5 staff-months for someone experienced with VPO. The difference in time required depends mainly on differences between the different target machines. For example, the ARM processor has limited support for immediate values in instructions. This means that constant addresses and values larger than 8 bits required additional work to implement. As another example, the ARM procedure call standard can cause a 64 bit floating point parameter to be split between memory and the stack, and extra translator code was required to deal with these.

The Object Code Back End

Instantiation of an object code back end requires two steps: generating a HRTL to M_t translator (which provides the same challenges as generating a HRTL to M_t-RTL translator), and encoding the generated M_t machine code instructions into a binary stream that is then passed to a binary file encoder.

Translation of HRTL to M_t assembly code takes roughly one staff-month for integer-only support. If a SLED specification for the target machine can be reused, encoding into an M_t binary stream takes one day; otherwise, support for that encoding can take up to a month.

The JVML Back End

Using the JVML back end is as straightforward as using the C back end, except that the binary translator writer must be aware that the JVML back end generates code for data addresses by allocating a large array and placing read-only data at those addresses. Hence, if addresses in the source binary are too large, the target program will attempt to reserve a large amount of space when it starts, which might result in an out-of-memory condition. As previously mentioned, the

JVML back end was only tested when translating (SPARC,Solaris) binaries that made limited use of pointer operations; it did not support the full C memory model.

4 Experience

Researchers, students, and engineers have used the UQBT framework to experiment with translations between a number of different architectures. As researchers, our main focus was to experiment with new machine features in order to capture those features in a general framework, so that more machines could be supported by the specification languages and APIs we used. As engineers, we were interested in seeing how well the generated target binaries performed. We always tested their execution time against that of a natively-compiled binary running on the target platform to determine what overhead was introduced by the translation process. We were also interested in how well the translated binaries ran: that is, what experience a user would get from running them.

This section describes our experiences and those of our students in using the UQBT framework. We group those experiences based on the focus of interest when developing a particular translator: that is, whether most of the work was done in instantiating a new front end or in writing a new back end. Once such front ends and back ends are available, they can be mixed and matched to instantiate a particular translator of choice, by choosing the source M_s and target M_t platforms. We have made an effort to quantify the amount of time spent in writing such translators, noting that different levels of experience with UQBT imply differences in time needed to develop a particular component.

4.1 Experience with the Development of the Initial UQBT Framework

The initial UQBT framework was developed with the (SPARC,Solaris) and (x86,Solaris) platforms in mind. The SPARC and x86 architectures provided us with two significantly different (RISC and CISC) source and target machines. We used the C back end and Solaris as the multiplatform operating environment.

The initial UQBT framework took 3 years of design and development by 2 part-time researchers, 1 engineer, and 3 undergraduate students. The students either worked on a 1-year project or during the summer break. A total of 5.7 staff-years were spent during those 3 years. Four different translators for the Solaris operating environment were instantiated in the initial development; namely

- `uqbtss` (a SPARC to SPARC translator),
- `uqbtsp` (a SPARC to Pentium translator),
- `uqbtpp` (a Pentium to Pentium translator) and
- `uqbtps` (a Pentium to SPARC translator).

A translator from and to the same architecture, e.g., `uqbtss` and `uqbtpp`, not only serves as a useful test vehicle but helps us to determine the overhead introduced by translation.

Developing the UQBT framework was an order of magnitude more difficult than writing a hand-crafted binary translator from scratch. This was due to the extra complexity required to support retargetability and to separate machine-dependent concerns from machine-independent analyses.

The use of specifications allowed us to quickly instantiate binary translators for other machines. We originally worked on the `uqbtss` translator. Once that translator worked, we were able to instantiate a `uqbtps` translator in a few months. This required writing the Pentium SLED and SSL specifications and the PAL specification for the Solaris-x86 environment, and implementing the analysis to transform floating point stack-based references to a flat register model. Once we had both translators, the SPARC and Pentium machine descriptions, and the PAL descriptions for the Solaris-SPARC and Solaris-x86 environments, instantiating `uqbtsp` and a `uqbtpp` translators was straightforward.

	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

Static SPARC to Pentium Translation

	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

Static Pentium to Pentium Translation

Figure 7: Running Times and Code Sizes for Static Translators Instantiated from the UQBT Framework in late 1999

The results reported in [CEU$^+$99] for the initial framework are duplicated in Figure 6. The test programs are: Fibo(40), which calculates the Fibonacci of number 40 and has 63 lines of assembly code; Sieve(3000), which calculates the first 3000 primes and has 61 lines of assembly code; and Mbanner(500K), a modified version of the command-line `banner(1)` program, which loops 500,000 times to display argv[1] ("1234567890" in this case) and has 204 lines of assembly code and a read-only data section of 336 bytes. For all programs, the time in seconds to execute the program on the target machine was compared to the time needed for the same source program compiled by a native compiler on that target machine; this allows us to determine 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(R) II, 250MHz machine with 320 MB RAM running the Solaris 2.6 environment. Pentium results were obtained on a Pentium MMX, 250 MHz machine with 128 MB RAM running the Solaris 2.6 environment. The source binary programs (input to the translator) were all compiled with GNU `gcc` 2.8.1 -O4.

Translated code programs used two different optimizing C compilers; GNU `gcc` 2.8.1 and Sun Microsystems `cc` 4.2, on both SPARC and Pentium machines. Source programs were compiled on the target machines (for comparison purposes) using `gcc` 2.8.1 with -O0 and -O4 options, on both SPARC and Pentium machines.

These preliminary results, and those obtained later on with other binaries and translators, show that generating good quality target binaries is possible for programs that do not rely on a runtime interpreter. These results also show that the quality of the generated code is comparable to that produced by hand-written binary translators. In fact, the separation of machine dependent from machine independent concerns aids in the development of better analyses techniques.

Translators instantiated from the UQBT framework cannot translate all source binaries, mainly due to bugs that are time consuming to eliminate due to a lack of debugging support in the framework and the small number of people working on the project at any one point in time. The largest programs we successfully translated were medium sized: e.g., the SPEC `go` benchmark is about 350 KB. Programs as large as the SPEC `gcc` benchmark (1.6 MB for the SPARC binary) did not fail to translate because of any inherent limit of the translator, such as a data structure overflow; they failed because the probability of not coming across a translation problem becomes vanishingly small, and the cost of debugging the translation (without better debugging support) becomes very high. This points to the need for significantly better debugging support in a translation framework such as UQBT (see §5.3).

4.2 Front End Experience

The two front ends that we describe here were written after the initial SPARC and Pentium translators had been tested. These front ends for the Motorola 68328 and the HP PA-RISC architectures were intended to be used with the C back end. As a result, our emphasis, besides getting the new front ends to run, was to get experience with reuse in the system, to know how long it would take to support other platforms, and to prove that we could support translators for these architectures that differ in so many ways from the earlier ones we supported.

4.2.1 The (68328, PalmOS) Front End

An experiment to support the Palm Pilot PRC binaries was made in early 2000 to determine how hard it would be to translate these binaries to another architecture, particularly at a time when other architectures were becoming popular in the personal digital assistant (PDA) market. The Palm Pilot makes use of the Motorola 68328 processor which is a CISC processor that runs at 16MHz (later models 20 or 33MHz) and has from 2 to 8 MB of RAM. The experiment consisted of translating (68328,PalmOS) PRC binaries to PRC binaries on the same platform. This would allow us to determine what problems, if any, would be encountered in processing PRC binaries.

This experiment lasted for 3 months with 2 engineers working in parallel, but in different countries and in non-overlapping time zones (California, US and Queensland, Australia). Both engineers were familiar with the UQBT framework, its APIs, and specification languages; therefore, no learning curve was incurred in that respect. Throughout the process, significant amounts of time were lost due to the inability to find the right documentation or tools. For example, there was little documentation about the PRC binary file format itself, as well as the ABI conventions followed by those binaries. In terms of tools, since we were going to use the C back end, we wanted a C compiler that could cross-compile to the 68328 architecture in a batch way: i.e., that made use of the UQBT-generated Makefile instead of requiring options to be entered interactively by a user (as did the then standard C compiler for the Palm Pilot—the Metrowerks Compiler). We also needed to learn how to link and generate PRC files that would run on a Palm Pilot, as different programs require a different ID number.

Overall, we estimate four staff-months were spent in the following activities: supporting the binary file API for the PRC format, adapting an existing 68000 SLED specification and integrating it into the skeleton decoders used by the UQBT system, writing an SSL specification for the 68328, writing a PAL specification for PalmOS conventions, and extending the removal of the frame pointer analysis to support the removal of the data pointer (transforming the code to use offsets from `%agp`). This was the first architecture we had used that made use of pre-decrement and post-increment addressing modes, hence some changes to the decoders were needed. The size of an integer is defined by the PalmOS API as 16 bits, but 68000 instructions come in 8, 16, and 32 bit sizes. The Palm Pilot compilers would sometimes pass two constant integer parameters using a single 32-bit push instruction. This necessitated some extra logic in the parameter processing code.

This was the first architecture we described that had PC-relative addressing modes. These seemed to be used only to reference objects in the code segment, which were mostly callback functions and constant strings. When a PC-relative addressing mode was decoded, the destination address was calculated as usual, but a special identifier was used to distinguish it. When the back end found one of these constants, it looked for a matching function prologue. If a match was found, it assumed that the destination was a function. If not yet decoded, it was decoded immediately. The name of the function was used in the C output, thereby “translating” the address from source machine address space to the target machine address space. If no function prologue was found, some heuristics were used to guess if the destination was a string. If so, a C string constant was emitted in the output C code. In the case where the destination did not appear to be a function or a string, it was translated as a constant with a warning comment.

This was also the first architecture we encountered that regularly uses callback functions since all Palm Pilot-based programs are GUI-based. This caused some problems with parameters and return values, which we worked around manually since we did not have a fallback interpreter implemented. We manually re-ran the UQBT translator, giving it the callback function address explicitly; such a re-run of the UQBT translator generated C code for that function, which we manually added to the Makefile. The automated solution would have been to use knowledge of the PalmOS library calls that take pointers to functions, to effectively prototype the callback functions when these calls were encountered.

We were able to translate small Palm Pilot binaries that are distributed with each Palm device sold.

In theory, if we used a back end that generated code for the ARM, a translator from 68328 to ARM could be instantiated. In such case, if the same operating system is available on both platforms, the same set of libraries would be available and a cross-translator could be instantiated. If the operating systems were different, a compatibility library would need to be built to support differences between the operating systems and their support libraries. In our experiments, we could not instantiate a (68328,PalmOS) to (ARM,Linux) translator because we did not write a compatibility library for the PalmOS-Linux operating systems. If, instead, an (ARM,PalmOS) platform became available, instantiation of a (68328,PalmOS) to (ARM,PalmOS) translator would be immediate.

4.2.2 The (PA-RISC, HP-UX) Front End

The Hewlett Packard Precision Architecture RISC (PA-RISC) front end for UQBT took approximately twice the time to write as other front ends. There were many reasons for this; most could be summarized as surprising aspects of the architecture that were not planned for in the original UQBT design. One undergraduate student and two engineers worked on this front end for a total of 10 staff-months, 6 of them by the student. The student was not familiar with the rest of the UQBT framework or its specification languages. His main job was to write SLED and SSL descriptions for the PA-RISC instruction set. Three months were spent in joint work by the student and one of the engineers. These experiences are described in the following paragraphs.

HP SOM

PA-RISC has its own proprietary binary file format (HP SOM, for Standard Object Module). The HP SOM format is simple, though coping with the various details was time consuming: e.g., some symbols exist multiple times in the symbol table but with different values. SOM support took about 0.5 staff-months. The SOM format lacks a read only data section, so string constants and other data appear in the text section. This requires that the text section be copied to the target binary for data references.

SLED

Features of the PA-RISC instruction set made it hard to write a SLED specification for this instruction set. For example, practically every instruction field is non-contiguous (some fields are split into 3 subfields). The non contiguous fields complicate the SLED descriptions, though the SLED language is general enough to support them with some verbosity. About 3 staff-months were spent on writing this specification file.

SSL

Almost all PA-RISC instructions have “completers” that are small fields specifying variations to the instruction semantics. Many of these have effects on registers other than the destination register. For example, the MA (Modify After) and MB (Modify Before) completers affect an address register. Other completers optionally annul the next sequential instruction. These completers necessitated changes to the SSL language. We added guarded assignments to support conditional execution of assignments. About two staff-months were spent on extensions to the SSL language and the necessary changes to existing SSL files. Changes to the back ends were also needed to support guarded assignments; for example, the C back end emits an `if` statement around any assignment that is guarded.

Helper functions

The PA-RISC architecture has a series of “helper” functions (e.g., `$$RemU` for unsigned remainder), similar to those in the SPARC architecture. However, in the PA-RISC architecture, these are so-called “millicode” routines with a different calling convention than other functions. These were easy to implement since fixed registers for input and output are used, so their semantics are readily encoded as a sequence of RTLs.

PAL

HP-UX has a calling convention that directs parameters of different types to be placed into different registers. For example, a double precision floating point value is passed in a different register to that of a single precision float, and integers are passed in another register again. In addition, a double precision parameter “uses up” two integer slots and is “aligned” (which complicates the rules for what register will be used for later parameters). This calling convention complicates analysis of parameters in the context of register calls. Basically any integer, float or double register could be a potential parameter, and if two or more of these are live at the call, it is not possible to determine which is the actual parameter. Many register calls are in fact virtual function invocations. Other research based on the UQBT framework has shown that it is often possible to find a candidate callee for a register call, using analysis similar to that used for register jumps that implement `switch` statements [TC02].

Under some conditions, including simple test programs, the linker automatically inserts a “parameter relocation stub”. These stubs do not contain standard prologues or epilogues, and end with a jump instruction. The whole stub is currently recognized as a idiom (pattern), and is dealt with as a special case.

A small set of prologues and epilogues seems to suffice in the PAL specification. There is one caller prologue (the standard call), no caller epilogue, two callee prologues (three if you count the parameter relocation stub), and five callee epilogues. Most of these have plenty of variations (e.g., in the number of registers that are saved), but these variations are comfortably handled with the PAL language.

The fact that the PA-RISC stack grows upwards, towards higher memory, meant that support for "inverted stacks" was needed. We added support in the PAL language for a `STACK IS INVERTED` keyword which prompts the procedural analysis to use different stack offsets.

Code Generation

Most PA-RISC programs begin their text section at address 0x1000, which is just 4 KB above address zero. The SPARC platforms we targeted use a page size of 8K, so when we copy the source PA-RISC binary's text segments into the translated SPARC binary, that PA-RISC code begins halfway through the first SPARC memory page. This necessitated some changes to the code that generates target binary files. In effect, we had to not copy the first 0x200 bytes of the code section (in the example code we used, this was never a problem), to make room for the ELF header. This appears to be a limitation of the way that Solaris and Linux environments map ELF files into memory. On other architectures, this is not a problem, because the first section is at a higher address (e.g., 0x10000 for the SPARC architecture).

4.3 Back End Experience

Our experience with writing different back ends was driven by the fact that we wanted to determine whether hand-written back ends would provide better results than the C back end, and whether it was easy to exercise more control over such back ends by performing optimizations at the RTL or the machine code level rather than in a more abstract representation. For all back ends, we were interested in providing a retargetable solution, and so the back ends were written in such a way that more than one target machine was supported.

4.3.1 The JVML Back End

There were different incarnations of the JVML back end throughout the lifetime of the UQBT project. Although this back end has been experimental at best, we were interested in determining the limits of translation for (SPARC,Solaris) binaries to the Java virtual machine (JVM) model.

The JVML back end was originally developed as a proof of concept; one month was spent in determining whether small (SPARC,Solaris) programs could be translated to JVML. Once this was found to be true, three staff-months were spent by an undergraduate student to modify the GNU `gcc` compiler to generate JVML code. This allowed us to use the C back end to generate C code that could then be translated into JVML and assembled by the Jasmin assembler. The student wrote and tested a GNU `gcc` machine description for JVML. Unfortunately, the GPL nature of `gcc` prevented us from including this back end with the rest of the UQBT framework distribution.

A new version of the back end was written using the Java programming language. The experiment consisted of 5 staff-months of work; 3 staff-months by an undergraduate student who was not familiar with the UQBT system, and 2 staff-months by an engineer who was learning about the system. No parallel work was done in the 5 months. This final back end supports integer and floating point operations and all operand sizes, a range of type conversions, a full set of HRTL operator support, simplified memory management support, and an extended runtime library that supports a wider variety of source Solaris environment programs.

A number of small (SPARC,Solaris) programs were translated to the JVM platform. Performance of the translated programs was compared on a SPARC machine against the source SPARC binaries as well as native Java language programs that implemented the same functionality. Most translated programs have runtimes that are 5-10 times longer than those for the source SPARC binaries when run on the same hardware. This was largely due to the need to assemble multibyte values a byte at a time when reading them from the byte arrays holding the source program's data sections. This byte assembly was done to cope with alignment restrictions. Programs such as `fibonacci` that used few data section values performed faster and sometimes approached the speed of the source program. Execution time for the translated programs, compared with native Java bytecode binaries running on the Java HotSpot(TM) 1.4 virtual machine on a SPARC processor was varied: some were as fast as the Java language version of the programs, others were slower. These differences had, in part, to do with our (limited) support for C memory management, which is not native to the Java programming language.

4.3.2 The Object Code Back End

The object code back end was a 3 staff-month experiment by a graduate student who was experienced with binary translation but not familiar with the UQBT framework. The aim was to integrate UQBT with a proprietary post-link optimizer to experiment with integration at the machine code level via object code files.

This back end supported integer-based programs on (SPARC,Solaris). It transformed all variable and register locations into local variables, i.e., into stack locations in a procedure's local stack frame. This resulted in simple code that relied on a post-link optimizer to perform register allocation as well as all optimization of the code. The back end reused the SLED SPARC specification and the encoding routines available through the New Jersey Machine Code tookit (NJMCTK), in order to generate the output binary instruction stream. Encoding in the ELF binary file format was written. A number of small programs were tested, which showed that the approach was feasibile and allowed us to reuse an existing (proprietary) solution. However, the resulting code was no better than that emitted by the C back end. The back end also lacked control over what optimizations were done and where instructions were placed. We attempted to address these problems in a later back end, the RTL back end.

4.3.3 The RTL Back End

Our goals for the RTL back end were twofold: testing and optimization support. These experiments were conducted by an engineer who was familiar with the UQBT framework but unfamiliar with the very portable optimizer (VPO) framework.

Testing

For testing support, we were interested in having a way to test the result of the decoding process into M_s-RTLs in a more automated and effective way, as we previously checked M_s-RTLs by manually comparing their semantics against the original decoded assembly instructions. The idea was simple, to short-circuit the analysis phase of UQBT, namely, the M_s-RTL to HRTL translation phase, and instead immediately translate M_s-RTLs into the corresponding M_s machine instructions and pack those instructions and the data from the source program into a binary for the same M_s platform. Since the VPO framework had an RTL input interface and could generate code for multiple machines, we decided to use it for this experiment.

We found it was necessary to do a small amount of analysis in order to use VPO with the SPARC processor. We had to remove delayed transfers of control since VPO has no support for these in its input RTLs. We generated an intermediate RTL form, I-RTL, that is equivalent to M_s-RTLs after a few simple analyses such as the recovery of control transfer instructions and the removal of those delayed control transfers. No procedural analysis is performed in order to generate I-RTLs.

The experiment took 3 staff-months in part due to the need to become familiar with the VPO framework and its RTL interface. VPO requires extensive use of temporary registers in order to simplify expressions so they could be represented by a sequence of target machine instructions. Each VPO-RTL is *one machine instruction*, whereas each UQBT-RTL is one or more individual register transfer assignments, each potentially containing expressions that cannot be implemented using a single machine instruction. This meant we needed to transform (“down shift”) some UQBT-RTLs into VPO-RTLs instead of having a 1:1 RTL mapping.

Although we successfully translated a number of small and medium sized (SPARC,Solaris) binaries to the same platform, we were unable to translate some programs. This was due to the need to have accurate information about what parameters are passed and returned by called procedures and in what register and memory locations. Without such location information, for example, it is not possible to reliably translate some programs where VPO chooses to make a procedure a SPARC leaf procedure—one that shares the stack frame and registers of its caller. This is because the I-RTL back end could not ensure that parameters and result values were stored in the correct registers. Accurate parameter information is not available in I-RTL since additional HRTL analysis is required.

This experiment showed that too much extra analysis was needed for VPO to be used effectively as a SPARC-RTL encoder.

Optimization

The experiment dealing with optimizations at the M_t-RTL level was successful. It took 4 staff-months to build an ARM-RTL back end using the VPO-RTL interface. VPO provides support for the ARM7TDMI, which is implemented, for example, by Intel's StrongARM processor. This architecture has no floating point or integer divide hardware and only supports 32-bit integers.

One limitation that we found with the ARM-VPO optimizer is its limited support for ARM addressing modes. For example, this made the code sequences for byte swaps longer than would otherwise be possible. A 4 byte swap requires 10 ARM instructions currently. With support for the ARM addressing modes that support shifted operands, this could be done in 7 instructions. If the ARM back end supported conditional (predicated) instructions, just 4 instructions would be needed. The VPO team at the University of Virginia is currently adding support for additional ARM addressing modes.

Some ARM architecture quirks also contributed towards the generation of longer code sequences than originally expected. The ARM has limited support for immediate values in instructions. The ARM only supports 8-bit immediate values in data processing instructions and only 12-bit address offsets in load and store instructions. This means that constant addresses and values larger than 8 bits must be stored in memory and at a location close to the instructions that reference them. These constants must typically be stored in the code stream (with a branch around them at the start) and PC-relative addressing used to access them.

Further, the ARM procedure calling convention passes arguments using a combination of registers and the stack. It is possible for the two words of a 64-bit floating point value to be split between a register and the stack, which significantly complicates parameter passing.

In our experience, VPO's optimizations were not as sophisticated as, e.g., Sun Microsystems' C language compiler at optimization level -xO4, but it was able to produce code for several small benchmarks with the I-RTL backend that were nearly as good or as good as that produced by `gcc -O4`.

One reason we originally chose to use VPO was to have better control over the instructions emitted than, say, with the C back end. We thought this would be important in order to support a wider range of programs. For example, to accurately emulate the source program's behavior after a signal or trap, it is necessary to have accurate register and other context information, and this requires knowing exactly what information is cached in each target machine register. While VPO did give us greater control, its optimizations still replaced and eliminated instructions, moved values to memory that were in registers and vice-versa, and sometimes moved instructions. It was necessary on some occasions to work around VPO's optimizations. For example, in several cases, we emitted array instructions rather than more straightforward code in order to ensure that locations that needed to be in memory in a particular order were not reordered or stored in registers. Finally, to prevent instructions from being moved too far, we modified VPO to support instruction barriers beyond which instructions could not be moved.

4.4 Summary

Milestones	Time	Personnel
(SPARC,Solaris) and (x86,Solaris) front ends and C back end	5.7 staff-years	18 researcher-months 24 engineer-months 3.6 researcher-months 4.8 student-months 6 student-months 12 student-months
(68328,PalmOS) front end	6 staff-months	3 engineer-months 3 engineer-months
JVML back end (C version)	3 staff-months	3 student-months
JVML back end (Java version)	5 staff-months	3 student-months 2 engineer-months
Object code back end	3 staff-months	3 student-months
RTL back end	7 staff-months	3 researcher-months (SPARC) 4 engineer-months (ARM)
(PA-RISC,HP-UX) front end	10 staff-months	6 student-months 3 engineer-months 1 researcher-month

Figure 8: UQBT Milestones and Effort Spent in Terms of Time and Personnel

Figure 8 summarizes the milestones of the UQBT project and the effort spent in meeting these milestones, including a breakdown of the time spent by different personnel (namely, researcher, student or engineer) who worked on this project.

As described in §4.1, the initial framework was built with the SPARC and x86 platforms in mind. This framework was built in 5.7 staff-years over a period of 3 years, with input from 2 researchers, 1 engineer, and 3 students (at different points in time). Once the initial framework was built, the next milestones were for different front ends or back ends. Figure 8 shows these in chronological order.

Various amounts of times were used in supervision, guidance and documentation; those figures are not reported in Figure 8.

5 Discussion

As researchers and engineers we learned several things from the UQBT project, which we try to summarize in this section. We hope these will benefit people working in similar or related areas of technology. Experiences are grouped in terms of what worked well, what did not work as expected, what was missing, and what could be improved. There is also a section on the merits of specification languages versus APIs or class hierarchies.

5.1 What Worked Well

In looking back at the original goals of the UQBT project (see §1), we were successful in achieving most of the goals set for this project, as well as others. We discuss each in turn.

Goals

In learning what aspects of instruction representation and semantics were needed to perform binary translation, given that we were mainly interested in user-level code, we concluded that the machine instruction set and some of the associated machine state was needed for translation purposes. We needed to be able to represent machine instructions in a way that made explicit every behaviour of an instruction, and be able to reason about the locations and state that the instruction modifies. These locations are registers and memory locations.

In writing those aspects of instruction representation and semantics as formal machine descriptions, we ended up with two complementary views of instructions: a syntactic representation reused from the NJMCTK project (namely, SLED specifications), and a semantic representation in the form of the SSL language. SLED specifications allow users to specify the mapping between the binary and the assembly representation of a machine instruction set, as well as the machine's registers and names for those registers. SSL specifications allow users to specify the mapping between assembly instructions and their equivalent RTLs, to name new registers and declare overlaps, to define superoperators in the from of macros for commonly set condition codes, and to specify the fetch-execute cycle for the machine. The combined SLED and SSL specifications provided us with enough information to perform analyses on the generated RTLs, as every register transfer is made explicit in this representation.

Some experiments were done in deriving binary translator components from the machine descriptions. For example, machine instruction decoders have been automatically generated from SLED specifications. These decoders have been used to automatically build disassemblers. Also, machine interpreters have been generated from a combination of both SLED and SSL specifications. These interpreters have been used in a research project at Sun Microsystems Laboratories [CU02].

To understand how to implement existing machine-dependent analyses on a machine-independent RTL representation, we first experimented with writing disassemblers manually, then saw how to make this more automated and generic. We also experimented with writing binary file loaders and determining how to abstract that information in a generic way. We quickly gained the experience needed to understand how to make many machine-dependent analyses machine-independent. We ended up aiming for very generic types of analyses that would allow us to abstract away from some of the machine-dependencies expressed in the RTLs themselves. Our experience resulted in several of the analyses that transform the source binary stream into a HRTL representation.

To understand which of the machine-dependent analyses could be made machine-independent and how, we designed and implemented several analyses that transformed the RTLs into somewhat more abstract RTLs. These analyses included the computed branch analysis, and the removal of delayed branches on SPARC and PA-RISC machines. Both of these analyses were very successful. They worked well in practice for medium-sized programs (50 KB-500 KB of text section).

We further extended this concept to transform RTLs into HRTLs, a more abstract, higher-level representation of instructions. One key analysis to further abstract RTL code was to represent control transfer instructions as high-level instructions rather than as assignments to the PC and other effects on locations. This representation allowed a richer semantic meaning for groups of instructions: e.g., instead of having three RTLs for a SPARC architecture conditional branch instruction, one HRTL instruction suffices and more explicitly captures the meaning of the instruction. Other key analyses dealt with being able to attach high-level operators and operands to control transfer instructions. One analysis replaces both—the conditional branch instruction that tests some condition codes and the instruction that sets those codes—with a new HRTL instruction that does a conditional branch based on a high-level expression. Another key analysis determines the arguments and return values for procedure calls. This analysis is performed in a machine-independent way based on the PAL specifications that describe OS calling conventions, the stack frame organization, and the valid locations for passing and returning values to and from a procedure. *This analysis was the single most beneficial analysis among all the ones we developed for translation.* It allows code to be represented in a way that allows efficient target native code to be generated: it removes source machine dependencies and supports procedure calls on a target machine that use that machine's conventions, and do not require emulating features of the source machine.

Our final goal was to develop a framework for experimenting with binary manipulation ideas. The UQBT framework and its components have allowed us to experiment with various ideas and abstractions in the area of static binary translation, and to experiment with machines and platforms that we never thought of when starting the project. We have also been able to experiment with the automatic generation of disassemblers and interpreters from SLED and SSL specifications, which has worked well in practice. One of our former students has experimented with decompilation ideas based on the UQBT framework.

Other Successes

Although we mainly focussed on the use of machine descriptions, it became clear that operating system conventions could also be abstracted into OS descriptions and APIs. We created a `BinaryFile` API for the loading of binary programs that provides the minimum information needed to translate a program. We identified the intersection of information that is normally stored in binary files and made it our API. This has worked quite well. It is normally easy to add support for a new binary file format using this API.

We also ended up specifying procedural information such as calling conventions, stack frames, and parameter and return locations, in the form of PAL descriptions. As mentioned previously, the associated procedure abstraction analysis was one of the big wins in the translation process, as better target native code could be generated as a result of that analysis.

Perhaps the most surprising success was our use of the C language and C compiler as our original back end. This was suggested by a student and while we thought at the time that it *might* work as a temporary measure, it turned out to prove very useful and robust. It is relatively easy to generate the required C code, although it requires a bit of tweaking with expression casts to ensure that the C compiler would not modify the intended meaning of our HRTL instructions. C compilers are able to perform the optimizations needed to not only produce good target machine code but also to clean up and remove inefficiencies in the kind of C code the back end produces for each HRTL instruction. Our experience was that, indeed, the C compiler is a very good macro assembler.

Implementating overlapping registers was a big step in allowing us to progress from small programs (e.g., the UNIX system command-line program `banner`) to medium-sized programs (e.g., the SPEC 95 `compress` and `go` programs). Overlapping registers exist in machines like the x86, where physical overlapping registers have been created over time, such as the `$al`, `$ax` and `$eax`. They also exist on machines such as SPARC and most RISC machines, where double and quad precision floating point operations operate on several single precision floating point registers. The solution was simple: in the RTL and HRTL representation, number the registers differently (e.g., `$ax` gets a different register number than `$eax`), and at code generation time allocate overlapping registers in memory that is overlapping. The solution works well in practice, but was initially hard to implement correctly.

5.2 What Did Not Work as Well as Expected

The PAL descriptions allow users to specify ABI-compliant calling conventions as well as commonly seen patterns or idioms. However, if a pattern is not specified, the analysis code has no way of coping with that pattern, as its operation often relies on non-portable aspects of the architecture (e.g., register windows or delay slots) whose semantics are not modelled directly. Complicated, assembler-like conventions that rely on knowing the location on the stack where particular values are stored, are not normally representable in the PAL language. However, our aim was to support only user-level code, and these more complicated conventions are normally found in systems code: namely, assembly code and library code.

Our implementation support for endianness was incomplete and somewhat inefficient, which results in some programs having too much overhead from the large number of target machine instructions generated for each endianness swap. For example, on the SPARC architecture, we made use of 11 SPARC instructions for the first swap and 8 instructions on subsequent swaps. However, if we had implemented a more efficient swap mechanism using SPARC V9 alternate address spaces, the number of instructions would have been reduced to two or one. Further, our implementation of endianness swaps requires a swap at each load and store. While this is sufficient for most code, more information is required when dealing with library calls where the target, rather than the source, endianness is expected.

While VPO is well suited to serve as the back end for a compiler, it has a few limitations when used in a binary translator. To generate high-quality code, VPO requires accurate liveness information, especially for procedure calls. This is relatively easy for a compiler front end to produce but is generally difficult to generate when doing binary translation. In addition, a binary translator sometimes needs to control what code is produced. For example, it is sometimes necessary to ensure, when translating to the SPARC architecture, that a particular procedure have its own stack frame (i.e., that it is not compiled as a leaf procedure). VPO does not provide this level of control over the final code it emits. As another example, we sometimes found that VPO optimized too well: it would sometimes move an instruction further than we wanted in code where we needed to precisely control what values were in each register.

Speculative decoding of procedures, while it achieved its main goal of being able to run programs with register calls, was disappointing in other respects. Much more code than necessary was often decoded (at times, compilers seemed to copy hundreds of kilobytes of unnecessary code to the input binary), and the “noise level” of invalid instructions and the like was disruptive. Even if this could be muted, there would often still be long delays at translation time during speculation.

Another problem with speculation is the lack of parameter matching that can be done between callers and speculatively decoded callees. This is because there is by definition no caller to match with a speculatively decoded callee. For SPARC architecture programs, this may mean passing too many parameters (all potential parameter locations that are live at the call site), but this is harmless and the program still works correctly. However, for PA-RISC programs, simply passing all live potential parameters does not work. The caller has no way of knowing which of up to three register locations might be used by the callee (depending on the type of the parameter). This is essentially a limitation of static binary translation. Some form of run time feedback may be able to overcome this problem.

In some of our earlier papers on the UQBT framework, we used the term *resourceable* by analogy to the term retargetable. Essentially, a retargetable tool is one that can be targeted to a different target machine. In a similar way, we reasoned that resourceable would mean a tool that is capable of being targeted to a different source machine. In practice, this term was confusing, as people thought it referred to source code rather than source binary program. Hence, we have ended up using the term retargetable for both source and target machines.

5.3 What Was Missing

One of the biggest design decisions that we did not take into account when the framework was originally developed was that of debugging support. This was a big mistake. We relied on standard debuggers to debug the generated target code. This proved very time consuming. In order to track where code came from, we relied on source addresses appearing as comments in each basic block; some of the transformations would reorder basic blocks, making the tracking of addresses hard to do. In general, there was no unified approach for tracking how code had been transformed throughout the framework.

Any tool that is to perform binary manipulation of code ought to have a proper debugging infrastructure to be able to more easily detect where problems with the translation lie. We believe that if we would have had a debugging infrastructure to help us track down code transformations, we would have been able to more easily test larger programs. In the end, it is a matter of time spent upfront on the infrastructure versus (more) time spent afterwards on individual programs.

The fallback interpreter that a full static binary translator needs to rely on at runtime was also missing. Although we had a student write one as a summer project, and have tools to generate an interpreter automatically now, these were never integrated into the final translator. Apart from a lack of time, one reason for this was because we did not have available a mapping of source registers to the translated virtual (in memory) and physical (in target machine register) locations. For the programs that UQBT-generated translators generate (most programs less than about 20 KB of real code, and some up to 350 KB), no interpreter is needed.

Atomicity support for instructions was not part of the semantic specifications or part of the translation, and such support is sometimes needed. Similar support for traps and signals is also not included. For example, in order to support the translation of programs that depend on traps, it must be possible to accurately reconstruct the original program's registers and other *source* machine state. This would require runtime support to map the target machine state back to that of the source machine. It would also require that the UQBT framework and the various back ends not perform any optimizations that would make it impossible to recover that source machine state.

5.4 What Could Be Improved

If we were going to do this all over again, we would improve some things that worked well, as well as to add things that were missing in our framework. Most of these involve details of our implementation rather than design decisions.

The SSL language makes implicit typing information for machine instructions in the form of the tuple (type, size, sign). Most of this information is not made explicit at each location, hence some writers of SSL specifications sometimes omit necessary information. Our SSL typing engine ensures users are warned about these problems. RTLs include the minimum amount of typing information needed to transform code. Basically, typing information is effectively only propagated at the intra-assignment level. This is enough to express the various bit manipulations to the back ends, e.g., to force a signed or unsigned right shift as required, or to cast memory to the appropriate size when reading or writing.

At one time we experimented with a version of UQBT that included copious typing information in the SSL representation. That information proved to be useless once we had a typing engine in place; in fact, all the right assumptions about the SSL types were inferred correctly into the RTL representation. However, type propagation needs to extended in order to support the translation of programs that make use of callback or function pointers passed as arguments to procedures. We started implementing a type propagation algorithm that was interprocedural in nature; however, that experiment was not completed in time to report how much propagation is actually needed. We observe that only integers and pointers are essential types of interest; all other types are explicit in

RTL assignment instructions. We also believe that some further type analyses would allow us to determine more function pointer addresses. Such pointers (to source machine functions) have to be changed to pointers to the equivalent translated functions.

The UQBT internal representation of RTLs are “semantic strings”, which are a linearized string representation of an expression, as described in [Fra94], Chapter 2. Our experience was contrary to theirs. Linearized strings, while convenient in several ways, are just not as suitable as tree-based representations for RTLs. The need to get at a particular sub-expression of a given expression is just too common, and is costly and error-prone when using linearized strings. In a sense, it is worse than using a list where an array is needed, because of the cost of traversing a sub-expression: the cost is generally much higher than following a pointer.

Some of the liveness analyses are prohibitively expensive, but since the translator is meant to be run once, statically, the time taken to run the translation was not an major concern.

Although we had regression testing support in the framework, we could have had more unit testing and testing of individual modules, not just testing of the “tricky” cases for the overall implementation.

5.5 On the Merits of Specification Languages

In supporting machine instruction syntax and semantics, or operating system conventions for binary file formats and calling conventions, there are two ways to capture this information in a retargetable way: by using specification languages or by using an object oriented, abstract class approach (or its equivalent in a non-object oriented language).

In order to create a specification language or an abstract class, several architectures need to be considered to decide what key basic features to specify. Once those features are identified, a language can be described or the data structures in the abstract class can be specified. Generality of the solution is important, as the specification language or abstract class should be complete and need not require changes. This is a hard problem to solve in general as you cannot provide a solution that works for all existing machines and all possible future machines to be developed, but at least the problem can be solved for existing machines of interest.

Both approaches provide advantages and disadvantages. In understanding these approaches, we provide our experiences by answering a series of questions.

What types of overhead does each solution introduce?

In understanding the overhead that we incurred in using the specification language approach, we refer to the size of the specification files themselves (see Figure 9) as well as the time taken to write such specifications (based on §4.2).

The size of the specifications indicates an order of magnitude less code must be written when compared to a hand-crafted approach or one based on abstract classes; the machinery to transform

Machine	SLED	SSL	PAL
SPARC V8	280	636	206
80386	746	1854	171
mc68328	1173	758	149
PA-RISC V1.1	832	588	127

Figure 9: Number of Lines of Specification for the SLED, SSL and PAL Specifications for the SPARC, 80386, mc68328 and PA-RISC Architectures.

the specification file into information is hidden in a separate tool that interacts with the tool to be written (e.g., the binary translator), rather than being embedded into the tool that is being written.

The amount of time taken to write the specification files can be longer than if using an abstract class approach. In our experience, it takes about 1 to 2 months to write and test a SLED specification, 2 to 4 weeks for the SSL specification and less than 1 week for the PAL specification, for a person experienced in the languages. A hand-crafted approach, say to write a disassembler, would take about 2 weeks for writing and basic testing; i.e., up to half the time of writing the equivalent SLED specification. Testing of a hand-crafted disassembler may take considerably more time though. An abstract class approach for writing a disassembler would take about 3 weeks for writing and testing the subclass that describes the syntax of the given machine; i.e., up to three quarters of the time of writing the equivalent SLED specification.

The overhead spent in writing specifications includes the time taken to learn the specification language itself. This overhead is not incurred in hand-crafted or abstract class solutions, as in those cases, the solution is written using the same language that is being used to write the tool at hand (e.g., the binary translator in our case).

Once a solution is available, is it reusable?

Specifications written in the SLED language have been used for multiple purposes, as part of encoders and decoders of machine instructions, such as parts of a compiler, a disassembler or a binary translator. Reusing a specification for these purposes has very little maintenance cost overall, as the specification is complete in describing the syntax of machine instructions, for both encoding and decoding purposes.

SLED was also reused in the PAL language. Basically, PAL implements a regular expression language over SLED, to describe the prologues and epilogues of a caller or callee. PAL support in the UQBT framework is in the form of translation of such PAL regular expressions into NJMCTK matching statements that use SLED names in each arm. The matching statement is then transformed by the NJMCTK into C code for integration into the rest of the UQBT framework. PAL specifications are fairly small for this reason.

In the case of SSL specifications, the language was designed to be able to describe the semantics of user-level machine instructions; i.e., it was not designed to support the specification of a complete machine or systems-level type of instructions. The specification has been useful not only for binary translation purposes; combined with the SLED specification, we have been able to automatically generate interpreters for the SPARC and Pentium architectures [CU02]. These interpreters have been tested against the SPEC benchmark programs and such programs can be interpreted correctly on the automatically-generated interpreters. This shows reuse of the specifications.

Let us contrast the specification approach with the solution required in an abstract class approach. A similar solution is achieved in this case, as the code that generates an interpreter, or some other tool, needs to be written using either approach. In both cases, the information that has been specified or abstracted is reused. Therefore, the maintenance overhead is the same.

Should a solution be designed to be used in one particular tool or one particular domain area?

The specification-based approach tends to lend itself to be used in multiple tools and domain areas. Multiple tools because the specification is not tied into any particular programming language or library; the specification's own tool can generate the required information and be integrated into different tools. Reuse in multiple domain areas tends to be a desire of some but this is a harder goal to reach, even though at first glance the languages look general enough to support other areas.

For example, the SSL language would not be reusable if wanting to write a simulator, as more machine state information would be needed to be described by SSL, as well as be able to describe system-level instructions. This was a limitation imposed in the design of the SSL language; the main interest was to be integrated into the binary translation framework.

The PAL specification was written with decoding purposes in mind, to capture procedural type of information that relates to parameter passing, stack frame setup and calling conventions. The PAL language was designed after contemplating reusing an existing language, CCL (calling conventions language) [BD95], that describes calling conventions and parameter information. CCL was developed in the context of a retargetable optimizing compiler, it allows writers to specify the calling conventions used for different high-level languages, the number of parameters passed and their types.

PAL was designed because CCL was not suitable for decoding purposes; i.e., CCL was suitable only for encoding purposes of procedural type of information. When analyzing binary programs, the types and number of arguments are not explicitly described in the binary program; these need to be recovered through analysis. The shortcomings of the CCL specification for analysis of calling conventions in binary programs is that the specifications assume prior knowledge of number of parameters and their types: exactly what needs to be recovered during procedure abstraction of a binary program.

In general, it is hard to develop specification languages that can be applied to multiple domains, unless those domains are well specified, such as in the case of the SLED language, where descriptions of syntax are unambiguous and complete. In the case of semantic descriptions, this is a harder problem to solve. Even machine descriptions that are commonly used in the form of the GNU compiler suite are not very reusable at all. The authors and others evaluated the reuse possibility of such descriptions for decoding purposes without positive results.

We believe that similar limitations are experienced in the abstract class approach, where extensions in the form of subclasses would not be enough to support a different domain, but rather changes to the core abstract class would be required.

A similar question is raised in terms of whether the solution should be specific to one particular programming language or not; i.e., whether it can be integrated with other tools or only with tools that speak a particular language. In both, the specification and abstract class approach, the code can be integrated with other tools provided an API has been made available. Most specification-based approaches make available such API, whereas, most abstract class-based approaches tend to make APIs available only for the language that the class was written in, making it less accessible to a wider audience. The integration question determines how many people end up using the provided solution—the specification language approach tends to be more widely used due to its higher integration capabilities with multiple languages.

6 Conclusions

The University of Queensland Binary Translation (UQBT) framework's design met the goals for a translation framework that was based on the use of specifications to support retargetability. It separates machine-dependent from machine-independent concerns, and allows users to instantiate different translators for experimentation with binary translation ideas and techniques. We believe the design of this framework worked well and is capable of supporting a variety of different machines.

Our experience in developing and using the UQBT framework points to the usefulness of the approach: simple experiments are now possible in less time since there is no need to hand-craft a solution from scratch. We have used the system with machines and in contexts we had never thought of supporting initially. This includes the (ms68328,PalmOS) translator, which is a small memory PDA device, and the (PA-RISC,HP-UX) translator, which had to deal with cross-OS issues.

Finally, the UQBT framework can be used for experimenting with a wider range of binary manipulation tasks than binary translation. It can be used for retargetable decompilation, binary instrumentation, binary rewriting, reoptimization, binary code obfuscation, and more.

Acknowledgments

Our thanks to all the students who were part of this project, in order of year: Shane Sendall, David Ung, Ian Walters, Shawn Coyne, Trent Waddington, Doug Simon, Simon Long, Sameer Gulrajani, Pavan Kumar, Manel Fernández, Nathan Keynes and Bernard Wong. Special thanks go to Shane, Doug, Nathan and Bernard for all the fun we had while spending our Australian summers working on this project. We thank Trent for his idea of using the C language and the C compiler as our first back end.

This work was possible thanks to funding from the Australian Research Council (ARC Grant No.A49702762, 1997-99), Sun Microsystems Laboratories (sponsored by Neil Wilhelm and Mario Wolczko, 1996-2001), and The University of Queensland (Grant No.97/UQNSRG009G, 1997).

The UQBT distribution is available online under a BSD open source license. Please refer to the UQBT home site for more information about this project and for links to the distribution:
`http://www.itee.uq.edu.au/csm/uqbt.html`

References

[BD88] 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.

[BD95] 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.

[CE01] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40:171–188, March 2001.

[CERL01] C. Cifuentes, M. Van Emmerik, N. Ramsey, and B. Lewis. The University of Queensland Binary Translator (UQBT) framework, December 2001. Documentation from the UQBT open source distribution, available from `http://www.itee.uq.edu.au/csm/uqbt.html`.

[CEU+99] C. Cifuentes, M. Van Emmerik, D. Ung, D. Simon, and T. Waddington. Preliminary experiences with the use of the UQBT binary translation framework. In *Proceedings of the Workshop on Binary Translation, Oct 16, 1999*, Technical Committee on Computer Architecture News, pages 12–22, Newport Beach, USA, December 1999. IEEE CS Press.

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

[CR02] C. Cifuentes and N. Ramsey. A transformational approach to binary translation of delayed branches. Technical Report TR-2002-104, Sun Microsytems Laboratories, Palo Alto, CA 94303, January 2002.

[CS98] 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.

[CS00] C. Cifuentes and D. Simon. Procedure abstraction recovery from binary code. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 55–64, Zurich, Switzerland, March 2000. IEEE CS Press.

[CU02] C. Cifuentes and D Ung. Walkabout - a retargetable dynamic binary translation framework. Technical Report TR-2002-106, Sun Microsytems Laboratories, Palo Alto, CA 94303, February 2002.

[Dig95] Alpha migration tools. Freeport Express. `http://www.support.compaq.com/amt/freeport/index.html`, 1995.

[Fra94] M. Franz. *Code-Generation On-The-Fly: A Key to Portable Software*. PhD dissertation, Swiss Federal Institute of Technology Zurich, 1994. ETH No. 10497.

[HH97] R.J. Hookway and M.A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[HM79] R.N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1979.

[Mey97] J. Meyer. Jasmin. `http://mrl.nyu.edu/˜meyer/jasmin/`, 1997.

[Pro95] T.A. Proebsting. Optimizing and ANSI C interpreter with superoperators. In *Proceedings of the 22nd Symposium on Principles of Programming Languages*, pages 322–332. ACM Press, January 1995.

[RF97] N. Ramsey and M. Fernández. Specifying representations of machine instructions. *ACM Transactions of Programming Languages and Systems*, 19(3):492–524, 1997.

[SCK$^+$93] 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.

[Sta93] R.M. Stallman. *Using and Porting GNU CC*, version 2.5 edition, October 1993.

[TC02] J. Tröger and C. Cifuentes. Analysis of virtual method invocation for binary translation. Technical Report FIT-TR-2002-01, Queensland University of Technology, Brisbane, Australia, 2002.

[Tho96] T. Thompson. An Alpha in PC clothing. *Byte*, pages 195–196, February 1996.

[Tra97] TracePoint. HiProf hierarchical profiler. `http://www.tracepoint.com/noframes/hiprof/products/hiprof`, 1997.

[VPO98] Machine-independent optimization. `http://www.cs.virginia.edu/zephyr/vpo`, 1998.

A An Example

We illustrate the transformation process by providing code for the translation of a recursive fibonacci routine, originally compiled for the Pentium architecture, translated into the SPARC architecture. We show the main intermediate representations of the code in the ASCII representation provided by the UQBT framework. Throughout this presentation, bolded text relates to code that represents the invocation of the `fib(x-1)` routine of the original C language program.

The original C language fibonacci routine looks like this

```
int fib (int x)
{
   if (x > 1)
      return (fib (x-1) + fib (x-2));
   else
      return (x);
}
```

The code was compiled on a (Pentium,Solaris) machine using the GNU `gcc` compiler. The disassembly of the `fib` procedure shows the following code

```
08048798:       55                  push   %ebp
08048799:       89 e5               mov    %esp,%ebp
0804879b:       53                  push   %ebx
0804879c:       83 7d 08 01         cmpl   $0x1,0x8(%ebp)
080487a0:       7e 2e               jle    80487d0 <fib+0x38>
080487a2:       8b 45 08            mov    0x8(%ebp),%eax
080487a5:       48                  dec    %eax
080487a6:       50                  push   %eax
080487a7:       e8 ec ff ff ff      call   8048798 <fib>
080487ac:       83 c4 04            add    $0x4,%esp
080487af:       89 c3               mov    %eax,%ebx
080487b1:       8b 45 08            mov    0x8(%ebp),%eax
080487b4:       83 c0 fe            add    $0xfffffffe,%eax
080487b7:       50                  push   %eax
080487b8:       e8 db ff ff ff      call   8048798 <fib>
080487bd:       83 c4 04            add    $0x4,%esp
080487c0:       89 c0               mov    %eax,%eax
080487c2:       8d 14 18            lea    (%eax,%ebx,1),%edx
080487c5:       89 d0               mov    %edx,%eax
080487c7:       eb 0f               jmp    80487d8 <fib+0x40>
080487c9:       8d 76 00            lea    0x0(%esi),%esi
080487cc:       eb 0a               jmp    80487d8 <fib+0x40>
080487ce:       8d 36               lea    (%esi),%esi
080487d0:       8b 45 08            mov    0x8(%ebp),%eax
080487d3:       eb 03               jmp    80487d8 <fib+0x40>
```

```
080487d5:       8d 76 00             lea    0x0(%esi),%esi
080487d8:       8b 5d fc             mov    0xfffffffc(%ebp),%ebx
080487db:       c9                   leave
080487dc:       c3                   ret
080487dd:       8d 76 00             lea    0x0(%esi),%esi
```

After transformation of the assembly code to Pentium-RTL, the code is voluminous due to the need to make explicit every assignment to a condition code; which are commonly used in the Pentium instruction set. We do not show here all the generated Pentium-RTL code but given an idea of the amount of code generated per one assembly instruction

```
08048798 *32* r[28] := r[28] - 4
         *32* m[r[28]] := r[29]
08048799 *32* r[29] := r[28]
0804879b *32* r[28] := r[28]
...
...
...
080487a0 *32* %pc := (((%NF ^ %OF) | %ZF) = 1) ? 80487d0 : %pc
080487a2 *32* r[24] := m[r[29] + 8]
080487a5 *32* r[tmp1] := r[24]
         *32* r[24] := r[24] - 1
          *1* %CF := (~(r[tmp1]@[31:31]) and (1@[31:31]))
                 or ((r[24]@[31:31]) and (~(r[tmp1]@[31:31]) or (1@[31:31]))
          *1* %OF := ((r[tmp1]@[31:31]) and ~(1@[31:31]) and ~(r[24]@[31:31]))
                 or (~(r[tmp1]@[31:31]) and (1@[31:31]) and (r[24]@[31:31]))
          *1* %NF := r[24]@[31:31]
          *1* %ZF := r[24] = 0 ? 1 : 0
080487a6 *32* r[28] := r[28] - 4
         *32* m[r[28]] := r[24]
080487a7 *32* r[28] := r[28] - 4
         *32* m[r[28]] := %pc
         *32* %pc := fib
080487ac *32* r[28] := r[28] + 4
080487af *32* r[27] := r[24]
080487b1 *32* r[24] := m[r[29] + 8]
...
...
...
080487d8 *32* r[27] := m[r[29] - 4]
080487db *32* r[28] := r[29]
         *32* r[29] := m[r[28]]
         *32* r[28] := r[28] + 4
080487dc *32* %pc := m[r[28]]
         *32* r[28] := r[28] + 4
```

The transformed HRTL representation is much smaller than the RTL one. Left-hand side addresses are made available for comparison purposes

```
High level RTLs for procedure fib(v0)
Twoway BB (0x3f96a0):
0804879c *32* v4 := 1
         *32* v3 := v0
080487a0  JCOND (v3 <= v4) 80487d0

Call BB (0x3ee9f0):
080487a2 *32* v1 := v0
080487a5 *32* r[tmp1] := v1
         *32* v1 := v1 - 1
080487a6 *32* v2 := v1
080487a7  v1 := CALL fib(v2)

Call BB (0x3ed5f0):
080487af *32* r[27] := v1
080487b1 *32* v1 := v0
080487b4 *32* r[tmp1] := v1
         *32* v1 := v1 - 2
080487b7 *32* v2 := v1
080487b8  v1 := CALL fib(v2)

Oneway BB (0x3ec5f8):
080487c0 *32* v1 := v1
080487c2 *32* r[26] := v1 + (r[27] * 1)
080487c5 *32* v1 := r[26]
080487c7  JUMP 80487d8

L1: Oneway BB (0x3f7ab0):
080487d0 *32* v1 := v0
080487d3  JUMP 80487d8

L2: Ret BB (0x3ea400):
080487d8  RET     Return location is v1 (type int32)
```

We used the C back end to generate the following code

```
#include "uqbt.h"
int32 fib(int32 v0) {
char _locals[4];
        int32 v1;  int32 v2;
        int32 v3;  int32 v4;
        int32 r26, r27;  int32 tmp1;
        /* 8048798 */
        v4=1;
        v3=v0;
        tmp1=(v0)-(1);
        if ((v3)<=(v4)) goto L1;
        v1=v0;
        tmp1=v1;
        v1=(v1)-(1);
        v2=v1;
        v1=fib(v2);
        r27=v1;
        v1=v0;
        tmp1=v1;
        v1=(v1)-(2);
        v2=v1;
        v1=fib(v2);
        v1=v1;
        r26=(v1)+(r27);
        v1=r26;
        goto L2;
L1:             /* 80487d0 */
        v1=v0;
        goto L2;
L2:             /* 80487d8 */
        return v1;
}
```

For interest, we show the generated map file for the translated program

```
dummy = V0;
dummy : .interp;
s.uqbt.data = LOAD V0x80488b4;
s.uqbt.data: .uqbt.data;
```

After compilation of the generated code on a (SPARC,Solaris) machine, and the subsequent disassembly of such code, we obtain the following code

```
 8051928:  9d e3 bf 88      save        %sp, -120, %sp
 805192c:  80 a6 20 01      cmp         %i0, 1
 8051930:  04 80 00 08      ble         0x8051950
 8051934:  01 00 00 00      nop
 8051938:  7f ff ff fc      call        fib
 805193c:  90 06 3f ff      add         %i0, -1, %o0
 8051940:  a0 10 00 08      mov         %o0, %l0
 8051944:  7f ff ff f9      call        fib
 8051948:  90 06 3f fe      add         %i0, -2, %o0
 805194c:  b0 02 00 10      add          %o0, %l0, %i0
 8051950:  81 c7 e0 08      ret
 8051954:  81 e8 00 00      restore
```

This code compares favourably not only to that generated by a native SPARC compiler, but also performs well, as no emulation of Pentium features are done in the code. The main differences between the two machines are: the Pentium is a CISC machine whereas the SPARC architecture is a RISC machine, the Pentium passes parameters on the stack whereas the SPARC architecture makes use of register windows, the Pentium is little endian whereas the SPARC architecture is big endian, and the Pentium makes extensive use of condition code side effects whereas the SPARC architecture does not. Note that since this program has no initialized data, endianness swaps have been disabled.

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.

Mike Van Emmerik was a Senior Research Assistant at the School of Information Technology and Electrical Engineering at The University of Queensland during the UQBT research. He has ten years experience in engineering, and ten years as a software developer and researcher. His interests are in the area of low-level code manipulation, including disassembly, decompilation, and binary translation. Mike has a B.Sc. in Computer Science and a B.E. in Electrical Engineering, both from The University of Queensland, where he currently teaches.

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 Ph.D. 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."

Brian Lewis is a Senior Staff Software Engineer in the Programming Systems Lab of Intel's Microprocessor Research Labs. His interests include binary translation as well as virtual machine and programming language implementation. Previously at Sun, he implemented automatic checkpointing in a high-performance Java virtual machine. He implemented the Tcl 8.0 and the Clarity C++ virtual machines. He also developed monitoring and debugging tools for the Spring distributed operating system. Prior to joining Sun, Brian worked at Olivetti and Acorn Research, where he implemented a user interface toolkit and runtime software. At Xerox, he led the team that developed the Shared Books distributed, multi-user publication management system. He implemented portions of the Mesa programming system and developed software version management tools. He received the Ph.D. in Computer Science from the University of Washington.