UQBT: Adaptable Binary Translation at Low Cost

A reusable, component-based binary-translation framework lets engineers quickly and inexpensively migrate existing software from one processor to another. This framework supports various processors, including CISC, RISC, and stack-based machines.

Cristina Cifuentes
University of Queensland
Sun Microsystems Inc.

Mike Van Emmerik
University of Queensland

Developments in the semiconductor industry have made possible smaller and faster processors for general-purpose computing, portable devices, multipurpose appliances, and system-on-chip designs. Binary translation offers a quick, inexpensive way to migrate software from one processor to another.

Although binary-translation techniques are still in their infancy compared to their compiler counterparts, engineers have been using them for 15 years. Just as engineers build compilers partly on the basis of specifications, we are developing the University of Queensland Binary Translator (UQBT) on the basis of machine specifications and properties of machines and operating systems. This static binary-translation framework supports various processors, including complex-instruction-set computers (CISC), reduced-instruction-set computers (RISC), and stack-based machines.

In this article, we describe the UQBT framework and discuss our observations while using it to instantiate six different translators across Sun Sparc, Intel Pentium, and Java virtual-machine architectures.

DESIGN GOALS

Hardware manufacturers have driven binary-translation developments. Until recently, these manufacturers focused on merely migrating their customer base from their soon-to-be legacy hardware architecture to their new architecture. Now hardware manufacturers are also using these techniques to migrate from a competitor's hardware platform to their own machines. Examples of well-known translators include

- Digital's Vest and mx for migrating VMS VAX and Ultrix MIPS (million instructions per second) code to the Alpha running under Digital Unix,[1]
- Tandem's Accelerator for migrating Guardian 90 TNS (Tandem Nonstop Systems) CISC binaries to the TNS/R (a RISC machine),[2] and
- Digital's FX!32 for translating Windows NT Intel x86 binaries to run on Alphas running Windows NT.[3]

Many of these technologies remain proprietary and unpublished. Moreover, each new binary translator has been handcrafted from scratch; therefore, reuse is difficult, because binary translators are highly machine dependent.

In contrast, UQBT is constructed from well-specified components. These components are either reusable across different machines or partially generated from specialized specifications, which dramatically reduces the cost of handling machine idiosyncrasies. One of our goals in developing UQBT was to design a framework for experimentation in static binary translation. We designed UQBT to adapt easily and inexpensively to changes in both source and target machines, including translations to register- and stack-based machines. When compilers and other tools support multiple target machines at a low cost, researchers and developers call them *retargetable*. By extension, we call a binary-analysis tool *re-sourceable* if it can analyze binaries from multiple source machines at a low cost. Our work makes UQBT both retargetable and re-sourceable.

UQBT separates machine-dependent and machine-independent concerns. Description (or specification)

0018-9162/00/$10.00 © 2000 IEEE

languages support machine-dependent components. Machine-independent components are reusable. In this way, a binary-translation writer can concentrate on describing machine properties and experimenting with different algorithms at a machine-independent level.

To focus on the core problems of translation, we limited the UQBT project's scope in two ways. First, UQBT's design addresses differences in instruction sets, endian (byte ordering of the data) convention, calling conventions, and binary-file formats, but it does not address operating-system differences. Our design is currently restricted to multiplatform operating systems such as Solaris and Linux. Second, UQBT handles user code (application programs) but not kernel code or dynamically linked libraries.

SUPPORTING MULTIPLE MACHINES INEXPENSIVELY

Like compilation, binary translation is conceptually a series of phases. A front end decodes the source binary program, an analyzer transforms the program's representation, and a back end optimizes and encodes the output (see Figure 1).

UQBT uses two intermediate representations to transform the source program into a high-level representation similar to that used by compilers (at this level, binary-translation writers can perform machine-independent analyses and transformations):

- Register transfer lists (RTLs) describe the machine instructions' effects on hardware registers. An M-RTL is an RTL with features unique to a given machine, M.
- Higher-level register transfer language (HRTL) is a machine-independent representation that

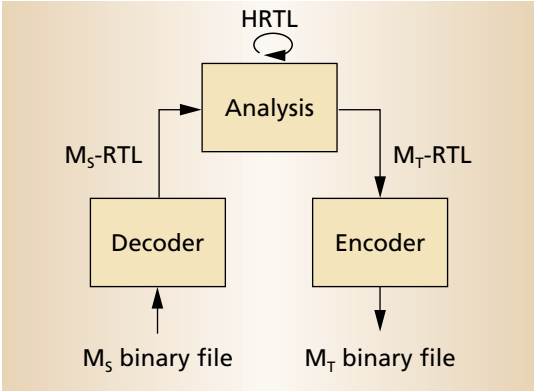

Figure 1. UQBT's phases. A binary translator decodes source binary programs into M_s-RTLs (register transfer lists for a source machine) and transforms them into HRTL (higher-level register transfer language) form. After analysis, the translator encodes the programs into M_t-RTLs (RTLs for a target machine) and changes them into machine code for the target machine.

Specifications Summary

A re-sourceable and retargetable binary translation framework must help describe machine instruction sets and their properties, as well as the operating system's conventions set.

Machine specifications

A machine instruction set's characteristics include its syntax (which bits match which assembly instructions), its semantics (what a particular assembly instruction means), and its control-transfer instructions (which instructions change a program's control flow and how). Machines that exhibit delayed control transfers can specify this fact as well. For historical reasons, we have specified these characteristics in various languages, but they can be specified in one. UQBT uses the following implementation languages:

- Specification Language for Encoding and Decoding (SLED) describes the machine instructions' syntax.[1]
- Semantic Specification Language (SSL) describes the machine instructions' semantics in terms of register transfer levels (RTLs).[2] Optimizers such as gcc and vpo use machine descriptions that specify semantics of a subset of the instruction set.
- Control-Transfer Language (CTL) enumerates the instructions that perform basic control transfers (conditional jumps, jumps, calls, or returns).
- Delayed Control-Transfer Language (DCTL) describes simple transformations needed to remove dependencies on delayed instructions (for machines that have them).

OS specifications

An operating system's conventions and formats come in the form of calling conventions, including where parameters are passed (for example, a stack or registers), descriptions of a procedure's stack frame and where local variables are, and the format of the binary file that the operating system expects. Two implementation languages represent this information:

- Procedure Abstraction Language (PAL) describes calling conventions, parameter-passing conventions, stack frames, and local variable locations for procedures.[3]
- Binary File Format (BFF) describes the internal format of a binary file, such as executable and linking format (ELF) for Solaris and Linux systems.[4]

References

1. N. Ramsey and M. Fernández, "Specifying Representations of Machine Instructions," *ACM Trans. Programming Languages and Systems*, May 1997, pp. 492-524.
2. C. Cifuentes and S. Sendall, "Specifying the Semantics of Machine Instructions," *Proc. Int'l Workshop Program Comprehension*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 126-133.
3. C. Cifuentes and D. Simon, *Procedural Abstraction Recovery from Binary Code*, Tech. Report No. 448, Dept. of Computer Science, Univ. of Queensland, Brisbane, Australia, 1999.
4. D. Ung and C. Cifuentes, "SRL—A Simple Retargetable Loader," *Proc. Australian Software Eng. Conf.*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 60-69.

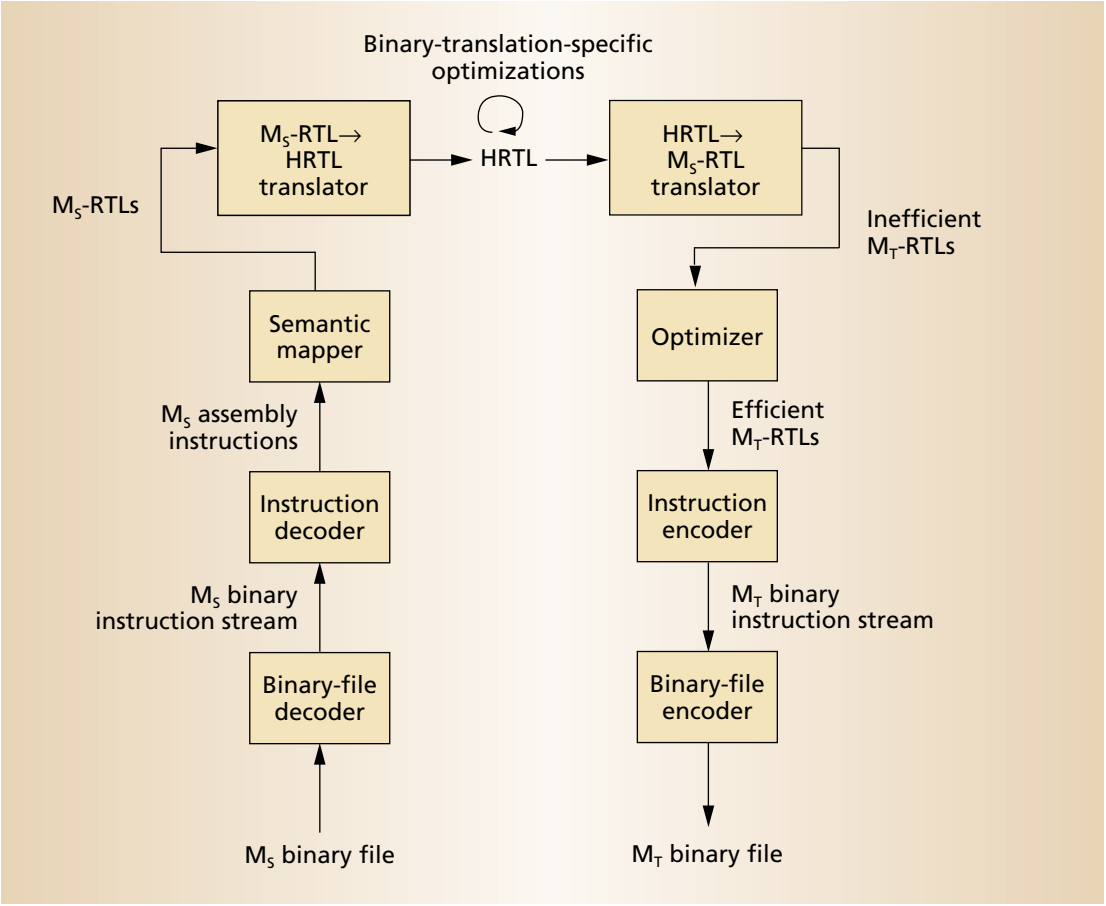

Figure 2. A re-sourceable and retargetable binary translation framework.

supports elementary control transfers (conditional and unconditional jumps, calls, and returns) and an infinite number of registers.

As Figure 1 shows, decoding and encoding work with a particular machine's RTLs, and the analysis phase elevates the level of representation from M_S-RTLs (RTLs for a source machine), to HRTL, and then down to M_T-RTLs (RTLs for a target machine). The analysis phase removes hardware dependencies from the source machine to generate good quality code for the target machine. For example, rather than imposing a stack parameter system on a machine that prefers passing parameters in registers, we transform the code from stack- to register-based, thus using the target machine's native conventions.

We support multiple source and target machines by using specifications that describe the operating system's conventions and the specifications that describe a machine instruction set's properties.

AN OVERVIEW OF UQBT

We provide insight into some of the specifications and transformations used by UQBT. Because of space limitations, we present examples only in terms of the Sparc architecture.

Decoding binary code to M_S-RTLs

UQBT translates binary code to M_S-RTLs in three steps (see Figure 2). The first step is to decode the source binary file into its components and feed the binary instruction stream to the instruction decoder. We can build the binary-file decoder from a specification describing the contents of the binary-file format, such as the executable and linking format (ELF) and the portable executable (PE) format.

The instruction decoder identifies each instruction and its operands. The UQBT framework uses the New Jersey Machine-Code toolkit to automatically generate part of this decoder from a syntactic Specification Language for Encoding and Decoding (SLED) specification that describes each instruction's binary representation.[4] The semantic mapper then maps each instruction to its semantic representation in the form of an M_S-RTL. A Semantic Specification Language (SSL) specification associating an M_S-RTL with each instruction drives the semantic mapper.[5]

Figure 3 shows tiny excerpts from SLED, SSL, and Control-Transfer Language (CTL) specifications for the Sparc processor. (We discuss CTL in the "Translating Ms-RTLs up to HRTL" section.) In SLED, fields of instruction, patterns, and constructors introduce a machine instruction's fields, the binary representations of instruction opcodes in terms of fields, and a full instruction's construction from opcodes and operands. The example in Figure 3 shows a constructor for an immediate call instruction. In SSL, semantics describes the register transfers associated with an instruction. The fragment of SSL shows the immediate call instruction's semantics. The * 32* notation indicates the number of

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

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

  constructors
    call__ disp30                      is CALL & disp30

# Semantic specification for immediate call instruction
  semantics
    call__ disp30                      *32* r[15] := %pc
                                       *32* %pc := %npc
                                       *32* %npc := r[15] + (4 * disp30);
# Transfer of control specification for call instructions
  call mapping
    call__ disp30                      & address = %npc.
```

Figure 3. Sample syntactic, semantic, and control-transfer specifications for a Sparc-architecture immediate call instruction.

bits assigned in each effect. We have explicitly named registers %pc and %npc (program counter and next program counter) for easier reading.

Translating M_s-RTLs up to HRTL

A series of transformations and analyses rewrite the machine-dependent M_s-RTL representation into HRTL. Most of the analyses are reusable for different machines—for example, identifying the targets of indirect control transfers,[6] transforming basic control transfers into HRTL instructions, and determining a procedure call's parameters and return value.[7]

Occasionally, custom analyses are necessary to remove a particular machine's peculiarities. An example would be transforming Pentium stack-based floating-point instructions (such as FADDP, which adds two floating-point registers and pops the floating-point register stack) to a conventional "flat" register model.

Control transfers. UQBT easily handles basic control transfers for each machine, such as conditional or unconditional jumps, calls, and returns. These instruction types have a higher-level semantic meaning than that achieved through raw register transfers. They modify the program counter and, thus, have a high-level meaning, which a flow graph can best represent. We use CTL for this purpose.

CTL describes which machine instructions map to HRTL's high-level control-transfer instructions. The call mapping identifies only the call instruction itself; UQBT later infers parameters to that call by analyzing the M_s-RTLs and the source machine's calling conventions The call mapping in Figure 3 shows a CTL specification for a Sparc immediate call instruction. This CTL specification identifies the target address as %npc, which after analysis will be the same as the program counter (at the start of the call) plus 4 • disp30, where disp30 is a 30-bit displacement value that is obtained after the SLED processing.

Recovering parameters and return values. UQBT uses reachability and liveness analysis to recover parameter and return-value information on the basis of the calling conventions and locations where parameters and return values can be passed. A Procedure Abstraction Language (PAL) description specifies such information and also describes a procedure's stack frame and the valid locations on the stack for local variables.7 We specify the calling conventions that the operating system's application binary interface (ABI) sets. The ABI describes the calling conventions allowed for a particular operating system, the distribution of the stack frame, the place where local variables can be allocated, and so on.

UQBT uses the concepts of prologues and epilogues (short idiomatic instruction sequences) for both callers and callees. Callee prologues and epilogues are at the beginning and end of a function. A caller prologue is the call itself, and the caller epilogue represents any cleaning up after a call.

Figure 4 illustrates sample caller and callee pro-

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

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

INCOMING PARAMETERS
    new_reg_win
    {
        AGGREGATE  -> [%afp – simm13 + 64]
        REGISTERS  -> %i0 %i1 %i2 %i3 %i4 %i5
        STACK      -> BASE = [%afp - simm13 + 92]
                      OFFSET = 4
    }
```

Figure 4. Procedure Abstraction Language (PAL) specification for common Sparc-architecture call, along with return instructions and parameter locations.

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

Figure 5. UQBT uses PAL to perform parameter analysis of (a) a sequence of Sparc RTLs, to produce (b) HRTL code. From the HRTL, UQBT can generate code for other processors.

logues and epilogues for the most common Sparc architecture call under a System V Unix system. PAL describes parameter locations from a caller's (outgoing) and a callee's (incoming) viewpoints, as well as the locations that can hold a return value and local variables. PAL uses an abstract-frame-pointer abstraction %afp to reference a fixed point in the stack frame, letting us remove explicit references to the stack and frame pointers.[7] Figure 4 shows a caller's outgoing parameter locations in terms of registers and offsets from %afp. The figure also shows the new_reg_win callee's incoming parameters in terms of the associated mapping of registers and memory locations.

UQBT uses PAL to identify parameters and return locations in the source program's HRTL representation A liveness analysis finds actual parameters and return values. For example, the left-hand side of Figure 5 shows the Sparc RTLs for a call to the procedure gcd, and the right-hand side shows the equivalent HRTL after analysis on this call's parameters. From this HRTL, UQBT can generate Pentium code that uses the native Pentium calling convention. This convention passes parameters on the stack, not in registers (as per Sparc).

Type analysis. UQBT's low-level type analysis distinguishes four types of values: integers, floating-point values, pointers to data, and pointers to code, along with their sizes (in bytes) and their sign. These four basic types, which our analysis recovers, are sufficient for identifying procedure parameters and result values. To translate calls to library functions, we summarize their low-level type signatures in a text file. Some architectures use helper routines for arithmetic that can be performed more efficiently with hand-optimized routines than with the machine's standard instructions. For example, Sparc V8 compilers call rem to perform an integer-remainder operation. These helper functions are nonstandard; they do not necessarily exist on other platforms. Moreover, they are not part of library header files. We replace calls to helper routines with their equivalent semantics in RTLs.

Manipulating HRTL. Lifting source-machine code to HRTL is advantageous because the code becomes machine independent, which allows decoupling of the translator's back end from the source machine. HRTL's high-level call construct lets the back end use the target's native calling conventions. Furthermore, it lets binary-translation-specific optimizations be plugged into the UQBT framework—for example, minimizing the amount of byte swapping at loads and stores when translating to a machine with a different endian type.

Translating HRTL down to binary code

This last phase is equivalent to that of an optimizing compiler, so we are using well-understood techniques. We have two options for the encoding phase: a standalone optimizer with its own interface language, or an optimizing C compiler with C as the interface language. As an example of the former, we have successfully interfaced to the very portable optimizer[8]—part of the Zephyr project[9]—whose RTL interface is similar to that of our RTLs and, hence, is easy to translate to. Generating low-level C code lets us experiment with different optimizers. For each translation, UQBT places the generated code and data into various C and assembly files, which can be compiled with any C compiler and assembler for the target machine. Each function is represented in a low-level C file. For each data section, UQBT generates an assembly file. A makefile (a file used for compilation of a program under many C compilers) packs the files into the appropriate binary format for the target machine.

Interpretation hooks. Static translations may need an interpreter or emulator to handle untranslated code discovered at runtime. The interpreter uses the M_S-to-M_T address mapping to determine when it can return to translated code; therefore, this mapping is stored in the target binary. Because the interpreter will use the original source text section, UQBT also copies this section to the target binary. The interpreter links dynamically and interprets M_S-RTLs.

Target binaries. Static analysis cannot solve all potential aliasing problems. Therefore, UQBT copies the data sections to the target binary as they are. A link-map file generated by the translator makes these data sections retain the same virtual memory address in the target binary as in the source binary. The target program calculates addresses exactly as the original source program did. Thus, the generated program correctly references the data (although the data may need to be referenced using a different endian convention).

EXPERIENCE WITH THE UQBT FRAMEWORK

UQBT is a work in progress. We have tested the current implementation with micro-benchmarks such as fibonacci, sieve, and banner. We are testing its scalability with the SPEC 95 benchmark suite; however, we need a full implementation of the system and the interpreter before we can complete the testing.

Translation quality

We have instantiated four translators from the UQBT framework to translate across Sparc and Pentium architecture binaries: uqbtss, uqbtsp, uqbtps, and uqbtpp. We use an s suffix for Sparc architecture binaries, and a p suffix for Pentium architecture binaries. For a given pair ab, a is the source architecture, and b is the target architecture. Translators to the same architecture, uqbtss and uqbtpp, are useful for determining the translation process's overhead, not in terms of time, but in terms of the generated code's quality compared with the original binary code. Our experiments with micro-benchmarks show that these translations produce binaries that perform at similar speeds to the original binary code.

Translators across the Sparc and Pentium architectures require programs with static data to handle endian conversion, because these two machines support big- and little-endian conventions, respectively. Our experiments with micro-benchmarks show that for programs lacking significant static data handling, the generated target programs perform favorably, compared with native code generated on that target machine.[10] Furthermore, programs that interact extensively with static data demonstrate slowdowns from 1.2 to 2 times that of native code on the target machine This is due to the number of target instructions necessary for byte swapping of a 4-byte word—one instruction for the Pentium compared with eight to 10 instructions for the Sparc V8 architecture. (Micro-benchmark results are available on the UQBT home page at http://www.csee.uq.edu.au/csm/uqbt. html.)

Effort

The UQBT framework gives binary-translation writers support for new architectures at a low cost. The effort to support a new architecture is minimal because it reuses most of the UQBT framework.

Framework development. Developing the UQBT framework has been an order of magnitude more time-consuming (in person-years) than writing a handcrafted binary translator from scratch. This complexity stemmed from the need to make the framework re-sourceable, as well as the need to separate machine-dependent concerns from machine-independent analyses. A small team (averaging 1.5 persons per year, some being relatively inexperienced students initially) developed UQBT over a period of four years.

Using specifications, we created new binary translators quickly by instantiating from the UQBT framework, specifying the source and target machine, reusing most system components, and providing a driver skeleton for the decoding phase. In general, supporting peculiarities of particular machines might require extensions to the system, either in the form of extra analyses (specific to one machine) or extra constructs in the semantic language.

We expect adding DCTL (Delayed Control-Transfer Language) support to be a 0.2 person-year effort. DCTL is a simple transformational language that is easy to describe and use. It can replace about 1,000 lines of transformational code (which remove delayed branch instructions). DCTL can replace all this code by supporting a simple language to specify the transformations rather than having to write the code for all the possible cases. The present code (1,000 lines) was written for Sparc delayed branches. The MIPS also have delayed branches, but the code will not support them as they are. With DCTL, both the present code and the MIPS could easily be specified and supported by the UQBT framework. Because few modern architectures use delayed control transfers, we have delayed implementing this language until now.

The UQBT framework gives binary-translation writers support for new architectures at a low cost.

Reusing the framework. Developers can use the specifications' size to estimate how much code they'll need to write to support a given pair of machines. Current machine-instruction descriptions (syntax, semantics, and control flow) for Sparc and Pentium architectures are between 1,100 and 2,600 lines. Calling-convention and stack-frame specifications are around 200 lines each. This is in contrast to the code they would need to write a complete translator: 26,500 lines of source code, 6,200 lines of definitions in header files, 3,900 lines of specification files, and about 2,000 lines for the framework-generated skeleton driver. This highlights the reduced amount of code that a binary-translation writer would have to develop. Furthermore, reusing specifications is also possible, particularly when other people have already written such specifications.

Translation to Java bytecodes exhibits framework adaptability

One of our students, Trent Waddington, tested the UQBT framework's adaptability by supporting translations to *bytecodes* (the Java virtual machine's assembly language). We instantiated two new translators, uqbtsb and uqbtpb, to translate from Sparc and Pentium architecture binaries to bytecodes. We reused the specifications for the source machines, and Trent wrote a back end for gcc (the GNU C compiler) to generate bytecodes in assembly files linkable by jasmin into Java class files. To run bytecode binaries, Trent wrote an extra support environment that supports memory access and pointers to memory and that translates some library calls.

The results of this experiment were good. The performance of the translated programs on non-memory-intensive micro-benchmarks was comparable to that of native C code compiled on the machine where

the Java virtual machine (JVM) was running. Memory-intensive programs showed performance degradation because of our simple memory-management support and the JVM's lack of unsigned integral types.

The overall effort of this experiment was 0.5 person-years. This included the time to develop the gcc JVM back end and the JVM runtime support environment, and the time to test the micro-benchmarks.

The UQBT project demonstrates the feasibility of creating adaptable binary-translation environments by using specifications to support various machines at a low cost. The work of writing the specifications is an order of magnitude smaller (in terms of time and the amount of code) than writing source code for the machine-dependent components, and developers can reuse the machine-independent analyses provided by the framework. Although work is still in progress, results thus far show that the quality of the generated code is close to that produced by hand-written translators. Moreover, engineers do not need to write assembly code to achieve performance or rewrite from scratch a new translator to support a different pair of machines.

Our hope is that a deeper understanding of description requirements will eventually let tool writers share and reuse existing descriptions. ✸

Acknowledgments

We thank David Ung, a member of the Binary group who is working on the dynamic extensions to the framework. We also thank former members Doug Simon (who worked on PAL), Shane Sendall (who worked on SSL), and Trent Waddington (who wrote the gcc Java virtual machine back end). Norman Ramsey provided his NJMC toolkit, answered questions about SLED and ŁRTL languages, and helped in the initial UQBT design. Our research is supported by Sun Microsystems Inc. and the Australian Research Council under Grant A49702762.

References

1. R.L. Sites et al., "Binary Translation," *Comm. ACM*, Feb. 1993, pp. 69-81.
2. K. Andrews and D. Sand, "Migrating a CISC Computer Family onto RISC via Object Code Translation," *Proc. ASPLOS V , Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1992, pp. 213-222.
3. R.J. Hookway and M.A. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation," *Digital Technical J.*, Vol. 9, No. 1, 1997, pp. 3-12.
4. N. Ramsey and M. Fernández, "Specifying Representations of Machine Instructions," *ACM Trans. Programming Languages and Systems*, May 1997, pp. 492-524.
5. C. Cifuentes and S. Sendall, "Specifying the Semantics of Machine Instructions," *Proc. Int'l Workshop Program Comprehension*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 126-133.
6. C. Cifuentes and M. Van Emmerik, "Recovery of Jump Table Case Statements from Binary Code," *Proc. Int'l Workshop Program Comprehension*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 192-199.
7. C. Cifuentes and D. Simon, *Procedural Abstraction Recovery from Binary Code*, Tech. Report No. 448, Dept. of Computer Science, Univ. of Queensland, Brisbane, Australia, 1999.
8. M.E. Benitez and J.W. Davidson, "A Portable Global Optimizer and Linker," *Proc. Conf. Programming Languages, Design and Implementation*, ACM Press, New York, 1988, pp. 329-338.
9. *Machine-Independent Optimization*, http://www.cs.virginia.edu/zephyr/vpo/, Dept. of Computer Science, School of Eng., University of Virginia, Charlottesville, Va., 1998.
10. C. Cifuentes et al., "Preliminary Experiences with the Use of the UQBT Binary Translation Framework," *Proc. Workshop on Binary Translation, Technical Committee on Computer Architecture News*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 12-22.

Cristina Cifuentes *is a faculty member in the Department of Computer Science and Electrical Engineering at the University of Queensland in Brisbane, Australia. She is on sabbatical leave at Sun Microsystems Inc. Her research interests are reverse engineering of binary and assembly code, software engineering, and the legal aspects of computing. Cifuentes has a PhD from the Queensland University of Technology. She is a member of the IEEE Computer Society and the ACM. Contact her at cristina@csee.uq.edu.au.*

Mike Van Emmerik *is a senior research assistant in the Department of Computer Science and Electrical Engineering at the University of Queensland. His research interests are in the area of low-level code and disassembling (including decompilation and binary translation), reengineering, and embedded systems. Van Emmerik has a BSc in computer science and a BE in electrical engineering, both from the University of Queensland. Contact him at emmerik@csee.uq. edu.au.*