Machine-Adaptable Dynamic Binary Translation*

David Ung and Cristina Cifuentes†
Department of Computer Science and Electrical Engineering
University of Queensland, QLD, Australia
{davidu,cristina}@csee.uq.edu.au

ABSTRACT

Dynamic binary translation is the process of translating and optimizing executable code for one machine to another at runtime, while the program is "executing" on the target machine.

Dynamic translation techniques have normally been limited to two particular machines; a competitor's machine and the hardware manufacturer's machine. This research provides for a more general framework for dynamic translations, by providing a framework based on specifications of machines that can be reused or adapted to new hardware architectures. In this way, developers of such techniques can isolate design issues from machine descriptions and reuse many components and analyses.

We describe our dynamic translation framework and provide some initial results obtained by using this system.

Keywords

Dynamic compilation, emulation, interpretation, dynamic execution, binary translation.

1. INTRODUCTION

Binary translation is a migration technique that allows software to run on other machines achieving near native code performance. Binary translation grew out of emulation techniques in the late 1980s in order to provide for a migration path from legacy CISC machines to the newer RISC machines. Such techniques were developed by hardware manufacturers interested in marketing their new RISC platforms. From mid 1990, binary translation techniques have been used to translate competitors' applications to the desired hardware platform. In the near future, we can expect to see such techniques being used to optimize programs within a family of computers, for example, by optimizing Sparc architecture binaries to UltraSparc architecture binaries.

UQBT, the University of Queensland Binary Translator, has developed techniques, specification languages and a complete framework for performing static translations of code [14,19]. In static binary translation, the code is translated off-line, before the program is run, by creating a new program that uses the machine instructions of the target machine. However, static translation has its limitations. Due to the nature of the von Neumann machine, where code and data are represented in the same way, it is not always possible to discover all the code of a program statically. For example, the target(s) of indirect transfers of control such as jumps on registers are sometimes hard to analyse statically. Therefore, a fall-back mechanism is commonly used with a statically translated program, in the form of an interpreter. The interpreter processes any untranslated code at runtime and returns to translated code once a suitable path is found.

The limitations of static binary translation are overcomed with dynamic translation, at the expense of performance. In a dynamic binary translator, code gets translated "on the fly", at runtime, while the user perceives ordinary execution of the program on the target machine. As oppossed to emulation, dynamic translation generates native code and performs on-demand optimizations of the code. Hot spots in the code are optimized at runtime to increase the performance of execution of such code. Further, some optimizations that are not possible statically are possible dynamically.

In this paper we describe the design of a *machine-adaptable* dynamic binary translator based on the static UQBT framework – UQDBT. A tool is said to be machine-adaptable when it can be "configured" to handle different source and/or target machines. In this way, a machine-adaptable dynamic binary translator is capable of being configured for different source and target machines through the specification of properties of these machines and their instruction sets. In other words, the translator is not bound to two particular machines (as per existing translators) but is capable of supporting a variety of source and target machines.

UQDBT differs from other dynamic translators in that it provides a clean separation of concerns, by allowing machine-dependent information to be specified, as well as performing machine-independent analyses to support machine-adaptability. In this way, UQDBT can support a variety of CISC and RISC machines at low cost. To support a new machine, the specifications for that machine need to be written and most of the UQDBT framework can be reused. New machine-specific modules may need to be added if a particular feature of a machine is not supported by the UQDBT framework and such feature is not generic across different architectures.

Dynamo '00 1/00 Boston, Massachusetts, USA
© 2000 ACM ISBN 1-58113-241-7/00/0001...$5.00

* The first author has support from an Australian Postgraduate Award. This work has further support from the Australian Research Council under grant No. A49702762 and Sun Microsystems, Inc.

† On sabbatical leave at Sun Microsystems, Inc.

This paper is structured in the following way. Section 2 discusses the static and dynamic frameworks for binary translation. Section 3 outlines the research problems of machine adaptable binary translation and what is addressed in UQDBT. Section 4 provides a case study of translation in the framework through an example program. Section 5 shows preliminary results in the use of the framework. Section 6 discusses effects of changing the granularity of translation and conclusions are given in Section 7. The work reported herein is work in progress.

1.1 Related work

In an attempt to improve on existing emulation techniques, companies in the late 1980s began using binary translation to achieve native code performance. Perhaps the most well known binary translators are Digital's VEST and mx[1], which translate VAX and MIPS machine instructions to 64-bit Alpha instructions. Both of these translators and others, like Apple's MAE[2] and Digital's Freeport Express[3] have a runtime environment that reproduces the old machine's operating environments. The runtime environment offers a fallback interpreter for processing old machine code that was not discovered at translation time, for example, due to indirect transfers of control.

In recent years, we have seen a transition to hybrid translators, which are proving to be extremely successful. The process of mixing translation with emulation and runtime profiling brought about some of the leading performers in the hybrid translation scene - Digital's FX!32[4], Executor by Ardi[5] and Sun's Wabi[6]. FX!32 emulates the program initially and statically translates it in the background, using information gathered during profiling. Embra[7], a machine simulator, is built using dynamic translation techniques that were developed in Shade; a fast instruction-set simulator for execution profiling [17]. Le[8] investigates out-of-order execution techniques in dynamic binary translators, though their results are based on an interpreter-based implementation. Many of the optimization techniques used in dynamic translators have been derived from dynamic compilers such as SELF[9] and tcc[10]. Runtime optimizations in such compilers can provide 0.9x – 2x the performance of statically compiled programs. Such techniques have also been used in Just-in-time (JIT) compilers for Java. JITs from Sun[11], Intel[12] and others dynamically generate native machine code at runtime.

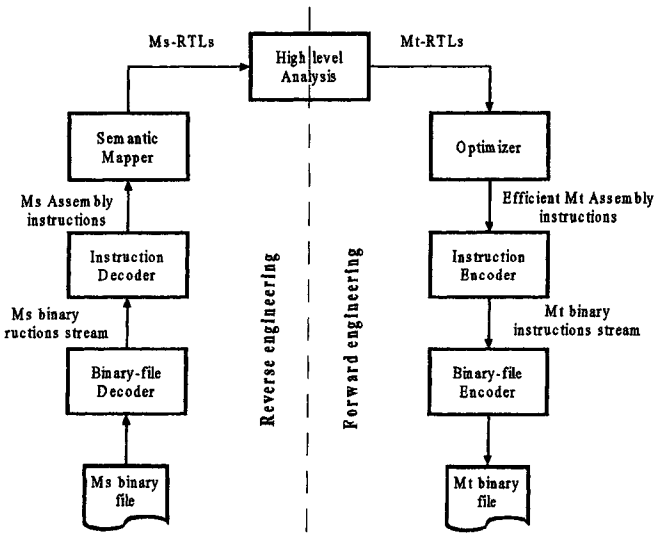

Figure 1. Static binary translation framework

To date, none of the current binary translators can generate code for more than one source and target machine pair. The machine-dependent aspects of the translation are hard coded into the translator, making it hard to reuse the translator's code for another set of machines. Our research differs from previous research in that machine-dependent issues are separated from machine-independent translation concerns, hence providing a way of specifying different machines (source and target machines) and supporting those specifications through reusable components, which implement the machine-independent analyses. This paper shows that this process is feasible and therefore enhances the reuse of code for the creation of dynamic binary translators. However, the machine-adaptability of the translator comes at the cost of performance, which is discussed in Section 5.

2. BINARY TRANSLATION FRAMEWORKS

Binary translation is a process of low-level re-engineering; that is, decoding to a higher level of abstraction, followed by encoding to a lower level of abstraction. Figure 1 gives a block-view of the UQBT static translation framework [14,19]. The re-engineering process is divided into the initial reverse engineering phase on the left-hand side and the forward engineering phase on the right-hand side. The reverse engineering steps recover the semantic meaning of the machine instructions by a three-step process of decoding the binary file, decoding the machine instructions of the code segment, and mapping such instructions to their semantic meaning in the form of register transfer lists (RTLs). The high-level analysis process lifts the level of representation of the code to a machine-independent form, performs binary translation specific optimizations on the code, and then brings down the level of abstraction to RTLs for the target machine. This is followed by the forward engineering process of optimizing the code, encoding the instructions into machine code and storing the code and data of the program in a binary file. The forward engineering process is standard optimizing compiler code generation technology.

RTL is a simple, low-level register-transfer representation of the effects of machine instructions. A single instruction corresponds to a register-transfer list, which in UQBT is a sequential composition of effects. Each effect assigns an expression to a location. All side effects are explicit at the top level; expressions are evaluated without side effects, using purely function RTL operators. An RTL language is a collection of locations and operators. For a machine M, the sub-language M-RTL is defined as those RTLs that represent instructions of machine M in a single RTL.

As previously mentioned, the problems with static binary translation are the inability to find all the code that belongs to a program and the limitation of optimizations to static ones, without taking advantage of dynamic optimization techniques. One of the hardest problems to solve during the decoding of the

machine instructions is the separation of code from data – any binary-manipulation tool faces the same problem. Unfortunately, this problem is not solvable in general as both code and data are represented in the same way in von Neumann machines. This makes static translation incomplete and hence a runtime support environment is needed in the form of an interpreter, for example.

2.1 Dynamic binary translation framework

In dynamic binary translation, the actual translation process takes place on an "as needed" basis, whereas static binary translation attempts to translate the entire program at once. Figure 2 illustrates a typical framework for a dynamic translator that uses a basic block as the unit of translation (i.e. its granularity). The left-hand side is similar to that of a static translator, but the processing of code is done at a different level of granularity (typically, one basic block at a time). The right-hand side is a little different to that of static translation. The first time a basic block is translated, assembly code for the target machine is emitted and encoded to binary form. This binary form is run directly on the target machine's memory as well as being kept in a cache. A mapping of the source and target addresses of the entire program for that basic block is stored in a map. If a basic block is executed several times, when the number of executions reaches a threshold, optimizations on the code are performed dynamically to generate better code for that hot spot. Different levels of optimization are possible depending on the number of times the code is executed.

Optimized code then replaces the cached version of that basic block's code. The processing of basic blocks is driven by a *switch manager*. The switch manager determines whether a new translation needs to be performed by determining whether there is an entry corresponding to a source machine address in the map. If an entry exists, the corresponding target machine address is retrieved and its translation is fetched from the cache. If a match is not found, the switch manager directs the decoding of another basic block at the required source address.

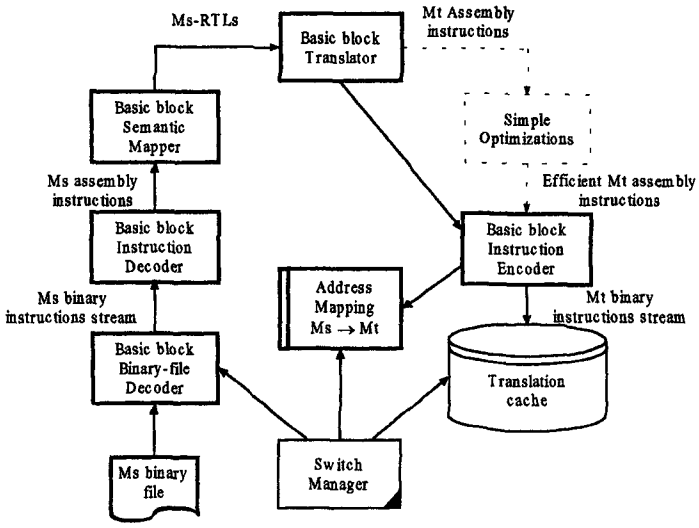

Figure 2. Dynamic binary translation framework

2.2 Machine-adaptable dynamic binary translation framework

Figure 3 extends Figure 2 to enable a dynamic translator to easily adapt to different source and target machines. This effort is achieved by a clean separation of concerns between machine-dependent information and machine-independent analyses. Through the use of specifications, a developer is able to concentrate on writing descriptions of properties of machines instead of having to (re)write the tool itself. The use of specifications to support machine-dependent information can also generate parts of the system automatically and provide a skeleton for the user to work on.

As seen in Figure 3, the decoding of the binary file to source

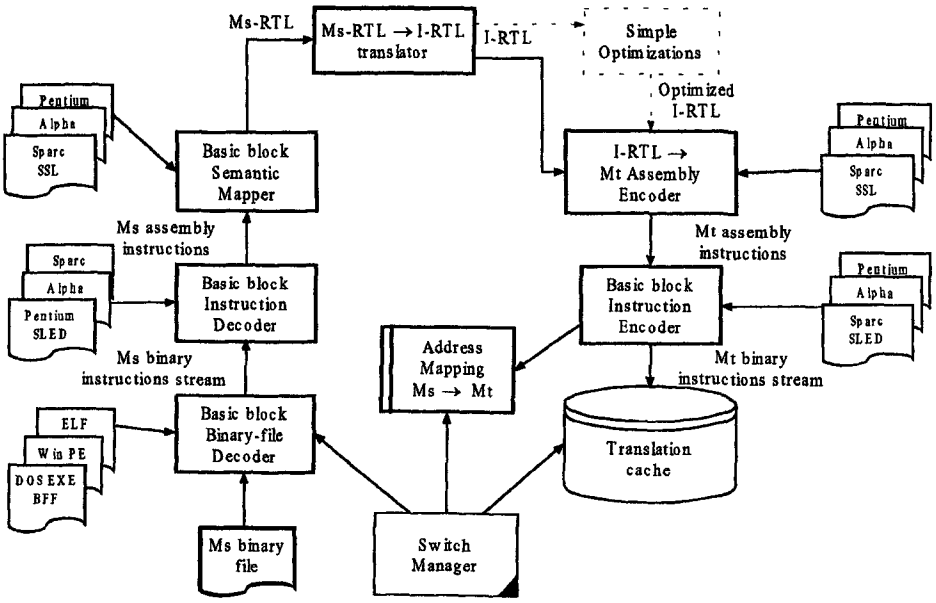

Figure 3. Machine-adaptable dynamic binary translation framework

machine RTLs (Ms-RTLs) requires the description of the binary-file format of the program, and the syntax and semantics of the machine instructions for a particular processor. We have experimented with three different languages, reusing the SLED language and developing our own BFF and SSL languages:

- BFF: the binary-file format language supports the description of a binary-file's structure [15]. Current formats supported are DOS EXE, Solaris ELF and to a certain extent, Windows PE. SRL, a simple resourceable loader, supports the automatic generation of code to decode files specified using the BFF language.
- SLED: the specification language for encoding and decoding supports the description of the syntax of machine instructions; ie. its binary to assembly mnemonic representation [18]. SLED is supported by the New Jersey machine-code toolkit [13]. The toolkit provides partial support for automatically generating an instruction decoder for a particular SLED specification. Current machines specified in this form include the Pentium, SPARC, MIPS and Alpha.
- SSL: the semantic specification language allows for the description of the semantics of machine instructions. SSL is supported by SRD [16]. SRD is the semantic mapper component, which supports the parsing of SSL files and the storing of such information in the form of a dictionary, which can be instantiated dynamically. The output of this stage is Ms-RTLs.

Ms-RTLs are converted to machine-independent RTLs (I-RTLs) through analyses, which remove machine dependent concepts of the source machine. This process identifies source machine's control transfers and maps it to the more general forms in I-RTL. For example: the following SPARC Ms-RTL for a call instruction

```
*32*    %o7 = %pc
*32*    %pc = %npc
*32*    %npc = 0x40000
```

is associated as a high-level call instruction in I-RTL. Other forms of transfers of control that exist in I-RTL are jumps, returns, conditional and unconditional branches. I-RTL supports register transfers, stack pushes and pops, high-level control transfers, and condition code functions. Some of these higher-level instructions allow for abstraction from the underlying machine. I-RTLs are converted into Mt-assembly instructions by mapping the functionality of such register transfers to the instructions available on the target machine, assisted by the SSL specifications. The instruction encoding process is supported by the SLED specification language, which maps assembly instructions into binary form. This code is then stored in the translator's cache for later reference.

As an example, Figure 4 shows the various instruction transformations during the translation of a Pentium machine instruction to a SPARC machine instruction. The reverse-engineering stage decodes the Pentium binary code (0000 0010 1101 1000) to produce Pentium assembly code, which is then lifted to Pentium-RTLs and finally abstracted to I-RTL by replacing machine-dependent registers with virtual registers. The forward-engineering phase encodes the I-RTL to SPARC-RTL, SPARC assembly instructions and finally SPARC binary code.

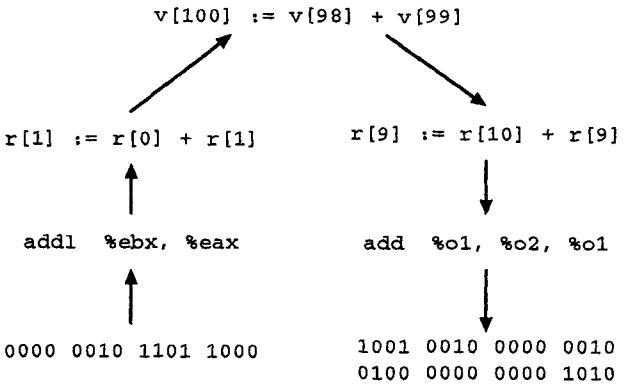

Figure 4. Pentium to SPARC example

2.3 Specification requirements in dynamic binary translation

Dynamic translation cannot afford time-consuming analyses to lift the level of representation to a stage that resembles a high-level language, as per UQBT [14]. In UQBT, static analyses recover procedure call signatures including parameters and return values, thereby allowing the generated code to use native calling and parameter conventions on the target machine. If such analyses were used in dynamic translation, high performance degradation would be experienced during the translation.

The alternative to costly analyses to remove properties of the underlying source machine is to go halfway to a high-level representation. We support inexpensive analyses to recover a basic form of high-level instructions (such as conditional branches and calls without parameters) and we emulate (rather than abstract away from) conventions used by the hardware and operating system in the source machine (i.e. without using the native conventions on the target machine). Both these steps are possible through the specification of features of the underlying hardware. For example, we emulate the SPARC architecture register windowing mechanism on a Pentium machine by specifying how this mechanism works. On a SPARC machine, we emulate the Pentium stack parameter passing convention. However, we do not emulate the SPARC processor delayed transfers of control as we support higher level branching instructions. Clearly, this compromise has a performance impact on the translated code, but it provides a fast way of translating code, which can then be optimized at runtime if it becomes a hotspot in the program.

In order to support the translation of Ms-RTLs to I-RTLs, the SSL language has been extended from machine instruction level semantics to include hardware semantics as well. For example, for the SPARC architecture, the effects of changing register windows and how each register in the current window is accessed were specified, and for the Pentium architecture, properties about the stack movement were specified. This information is currently not used by UQBT; only by UQDBT. UQBT relies on costly analysis to abstract higher-level information, without depending on very low-level details of the underlying hardware.

3. RESEARCH PROBLEMS

Unlike other dynamic binary translators that are written with a fixed set of source and destination machines in mind, UQDBT is designed to handle a wide range of CISC and RISC machine architectures. While some translators can directly map source machine-specific idioms to the target machine, such translators are bound to work only under that source/target pair. To extend those translators to support different machines, extensive rewriting of the code is needed, as the direct idiom mapping between machines is different. The goal of UQDBT is to provide a framework that can be modified and extended with ease to support additional source and target machines without the need to rewrite a new translator from scratch. The process of finding a generalization of all existing (and future) machines is non-trivial and cannot be fully predicted. UQDBT uses UQBT's approach of specifying properties of machine instruction sets that are widely available in today's machines and allowing the user to extend the specification language to support new features of (future) machines to reuse the rest of the translation framework. As with UQBT, we use multi-platform operating systems to concentrate on the more fundamental issues of instruction translations.

UQDBT's goal was to address the following types of research problems in dynamic machine-adaptable binary translation:

1. What is the best way of supporting the machine-dependent to machine-independent RTL translation? The main criteria in the translation is efficiency, hence expensive analyses are not an option. Further, the translation needs to be supported by the underlying specification language, in order to generate Ms-RTLs that contain enough information about the underlying Ms machine.

2. How much state of the source machine is needed for dynamic translation and what effects does this have on specification of such properties of the machine?

3. What is the best way of automating the transformation of I-RTLs down to Mt-assembly code? Can a code selector be automatically generated from a target machine specification?

4. Is it possible to efficiently use specifications that contain information about operating system conventions, such as calling and parameter conventions used by the OS to communicate with the program? For example, in order to use Pentium's stack parameter convention in code that was translated from a SPARC architecture binary (which passes parameters on registers), analysis to determine the parameters needs first to be performed.

3.1 Implementation of UQDBT

We have been experimenting with the right level of description required in order to support dynamic translation based on specifications. In our experience, too low-level or high-level a description of the underlying machine is unsuitable. We view UQBT's semantic specifications of machines as a high-level description, as they only describe the machine instruction semantics but do not specify the underlying hardware that supports the control transfer instructions (e.g. the register windows and delayed instructions on the SPARC architecture). In UQBT, such detailed level of information is not needed because of the specification and use of calling conventions and control transfer instruction. Further, other semantic description languages are used to describe all the low-level details of the underlying machine; such languages are suitable for emulation purposes but contain too much information for dynamic translation.

The first problem above has been addressed by specifying how the hardware works in relation to control transfer instructions; this provides for a fast translation of Ms-assembly instructions into information-rich Ms-RTLs (the extended ones), avoiding the need to recover information at runtime. The following are the types of information that have been described for SPARC and Pentium processors:

- The effects of the SPARC register windowing mechanism
- Stack properties
- Memory alignment
- Parameters and return locations.

The SPARC machine allocates a new set of working registers each time a SAVE instruction is called. In other words, it effectively provides an infinite number of registers for program use. The effect of the SPARC register windows is captured by extending SSL to specifying how each of the registers are accessed and how the register windows change during each `save` and `restore` instruction, hence providing a different set of working registers. This provides accurate simulation on target machines that only have a limited amount of usable registers.

The effects of the stack pointer are different on different types of machines. On Pentium machines, the stack pointer can change indefinitely within a given procedure. On RISC machines, the stack pointer is normally constrained to a pre-allocated stack frame's fixed size that includes enough space for all register spills of that procedure. Specifying how the stack changes in the original machine suggests ways for the code generator to generate stack manipulation instruction on the target machine. For example, simulating stack pushing and popping on a SPARC machine.

Memory alignment places constraints on how the machine state at a particular point in the program should be. In SPARC, the frame pointer and stack pointer need to be double word aligned. Thus, the code generator needs to enforce such conditions before entry to or exit from a call.

Differences in machine calling conventions, namely how the parameters are passed and where return values are stored, play a crucial part on how the code generator constructs the right setup when calling native library functions. SPARC generally passes parameters in registers while Pentium pushes them on the stack. This information is needed for both source and target machines to identify the transformation of parameters and return values.

Differences in endianness between source and target machines requires byte swapping to be performed when loading and storing data. Byte swapping is an expensive process. Although the Pentium can do byte swapping quite easily, it takes about 10 SPARC V8 instructions for a 32-bit swap. This is an expensive process and should be avoided if possible. In particular, when running a Pentium binary on a SPARC, every push and pop instruction (which appears quite often in Pentium programs) will require byte swapping. Heuristics are used in UQDBT to avoid byte swapping on pushing and popping to the stack.

```
main()
        8048918:  55                    pushl  %ebp
        8048919:  8b ec                 movl   %esp,%ebp
        804891b:  68 f8 93 04 08        pushl  $0x80493f8
        8048920:  e8 9b fe ff ff        call   0xfffffe9b <printf>
        8048925:  83 c4 04              addl   $0x4,%esp
        8048928:  33 c0                 xorl   %eax,%eax
        804892a:  eb 00                 jmp    0x0
        804892c:  c9                    leave
        804892d:  c3                    ret
```

Figure 5. "Hello World" x86 disassembly

The second problem above is related to the first one. The amount of source machine state that is carried across depends on the effectiveness of translation in the first problem. For areas that are not easily specified or unspecified, they are carried across and are apparent within the machine-independent RTL. In UQDBT, control transfer instructions will contain a tag indicating how to process its delayed slot instruction (in architectures that support delayed slots).

The third problem above is current work in progress. The goal is not only to automatically construct the code generator, but also determine the best performance heuristics for selecting target machine instruction when encountering similar patterns. Some patterns may never be matched or may be nearly impossible to match. For example, trying to pattern match a SPARC `save` instruction with some Pentium-RTLs.

Our experiences with the fourth problem above suggest that performance can be gain by using native OS conventions. UQDBT currently simulates the calling convention for Pentium programs on a SPARC machine, i.e. parameters are passed on the stack instead of in registers. To remove this simulated effect and convert it to use native conventions, one needs know:

- How much improvement does it offer over direct machine simulation?
- At what level should this conversion occur?
- Is it worth while doing such analysis in a dynamic binary translation environment?

4. CASE STUDY

In this section we show an example of a small Pentium program converted by UQDBTps (the "ps" postfix indicates translations from Pentium to SPARC architectures) to run on a SPARC machine. Both programs are for the Solaris operating system. The main differences of the two test machines are:

1. SPARC is a RISC architecture, whereas Pentium is CISC.
2. SPARC is big-endian, while Pentium is little-endian.
3. SPARC passes parameters in registers (and sometimes on the stack as well), while Pentium normally passes them on the stack.

```
8048918: PUSH r[29]
8048919: *32* r[29] := r[28]
804891b: PUSH 134517752
8048920: *32* r[28] := r[28] - 4
         *32* m[r[28]] := %pc
         CALL 0x80487c0

8048925: *32* r[tmp1] := r[28]
         *32* r[28] := r[28] + 4
         ADDFLAGS32( r[tmp1], 4, r[28] )
8048928: *32* r[24] := r[24] ^ r[24]
         LOGICALFLAGS32( r[24] )
804892a: JUMP 0x804892c

804892c: *32* r[28] := r[29]
         *32* r[29] := POP 32
804892d: RET
```

Figure 6: I-RTLs for "Hello World"

4.1 Basic block translations and address mappings

Figure 5 is the disassembly of a "Hello World" binary program compiled for the Pentium machine running Solaris. The first column is the source address seen by the Pentium processor. The second and third columns are the actual Pentium binaries and its corresponding assembly representation. There are 3 basic blocks

```
0x43676c8:  save  %sp, -132, %sp
0x43676cc:  add  %sp, -4, %sp
0x43676d0:  add  %i7, 8, %l0
0x43676d4:  st  %l0, [ %sp + 0x84 ]
0x43676d8:  mov  %sp, %l0
0x43676dc:  subcc  %l0, 4, %l0
0x43676e0:  mov  %l0, %sp
0x43676e4:  mov  %fp, %l0
0x43676e8:  st  %l0, [ %sp + 0x84 ]
0x43676ec:  mov  %sp, %l0
0x43676f0:  mov  %l0, %fp
0x43676f4:  mov  %sp, %l0
0x43676f8:  subcc  %l0, 4, %l0
0x43676fc:  mov  %l0, %sp
0x4367700:  sethi  %hi(0x8049000), %l0
0x4367704:  add  %l0, 0x3f8, %l0    ! 0x80493f8
0x4367708:  st  %l0, [ %sp + 0x84 ]
0x436770c:  ld  [ %sp + 0x84 ], %o0
0x4367710:  mov  %fp, %l0
0x4367714:  mov  %sp, %l1
0x4367718:  sethi  %hi(0xfffffc00), %l2
0x436771c:  add  %l2, 0x3f8, %l2    ! 0xffffffff8
0x4367720:  and  %sp, %l2, %sp
0x4367724:  and  %fp, %l2, %fp
0x4367728:  sethi  %hi(0xef663400), %g6
0x436772c:  call  %g6 + 0x1b8        ! <printf>
0x4367730:  nop
0x4367734:  mov  %l1, %sp
0x4367738:  mov  %l0, %fp
0x436773c:  sethi  %hi(0x8048800), %g5
0x4367740:  add  %g5, 0x125, %g5
0x4367744:  sethi  %hi(0x41bfc00), %g6
0x4367748:  call  %g6 + 0x370 !<switch_manager>
0x436774c:  nop
```

Figure 7. Generated Sparc assembly for the 1st BB of Figure 6

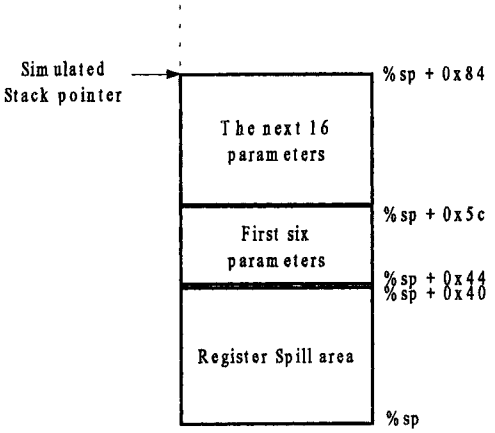

Figure 8. Sparc stack frame

(BBs) in this program:

- first BB - 4 instructions (0x8048918 to 0x8048920)
- second BB - 3 instructions (0x8048925 to 0x804892a)
- third BB - 2 instructions (0x804892c to 0x804892d)

Figure 6 shows the intermediate representation (I-RTLs) for these BBs. Note that the translation is done incrementally, i.e. each BB is decoded separately at runtime.

UQDBTps works in the source machine's address space and translates a basic block at a time. Both the data and text from the source Pentium program are mapped to the actual machine's source address space even though it is actually running on a SPARC machine. For example, a Pentium program with data and text sections located at 0x8040000 and 0x8048000 will be mapped exactly at these addresses even though a typical SPARC program expects the text and data at addresses 0x10000 and 0x20000. UQDBTps also simulates the Pentium machine's environment in the SPARC generated code, i.e. the pushing and popping of temporaries and parameters from the Pentium machine is preserved in the generated code. UQDBTps tries to generate code as quickly as it possibly can with little or no optimization.

4.2 Pentium stack simulation

Figure 7 is the SPARC code generated for the first BB of Figure 6. The first four instructions simulate the Pentium main prologue by setting up the stack and storing the return address (obtained by the value of `%i7 + 8`). 132 bytes of stack space are reserved in the initial `save %sp, -132, %sp`. This space is used by the SPARC processor to store parameters, return structures, local variables and register spills. Hence the actual simulated Pentium stack pointer `%esp` starts at `%sp+0x84` (see Figure 8), while Pentium's `%ebp` is mapped to the SPARC register `%fp`. Pushing is handled by subtracting the size of the value pushed from `%sp` and storing the result in `[%sp+0x84]`. Popping removes from `[%sp+0x84]` and increments `%sp` by the appropriate size.

4.3 Function calls and stack alignments

In Pentium, actual parameters to function calls are passed on the stack while SPARC parameters are passed in registers. The

```
0x435ff58:  sethi  %hi(0x4485c00), %l0
0x435ff5c:  add  %l0, 0x3e8, %l0     ! 0x4485fe8
0x435ff60:  mov  %sp, %l1
0x435ff64:  st  %l1, [ %l0 ]
0x435ff68:  mov  %sp, %l0
0x435ff6c:  addcc  %l0, 4, %l0
0x435ff70:  mov  %l0, %sp
0x435ff74:  sethi  %hi(0x42b7000), %l0
0x435ff78:  add  %l0, 0x3d4, %l0     ! 0x42b73d4
0x435ff7c:  rd  %ccr, %l1
0x435ff80:  st  %l1, [ %l0 ]
0x435ff84:  sethi  %hi(0x4486000), %l0
0x435ff88:  add  %l0, 0x18, %l0      ! 0x4486018
0x435ff8c:  sethi  %hi(0x4486000), %l1
0x435ff90:  add  %l1, 0x18, %l1      ! 0x4486018
0x435ff94:  ld  [ %l1 ], %l1
0x435ff98:  sethi  %hi(0x4486000), %l2
0x435ff9c:  add  %l2, 0x18, %l2      ! 0x4486018
0x435ffa0:  ld  [ %l2 ], %l2
0x435ffa4:  xorcc  %l1, %l2, %l1
0x435ffa8:  st  %l1, [ %l0 ]
0x435ffac:  sethi  %hi(0x42b7000), %l0
0x435ffb0:  add  %l0, 0x3d4, %l0     ! 0x42b73d4
0x435ffb4:  rd  %ccr, %l1
0x435ffb8:  st  %l1, [ %l0 ]
0x435ffbc:  sethi  %hi(0x8048800), %g5
0x435ffc0:  add  %g5, 0x12c, %g5     ! 0x804892c
0x435ffc4:  sethi  %hi(0x41bfc00), %g6
0x435ffc8:  call  %g6 + 0x370 !<switch_manager>
0x435ffcc:  nop
```

Figure 9. Generated Sparc assembly for the 2nd BB of figure 6

`printf` format string to "Hello World" is at address `$0x80493f8` and is pushed by the instruction `804891b` (see Figure 5). To successfully call the native SPARC `printf` function, this address must be stored in register `%o0` (instruction `0x436770c` in Figure 7). The equivalent `printf` in SPARC is at `0xef6635b8` and a call to this function is made (instruction `0x436772c`). Calls to library functions such as `printf` are assumed by UQDBTps to exist on the source as well as the target machine. This assumption is not restrictive as long as there is a mapping from the source library function to an equivalent function on the target machine; i.e. libraries can be reproduced on the target machine by translation or rewriting to produce such a mapping. Translators such as FX!32 make the same assumption.

SPARC machines expect `%sp` and `%fp` to be aligned to double word (64 bits) boundaries. Therefore, before calling a native SPARC library function, `%sp` and `%fp` need to be 8-byte aligned. The current values are restored after the function call returns (instructions `0x4367734` and `0x4367738`).

At the end of a basic block, control is passed back to UQDBTps' switch manager, with the indication of the next basic block address to be processed in `%g5` (`0x8048925` in the above case). It is the role of the switch manager to decide whether to start the translation indicated in `%g5` or fetch an already translated BB from the translation cache. In the above case (where the next BB starts at `0x8048925`), the address is not in the translation map, hence the translation starts at that new address. Figure 9 shows the generated SPARC assembly for the next BB for the Pentium program.

4.4 Register mapping and condition codes

During the translation, all Pentium registers are mapped to virtual registers (i.e. memory locations). To access a virtual register on a SPARC machine, a `sethi` and an `add` instruction are used. For example, instructions `0x435ff84` and `0x435ff88` in Figure 9 are used to access the virtual register representing the x86 register `%eax`.

While most instructions on SPARC do not affect condition codes (flags) unless explicitly indicated by the instruction, almost all Pentium instructions affect the flags. Each Pentium instruction that affects the status of the flags is simulated using the equivalent condition code version of the same instruction on the SPARC machine (instructions `0x435ff6c` and `0x435ffa4`). The condition codes are read (instruction `0x435ff7c`) and saved to the virtual flag register (instruction `0x435ff80`) after these instructions, to preserve its current value, which can be retrieved later if required.

A closer look at the above example shows that the generated code is not very efficient. Simple optimizations such as forward substitutions and dead code elimination can greatly reduce the size of the generated code. Such optimizations can yield better code but will take longer to generate – a trade off between code quality and speed. The code generator back-end of UQDBTps is very fast despite the poor quality of the generated code. We are currently implementing on-demand optimizations, to the hotspots in the program, as well as performing register allocation.

5. PRELIMINARY RESULTS

UQDBT is based on the UQBT framework, as such, its front-end is re-used from UQBT, changing its granularity of decoding from the procedure level to the basic block level. The front-end uses the extended SSL specifications and generates Ms-RTLs. The machine instruction encoding routines of the back-end are automatically generated from SLED specifications using the NJMC toolkit.

This section shows some preliminary results obtained by two dynamic translators instantiated from the UQDBT framework; UQDBTps (Pentium to SPARC) and UQDBTss (SPARC to SPARC). It then looks at the types of optimizations that need to be introduced in order to improve the performance of frequently executed code, and it gives the reader an idea of effort gone into the development of the framework and the amount of reuse expected.

5.1 Performance

Micro-benchmark results were obtained using a Pentium MMX 250 MHz machine and an UltraSparc II 250 MHz machine, both running the Solaris operating system. The results reported herein are those of the translation overhead and do not currently make use of dynamic optimizations, only register caching within basic blocks. Clearly, the performance of generated binaries from UQDBT without optimization is inferior to direct native compilation. A typical 1:10 ratio (i.e. 1 source machine instruction to 10 target machine instructions) is expected in a typical emulator/interpreter without caching. UQDBTps gives figures close to this ratio. For example, for the 9 Pentium instructions of Figure 5, 95 SPARC instructions were generated. While this ratio is similar to that of emulation, the speed up gained from UQDBT comes from reusing already translated BBs from the translation cache when the same piece of code is executed again.

In its present form, UQDBT is still in its early development and hence we provide preliminary results for UQDBTps, a Pentium to SPARC translator, and UQDBTss, a SPARC to SPARC translator. It is undoubtedly true that there is little practical use for SPARC to SPARC translation unless runtime optimizations can significantly speed up translated programs. The inclusion of this translation is to show the effect of machine-adaptability in UQDBT. Further, the translation from SPARC binaries to I-RTL removes any machine dependencies and thus, during I-RTL to SPARC code generation, the UQDBTss is unaware of the fact that the source machine is SPARC. This is also true for UQDBTps. Since little analysis is done to the decoded instructions and processing is concentrated on decoding and code generation requested by the switch manager, it better reflects the performance impact on the use of on-demand techniques prior to introducing optimizations.

The test programs showed in the tables are:

- Sieve 3000 (prints the first 3,000 prime numbers),
- Fibonacci of 40, and
- Mbanner (prints the banner for the "ELF" string 500,000 times).

Sieve mainly contains register to register manipulation, while Fibonacci has a lot of recursive calls and Mbanner has a lot of stack operations and accesses to an array of data.

Tables 1 and 2 show the times of translation and execution of programs using UQDBTps and UQDBTss, compared to natively gcc O0 compiled programs. The source programs were also O0 compiled. Column 2 shows the preprocessing time that is needed before the actual translation takes place. Note that UQDBTps takes longer to start than UQDBTss. This is because Pentium has a larger instruction set (hence a larger SSL specification file) which takes longer to process. It is also caused by different page alignment sizes between the Pentium and SPARC; as a result, extra steps are taken to ensure that both text and data sections are loaded correctly on the SPARC machine. Column 3 shows the total time spent decoding the source instructions, transforming them to I-RTLs and generating the final SPARC code. Column 4 shows the execution time in the generated SPARC code without using register caching, i.e. every register access was done through virtual registers. Column 5 shows the execution time of the generated SPARC code with register caching, which yields between 15 to 50 percent performance gain. Column 6 is the natively compiled gcc version of the same program on SPARC. Comparing columns 5 and 6 gives the relative performance of the translators. The figures suggest 2 to 6 times slowdown when running programs using UQDBTps and UQDBTss. The slow performance of the translated Fibonacci program under UQDBTss is caused by the effects of the register windowing mechanism in SPARC, which are carried forth to the I-RTLs. Since the I-RTLs are unaware of the fact that the source and target machines are the same, this causes the entire register windowing system to be simulated in the generated code. Given that on-demand optimizations have not been performed yet, the quality of the generated code is comparable to O0 optimization level of a traditional compiler.

Tables 3 and 4 show the efficiency of the translators relative to the size of the original program. Column 2 is the size of the

Test programs	Pre-processing	Translation time	Execution time without reg caching	Execution time - simple reg caching	Native gcc compiled
Sieve3000	0.52	0.12	79.95	66.98	29.22
Fibonacci	0.52	0.07	162.23	139.35	41.18
mbanner	0.50	0.34	191.00	126.28	22.85

Table 1: UQDBTps - Pentium to SPARC translation (second)

Test programs	Pre-processing	Translation time	Execution time without reg caching	Execution time - simple reg caching	Native gcc compiled
Sieve3000	0.20	0.17	77.38	56.89	29.22
Fibonacci	0.22	0.12	256.05	198.97	41.18
mbanner	0.21	0.50	204.09	97.09	22.85

Table 2: UQDBTss - SPARC to SPARC translation (seconds)

Test programs	Original program size	Source bytes decoded	Target bytes generated w/o reg caching	Target bytes generated w/- reg caching	1,000 cycles / source byte
Sieve3000	118	145	1560	1472	206
Fibonacci	102	94	1188	1104	186
mbanner	483	467	4548	3816	182

Table 3: UQDBTps - Pentium to SPARC translation

Test programs	Original program size	Source bytes decoded	Target bytes generated w/o reg caching	Target bytes generated w/- reg caching	1,000 cycles / source byte
Sieve3000	216	260	2256	1716	163
Fibonacci	220	176	1764	1468	170
mbanner	748	760	7112	5032	164

Table 4: UQDBTss - SPARC to SPARC translation

program's text area. Note that not all bytes necessarily represent instructions and that not all code is necessarily reachable or executed at runtime. Column 3 shows the actual bytes decoded by the translator at runtime. This number varies from Column 2 since only valid paths at runtime are translated, and sometimes re-translation is needed when a jump into the middle of a BB is made. Columns 4 and 5 show the number of bytes of code generated by the translator without/with register caching. Register caching has been done at a basic block level; cached registers are copied back to their memory locations at the end of each basic block. Comparing column 5 with column 3 gives the relative ratio of bytes generated versus bytes decoded. The above figures suggest that on average, each byte from the source machine translates to around 7 to 10 bytes of target SPARC code. The last column is the ratio of machine cycles to bytes of source code. It gives a rough indication of the performance of the translation. On an UltraSparc II 250MHZ, the translators require about 180,000 machine cycles per byte of input source, which is about 10 times more cycles used than in a traditional O0 compiler.

5.2 Optimizations – future work

Most programs spend 90% of the time in a small section of the code. It is these hotspots that are worthwhile for a dynamic binary translator to spend time optimizing. UQDBT currently does not perform any optimizations. The next revision of UQDBT will contain optimizations that are triggered by counters. Counters are inserted in basic blocks to indicate the number of times a particular basic block is executed at runtime. When a certain threshold is reached (indicating that the program spends significant time in a piece of code), the optimizer will be invoked in an attempt to produce efficient code. Four levels of optimization will be provided by UQDBT progressively when certain thresholds are reached:

1. Register liveness analysis, forward substitution, constant propagation – improves the quality of the generated code and reduces the number of instructions executed.
2. Register allocation – a more rigorous process for removing access to virtual registers and replacement with allocation to hardware registers on the target machine, assisted by liveness information, rather than just caching registers at a basic block level.
3. Code movement – moving and joining frequently executed BBs closer together, thus reducing transition costs (calls and jumps).
4. Customization – create specialized versions of the BBs that are found to have a fixed range of runtime values within the BB, e.g. on repeated entry to a BB, a register or variable contains the same value 90% of the time.

5.3 Effort

In order to give the reader an idea of the effort that has gone into the development of UQDBT and the effort of reusing the system, we quantify such effort as follows.

UQDBT has been the effort of 1 person over a period of 1.5 years, experimenting with the amount of specification required at the semantic level for different machines. This effort was performed by a person who was already familiar with the UQBT framework; having worked on SSL in the past.

UQDBT's current implementation size is of 18,500 lines of source code in C++, 3,300 lines of partially-generated code, and 3,500 lines (1,000 for SPARC and 2,500 for Pentium) of

specification files. A user of the UQDBT framework would be able to reuse most of this source code and would need to write syntax and semantics specification files for new machines (or reuse existing ones). These figures are not final at this stage, as most dynamic optimizations have not been implemented yet. It nevertheless gives an indication of the amount of reuse of code in the system.

6. DISCUSSION

The preliminary results of UQDBT point at the tradeoffs of machine-adaptability. In return for writing less code to support two particular machines, a performance penalty in the generated code is seen at this stage. A binary translation writer would be expected to write specifications for new machines, which are in the order of a few thousand lines of code, and reuse a good part of 18,500 lines of code, reaping the benefits of reuse and time efficiency. However, at this stage, UQDBT generates code that performs at about the same speed as emulated code, therefore a user seeing a 10x performance degradation in their translated programs. The introduction of register caching in the generated code has brought down this factor to 6. We expect that the introduction of on-demand optimizations on hotspots of the program will improve the performance of the generated code, bringing down the performance factor to 2x-3x.

One of the main questions we have dealt with throughout experiments in this area has been how much should be specified and how much should be supported by hand. The level of detail in a specification can make a translator faster or slower. If the full details of a machine are specified, the specification is suitable for generating an emulator that supports 100% that machine. However, if we can provide a means for eliminating part of that emulation process, then a different type of specification is needed. This is what we have tried to achieve through our semantic specifications and the use of two intermediate languages. The RTL language describes low-level and machine specific aspects of a machine, and UQDBT finds support in the specifications to perform simple analyses to lift the level of the representation to I-RTL. The aim is to perform simple transformations of the code that are not expensive on time and that are generic enough to be suitable for our intermediate representations. This is why I-RTL is different to HRTL, the high-level intermediate representation used by the static UQBT framework. In HRTL, expensive analyses recover parameters to procedures and return values, hence allowing the code generator to use native calling conventions on the target machine. In I-RTL, the code generator makes use of the specification of a stack for example, in order to pass parameters on the stack, without ever determining which locations are parameters to procedure calls. However, some notion of parameters is needed in order to interface correctly to native library functions, and to pass parameters in the right locations.

It has also been our experience that some modules may be better off written by hand, without specifying the complete semantics of features of a machine that are too unique. For example, one could consider implementing an SPARC-specific module for supporting the register windowing semantics, so that better register allocation is performed in this case. At present, we have specified the register windowing mechanism and generated code that puts all these registers in virtual (memory) locations. Through the use of register caching, some of the memory locations are mirrored to hardware registers of the target machine, improving somewhat the performance of the program. However, there is still a large overhead in the copying of the registers to virtual locations at each call and return. This can be reduced with dead code elimination, but perhaps hand written code would have achieved better code.

Another aspect to take into consideration is the granularity of translation. In UQDBT, the granularity unit for processing is a basic block (BB) at a time. Just after code generation, a link is made to the switch manager at the exit of each BB and flushing of cached registers to virtual registers is performed. This is needed to keep data accurately stored and consistent across transitions from one BB to another. Transitions from one BB to the next will go via the switch manager if the next BB has not been translated yet. Using BB as the unit of translation restricts the effectiveness of register allocation. Since BBs are relatively small, it is difficult to determine register live-ness information, as data is not collected across BB boundaries. If the unit of granularity is changed, this could yield better code in some cases while worse code in others. For example, the translation unit might be changed from a BB to a procedure at a time. This would allow the code generator to reduce the amount of flushing of cached registers and hence reduce the number of instructions that need to be executed at runtime. It would improve the effectiveness of allocating registers during code generation since more live-ness information could be collected. But using a larger unit of translation such as a procedure may involve decoding paths that may not be ever taken at runtime, thus generating code that is not executed. It is not obvious what the granularity unit should be since some types of programs will benefit by using a particular granularity unit while others may suffer. Program with a lot of small procedures will benefit if the unit of translation is a procedure, but suffer if the program has a lot of conditional branches.

7. CONCLUSION

UQDBT is a machine-adaptable dynamic binary translator framework that is capable of being configured for different source and target machines through specifications of properties of those machines. The UQDBT framework can be modified and extended with ease to support additional source and target machine architectures without the need to write a new translator from scratch.

Our case study shows that the translation process between two different architectures is both complex and challenging using machine-adaptable dynamic translation techniques. Nevertheless, preliminary results suggest that performance of implementing on-demand processing in a dynamic system can be done efficiently. Despite that, some research problems remain in building a fully machine-adaptable dynamic translation framework. UQDBT appears to be a promising model to provide a generic dynamic binary translation framework.

8. ACKNOWLEDGMENTS

The authors wish to thank Mike Van Emmerik for his helpful discussions in implementation and testing strategies and the members of the Kanban group at Sun Microsystems, Inc.; whose

Self system motivated some of this work. This work is part of the University of Queensland Binary Translation (UQBT) project. More information can be obtained about the project by visiting the following URL:

http://www.csee.uq.edu.au/csm/uqbt.html.

9. REFERENCES

1. 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.
2. Apple Corporation. *Macintosh application environment*. http://www.mae.apple.com/, 1994.
3. Digital. *Freeport express*. http://www.digital.com/amt/freeport/, 1995.
4. R.J. Hookway and M.A. Herdeg. *Digital FX!32: Combining emulation and binary translation*. Digital Technical Journal, 9(1):3-12, 1997.
5. ARDI. *Executor Internals: How to Efficiently Run Mac Programs on PCs*. http://www.ardi.com/MacHack/machack.html, 1996.
6. SunSoft. *Wabi*. http://www.sun.com/sunsoft/Products/PC-Integration-products/, 1994.
7. Emmett Witchel and Mendel Rosenblum, *Embra: Fast and Flexible Machine Simulation*. The proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems, Philadelphia, 1996.
8. Bich C. Le. *An out-of-order execution technique for runtime binary translators*. In Proceedings of the 8th international conference on Architectural support for programming languages and operating systems, pages 151-158, San Jose, CA. Oct 1998.
9. D. Ungar and R.B. Smith. *SELF: The power of simplicity*. In Conference on Object-Oriented Programming Systems, Languages and Applications, pages 227-241. ACM Press, October 1987.
10. Massimiliano Poletto, Dawson R. Engler and M. Frans Kaashoek. *tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation*. In PLDI '97. Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation, pages 109-121, Las Vegas, NV, June 1997.
11. Sun, *Java JIT compiler*, http://www.sun.com/solaris/jit.
12. Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh and James M. Stichnoth. *Fast, effective code generation in a just-in-time Java compiler*, In PLDI '98, Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation, pages 280-290, Montreal, Canada, June 1998.
13. Norman Ramsey and Mary Fernández. *The New Jersey Machine-Code Toolkit*. Proceedings of the 1995 USENIX Technical Conference, pages 289-302, New Orleans, LA, January 1995.
14. C. Cifuentes, M. Van Emmerik and N. Ramsey, *The Design of a Resourceable and Retargetable Binary Translator*. In Proceedings of the Working Conference on Reverse Engineering, pages 280-291, Atlanta, USA, Oct 1999. IEEE CS Press.
15. D. Ung and C. Cifuentes. SRL – *a simple retargetable loader*. In Proceedings of the Australia Software Engineering Conference, pages 60-69, Sydney, Australia, Sept 1997. IEEE CS Press.
16. 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.
17. B. Cmelik and D. Keppel. Shade: *A Fast Instruction-Set Simulated for Execution Profiling*. SIGMETRICS, Nashville, TN, 1994.
18. Norman Ramsey and Mary Fernández. *Specifying representation of machine instructions*. ACM Transactions of Programming Languages and Systems, 19(3):492-524, 1997.
19. 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, Newport Beach, USA, Oct 1999. Published in Technical Committee on Computer Architecture News, pages 12-22, Dec 1999, IEEE CS Press.