Dynamic binary translation using run-time feedbacks

David Ung [a,*], Cristina Cifuentes [b]

[a] *Department of Information Technology and Electrical Engineering, University of Queensland, QLD, Australia*
[b] *Sun Microsystems Laboratories, Menlo Park, CA 94025, USA*

Available online 16 November 2005

Abstract

Dynamic binary translation is the process of translating, modifying and rewriting executable (binary) code from one machine to another at run-time. This process of low-level re-engineering consists of a reverse engineering phase followed by a forward engineering phase.

UQDBT, the University of Queensland Dynamic Binary Translator, is a machine-adaptable translator. Adaptability is provided through the specification of properties of machines and their instruction sets, allowing the support of different pairs of source and target machines. Most binary translators are closely bound to a pair of machines, making analyses and code hard to reuse.

Like most virtual machines, UQDBT performs generic optimizations that apply to a variety of machines. Frequently executed code is translated to native code by the use of edge weight instrumentation, which makes UQDBT converge more quickly than systems based on instruction speculation.

In this paper, we describe the architecture and run-time feedback optimizations performed by the UQDBT system, and provide results obtained in the x86 and SPARC® platforms.
© 2005 Published by Elsevier B.V.

Keywords: Dynamic compilation; Run-time profiling; Dynamic execution; Binary translation; Reverse engineering; Re-engineering; Virtual machine

1. Introduction

The term binary translation is used to describe a technique that allows software compiled for one machine to be converted to run on another machine while achieving reasonable performance on that machine. In the 1980s, the newer RISC machines were replacing the legacy CISC machines. In an effort to support programs that were written for the now outdated machines, manufacturers looked for solutions for marketing their new RISC platforms. Originally, legacy programs were supported on the new machine through a process of emulating the old hardware on the new one, but emulation was slow, even on the newer and faster RISC machines. Subsequently, hardware manufacturers introduced binary translation in order to provide a better migration path for running legacy programs. Over the years, binary translation techniques have been used to translate competitors' applications to the desired hardware platform, as well as to continue to provide a migration path from a legacy hardware environment to a newer one.

---

* Corresponding author.
  E-mail addresses: davidu@gcc.gnu.org (D. Ung), cristina.cifuentes@sun.com (C. Cifuentes).
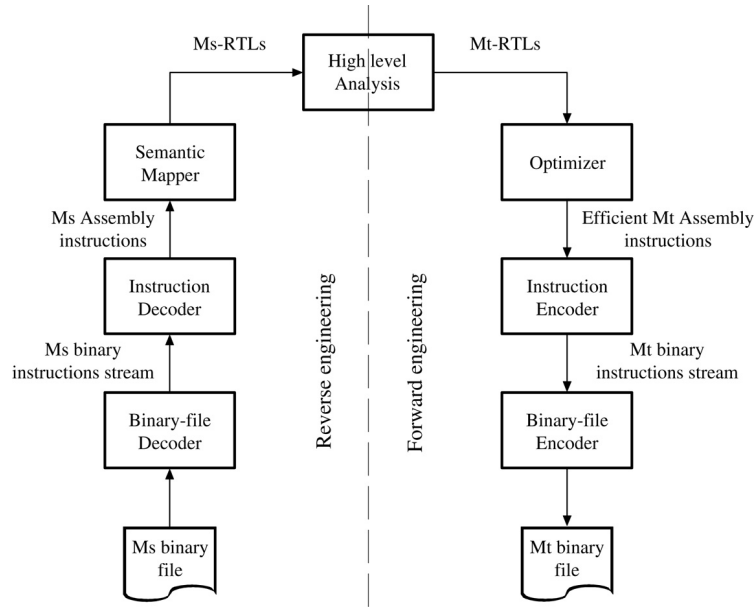
Fig. 1. UQBT static framework.

Binary translation is a process of low level re-engineering, which consists of a reverse engineering phase followed by a forward engineering phase. The reverse engineering phase decodes the source machine binary instructions to a higher level of abstraction, and the forward engineering phase encodes the abstraction into another binary form. The reverse engineering phase deals with the analysis of binary code, and has to unravel the program's semantics. The binary translation process can be performed in a static (i.e., a priori, without running the program being translated), or a dynamic (i.e., at run-time, while running the program being translated) way.

Fig. 1 shows the translation phases and components of the University of Queensland Binary Translator (UQBT) *static* translation framework [4,5]. Given a source machine (Ms) and a target machine (Mt) specification, the UQBT framework can be instantiated for this pair of machines to create a translator from machine Ms to machine Mt, assuming the same operating system. The reverse engineering phase recovers the semantic meaning of machine instructions by a three-step process: loading the binary file, decoding the machine instructions of the code segment, and mapping these instructions to their semantic meaning in the form of register transfer lists (RTLs). The semantic RTL transition from a source machine (Ms-RTL) to a target machine (Mt-RTL) is handled by the high-level analysis phase, which lifts the Ms-RTL representation to a machine-independent representation called HRTL, and then generates RTL for the target machine (Mt-RTL). The last phase, a forward engineering phase of code generation and binary file assembly, performs equivalent functions to a compiler backend: optimizing the code, encoding into machine instructions, and assembling program code and data to a binary file.

In static binary translation, the code is translated off-line, in a similar way to a compiler for most third generation languages such as Pascal, C and C++. The source program is used to create a new program (called a translation) 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 analyze statically, as the value of the target address is dependent on the contents of a register (which gets changed at run-time in the program). Therefore, a fall-back mechanism is commonly used with a statically translated program, in the form of a run-time interpreter. The interpreter handles any un-translated code at run-time and returns to translated code whenever it is safe to do so. The delay introduced by the interpreter may be noticeable/significant to the user.

Dynamic binary translation can overcome the limitations of static translation, but may come at the expense of some performance. A dynamic binary translator translates code "on the fly" at run-time, while the user perceives the ordinary execution of the program on the target machine. Different from emulation or interpretation, dynamic translation is performed in three stages: interpreting code, translating frequently interpreted code (thereby generating
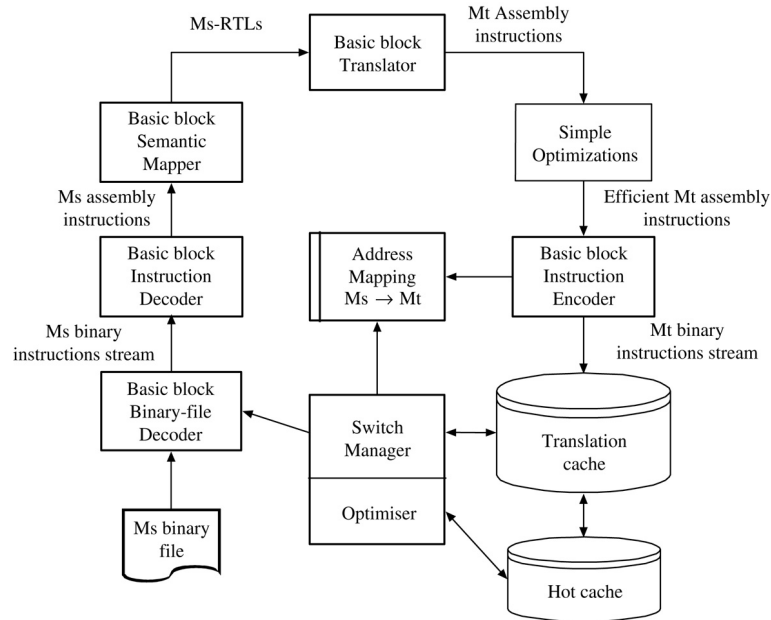
Fig. 2. Dynamic binary translation framework.

native code), and executing the generated code. Since actual translation is carried out as part of program execution, the amount of effort that can be spent analyzing the source program is now restricted such that it does not jeopardize system response time. Sluggish system response is more apparent and is harder to deal with for a dynamic system. A balance between translation cost and code quality needs particular attention. As such, the full set of classical optimizations used by compilers is restricted in dynamic binary translation. However, some optimizations that are not possible statically are possible dynamically. For example, optimizing variables that remain the same throughout some part of the program after its initial assignment. A dynamic translator may perform on-demand optimizations during code execution, where frequently executed code pieces are optimized at run-time to increase the executing performance of such code.

In this paper we describe the architecture and optimizations performed by UQDBT [21]—a *machine-adaptable* dynamic binary translator, which is based on the static UQBT framework. UQDBT can support different source and/or target machines through the specification of properties of these machines and their instruction sets. Unlike other dynamic systems such as Dynamo [2] and Wiggins/Redstone [7], which are closely bound to the underlying machine, optimizations performed in UQDBT are generic and can be applied to other machines. Also, the identification of frequently executed code converges more quickly in UQDBT using edge weight instrumentation than in systems that are based on instruction speculation (sampling by frequent interrupts to identify frequent values of program counter).

This paper is structured in the following way. Section 2 discusses the UQDBT framework for dynamic binary translation and the support of machine adaptability. Section 3 describes the optimizations that are performed in UQDBT. Section 4 discusses some of the work related to the area of binary translation and conclusions are given in Section 5.

2. The UQDBT framework

Fig. 2 illustrates the UQDBT framework, which uses a granularity of a basic block as the unit of translation. The left-hand side of Fig. 2 is similar to that of a static translator (see Fig. 1), only that 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. Simple optimizations that can be done quickly may be performed. Register caching at the basic block level is the only optimization performed by UQDBT. The translated binary is run directly from 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 is stored in a map. If a particular set of basic blocks is executed repeatedly, when the number of executions

reaches a threshold, optimizations on the code are performed dynamically to generate better code; this is commonly referred to as a hot path. The system invokes the *optimizer* to determine the set of basic blocks that contributes to this hot path, thus creating a specialized version of the code in the hot cache. Different levels of optimization are possible depending on the number of times the code is executed.

The *switch manager* drives the processing of basic blocks. It determines whether a new translation needs to be performed by checking 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 the 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.

2.1. Machine-adaptability

Machine-adaptability is achieved in the system by providing a clean separation of concerns; allowing machine-dependent information to be specified, and performing machine-independent analyses. Unlike existing translators, the UQDBT translator is not bound to any hardware, and is capable of supporting a variety of source and target machines. UQDBT supports a variety of CISC and RISC machines at low cost through reuse of the framework. New machines can be supported in the framework by writing their specifications and machine-specific modules (if not already part of UQDBT).

To provide for machine adaptability, extensions were made to Fig. 2 that enable UQDBT to easily adapt to different source and target machines. The machine-dependent parts of the system were isolated and identified in the form of specifications. In this way, a developer is able to concentrate on writing descriptions of properties of machines instead of having to (re)write the tool itself. These machine-dependent specifications can generate parts of the system automatically, which are integrated into UQDBT.

Machine-dependent information applies to three different levels of the system during decoding and encoding. These are:

1. binary-file format of the executable program,
2. syntax of the processor's machine instructions, and
3. semantics of the processor's machine instructions.

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 [20] and provides automatic generation of a loader that can decode binary files in that format.
- SLED: the specification language for encoding and decoding [17] supports the description of the syntax of machine instructions. SLED is supported by the New Jersey machine-code toolkit [16], which provides partial support for automatically generating an instruction decoder and its components.
- SSL: the semantic specification language [3] allows for the description of the semantics of machine instructions. It provides the assembly translation between RTLs and binary instructions.

Differences in machine calling conventions, namely how function parameters are passed and where return values are stored, affect how the translator generates the correct setup when calling native library functions. RISC machines generally pass parameters in registers while CISC machines push them on the stack. This information is used for both source and target machines to identify the transformation of parameters and return values.

As an example, Fig. 3 illustrates 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.

3. Architecture and optimizations

In a dynamic system, optimization is done at run-time. Although optimizations can be applied as early as generating code at translation time (the simple optimization phase of UQDBT), the more significant improvements come from

```
              v[100] := v[98] + v[99]


  r[1] := r[0] + r[1]          r[9] := r[10] + r[9]


  addl   %ebx, %eax             add   %o1, %o2, %o1


  0000 0010 1101 1000          1001 0010 0000 0010
                               0100 0000 0000 1010
```
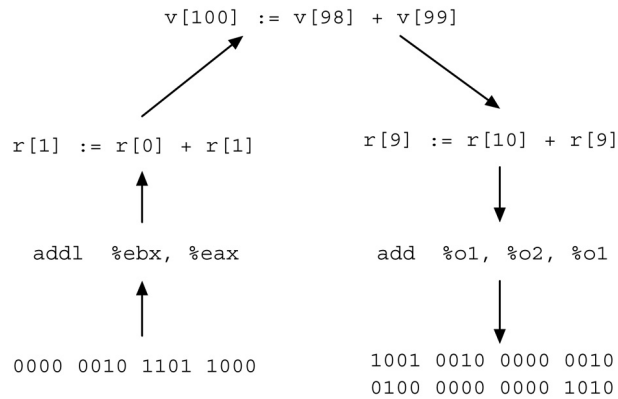
Fig. 3. Example re-engineering process: Pentium to SPARC translation.

```
08048918      pushl   %ebp
08048919      movl    %esp,%ebp
0804891b      subl    $0x4,%esp
0804891e      movl    $0x0,0xfffffffc(%ebp)
08048925      cmpl    $0x63,0xfffffffc(%ebp)
08048929      jle     08048930
0804892b      jmp     08048948
0804892d      nop
0804892e      nop
0804892f      nop
08048930      movl    0xfffffffc(%ebp),%eax
08048933      pushl   %eax
08048934      pushl   $0x8049418
08048939      call    080487c0 <printf>
0804893e      addl    $0x8,%esp
08048941      incl    0xfffffffc(%ebp)
08048944      jmp     08048925
08048946      nop
08048947      nop
08048948      xorl    %eax,%eax
0804894a      jmp     0804894c
0804894c      leave
0804894d      ret
```

Fig. 4. Assembly instructions of an x86 example.

optimizing frequently executed pieces of code. This is usually found by examining the program's execution behavior and picking out the most relevant code areas for optimizations to achieve the best performance. In UQDBT, few optimizations apart from register caching are done during the initial code generation in the translation phase. More aggressive optimizations are only done to pieces of code that are found to be executing very frequently. As most programs spend a significant amount of their execution time in a small region of code, identifying and optimizing that region of code is almost certain to boost the performance of the program. Rather than trying to optimize every piece of code that the translation comes across (as in a static system), recognizing what to optimize is an important factor to speed boost in a dynamic system.

In this section, we will look at some of the optimizations performed by UQDBT when translating a small piece of Intel x86 binary to run on a Sun SPARC machine. Fig. 4 shows a snippet of a function in x86 assembly code that loops 100 times and calls the library function *printf*. In the x86 assembly code, the destination register is on the right-hand side of an instruction's operand list. Addresses in boldface denote the start of a basic block: 8 basic blocks form part of this snippet of code, two of which (0x0804892d and 0x08048946) are NOP (do nothing) basic blocks. Fig. 5 shows the control flow graph of the resulting SPARC translation, which the dynamic translator generates from Fig. 4. Each rectangle in Fig. 5 represents a basic block (a sequence of instructions that ends with a control flow instruction) of translated SPARC code, while the arrows represent the flow of control between the basic blocks. A branch into the middle of an already translated basic block creates a new basic block starting at that point, duplicating some of the code. During program execution, the translator generates only blocks that are needed at run-time, hence, the NOP
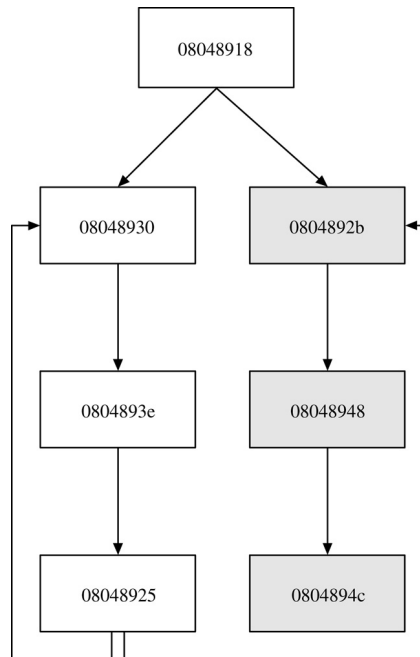
Fig. 5. Control flow graph for example in Fig. 4.

basic blocks are not included. While the execution is looping within the above code, the program spends most of its time within the generated blocks (white blocks in Fig. 5). When the program exits the loop, the rest of the code is then generated (represented by shaded blocks).

3.1. Instrumentation

Triggering of optimizations happens when a hot sequence (a collection of code that executes frequently in sequence at run-time) is found. As most programs spend the majority of their execution time in a localized region of code, identifying it can significantly improve system performance. In UQDBT, this triggering of hot sequence identification is done through the instrumentation and profiling of the program's execution path. Two possible profiling techniques could be used: node weight or edge weight profiling. Node weight profiling can capture the frequency of a node (a basic block) being executed, while edge weight profiling captures the executing frequency of the path taken between two adjacent nodes. We use edge weight profiling to capture the relationship between the basic blocks and record the frequency of branch and jump targets. Each edge weight contains an address of the source and destination basic blocks, plus a frequency counter that is initially set to zero. At run-time, each execution of an edge increments that edge weight's frequency counter. Fig. 6 shows the instrumentation code added to the generated translation for a two-way node—a basic block with two out-edges (i.e., an if–then basic block). In a two-way node, if the conditional jump evaluates to true, the branch is taken, otherwise, the code falls through to the next basic block. Code stubs are created for each edge leaving from the translated code, in order to add instrumentation and linking code to the corresponding basic block once it becomes available. Initially, the jump instruction at the end of the code stub points to the *switch manager*. This jump instruction is later patched by the run-time system when the program executes this path, hence triggering the translation of the appropriate code.

The *optimizer* is invoked by the stub code when a particular edge frequency counter reaches a predefined threshold. At this point, the *optimizer* is called to determine the set of flow edges that contributes to the execution hot path. The set of flow edges found by the algorithm is a set of relationships between basic blocks in the program that are being executed frequently. This set is then used to guide the *optimizer* to generate optimized translations to the hot cache. The hot path finding algorithm is an iterative traversal of flow edges for inclusion, as follows:

```
find_hot_paths (set_of_edges, edge)
{
```
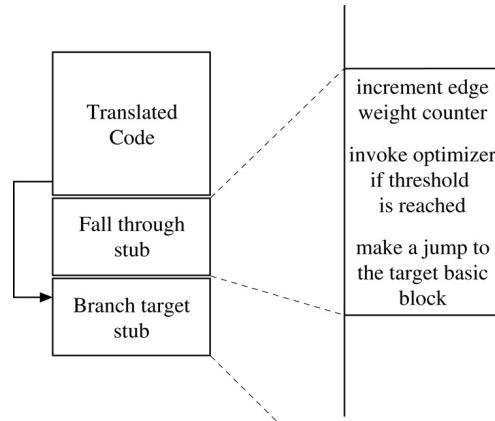
Fig. 6. Instrumentation of a two-way basic block.

```
   if ( edge in set_of_edges ) then return;
   currentBB = edge.destinationBB;
   for ( all out-edges E of currentBB ) do {
       if ( E.weight > path-inclusion-threshold ) then {
          add E to set_of_edges;
          find_hot_paths (set_of_edges, E);
       }
   }
}
```

The *optimizer* calls find_hot_paths starting with an empty set and the edge weight that triggered the optimization. The algorithm examines each out-edge of the current basic block and only follows paths that are potentially hot (i.e., paths within the path inclusion threshold—a high percentage value of the counter threshold). Fig. 7 shows the state of the system when the counter threshold for optimization is set to 50 and the path-inclusion-threshold is set to 40% of the previous edge weight. Each edge in the figure is annotated with its frequency counter. The initial edge 0 8 0 4 8 9 3 0 to 0 8 0 4 8 9 3 e will trigger the optimization to take place. Two other edges, 0 8 0 4 8 9 3 e to 0 8 0 4 8 9 2 5 and 0 8 0 4 8 9 2 5 to 0 8 0 4 8 9 3 0 (both with a frequency counter of 49) are added to the hot path.

3.2. Improving code locality

The discovery of hot paths not only allows the *optimizer* to identify which section of the program is most beneficial for optimization; it can also improve the locality of the translated code through moving and merging of basic blocks. By moving the frequently executed basic blocks closer together, system cache is improved. Fig. 8A shows the code initially generated by the translator. Each generated basic block occupies a space of its own in memory, and lies some distance from the others. By moving those basic blocks together, the locality of the basic block translation is improved, and hence better cache utilization is achieved. An added bonus to having localized basic blocks together is that there is the opportunity for reducing the flow of control between them. This can be achieved by merging the basic blocks together, thus eliminating unnecessary intermediate control flow instructions. Fig. 8B shows the result after moving and merging the basic blocks together.

After the *optimizer* determines the hot path, each basic block is re-translated and put into the hot cache. In the example, the basic blocks 0 8 0 4 8 9 3 0, 0 8 0 4 8 9 3 e and 0 8 0 4 8 9 2 5 are re-translated. Re-translation is necessary as the *optimizer* uses the information gathered for the program's behavior to create new translations that better reflect actual program execution, thus improving performance. An interesting problem arises when a large number of basic blocks can be merged or moved. We know that the set of hot path blocks can be calculated from the algorithm described in the previous section, but how can they be efficiently placed in the hot cache? What are the consequences of moving blocks A and B closer to block C as it may now be further apart? Fortunately, the edge information provides some clues as to what the layout should be, and allows the translator to produce an interim placement scheme that may be close to optimal placement.
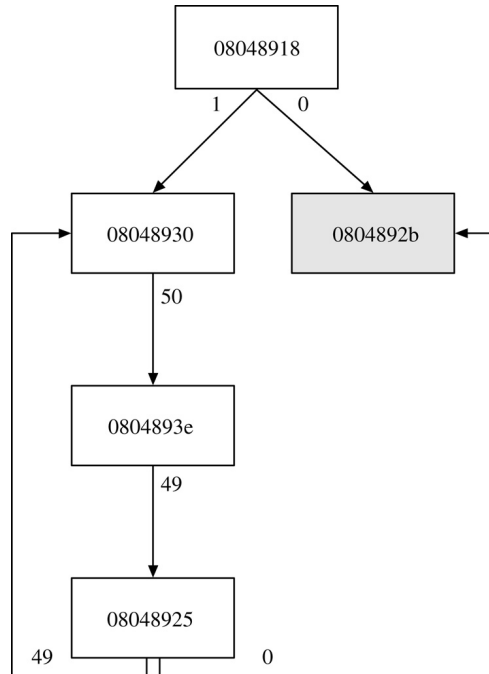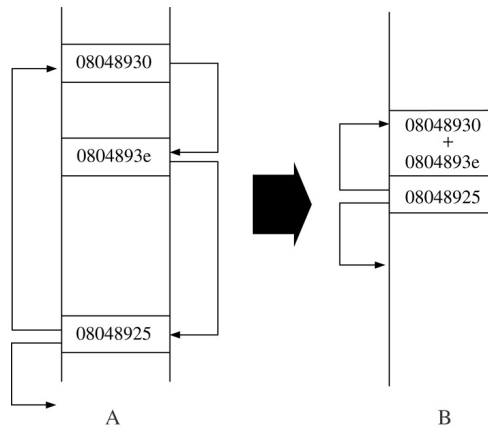
Fig. 7. Control flow graph with edge weights.

Fig. 8. Moving and merging of frequently executed basic blocks to improve code locality.

3.2.1. Block re-ordering

A simple example demonstrating the hot cache placement problem can be seen from Fig. 9. The graph of blocks in the figure is simplified to only show out-edges that are executed at run-time. For example basic block B may jump to block D and J, but since block J is never executed it will not be translated and is removed from the figure to simplify display (this conforms to the definition of basic blocks which ends with a change of flow control). In the figure, if the path-inclusion-threshold for hot path selection is (30% * edge.weight), then all the blocks from A to I are part of a hot path with the edge I–A being the edge that triggered the call to find_hot_path. The edge information allows for an easy identification of the hotter trace within the loop which is IABDEGI, but the rest of the blocks; C, F, and H; do not have a suitable scheme for placement and their order is therefore chosen at random. There are 6 possible permutations for placement of blocks C, F, and H: IABDEG:CFH, IABDEG:CHF, IABDEG:HCF, IABDEG:HFC, IABDEG:FCH, and IABDEG:FHC. One simple placement scheme is to pick the edge with the next highest weight, which in the example gives the placement of IABDEGFCH. It can clearly be seen that this is not the best solution. We can modify the
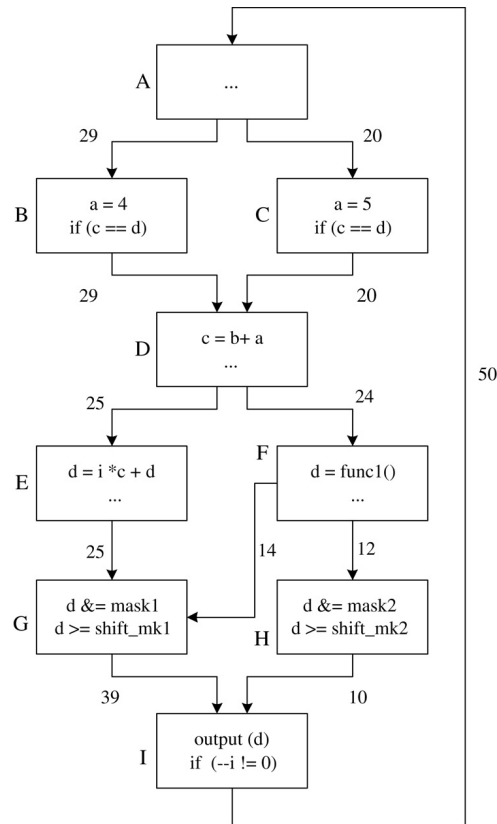
Fig. 9. Trace placement example.

scheme to select the next highest weight and follow through all of its out-edges until exhaustion, before re-selecting the next highest weight in the remaining edges; this yields the placement IABDEGFHC. Unfortunately, this does not guarantee to be optimal as block C is now at the end of the list and edge AC may now be further apart; a simple branch cannot be used to link the blocks. Trampoline code may be used as a work around to solve such cases. The example in Fig. 9 only contains 3 loose nodes after the main trace is identified, whereas often there are many more nodes that remain loose after removing the critical trace nodes. The large number of permutations do not allow the translator to select the best placement in reasonable time. As such, heuristics such as the one above are used to select a "good" placement in lieu of computational cost.

3.2.2. Re-ordering via duplication

An alternative to the above scheme is to duplicate some blocks on the critical traces; the duplicated blocks reduce the internal branching among the hot traces, thus maintaining the contiguous trace properties. With full duplication, each possible path becomes an execution trace of the program, making the trace carry some run-time characteristics of its execution flow, which often creates opportunities for customized optimizations. In Fig. 9, by duplicating block D, the values of 'a' can be constant propagated into the edges BD and CD among the traces IABDEG:FH:CD. As seen, the benefit of duplication allows for more optimized versions of the trace to be placed in the hot cache, while the locality of block C and D is improved by removing the long branch in the original placement. The benefits of value and flow optimization in traces is not to be disregarded, however, these optimizations are outside the scope of this paper. In order to determine which block needs duplication is difficult, which again involves selecting from a large number of placement permutations. The increase in code size is also an important parameter to be considered when balancing between code expansion and possible performance gain. Nevertheless, the combination of using a good selection scheme as mentioned in the last section and some block duplication effectively captures the placement for the most executed critical traces.
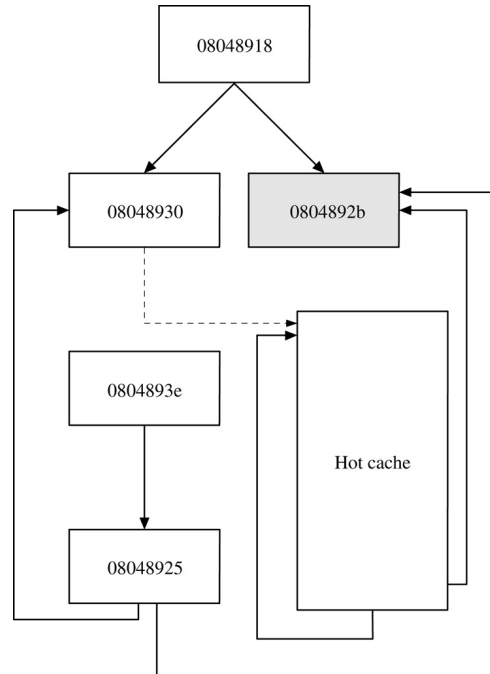
Fig. 10. Generation of hot cache code.

3.2.3. Linking existing translations to the hot cache

The hot cache itself is a special version of a basic block in that it allows control flow exits from anywhere within the block instead of only at the end of the basic block. The block containing the triggering edge is modified such that program execution is redirected to the beginning of the hot cache. All other blocks remain unchanged, hence code that branches to those blocks is unaffected. The hot cache is a specialized version of the execution path consisting of the customized and merged basic blocks. Fig. 10 shows the state of the translated code after generation of a hot cache. The original block 08048930 is patched with a jump to the hot cache. The edge 08048925 to 08048930 is represented by the last backward jump at the end of the hot cache.

3.3. Path prediction feedback

During instrumentation and profiling, the system may collect execution profiles of frequently taken branches. The *optimizer* may then use the information gathered during the program run to modify certain branch instructions such that the code locality can be further improved through path prediction. For example, assume the following piece of SPARC assembly branch code:

```
               ld    %sp+56, %o0
               cmp   %o0, 100
               ble   LabelA
               nop
fallthrough:
               add   %o2, 1, %o2
LabelA:
               jmp   LabelB
```

If during program execution the branch less and equal (`ble`) is taken the majority of the time, it is worthwhile inverting the branch instruction to make LabelA the fall through code instead, hence moving more frequently executed code closer together. Feedback of information is provided by UQDBT instrumentation, which allows the *optimizer* to make the necessary modifications. Target machine instructions that provide hints to the processor about branch direction can be utilized to decrease the chance of branch mis-prediction. Execution time spent within the hot cache can improve up to 20% over the original scattered translations.

Fig. 11 shows the translated SPARC assembly code for the x86 example in Fig. 4, as written to the hot cache by the translator. The hot path of Fig. 4, denoted by the basic blocks starting at addresses 0x08048930, 0x0804893e and 0x08048925 is translated into the code in Fig. 11, denoted by the basic block addresses 0x4308ed8, 0x4308f24, and 0x4308fe4. Fig. 11 provides some annotations of the translation, to see the relationship to Fig. 4 and the location of the original x86 basic blocks. The translator caches x86 registers in the SPARC processor registers in the following way: global x86 registers such as %ebp and %esp are stored in global SPARC processor registers such as %g7 and %g4, and general purpose x86 registers such as %eax, are stored in local SPARC processor registers, such as %l1.

The amount of code in Fig. 11 is voluminous compared to that in Fig. 4 because all states of the original x86 instructions need to be saved. In particular, two calling conventions are being used: the x86 one (where parameters are pushed to the stack), and the native SPARC calling convention (where parameters are passed on the output %o registers). In Fig. 11, we can see that any x86 pushes onto the stack are "emulated" by pushing onto a local stack area. However, given that the library routine invoked, printf, is a native library using the SPARC calling conventions, the parameters to that routine need to be passed on the %o registers. The code also shows the need to keep track of condition codes and to update them as needed.

3.4. Optimization opportunities

The moving and merging of smaller basic blocks into the hot cache may provide further optimization opportunities. Similar to the effect of function in-lining, a hot path sequence increases the translation granularity with information that can be propagated along blocks within the hot path. Additional information collected from instrumentation profiles provides a customized guide for code placement and specialization. Aggressive optimizations are justified for the benefits gained from better code quality, which would eventually overcome the extra cost of re-translation. Register allocation policies can also use run-time profiles to favor highly executed paths and reduce the register spilling to colder paths.

During the initial translation process (before the *optimizer* is called), extra care must be taken to ensure that the control flow of the translation is maintained such that transition states from one block to another reflect that of the original source program. Extra housekeeping is necessary to ensure that the value of virtual registers holds the same state as the original program at each exit of a basic block. This safe state is critical because the next block address may not be translated and could potentially call other components of the system like the *switch manager*. Code within the hot cache is less restricted as long as the safe state can be reached when leaving the hot cache. The basic blocks that spread across the hot cache are part of the global structure instead of individual block translations. This leads to better utilization of program control flow during translation and opens up the next stage of optimizations, which can be perceived as inter-basic block rather than intra-basic block optimizations. With a larger set of data that is gathered within these basic blocks, the register caching strategy is vastly improved as more code is revealed through path feedback.

While all these optimizations are happening within the hot cache, branch exits out of the hot cache still need to link back to previous translations that are less frequently executed. Since the hot cache contains only the most frequently executed regions, any infrequently executed code and any housekeeping code is to be moved outside of the hot cache. UQDBT implements this by creating a stub block that links between the hot cache and any other blocks coming out of it. The stub block may contain any housekeeping code and instrumentation code for triggering re-optimization, for the case when an exit from the hot cache becomes hot. The actual linking of blocks to other hot paths are delayed until the system can determine that the link is beneficial, hence reducing work from each exit code stub.

3.5. Endianness

Translating cross platform can be expensive if the source and target machines are of different endianness. There are two main camps of memory byte ordering that are used by present machine architectures: little-endian byte ordering (e.g. Intel IA32, PDP-11) and big-endian byte ordering (e.g. SPARC, PA-RISC). The value 0xaabbccdd in big-endian order is seen as the value 0xddccbbaa if read by a little-endian order machine. As such, adjustment is necessary in order to correctly process data that is different to the native (target) machine byte ordering. Cross-endian translation can be achieved by either modifying the effective addresses of memory accessing instructions, or by converting the result from stores and loads to the native target machine endian ordering. The former technique is known as address swizzling, and the latter can be achieved by byte swapping.

```
0x4308ed8:     mov  %g7, %l0                    # translation for 0x08048930
0x4308edc:     addcc %l0, -4, %l1              # %g7 == %ebp, %g4 == %esp
0x4308ee0:     ld  [ %l1 ], %l2
0x4308ee4:     mov  %l2, %o2                    # pushl %eax
0x4308ee8:     sub  %g4, 4, %g4
0x4308eec:     st  %o2, [ %g4 ]
0x4308ef0:     sethi %hi(0x8049400), %l0        # pushl $0x8049418
0x4308ef4:     add  %l0, 0x18, %l0
0x4308ef8:     sub  %g4, 4, %g4
0x4308efc:     st  %l0, [ %g4 ]
0x4308f00:     sethi %hi(0x442b000), %l0
0x4308f04:     or  %l0, 0x20, %l0
0x4308f08:     st  %o2, [ %l0 ]
0x4308f0c:     ld  [ %g4 ], %o0                 # call setup, parameters in
0x4308f10:     ld  [ %g4 + 4 ], %o1             # %o0, %o1 and %o2
0x4308f14:     ld  [ %g4 + 8 ], %o2
0x4308f18:     sethi %hi(0xef663400), %g6
0x4308f1c:     call  %g6 + 0x2d8 <printf>       # call printf
0x4308f20:     nop
0x4308f24:     nop
0x4308f28:     mov  %g4, %l0                    # return value handling
0x4308f2c:     sethi %hi(0x442b000), %l1
0x4308f30:     or  %l1, 0x60, %l1
0x4308f34:     st  %l0, [ %l1 ]
0x4308f38:     mov  %g4, %l0                    # translation for 0x0804893e
0x4308f3c:     addcc %l0, 8, %l1               # addl $0x8,%esp
0x4308f40:     mov  %l1, %g4
0x4308f44:     sethi %hi(0xefffb400), %l0
0x4308f48:     add  %l0, 0x50, %l0
0x4308f4c:     rd  %ccr, %l1
0x4308f50:     st  %l1, [ %l0 ]                 # condition code save
0x4308f54:     mov  %g7, %l0
0x4308f58:     addcc %l0, -4, %l1
0x4308f5c:     ld  [ %l1 ], %l2
0x4308f60:     sethi %hi(0x442b000), %l3
0x4308f64:     or  %l3, 0x60, %l3
0x4308f68:     st  %l2, [ %l3 ]
0x4308f6c:     mov  %g7, %l0
0x4308f70:     addcc %l0, -4, %l1
0x4308f74:     mov  %g7, %l2
0x4308f78:     addcc %l2, -4, %l3
0x4308f7c:     ld  [ %l3 ], %l4
0x4308f80:     addcc %l4, 1, %l5               # incl 0xfffffffc(%ebp)
0x4308f84:     st  %l5, [ %l1 ]
0x4308f88:     sethi %hi(0xefffb400), %l0
0x4308f8c:     add  %l0, 0x50, %l0
0x4308f90:     rd  %ccr, %l1
0x4308f94:     st  %l1, [ %l0 ]
0x4308f98:     nop
0x4308f9c:     nop
0x4308fa0:     mov  %g7, %l0                    # translation for 0x08048925
0x4308fa4:     addcc %l0, -4, %l1
0x4308fa8:     ld  [ %l1 ], %l2
0x4308fac:     subcc %l2, 0x63, %l3            # cmpl $0x63,0xfffffffc(%ebp)
0x4308fb0:     sethi %hi(0x442b000), %l4
0x4308fb4:     or  %l4, 0x80, %l4
0x4308fb8:     st  %l3, [ %l4 ]
0x4308fbc:     sethi %hi(0xefffb400), %l0
0x4308fc0:     add  %l0, 0x50, %l0
0x4308fc4:     rd  %ccr, %l1
0x4308fc8:     st  %l1, [ %l0 ]
0x4308fcc:     sethi %hi(0xefffb400), %l0
0x4308fd0:     add  %l0, 0x50, %l0
0x4308fd4:     ld  [ %l0 ], %l1
0x4308fd8:     wr  %l1, 0, %ccr
0x4308fdc:     bg  0x442f120                    # jle 08048930
0x4308fe0:     nop
0x4308fe4:     ba  0x4308ed8
0x4308fe8:     nop
```

Fig. 11. Translated SPARC assembly code in the hot cache for the example code in Fig. 4.

3.5.1. Address swizzling

In address swizzling, the original effective address for the memory access is adjusted to correct the "reversed" data values. Sometimes it is desirable to have memory organised in a different endian order to the native target machine's order. For example, when running x86 Microsoft Word on a big-endian machine. Since Word expects the data to be in

little-endian order, and without altering the physical data layout , data accesses will require modifiying those addresses and offsets. If address 0x1000 contains a 64-bit value stored in little-endian format, then the native big-endian machine will adjust its memory access by the following formula:

$$\text{swizzled address} = \text{HighWMark} - (\text{SIZE} + \text{EA} - \text{LowWMark})$$

where:

- LowWMark to HighWMark are the memory address ranges for which data is of a different endian order. LowWMark holds the lowest address of the range and HighWMark holds the highest address. LowWMark = 0x1000 and HighWMark = 0x1008 for the example.
- SIZE is the number of bytes the memory instruction is fetching/writing.
- EA is the effective address requested by the original program.

For example, if the original source instruction loads 16 bits from 0x1002 (i.e., 2 bytes), the swizzled address will be 0x1008 − (2 + 0x1002 − 0x1000), which is 0x1004. Address swizzling is used by Apple to retrieve data from PCI devices which have different byte ordering.

3.5.2. Byte swapping

The second option to cross-endian translation is to simply byte swap every read and write access to and from memory. Below is an example implementation of a 32-bit byte swap function:

```
unsigned int ByteSwap (unsigned int x)
{
    unsigned char b0, b1, b2, b3;
    b0 = (x & 0x000000FF) >> 0 ;
    b1 = (x & 0x0000FF00) >> 8 ;
    b2 = (x & 0x00FF0000) >> 16 ;
    b3 = (x & 0xFF000000) >> 24 ;
    return (b0 << 24) | (b1 << 16) | (b2 << 8) | (b3 << 0) ;
}
```

Although semantically correct and safe to do, calling the byte swap function each time a read or write operation occurs is expensive. As such, processors may provide special features that can ease the byte swapping process. Intel's 80486 and above processors provide a bswap instruction for cross-endian data handling, while SPARC-V9 [23] and PowerPC [14] processors have special load/store instructions that can access memory using a different endian ordering, which entirely avoids any byte swapping. However, issuing byte swap instructions is still a significant overhead, and even more so for machines that do not have the hardware support.

Some simple bit-wise/logical simplification and load–store aliasing can greatly reduce the number of byte swaps. Byte swapping can be avoided for load–store pairs that only define registers that are dead after the store instruction. For example, by removing byte swapping for memory copying functions that load data from one region of memory and write to another. Store–load paring can be applied similarly. This method requires tagging of byte ordering information to the virtual registers so that the code generator may apply lazy byte swapping [12].

Bit-wise operations involving byte swap can also be simplified. The following show that simplifications can be applied on operations involving exclusive or:

Operation	Simplification
ByteSwap (exprA) xor imm	ByteSwap (exprA xor ByteSwap (imm))
Imm xor ByteSwap (exprA)	ByteSwap (ByteSwap (imm) xor exprA)
ByteSwap (exprA) xor ByteSwap (exprB)	ByteSwap (exprA xor exprB)

The first two simplification rules appear to be more complex than the original expression, however, the ByteSwap(imm) can be evaluated at code generation time and may be folded by exprA (for example, if exprA = reg xor imm). The third rule simplifies the sub-expression in exprA with that in exprB, without being constrained by the ByteSwap(). Other bit-wise operations such as "bit-wise or" and "bit-wise and" follow the same simplification rules as above. The same rules apply to logical operations as well.

3.6. Re-optimization

The execution behavior of a program can change from one state to another. A program can take a particular path for a period of time, and then choose a different path later on. Hence, the optimization strategy needs to be able to cope with the changes of behavior that a program may exhibit. Therefore, finding only one hot path may not be optimal in many cases. To fully utilize the translation system, there is a need to identify these states and to add the new hot paths to the existing hot cache. For example, program execution leaves the hot cache and decides to spend time in un-optimized blocks. When those blocks become "hot", a new hot path is generated and is added to the existing hot cache.

3.7. Performance and overhead

The current implementation of UQDBT can translate programs compiled for Intel x86 or Sun SPARC processors, running under the Solaris[TM]Operating Systems (OS). By using the same operating system, we are able to concentrate on machine instruction translation. As a result, calls to library functions are not translated, but are replaced by their corresponding target (native) machine calls.

Instrumentation code costs up to 13 SPARC (10 for x86) instructions for each control flow stub in a basic block, though only half of these are usually executed at run-time. This incurs a 5%–20% execution overhead when running un-optimized code with profiling turned on. Code in the hot cache is not instrumented, hence no overhead is incurred once the code is optimized (which is the most frequently executed code).

Programs that are optimized using edge weight profiles run up to 15% faster than non-profiled translations (which run 2 to 6 times slower than native programs compiled on the target machine). Programs that do not optimize well simply incur the cost of instrumentation in un-optimized code, but not in hot cache code. This ensures that frequently executed code does not get penalized. These results are based on small/medium-sized programs that are less than 100 KB in size, as seen in the following tables for Pentium to SPARC processor and SPARC to Pentium processor translations.

Test programs	Startup time	Translation time	Execution time without register caching	Execution time---with register caching	Execution time---with profile info	Natively gcc compiled
Sieve3000	0.29	0.06	56.4	39.4	34.9	17.4
Fibonacci	0.29	0.03	94.7	80.8	79.7	25.1
Mbanner	0.28	0.19	108.9	72.5	65.1	13.7
UQDBTps---Pentium to SPARC translation (seconds)						

Test programs	Startup time	Translation time	Execution time without register caching	Execution time---with register caching	Execution time---with profile info	Natively gcc compiled
Sieve3000	0.45	0.16	89.1	77.4	76.5	25.6
Fibonacci	0.48	0.14	223.2	198.6	189.4	49.2
Mbanner	0.41	0.64	197.3	178.3	171.7	23.4
UQDBTsp---SPARC to Pentium translation (seconds)						

3.8. Future work

One of the major difficulties with dynamic translation from one machine to another is the effect of processor condition codes. In CISC machines like x86, many instructions set the condition codes as a side affect. Some of these condition codes have no effect in subsequent instructions and may be overwritten by the later instructions. When translating to another machine, it is desirable to eliminate any unnecessary simulation of the condition code from the original machine, especially when generating code for the hot cache. Determining whether the effects of a condition code can be removed involves examining the code to see if there are any uses that follow a definition. This process is difficult to complete in dynamic systems as it can involve analyzing across basic block boundaries, some of which

may not even have been generated at the time of the analysis. Although full removal of condition codes is impractical dynamically, it is possible to provide partial removal to provide some benefit to the system. It is a question of how deep the analysis process can go before it decides to be conservative and how to restore the original states of the condition code when exiting from the optimized code cache. To ensure program correctness, the current implementation of UQDBT conservatively emulates all side effects of condition codes on the target machine. Although most of the condition code assignments are killed by subsequent instructions or from instructions in the next block, the hot cache is penalized to maintain condition code information near exit points. Even at exit points, the dependences of condition codes restricts the effectiveness of optimization for the hot path.

4. Related work

The early binary translators were static in nature, such as Digital's VEST and mx [19], which translate VAX and MIPS machine instructions to 64-bit Alpha instructions. These translators and others like Apple's MAE [18] and Digital's Freeport Express [9] require a run-time environment that reproduces the source machine's operating environments.

In recent years, technology gave way to the new breed of hybrid translators like Executor by Ardi [12] and Sun's Wabi [8], which mix translation with emulation. Machine simulators such as Embra [24] are built using dynamic translation techniques that were developed in Shade [6]. Le [13] investigates out-of-order execution techniques in dynamic binary translators, though his results are based on an interpreter-based implementation. Other forms of systems that use dynamic translation to process non-native code are Just-in-time (JIT) compilers for the Java^TM programming language. JITs from Sun [10], Intel [1] and others dynamically generate native machine code at run-time.

Dynamic compilation is generally found in dynamic compilers such as SELF [22] and tcc [15]. Systems such as these either require the programmer to annotate special parts of the program for dynamic compilation at run-time, or the program must be written in a specialized language. In dynamic translation, the optimization process is transparent to the user.

Profiling the program's behavior during execution can assist in identifying hot paths for optimization. Digital's FX!32 [11] performs native optimizations from profile data that was collected during an initial run of the program through emulation. The program is then translated and optimized offline. The optimization process is static and its benefits are only available during later re-runs of the program. In UQDBT, the optimization is done at run-time, hence it is effective from the first run.

Dynamic optimizers like Dynamo [2] and Wiggins/Redstone [7] take a native binary program and optimize it on the same machine. In contrast, dynamic translators take code from one machine and translate it to a different machine. Dynamic optimizers must rely on the underlying hardware to provide efficient implementation of instruction profiling to identify hot paths. In UQDBT, instrumentation code is generated on the fly using edge weight profiling, which is hardware independent. It also converges more quickly when identifying hot paths than instruction speculation in dynamic optimizers.

5. Conclusion

Existing commercial binary translators cannot 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. The UQDBT dynamic binary translator framework supports different source and target machines through specifications of properties of those machines.

Optimizations performed in UQDBT are generic and can be applied to various different types of machines. The system does not rely on the underlying machine to provide support for profiling. Instead, UQDBT employs edge weight profiling by instrumenting translations to identify frequently executed paths. The main optimizations are concentrated within the hot cache by better utilization of caches and improving localization of hot paths. Despite the fact that some traditional optimizations are difficult to perform in a dynamic system, UQDBT provides a good generic dynamic binary translation framework which also performs generic optimization regardless of the underlying machine.

Acknowledgments

References

[1] A. Adl-Tabatabai, M. Cierniak, G.Y. Lueh, V.M. Parikh, J.M. Stichnoth, Fast, effective code generation in a just-in-time Java compiler, in: ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, PLDI 98, ACM Press, 1998, pp. 280–290.

[2] V. Bala, E. Duesterwald, S. Banerjia, Dynamo: a transparent dynamic optimization system, in: Proc. PLDI 2000, Programming Language Design and Implementation, ACM Press, 2000, pp. 1–12.

[3] C. Cifuentes, S. Sendall, Specifying the semantics of machine instructions, in: Proc International Workshop on Program Comprehension, IEEE CS Press, 1998, pp. 126–133.

[4] C. Cifuentes, M. Van Emmerik, UQBT: Adaptable binary translation at low cost, IEEE Computer 33 (3) (2000) 60–66.

[5] C. Cifuentes, M. Van Emmerik, D. Ung, D. Simon, T. Waddington, Preliminary experiences with the use of the UQBT binary translation framework, in: Proc. WBT99, Workshop on Binary Translation, Technical Committee on Computer Architecture News, IEEE CS Press, 1999, pp. 12–22.

[6] B. Cmelik, D. Keppel, Shade: a fast instruction-set simulated for execution profiling, in: Proc SIGMETRICS 94, Conference on the Measurement and Modeling of Computer Systems, ACM Press, 1994, pp. 128–137.

[7] D. Deaver, R. Gorton, N. Rubin, Wiggins/Redstone – An on-line program specializer, in: Proc. Hot Chips 11 - A Symposium on High performance chips, 1999.

[8] S. Fordin, S. Nolin, Wabi 2: Opening Windows, Prentice Hall PTR, 1996.

[9] R. Gorton, Digital Freeport Express V1.0. Archive news article in comp.compilers.
Available from http://compilers.iecc.com/comparch/article/96-01-052, 1996 (last accessed April 13, 2003).

[10] R. Griesemer, S. Mitrovic, A compiler for the Java HotSpot virtual machine, in: L. Boszormenyi, J. Gutknecht, G. Pomberger (Eds.), The School of Nicklaus Wirth: The Art of Simplicity, Morgan Kaufmann Publishers, Los Altos, California, 2000.

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

[12] M.J. Hostetter, C.T. Mathews, Executor Internals: How to Efficiently Run Mac Programs on PCs, http://www.ardi.com, 1996 (last accessed April 13, 2003).

[13] B.C. Le, An out-of-order execution technique for runtime binary translators, in: Proc ASPLOS 98, ACM SIGOPS Operating Systems Review 32 (5)(1998) 151–158.

[14] C. May, E. Shilha, R. Simpson, H. Warren, The PowerPC Architecture: A Specification for a New Family of RISC Processors, 2nd edition, Morgan Kaufmann Publishers, Inc., 1994.

[15] M. Poletto, D.R. Engler, M.F. Kaashoek, tcc: A system for fast, flexible, and high-level dynamic code generation, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 97, ACM Press, 1997, pp. 109–121.

[16] N. Ramsey, M. Fernández, The New Jersey machine-code toolkit, in: Proc. 1995 USENIX Technical Conference, USENIX, 1995, pp. 289–302.

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

[18] J. Seeger, Apples Mac-Emulator, Apfel im Netz, Macintosh Application Environment Version 2.0, 1995.

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

[20] D. Ung, C. Cifuentes, SRL – a simple retargetable loader, in: Proc ASWEC97, Australia Software Engineering Conference, IEEE CS Press, 1997, pp. 60–69.

[21] D. Ung, C. Cifuentes, Machine-adaptable dynamic binary translation, in: Proc. Workshop on Dynamic and Adaptive Compilation and Optimization, ACM Press, 2000, pp. 37–47.

[22] D. Ungar, R.B. Smith, SELF: The power of simplicity, in: Conference on Object-Oriented Programming Systems, Languages and Applications, ACM Press, 1987, pp. 227–241.

[23] D. Weaver, T. Germond (Eds.), The SPARC Architecture Manual – Version 9, Prentice Hall, Santa Clara, California, 1993.

[24] E. Witchel, M. Rosenblum, Embra: Fast and flexible machine simulation, in: Proc SIGMETRICS 96, Conference on Measurement and Modeling of Computer Systems, Philadelphia, 1996.