Characterization of DBT overhead

Conference Paper · November 2009

DOI: 10.1109/IISWC.2009.5306785 · Source: IEEE Xplore

Edson Borin
University of Campinas
120 PUBLICATIONS **621** CITATIONS
SEE PROFILE

Youfeng Wu
Intel
90 PUBLICATIONS **1,792** CITATIONS
SEE PROFILE

AMAS-BT

Architectural and Microarchitectural Support for Binary Translation

PROCEEDINGS -- AMAS-BT 2008

1st Workshop on Architectural and Microarchitectural Support for Binary Translation

Held in conjunction with the 35th Int'l Symposium on Computer Architecture

(ISCA-35)

Beijing, China -- June 21, 2008

http://amas-bt.cs.virginia.edu/

AMAS-BT
Architectural and Microarchitectural Support for Binary Translation

Word From The Organizing Committee

Long employed by industry, large scale use of binary translation and on-the-fly code generation is becoming pervasive both as an enabler for virtualization, processor migration and also as processor implementation technology. The emergence and expected growth of just-in-time compilation, virtualization and Web 2.0 scripting languages brings to the forefront a need for efficient execution of this class of applications. The availability of multiple execution threads brings new challenges and opportunities, as existing binaries need to be transformed to benefit from multiple processors, and extra processing resources enable continuous optimizations and translation.

The main goal of this half-day workshop is to bring together researchers and practitioners with the aim of stimulating the exchange of ideas and experiences on the potential and limits of Architectural and Microarchitectural Support for Binary Translation (hence the acronym AMAS-BT). The key focus is on challenges and opportunities for such assistance and opening new avenues of research. A secondary goal is to enable dissemination of hitherto unpublished techniques from commercial projects. A total of six high-quality papers were selected, and each paper received at least 3 reviews.

Welcome to Beijing and to the first workshop on Architectural and Microarchitectural Support for Binary Translation !

Mauricio Breternitz

Workshop Organizers

- Mauricio Breternitz, Intel
- Robert Cohn, Intel
- Youfeng Wu, Intel

Program Committee

- Erik Altman, IBM
- Mauricio Breternitz, Intel
- Mark Charney, Intel
- Robert Cohn, Intel
- Andy Glew, Intel
- Kim Hazelwood, University of Virginia
- David Kaeli, Northeastern University
- Chris J. Newburn, Intel
- Alex Skaletsky, Intel
- Chenggang Wu, CAS, China
- Youfeng Wu, Intel

Web development by Dan Upton, University of Virginia
Logo design by Raquel Breternitz, University of Texas

Table Of Contents

Characterization of Dynamic Binary Translation Overhead

Edson Borin

PSL - Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95054
edson.borin@intel.com

Youfeng Wu

PSL - Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95054
youfeng.wu@intel.com

Abstract

In the recent years, dynamic binary translation has become a hot research area. Besides supporting legacy binary code and ISA virtualization, it enables innovative co-designed microarchitectures and allows transparent binary instrumentation. The dynamic nature of the translation usually incurs extra execution overhead and many research works had proposed software and hardware solutions to minimize the overhead [1, 2]. In this paper, we analyze our dynamic binary translator (StarDBT) performance and depict the main sources of overhead in details. We classify the translation operations and associated overhead into five major categories, and quantify their contribution to the overall overhead. Based on the analysis and detailed evaluation, we identify and point out the most promising solutions to address the overhead problem. We believe this study is an important first step toward the grand goal of zero-overhead dynamic binary translation.

1. Introduction

Dynamic binary translation (DBT) has emerged as an important tool with many real world applications. For example, it can be used to support legacy binary code [3]; support ISA virtualization [4]; enable innovative co-designed microarchitectures [5, 6], and many others [7, 8, 9, 10, 11, 12]. As a research infrastructure, we developed a DBT system, named StarDBT [13], which translates from IA (Intel Architecture, a.k.a 'x86') to IA, including from IA32 to Intel64. In this paper, we focus on the IA32 to IA32 translation. This "same ISA" translation system is useful for dynamic binary optimization as well as program shepherding for reliability and security.

Like most other sophisticated DBT systems [9, 14, 15], StarDBT adopts the two-phase translation strategy. It uses a simple fast translator for cold code translation and once a workload hotspot is detected, it applies optimizations. The simple and fast translation tries to generate target code with minimal runtime overhead. Control transfer instructions such as branches, function calls, and returns need to be rewritten. This involves some translation table lookup and dispatch code. The simply translated code is also instrumented with profiling code to collect block frequency information to recognize program hotspots for optimizations.

For program hotspots, we form straight-line traces to optimize. Because the optimizations are conducted at runtime, we only implement a few effective optimizations such as code layout and partial redundancy elimination. In order to isolate the performance overhead, we turned off the optimizations in StarDBT for this study, except the code layout optimization implied by the hot trace formation process. The translator places generated code in a code cache for later reuse. The runtime dispatcher maintains a translation lookup table. The execution of the native IA32 code is achieved via the execution of the translated code and dispatching in the runtime system. In our system, we use a 64K-entry translation lookup table, as in [15], and we allocate 16MB of memory for code cache. Once the lookup table or the cache is full, we simply flush the code cache.

StarDBT's performance is very competitive to state-of-the-art DBT systems [14, 15, 16]. Figure 1 shows the overall performance overhead of the execution under StarDBT compared with the native execution, without StarDBT, for SPEC2000 [17] benchmarks. On average, the translated version runs about 10% slower than the native version for the entire SPEC2000 benchmarks, and 20% and 1% for SPEC2000int and SPEC2000fp, respectively. This overhead is very acceptable for program shepherding kinds of applications. For dynamic optimization, however, in order for dynamic binary optimization to improve program performance, the overhead represents a serious obstacle to overcome and show benefits from optimizations.

Some of the benchmarks (179.art and 178.galgel) execute faster under StarDBT than in native mode because StarDBT builds traces for hot code, which removes some direct branch instructions and improve instruction cache hit ratio due to better code locality.

This paper investigates StarDBT overhead in details. We

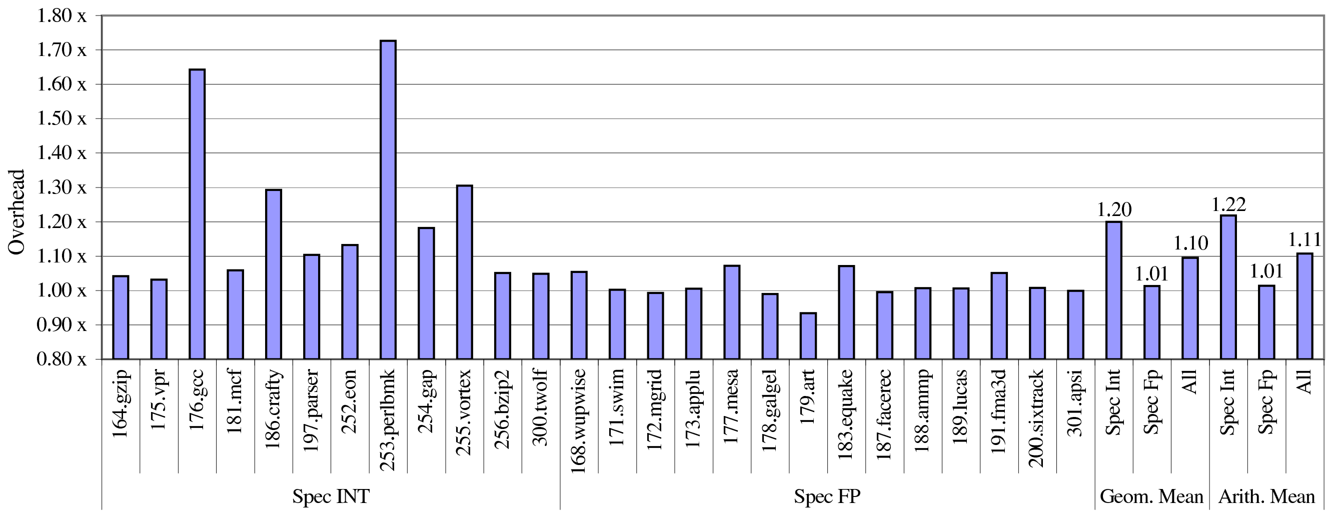


Figure 1: StarDBT overall performance for SPEC2000.

classify the translation operations and associated overhead into five major categories, and quantify their contribution to the overall overhead. We also point out the most promising solutions to eliminate the overhead, via hardware or software means. We believe this study is an important first step toward the grand goal of zero-overhead dynamic binary translation.

Particularly, the contributions of this work can be summarized as bellow:

- We developed a methodology to measure the overhead by running StarDBT on real systems. Specifically, we modify StarDBT to have special versions that manifest the overhead associated with particular DBT operations.

- We studied in detail the overhead associated with five major DBT operations and quantitatively reported the contribution of operations to the DBT overhead.

- We point out the most promising solutions to remove the overhead toward a potentially zero overhead DBT system.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 investigates the overhead in details, quantifying the sources of the overhead. Section 4 summarizes the results and points out the most promising solutions to remove the overhead. Finally, Section 5 concludes the paper.

2. Related Work

There have been a number of publications that tries to address the DBT overhead issue. For example, Kim and Smith [1] studied the translation overhead for indirect branches and function returns, and proposed interesting solutions to address them. Hu and Smith [2] investigated startup overhead in DBT systems and developed two solutions to alleviate the problem. Most of the early studies target specific overhead issues and studied them in a simulation environment. We present the “big picture” for the overall overhead in DBT systems, and quantify the overheads with real DBT runs. The solutions proposed by Kim and Hu are valuable to reduce the DBT overhead.

3. Tracking the Overhead

Dynamic binary translation naturally adds overhead to the application execution. In this section we identify the most important sources of overhead in StarDBT and present ideas to remove it. To analyze the sources of overhead, we break StarDBT functionality in five main steps: *initialization*, *cold code translation*, *code profiling*, *hot trace building* and *translated code execution*. We define the five DBT operations in more details below.

Initialization: Before translating and executing an application, the StarDBT itself (implemented as a dynamic library) must be loaded into memory and initialized. We call this overhead *initialization overhead*.

Cold Code Translation: Starting from the beginning of the application, every time StarDBT encounters a new basic block, it translates and places it into the code cache before executing. The overhead caused by translating this code is called *Cold Code Translation Overhead*.

Code Profiling: In order to accelerate the application execution, StarDBT uses runtime information to detect hot (frequently executed) code and optimize it. The most common approach when looking for hot code is to profile basic blocks and build single-entry, multiple-exit traces of code, also known as hot traces [7, 9, 13, 18, 19]. Unless hardware support is provided, profiling basic blocks requires code instrumentation, which adds two kind of overhead:

1. *Profiling instrumentation overhead*: the time required to perform the instrumentation (generally included in the translation time of the dynamic binary translators);
2. *Profiling execution overhead*: time spent executing the profiling instructions.

In order to reduce the profiling overhead, StarDBT, like other DBTs [7], limit profiling to a subset of basic blocks. These blocks generally have a high potential to become a trace header (trace entry point). As an example, back-edges targets are selected for profiling because they are very likely to be loop headers.

Hot Trace Building: After identifying a hot block, StarDBT builds a hot trace using the MRET[2] approach [7, 20]. In this approach, StarDBT records a trace by following the control execution until it reaches a stop condition (e.g. syscall instruction, etc). The control execution is followed twice and the common path is selected to become a trace. After recording the most recent executed trace, StarDBT builds a hot trace and updates the translated code to jump into the new trace instead of the old translated cold block. The overhead caused by following the execution and building the trace is called *Hot Trace Building Overhead*.

Translated Code Execution: In order to keep the original program behavior, StarDBT must emulate some of the instructions it translates. For example, x86 "call" instructions cannot be executed in translated code because it would automatically push the next IP in the translated code into the program stack. Notice that the stack must contain the same next IP as in the original program execution to guarantee the original program behavior. The `CALL target` instruction can be emulated by executing a `PUSH original NEXT_IP` and a `JMP target` instruction. Also, during execution, indirect branch instructions (including the return instruction) must resolve their target address every time they are executed. The address resolution takes the original target address and converts it into the translated target address. StarDBT may perform a search in a dispatch table containing the original-translated addresses pairs. Besides the emulation and control transfers overhead, the hot code building implicitly performs tail duplication to form single entry traces and the duplicated code incur performance overhead by increasing the instruction cache pressure and the branch prediction unit pressure. We classified these overheads as *Translated Code Execution Overhead*.

To quantify the StarDBT overheads during each of the above five operations, we modify StarDBT to have special versions that manifest the overhead associated with particular DBT operations. This allows us to measure the overhead on real machines with large inputs. For example, in order to measure the overhead of StarDBT trace building overhead, we develop a special version of StarDBT that takes persistent traces generated from a previous run, compare its execution time with that of the regular version, and report the overhead associated with hot trace building.

Assume that the overhead has n sources, S_1, S_2, $\cdots$, S_n. The overall StarDBT overhead = overhead(S_1, S_2, $\cdots$, S_n) = $DBT/Native$, where DBT is the execution time of the original binary under StarDBT and $Native$ is the execution time of the original binary without StarDBT. Our methodology separately measures the overhead from S_i, $i = 1, \cdots, n$, say overhead(S_i) = $DBT_i/Native$, where DBT_i is a special version of StarDBT to manifest overhead S_i. If the overhead from different sources are mutually independent, we should have overhead(S_1, S_2, $\cdots$, S_n) = overhread(S_1) + $\cdots$ + overhead(S_n). Our experiments actually show that, for most of the benchmarks, overhead(S_1, S_2, $\cdots$, S_n) is very similar to overhread(S_1) + $\cdots$ + overhead(S_n). This suggests that our overhead measurement is relatively accurate. Although it would be an interesting research to characterize the correlation between the overheads, this paper focuses on the relative importance of the overheads.

We evaluated the StarDBT overhead using the SPEC2000 [17] benchmarks with reference inputs on a 2.93 GHz Intel Xeon X7350 Linux machine with 2GB of RAM. The benchmarks were compiled using ICC 9.1 with aggressive compilation flags (-fast) and profiling feedback. Each benchmark was executed 10 times and we computed the average (geometric and arithmetic mean) execution time ratios and percentages. We also computed the 95% confidence interval and verified that the quality of the results was high.

Since the average overhead for SPEC2000fp benchmarks is very small (less than 1.4%), we concentrate our analysis on SPEC2000int benchmarks in the rest of the paper. We will also use percentage of execution time increase over the native execution to have finer grained comparison of the overhead. As some benchmarks have near 0% overhead, we have to use arithmetic mean instead of geometric mean to express the overhead. The following sub-sections discuss the StarDBT overheads.

3.1. Initialization Overhead

Before translating and executing the application, the StarDBT library is loaded into memory and initialized. In order to measure the *initialization overhead* we modified StarDBT to execute the native code right after the library initialization. We call this configuration DBT_{Native}. In this approach, the StarDBT library is loaded and initialized but, instead of translating code, the native code is executed.

We executed the SPEC2000 benchmarks using the DBT_{Native} configuration and compared the execution time with the native code execution (referred to as $Native$). Namely,

$$S_{Initialization} = \frac{DBT_{Native} - Native}{Native}$$

Our experimental results show that the average overhead caused by the StarDBT library initialization is less than 0.1%.

3.2. Profiling and Hot Trace Building Overhead

In order to measure the profiling and hot trace building overhead, we run StarDBT using three different configurations:

- $DBT_{DTraces}$: StarDBT dynamically profiles the code and build MRET^2 [20] traces. This configuration includes the overhead of profiling and building the hot traces.

- $DBT_{PTraces}$: StarDBT avoids profiling and trace building overhead by loading persistent traces from a file. In this approach, the application is executed twice: in the first run, StarDBT generates and dumps the MRET^2 traces into a file and, in the second run, it loads back the traces from the file in the beginning of the execution.

- $DBT_{LT+Native}$: similarly to DBT_{Native} configuration, the native code is executed right after the library initialization. However, this configuration loads the traces from file during the initialization. This configuration allows us to isolate and measure the overhead of loading persistent traces from file.

We measured the persistent trace loading overhead by comparing the execution time of the SPEC2000 benchmarks using the $DBT_{LT+Native}$ and the DBT_{Native} configurations. The $DBT_{PTraces}$ approach removes the profiling and hot trace building overhead but adds the overhead to load the persistent traces from file. In order to provide a fair comparison, we isolate the profiling and the hot trace building overhead by subtracting the overhead of loading persistent traces from file. Figure 2 shows the StarDBT profiling and hot trace building overhead for the SPEC2000int applications, measured using the following formulas:

$$S_{LT} = \overbrace{DBT_{LT+Native} - DBT_{Native}}^{\text{Load Traces Overhead}}$$

$$S_{Prf+HTBuild} = \frac{DBT_{DTraces} - (DBT_{PTrace} - S_{LT})}{Native}$$

The average slowdown caused by profiling and building the hot traces in SPEC2000int benchmarks is about 5.2%. The overhead is not high due to the fact that many of these

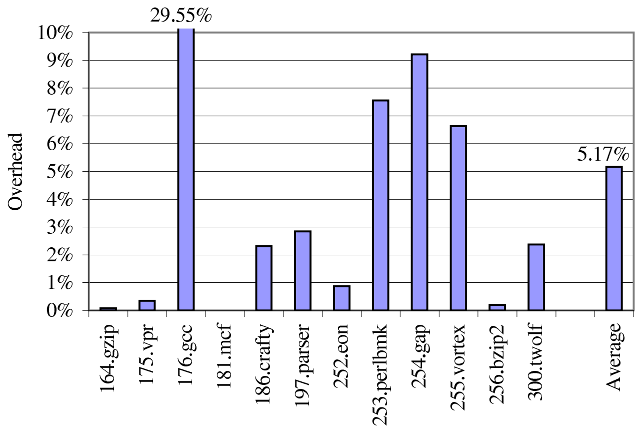


Figure 2: Profiling and Hot Trace Building Overhead.

benchmarks spend most of the executing time in a very small set of kernels. Moreover, these applications execute for a long time (Spec2000 reference input), which amortizes the overhead.

Applications with short run time and less code reuse suffer more from profiling and hot trace building overhead. The 176.gcc benchmark, with an overhead of almost 30%, is an example of such applications. It distributes the execution through a large set of kernels and presents one of the shortest execution times among the SPEC2000 benchmarks. We expect the profiling and trace building overhead to be higher on interactive applications [13], which indicates the importance of investigating efficient techniques to perform hot code profiling and hot trace building [3, 4, 21, 22, 23].

Aside from profiling and building hot traces, trace optimizations also introduce extra overhead and should be performed carefully. Since we have turned off StarDBT trace optimizations, we can omit the optimization overhead here. Next section discusses the cold code translation overhead.

3.3. Cold Code Translation Overhead

During execution, every time the application control reaches not-yet-translated code, StarDBT translates the code and places it into the code cache. StarDBT also updates the previously translated blocks to branch into the newly translated code. The overhead caused by translating this code is called *Cold Code Translation Overhead*.

The $DBT_{PTraces}$ configuration removes the profiling and hot trace building overhead by using persistent traces. However, this configuration still suffers from *cold code translation overhead*. In order to remove this overhead, we created the $DBT_{PTraces+CP}$ (Code Patching) configuration. In this configuration, the persistent hot traces are loaded and the native code is patched with instructions to jump to the loaded traces. After this, the patched code is executed. In this way, no cold code translation is performed.

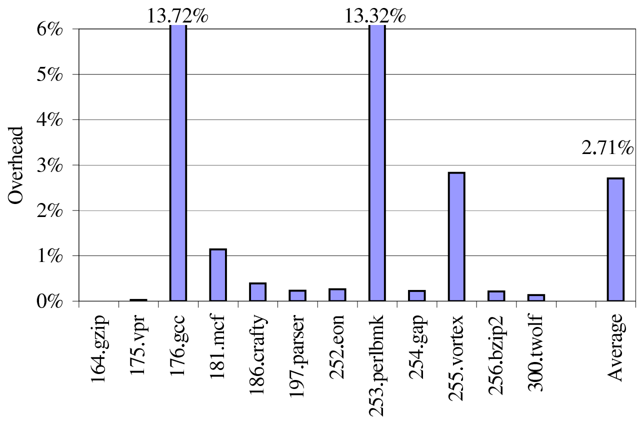


Figure 3: Cold code translation overhead.

Note that patching the original code is in general unsafe due to the well-known self-reference issues. We patch original code only for overhead measurement purpose. We estimate the cold code translation overhead by comparing the performance overhead when using the $DBT_{PTraces}$ and the $DBT_{PTraces+CP}$ configurations. Figure 3 shows the estimated cold code translation overhead, measured using the following formula:

$$S_{Cold\ Trans.} = \frac{DBT_{PTraces} - DBT_{PTrace+CP}}{Native}$$

Analogously to the profiling and hot trace building, applications with longer run time have less cold code translation overhead on the performance. Also, benchmarks that touch less code during the execution have less code translated. In fact, 176.gcc and 253.perlbmk, showing the worst results, have the shortest average execution time. 176.gcc (253.perlbmk) is executed using 5 (7) different input data sets with an average execution time about 21% (23%) of the average execution time for SPEC2000int benchmarks.

Applications with shorter run times, like GUI applications, may suffer even more from cold code translation overhead [2, 13]. This indicates that fast translation [2, 18] should be employed in order to reduce the cold code translation overhead. Another promising method to reduce cold code translation is to run native code directly during cold code execution for same-ISA dynamic optimizations. Native code execution however may require hardware support to avoid code patching and self-reference issue.

3.4. Translated Code Execution Overhead

Ideally, removing the overhead described in the previous sections would make the translated code run at least as fast as the native code. However, the translated code is not exactly the same as the original code and the lack of specialized hardware support may cause extra execution overhead when executing the translated code. In this section, we analyze the translated code execution overhead.

3.4.1 Code Emulation Overhead

In order to keep the original program behavior, the translated code must emulate some of the native instructions during execution. X86 "call" and "return" instructions are examples of instructions that must be emulated. The "call" instruction automatically pushes the next IP into the program stack. In the translated code, the next IP will be the instruction address in the translated code, which could produce a wrong program behavior if the original program expects the original instruction address in the stack. Table 1 shows the sequences of instructions used to emulate the "call", "return" and "indirect jump" instructions in the translated code.

Table 1: Instructions emulated in the translated code.

Native Instruction	Emulation (Translated Code)

`CALL target`	`PUSH ORIGINAL_NEXT_IP`
	`JMP target`

`RET imm16`	`LEA ESP, [ESP + 4 + imm16]`
	Resolve translated address
	...
	`IND. JMP TRANSLATED ADDR`

`IND. JMP reg`	*Resolve translated address* ... `IND. JMP TRANSLATED ADDR`

The instructions in Table 1 are the most frequently executed instructions that require emulation in our IA32 to IA32 StarDBT. However, the IA32 to EM64T StarDBT version [13] must emulate more instructions (push, pop, etc.) and the overhead caused by the extra instructions may be higher.

The return and the indirect jump instructions require the dispatcher to resolve the target address before jumping. Apart from the address resolution overhead, we do not expect the extra instructions added in the emulation to cause significant execution overhead in the IA32 to IA32 StarDBT. The address resolution overhead is part of the code cache control transfer overhead and is detailed in the next sub-section.

3.4.2 Code Cache Control Transfer Overhead

StarDBT employs three modes of operations to perform the control transfer between blocks inside the code cache:

- *Chaining*: whenever a new code block is translated, StarDBT updates the previously translated blocks to directly jump to the new translated block. During the translated code execution, the chained blocks directly

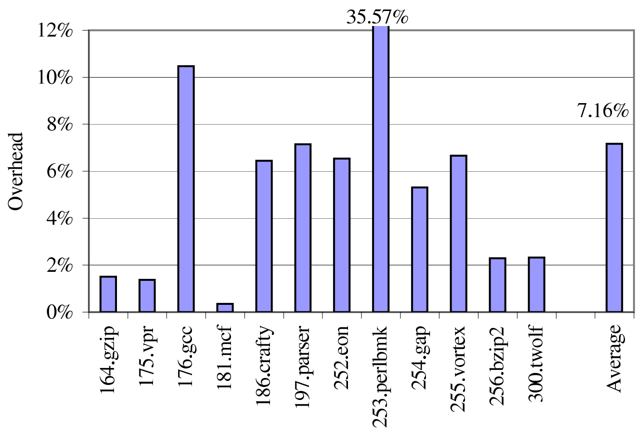


Figure 4: Inlined dispatching overhead.

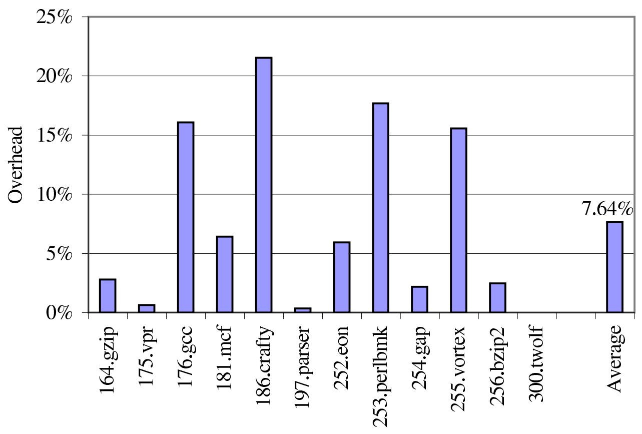


Figure 5: Offline-Patched binary overhead.

jump to each other and the control transfer does not incur extra overhead.

- *Dispatching*: system calls, not-yet-translated target addresses and indirect branches forbid the usage of chaining. In these cases, the StarDBT Code Dispatcher is called. The Code Dispatcher maintains a dispatch table that associates the original addresses with their respective translated addresses. For not-yet-translated target addresses, the Code Dispatcher translates the target block and updates the dispatch table. For indirect branches, the Code Dispatcher resolves the target address (by looking up in the dispatch table) and transfer the control to the translated target block. Calling the Code Dispatcher to resolve the indirect branches target addresses is costly, since it involves context saving (restoring) when entering (exiting) the Code Dispatcher handler.

- *Inlined dispatching*: In order to reduce the overhead of entering and exiting the Code Dispatcher, StarDBT aggressively inlines the dispatch table lookup code for indirect branches and function returns into the translated code when generating code for hot traces.

As we discussed before, chaining does not incur extra overhead during the translated code execution. Due to the inlined dispatching approach, the Code Dispatcher is only called to resolve addresses from cold code. However, cold code is rarely executed and the overhead caused by calls to the Code Dispatcher is minimized. In this way, the majority of the address resolution overhead occurs with the inlined dispatching, which is used inside frequently executed hot code, and may cause significant performance overhead.

In order to evaluate the inlined dispatching overhead, we derivate the $DBT_{PTraces+CP}$ configuration into two versions:

- $DBT_{PTraces+CP}$ (Code Patching): As described in Section 3.3, this configuration loads the persistent traces from file and patches the original native code with instructions to jump to the loaded traces. In this way, no translation is performed. This configuration also allows the indirect branches in the hot traces to directly jump into the original (patched) code, with no dispatching overhead. After jumping to the original code, the execution is redirected back to the hot traces in the first patched block.

- $DBT_{PTraces+CP+AR}$ (Address Resolution): This configuration is similar to the previous one, but the hot traces include the dispatch table lookup code to resolve the indirect jumps target addresses. Therefore, this configuration includes the inlined dispatching overhead.

We compared the performance of both configurations to determine the inlined dispatching overhead. Figure 4 shows the inlined dispatching overhead for the SPEC2000int applications, measured using the following formula:

$$S_{Addr.Res.} = \frac{DBT_{PTraces+CP+AR} - DBT_{PTrace+CP}}{Native}$$

The experimental results show that the address resolution overhead caused by indirect jumps and return instructions is 7.16%. It accounts for approximately 31.5% of total overhead. This indicates that code cache control transfer techniques, such as JTLB [1], should be employed to reduce the overhead.

3.4.3 Code Duplication and RAS Overhead

The $DBT_{PTraces+CP}$ configuration eliminates the profiling, the hot trace building, the code translation, and the inlined dispatching overhead. However, it still initializes the StarDBT library, loads the traces, and patches the original

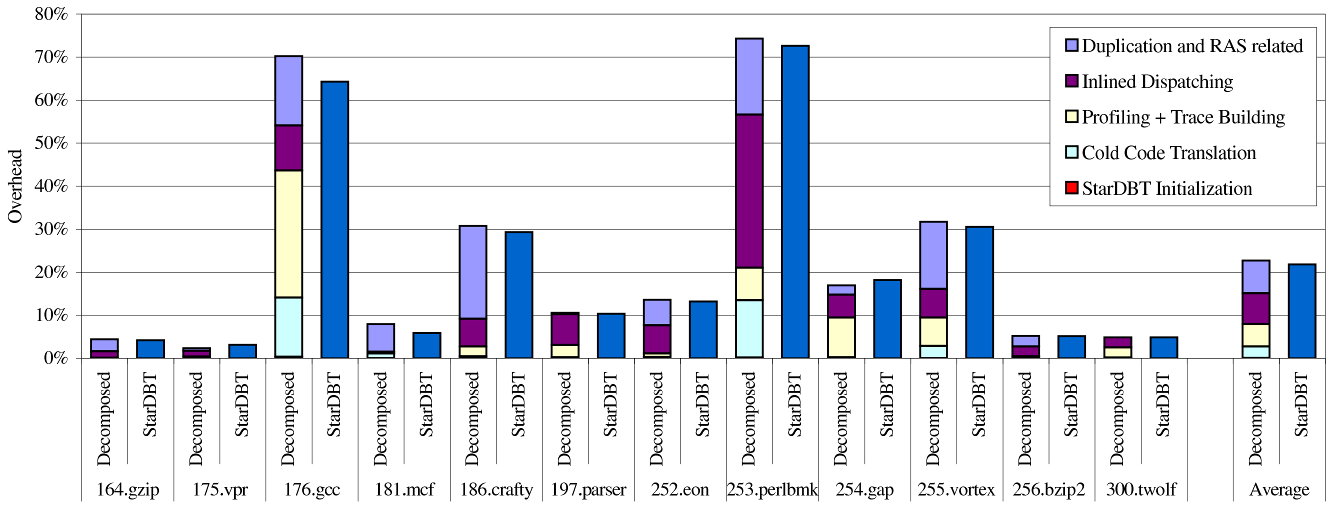


Figure 6: StarDBT total overhead versus decomposed overhead.

native code at runtime. In order to estimate the remaining overhead, we created an offline tool to insert the traces into the original executable file and patch the native binary code to jump to the traces. We call the resulting executable file: *offline-patched binary*.

Ideally, executing the *offline-patched binary* would be at least as efficient as executing the native code. However, the hot code is a morphed copy of the original code and may use the underline microarchitecture differently. Figure 5 shows the *offline-patched binary* overhead when compared to the native code execution.

Although the *offline-patched binary* does not involve dynamic binary translation, its execution is still about 7.6% slower than the native run. This overhead is primarily generated from two sources: overhead to predict return address, and the overhead related to code duplications. In this section we point out potential microarchitecture related overhead when executing translated code.

Instruction Cache Overhead Code duplication, due to aggressive trace formation or inlining dispatcher may increase the dynamic instruction footprint by about 20%. Increased instruction footprint increases the pressure on the instruction cache and cause performance degradation due to cache misses.

Branch Predictor Unit Overhead Branch miss-predictions cause the pipeline to execute wrong instructions and the recovery is usually costly in terms of processing cycles. The branch predictor unit relies on the branch target buffer (BTB) to keep track of the branching statistics for the most recent executed branches. Like the instruction cache, the BTB is a limited size cache and the code duplication may increase the pressure on it, causing performance degradation.

Return Address Stack Overhead The Intel Core 2 Duo microarchitecture, used in our experiments, relies on the return address stack (RAS) to efficiently predict the target address of return instructions. In fact, return address stack is a common technique used by most modern microprocessors to predict return addresses. However, as discussed before, return instructions cannot be executed inside the translated code and are normally emulated using indirect jump instructions (Table 1). It turns out that, replacing return instructions by indirect jumps greatly increase the pressure on the indirect branch predictor cache, because it has to keep track of more branch instructions. Moreover, the indirect jump predictor was not designed to predict return target addresses and the prediction accuracy is poor. In the next section we propose a technique to solve this and other problems.

4. Summary and Recommendations

In Section 3, we investigated the dynamic binary translation overhead in details, decomposing and quantifying the sources of the overhead in StarDBT. In order to check the accuracy of the decomposed overhead, we compared the total StarDBT overhead with the sum of the decomposed overhead. Figure 6 shows the total StarDBT overhead and the sum of the decomposed overhead side by side.

Notice that the sum of the decomposed overhead is very similar to the total StarDBT overhead. In fact, the average sum of the overheads is 22.73% while the average StarDBT overhead is 21.80%. This indicates that 1) most of the overhead associated with individual operations are relatively independent, 2) our methodology has captured the DBT over-

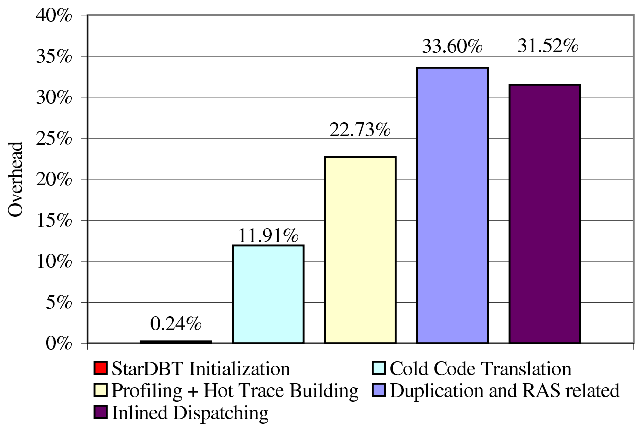


Figure 7: Contribution to the total overhead.

head reasonably accurately.

Figure 6 also shows that most of the overhead is caused by RAS/code duplication related overhead and inlined dispatching. In fact, the RAS/code duplication issue and the inlined dispatching are responsible for 33.6% and 31.5% of the total overhead, respectively, while profiling and hot trace building, and cold code translation are responsible for 22.7% and 11.9% of the overhead. StarDBT initialization has no significant performance impact. Figure 7 shows StarDBT main source of overhead when executing SPEC2000int benchmarks, compared to the baseline assuming the total overhead is 100%.

Although the cold code translation overhead is generally small in our experiments, applications with short run times or without hot spots, like GUI applications, may suffer more from cold code translation overhead [2, 13]. In fact, 176.gcc and 253.perlbmk benchmarks have the shortest average execution times among SPEC2000 applications, without many obvious hot spots, and showed the worst cold code translation overheads, 13.7% and 13.3% respectively. Therefore, it is still important to investigate techniques to solve this issue. Fast translation [2, 18] or native code execution support are examples of techniques that could be employed to reduce the cold code translation overhead. Another idea is to use interpretation [6] before performing the cold code translation. Some code are executed only once, and interpreting the code can be faster than translating and executing it. Besides, interpretation does not duplicate code and avoids code cache bloating.

Native code execution is especially attractive for "same ISA" translation. Since the target ISA is the same as the source ISA, we may run native code during cold code execution. This can potentially achieve zero overhead cold code execution, if we can efficiently solve the following two issues: 1) since the native code is not instrumented, we will need additional techniques to collect profile information for forming hot traces; 2) because of the "self-reference" issue, we need hardware ip-remapping support to link cold code with hot code without patching the original code. Recently advance in lightweight profiling support can address the first issue [21, 22]. The JTLB-like hardware [1] can help solve the second issue.

Applications with short run times and less code reuse are more likely to suffer from profiling and hot trace building overhead [13]. Moreover, our experimental results show that profiling and hot trace building caused more overhead than cold code translation. Therefore, it is also important to investigate efficient techniques to perform hot code profiling and hot trace building, such as proposed in [3, 4, 21, 22, 23].

Inlined dispatching, caused by indirect branches and return instructions in hot traces, is an important source of overhead in StarDBT. The increasing usage of languages like C++ and Java augment the number of return and indirect branches instructions in the programs. This indicates that this overhead will become more important and code cache control transfer techniques, such as in [1], should be investigated to reduce the overhead.

Code duplication and RAS related overhead, including the instruction cache miss, branch prediction miss, and the overhead of not using the return address stack, was one of the most important sources of overhead in our experiments. To accurately quantify the specific microarchitectural related overheads, we may need to use a cycle-accurate simulator. However, early work, such as [1], has already confirmed that not using the return address stack is a significant source of overhead. To reduce this overhead, we propose two new instructions for DBT:

- `RAS_PUSH [REG|IMM|MEM]`: pushes a value into the return address stack prediction hardware.
- `JMP_RAS [REG|MEM]`: similar to the X86 indirect jump instruction, but uses the return address stack to predict the jump target.

The `RAS_PUSH` instruction would be used to push the translated return address into the RAS every time we emulate a call instruction and the `JMP_RAS` instruction would be employed when emulating return instructions. In this way, the translated code would not burden the regular branch prediction hardware when emulating return instructions. Table 2 shows the call and return emulation using the new instructions.

5. Conclusions and Future Works

In this paper we investigated the most important sources of overhead in our dynamic binary translator (StarDBT). We first classified the1 translation operations and associated overhead into five major categories. Then, we quantified their contribution to the overall overhead by modifying

Table 2: Call/Return emulation using the new instructions.

Native Instruction	Emulation (Translated Code)

`CALL target`	`PUSH ORIGINAL_NEXT_IP`
	`RAS_PUSH TRANS_NEXT_IP`
	`JMP target`

`RET imm16`	`LEA ESP, [ESP + 4 + imm16]`
	Resolve translated address
	...
	`JMP_RAS TRANSLATED ADDR`

`IND. JMP reg`	*Resolve translated address* ... `IND. JMP TRANSLATED ADDR`

StarDBT to have special versions which manifest the overhead associated with particular DBT operations. Finally, we pointed out the most promising solutions to attack the DBT performance overhead problem.

Code duplication and RAS related overhead was one of the most important source of overhead in our experiments, causing an average overhead of 7.2% in the SPEC2000int benchmarks. Early study attributes majority of this overhead to the fact that the translated code for call and return does not use the RAS. We proposed a novel method to remedy this problem. We will evaluate this new method in the future.

Our experimental results also indicates that cold code translation, profiling, hot trace building overhead, and *inlined dispatching* for indirect branches are important sources of overhead in StarDBT. Therefore, more researches on fast code translation, e.g. enabling direct execution of native code, on profiling and hot trace building, e.g. lightweight profiling support, and on fast code emulation, e.g. providing hardware support for indirect branch and function return, should be employed in order to achieve near zero overhead DBT.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and discussions. We also appreciate the support provided by Jesse Fang at the Programming System Laboratory at Intel.

References

[1] Kim, H-S., and Smith, J. Hardware Support for Control Transfers in Code Caches. In proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. pp. 253–264. 2003, Washington, DC, USA.

[2] Hu, S., Smith, J. Reducing Startup Time in Co-Designed Virtual Machines. SIGARCH Comput. Archit. News. 34(2):277–288. 2006.

[3] Moseley, T., Shye, A., Reddi, V., Grunwald, D., and Peri, R. Shadow Profiling: Hiding Instrumentation Costs with Parallelism. In proceedings of the International Symposium on Code Generation and Optimization. pp. 198–208. 2007, Washington, DC, USA.

[4] Bond, M., and McKinley, K. Continuous Path and Edge Profiling. In proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture. pp. 130–140. 2005, Barcelona, Spain.

[5] Cmelik R. F., Ditzel D. R., Kelly E. J., Hunter, C. B., Laird, D. A., Wing, M. J., and Zyner, G. B. Combining Hardware and Software to Provide an Improved Microprocessor. US Patent # 6031992. 2000.

[6] Klaiber A. The Technology Behind Crusoe Processors. Transmeta Technical Brief. 2000.

[7] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A Transparent Runtime Optimization System. In proceedings of the 2000 ACM SIGPLAN conference on Programming language design and implementation. pp. 1–12. 2000, Vancouver, British Columbia, Canada.

[8] Borin, E., Wang C., Wu Y., and Araujo G. Software-Based Transparent and Comprehensive Control-Flow Error Detection. In proceedings of the international symposium on Code generation and optimization. pp. 333–345. 2006, New York, NY, USA.

[9] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser. A., Lowney. G., Wallace, S., Reddi, V., and Hazelwood, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 190–200. 2005, Chicago, IL, USA.

[10] Qin, F., Wang, C., Li, Z., Kim, H.-S., Zhou, Y., and Wu, Y. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In proceedings of the 39th annual IEEE/ACM International Symposium on Microarchitecture. pp. 135–148. 2006, Orlando, FL, USA.

[11] Wu, Q., Reddi, V., Wu, Y., Lee, J., Conners, D., Brooks, D., Martonosi, M., and Clark, D. Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture. pp. 271–282. 2005, Barcelona, Spain.

[12] Ying, V., Wang, C., Wu, Y., and Jiang, X. Dynamic Binary Translation and Optimization of Legacy Library Code in a STM Compilation Environment. In Workshop on Binary Instrumentation and Applications. 2006, San Jose, CA, USA.

[13] Wang, C., Hu, S., Kim, H.-S., Nair, S., Breternitz, M., Ying, Z., and Wu, Y. StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In proceedings of Asia-Pacific Computer Systems Architecture Conference. pp. 4–15. 2007, Seoul, Korea.

[14] Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skalesky, A., Wang, Y., and Zemach, Y. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. pp. 191–201. 2003, San Diego, CA, USA.

[15] Bruening, D. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D thesis, Massachusetts Institute of Technology, 2004.

[16] Sridhar, S., Shapiro, J. S., Northup, E., and Bungale, P. HDTrams: An Open Source, Low-Level Dynamic Instrumentation System. In proceedings of the 2nd international conference on Virtual execution environments. pp. 175–185. 2006, Ottawa, Ontario, Canada.

[17] Henning, J. SPEC CPU2000: Measuring CPU Performance in the New Millennium. Computer. 33(7):28–35. 2000.

[18] Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., and Mattson, J. The Transmeta Code MorphingTM Software: Using Speculation, recovery, and Adaptive Retranslation to Address Real-Life Challenges. In proceedings of the International Symposium on Code Generation and Optimization. pp. 15–24. 2003, San Francisco, CA, USA.

[19] Gal, A., Probst, C., and Franz, M. HotpathVM: an effective JIT compiler for resource-constrained devices. In proceedings of the 2nd international conference on Virtual execution environments. pp. 144–153. 2006, Ottawa, Ontario, Canada.

[20] Wang, C., Bixia, Z., Kim, H-S., Breternitz, M., and Wu, Y. Two-pass MRET trace selection for dynamic optimization. US Patent # 20070079293. 2007.

[21] Chen, H., Hsu, W-C., Lu, J., Yew P-C., and Chen D-Y. Dynamic trace selection using performance monitoring hardware sampling. In proceedings of the international symposium on Code generation and optimization. pp. 79–90. 2003, San Francisco, CA, USA.

[22] Merten, M., Trick, A., Nystrom, E., Barnes, N., and Hmu, W. A hardware mechanism for dynamic extraction and relayout of program hot spots, SIGARCH Comput. Archit. News. 28(2):59–70. 2000.

[23] Zhang, W., Calder, B., and Tullsen, D. M. An Event-Driven Multithreaded Dynamic Optimization Framework. In proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques. pp. 87–98. 2005. Saint Louis, MO, USA.

Cold Code Analysis

Wesley Attrot* Unicamp

Guido Araújo Unicamp

Abstract

Dynamic binary translators are programs that translate binary programs from one machine to another. The translation is done on the fly, so performance is a major issue in this kind of system. Identifying and optimizing hot traces is a way to achieve more performance, and also to compensate for the translation overhead. Aggressive optimizations need precise data/control-flow information about the code, otherwise they will be conservative and less effective. In this paper, we measure the amount of additional data-flow information one can obtain by going beyond hot trace boundaries into non-frequently executed (cold) code. We show that in some cases, as in liveness analysis, one can considerably improve the information available, thus creating more opportunities for trace optimization. Moreover, the amount of additional data-flow information decreases very fast as one departs from trace boundaries, limiting the overhead imposed by the cold code analysis.

1 Introduction

Dynamic binary translators (DBTs) are programs designed to execute in a host machine binaries from other machines, performing the translation on the fly. Alternatively, DBTs may be used to improve performance of binaries from the same architecture [4].

DBTs translate each instruction to be executed into a equivalent instruction in the host machine. Some DBTs have a codecache to store the translated instructions and to execute these instructions. The codecache optimization saves time, as it avoids the overhead of translating instructions repeatedly [4].

*This work was sponsored by CNPq and Intel

For optimization purpose DBTs can instrument the translated code so as to identify highly executed regions in the program and select them for analysis. Once optimized, these regions can improve the program runtime and hopefully hide the translation overhead.

Dynamic optimizations may include well-known optimizations like dead code elimination, copy propagation, CSE [2] and others that are more specific to the translation environment. Register allocation in particular, is an optimization that demands much more effort, particularly if the host and guest machines have very different register sets and sizes.

Like in off-line optimizers, on-the-fly optimizers need information about the code, which could be particularly hard to obtain in a dynamic environment. Dynamic environments create some new challenges and constraints to the optimizer. For example, it is not possible to construct the control flow graph (CFG) of the program, only program execution traces are available. Moreover, structures like variables and basic blocks are hidden behind the assembly code.

Off-line optimizers can spend a lot of time gathering information about the code and optimizing it, but dynamic optimizers need to be as fast as possible. The time spent optimizing the traces must pay itself in performance increase or the optimization will be useless. But even the fastest optimizer needs some kind of information about the code to do its job.

2 Trace Optimization

Traces are sequences of instructions, including branches but not including loops, that are executed for some input data [15]. If some trace is executed more times than a specified threshold we call it a *hot trace*. DBT's hot traces are important for optimizations because their high execution frequency considerably improves optimizations' effectiveness, thus improving the overall program execution time.

```
(01) shl esi, cl
(02) add edx, ebx
(03) or eax, esi
(04) movzx eax, ax
(05) mov DWORD PTR [esp+01ch], eax
(06) mov ebx, DWORD PTR [edi*4+080cd380h]
(07) test ebx, ebx
(08) jnz ...
(09) mov eax, DWORD PTR [esp+024h]
(10) movzx ebp, WORD PTR [eax*2+0810dae0h]
(11) add eax, 0x1h
(12) mov DWORD PTR [esp+024h], eax
(13) cmp ebp, 0x100h
(14) jb ...
(15) mov eax, ebp
(16) shr eax, 0x7h
(17) movzx edi, BYTE PTR [eax+080d4560h]
(18) mov ebx, DWORD PTR [esp+020h]
(19) movzx ebx, WORD PTR [ebx+edi*4+02h]
(20) mov eax, ebx
(21) neg eax
(22) add eax, 0x10h
(23) cmp edx, eax
(24) jg ...
(25) mov eax, DWORD PTR [esp+01ch]
(26) movzx esi, WORD PTR [ebx+edi*4]
(27) jmp ...
```

Figure 1: Trace

Figure 1 shows a trace from the SPEC CPU 2000 [1] program 164.gzip in x86 assembly. Since this is a hot trace, we expect that the executing flow enters into the first instruction `shl esi,cl` and leaves out of the last instruction `jmp ...` most of the time. The execution flow may leave the trace in one of the 3 branch instructions in the middle of the trace, but this behavior is expected to occur few times. We call these branches of **side exits** because they break the continuous execution flow of the trace.

Assuming that this trace will execute most of the time without taking a side exit, we can focus the optimizations on this assumption. If we do not care about memory and register aliasing, a simple optimization that can be performed on the example trace is to remove a memory access. The 5^{th} instruction of the trace stores `eax` to `esp+01c` and the 25^{th} instruction on this trace reads this memory position. If we find an empty register covering these two instructions, we can copy the store value into the 5^{th} instruction to this register and replace the memory access at the 25^{th} instruction for a reference to a register. In fact, we notice that register `esi` is used at the 3^{rd} instruction and defined at the 26^{th} instruction. From the 5^{th} to the 25^{th} instruction there is no definition or use of `esi`, but we can not ensure that register `esi` is dead along these instructions.

There are 3 side exists between the instructions under analysis. The value of `esi` may be alive outside the trace on one of the side exits. If this is true, then the proposed optimization is not possible.

The common liveness analysis on this trace is not enough to solve this issue. For conservative reasons we must assume that all registers are alive on the side exists and at the end of the trace. This assumption turns the proposed optimization impossible.

3 Cold Code Analysis

Situations like the presented in Figure 1 restrict the aggressiveness of trace optimizations, allowing optimizations only inside basic blocks. If we always assume, for every optimization, the conservative way in every optimization, we will miss many optimization opportunities.

The proposed optimization to remove the memory access can not be performed under conservative estimation of registers' liveness. But if we inspect some instructions outside the trace, we can increase our information about the register liveness and maybe discover that `esi` is dead on all side exists, thus allowing us to do the optimization.

We call the code outside a trace **cold code**, because it is not supposed to execute as many times as traces, so is not profitable to optimize it. But on the other hand it has information that may be useful to the hot traces, like register and flags usage.

The analysis of cold code consists in performing the required data flow analysis on this piece of cold and gather the required information to increase the data flow analysis precision in the hot trace. Since

DBTs do not generate a full CFG, some heuristics must be employed to walk on the cold code.

Diving into cold code (or walk into cold code) has its own issues involved. How deep must we dive into cold code and how much time spent on it? We have no guaranties that this extra time spent on increasing data flow information will allow us to perform further optimization.

4 Diving into Cold Code

Figure 2 shows the initial configuration for a trace. This trace has 4 basic blocks and 3 side exits. This can be called level 0 cold code diving, that is, no cold code analysis. If we think about liveness analysis, we must assume that all registers are live at each side exit.

Figure 3 corresponds to dive one basic block into cold code (**level 1**). At this level only one basic block into cold code is analyzed. The end of the block is limited by a branch instruction, a function call instruction, a ret instruction or a jump to a non translated address. When we find one of these instructions, we assume that the registers that could not be solved are alive after this point.

If we decide to dive one more level (**level 2**), we must analyze the successors of basic blocks in level 1, when possible, as shown in Figure 4, where we can see that each basic block in level 1 has its successors added to the analysis.

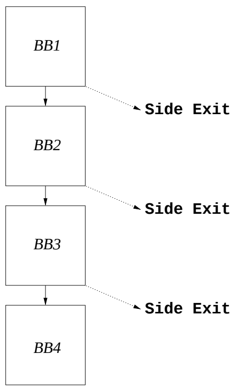


Figure 2: Trace for Optimization

We can expand the diving process to reach deeper levels, but it might not always be possible do dive so deep. Figure 5 shows a possible scenario.

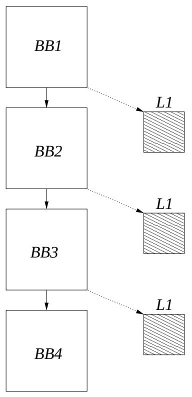


Figure 3: Diving one level into cold code

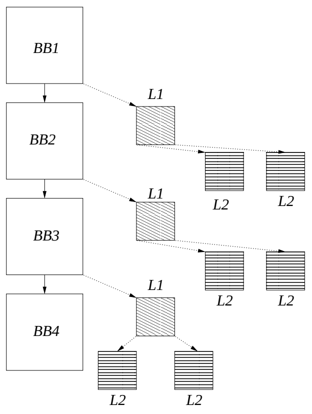


Figure 4: Diving two levels into cold code

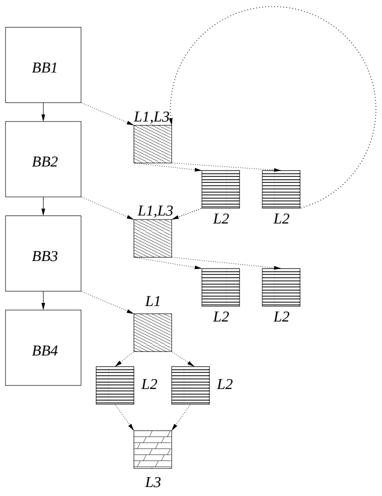


Figure 5: Diving three levels into cold code

If we reach a jump instruction that does not allow us to determine to target address, it is not possible to keep going with the analysis. The same happens with ret instructions, without looking at the stack. This put an extra cost to the cold code analysis.

In Figure 5 we can see that sometimes a deeper level can take us to a basic block at shallow levels. If we keep track of the information at each level entrance, we don't need to dive again, otherwise, we will be analyzing the same code again.

The blocks on the side exit of **BB1** go from level 2 to level 1 when trying to reach level 3. Sometimes they can reach the middle of a trace.

5 Experimental Results

Our experimental results are focused in the optimization proposed in the beginning of this paper, i.e. in finding available registers (dead registers) that can be used to replace memory accesses on a trace.

We implemented this technique into our DBT translating system, which performs code transformation from IA32 to IA32. Our DBT [6] runs on Linux and has its structure shown in Figure 6. It has a kernel module which is loaded and replaces the system call **execve**. Every time that a program executes, the module checks for a shared library in the same directory of the application, if this library is not found, then the original Linux execve is called. If the shared library is found, then it is loaded and starts the execution. This shared library is our DBT itself, which loads the application code and starts the translation process.

Our DBT has 3 main modules: *front-end*, *runtime* and *back-end*. The *front-end* module translates the application instructions and stores them into a code cache, controlling the program execution in this cache. The *runtime* module performs the communication between the DBT and OS, and between the application and the OS. To achieve that, it provides I/O interfaces, system calls, and support to system signals, dynamic shared objects loads and self-modifying code. The *front-end* and the *runtime* interact to handle system related features. The *front-end* is also responsible to select hot traces for runtime optimization by the *back-end*. The *back-end* module is responsible for trace optimization. It is able to dump hot-traces into files, so as to perform off-line optimization and profiling analysis.

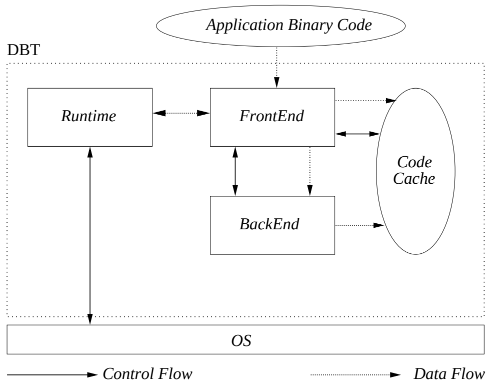


Figure 6: Our Dynamic Binary Translator Structure

5.1 Experiments

Our experiments used the SPEC CPU 2000 [1] benchmark. Programs were compiled with the Intel Compiler using the peak tuning and ref input. For each program the DBT identified hot traces, using MRET heuristics [4], and dove into cold code looking for available registers. The register set of x86 architecture creates some alias relationship among the registers. For example, a definition of register `EAX` also defines registers `AX`, `AH` and `AL`. A use of register `BH` implies in a use of register `EBX`. To handle these issues, we treat the registers as resources, and thus `EAX`, `AX` as well as the other registers are considered different resources.

In our experiments we analyze the set of registers presented in Table 1.

We analyzed every side exit of each trace constructed by our DBT. The set of experiments was performed using 8 dive levels. For the first run of SPEC, cold code analysis was done using just one dive level, a second run used two dive levels and so on, until we reach 8 dive levels. As shown in Table 1, at each side exit we are trying to determine the availability of 48 registers. The set of traces generated by our DBT produced a total of 240,612 side exits, so the maximum number of available registers is 11,549,376 registers.

In our implementation a basic block can be analyzed more than once because we do not keep track

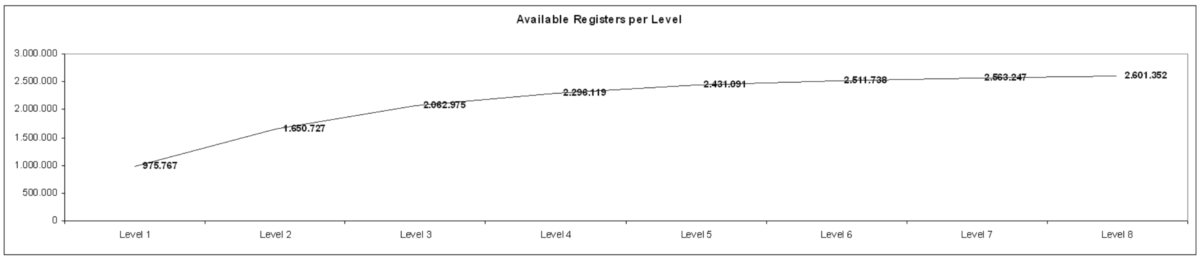


Figure 7: Registers available per Level

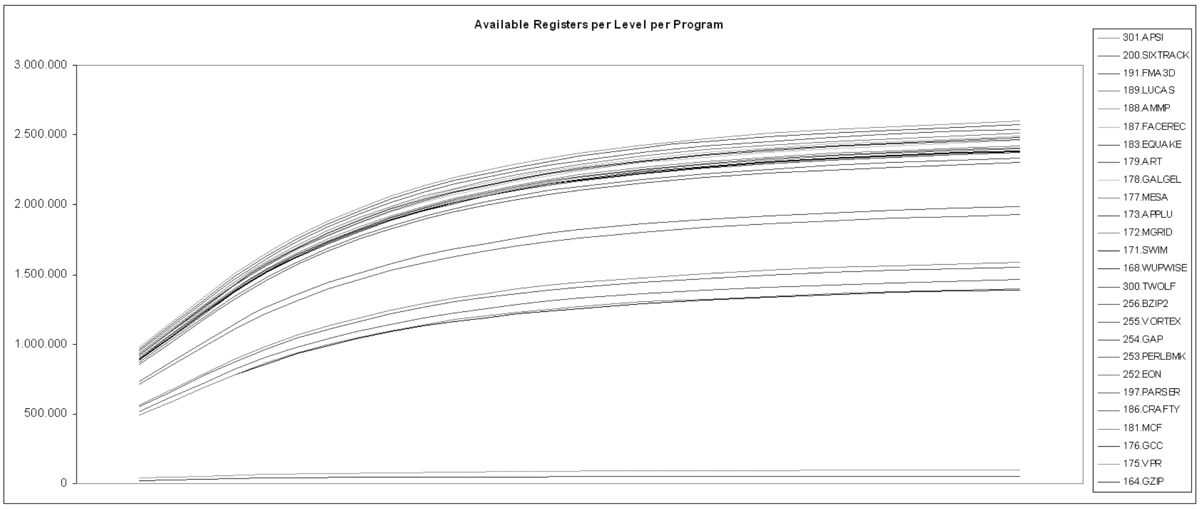


Figure 8: Registers available per Level per Program

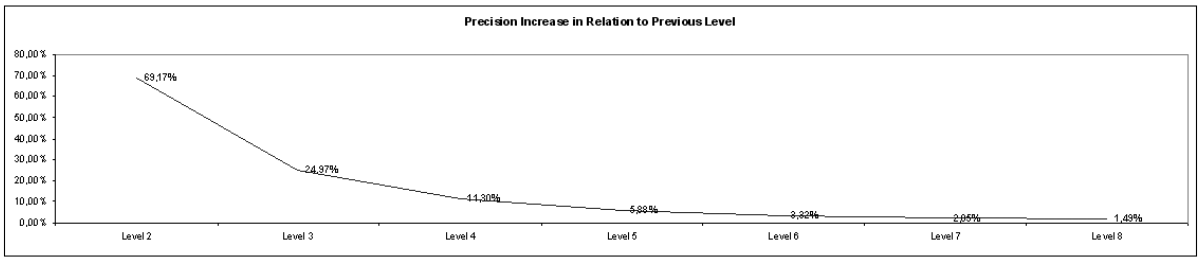


Figure 9: Increase of information accuracy

of the information collected at each block.

Figure 7 and Table 2 summarize the results that we found at each level. Diving 8 levels allowed us to discover that 22.52% of the registers are available (dead), that is, almost 1/4 of the whole register set. For an architecture with few registers, this could become an important resource which could be explored by a dynamic register allocator. Notice that we are discussing dead registers. If we try to determine unused live range intervals existing in live registers, this value could become larger than 22.52%.

By going beyond trace boundaries, cold code analysis can determine the live/dead status for almost all registers on a side exit, thus allowing the optimizer to make better choices of what to op-

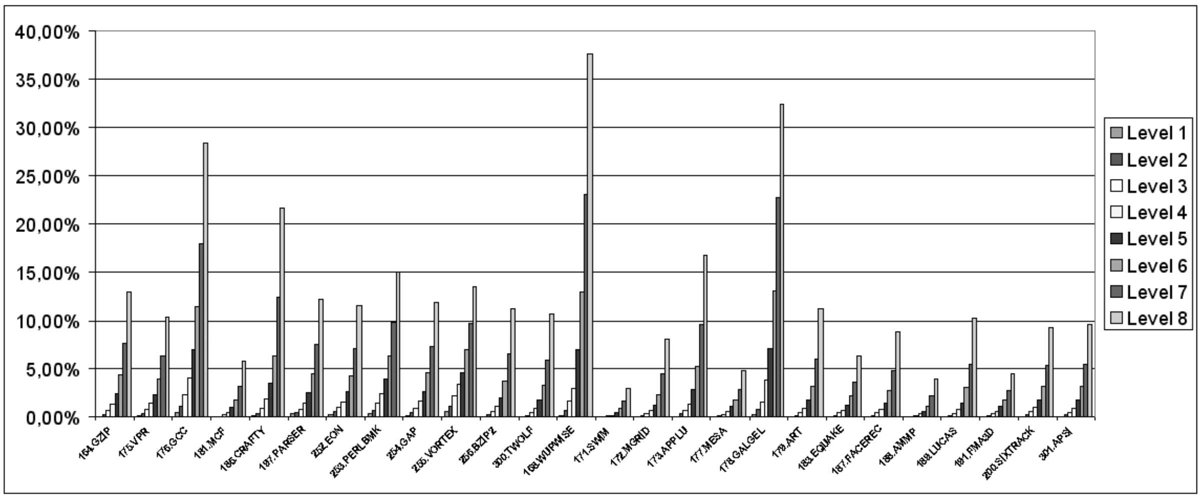


Figure 10: Cold Code Analysis Overhead

	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8
164.GZIP	0.11%	0.32%	0.75%	1.39%	2.48%	4.38%	7.63%	13.01%
175.VPR	0.17%	0.41%	0.86%	1.47%	2.34%	3.93%	6.42%	10.28%
176.GCC	0.49%	1.10%	2.31%	4.06%	6.91%	11.46%	17.96%	28.42%
181.MCF	0.09%	0.12%	0.27%	0.50%	1.00%	1.73%	3.23%	5.77%
186.CRAFTY	0.20%	0.41%	0.95%	1.88%	3.54%	6.33%	12.40%	21.62%
197.PARSER	0.40%	0.42%	0.83%	1.45%	2.57%	4.44%	7.45%	12.23%
252.EON	0.23%	0.58%	0.97%	1.59%	2.68%	4.30%	7.08%	11.51%
253.PERLBMK	0.34%	0.73%	1.43%	2.43%	3.94%	6.28%	9.76%	14.96%
254.GAP	0.17%	0.48%	0.93%	1.61%	2.66%	4.55%	7.27%	11.90%
255.VORTEX	0.51%	1.21%	2.17%	3.36%	4.69%	6.95%	9.70%	13.51%
256.BZIP2	0.14%	0.26%	0.61%	1.11%	2.02%	3.72%	6.58%	11.21%
300.TWOLF	0.11%	0.22%	0.47%	0.94%	1.78%	3.28%	5.97%	10.69%
168.WUPWISE	0.21%	0.76%	1.66%	3.05%	6.94%	13.00%	23.07%	37.59%
171.SWIM	0.01%	0.06%	0.14%	0.19%	0.47%	0.93%	1.67%	3.03%
172.MGRID	0.05%	0.16%	0.34%	0.69%	1.25%	2.40%	4.45%	8.05%
173.APPLU	0.13%	0.32%	0.69%	1.41%	2.84%	5.28%	9.57%	16.70%
177.MESA	0.05%	0.17%	0.24%	0.51%	1.07%	1.74%	2.91%	4.85%
178.GALGEL	0.29%	0.78%	1.58%	3.81%	7.15%	13.03%	22.76%	32.45%
179.ART	0.08%	0.16%	0.45%	0.92%	1.69%	3.19%	6.00%	11.23%
183.EQUAKE	0.07%	0.16%	0.44%	0.71%	1.26%	2.15%	3.65%	6.26%
187.FACEREC	0.04%	0.21%	0.42%	0.83%	1.48%	2.74%	4.86%	8.88%
188.AMMP	0.05%	0.10%	0.20%	0.36%	0.65%	1.22%	2.17%	3.88%
189.LUCAS	0.06%	0.19%	0.37%	0.83%	1.50%	3.09%	5.47%	10.27%
191.FMA3D	0.08%	0.19%	0.39%	0.67%	1.07%	1.73%	2.78%	4.47%
200.SIXTRACK	0.10%	0.23%	0.55%	1.01%	1.76%	3.16%	5.42%	9.33%
301.APSI	0.10%	0.24%	0.50%	0.94%	1.80%	3.18%	5.51%	9.57%

Table 4: Cold Code Analysis Overhead

Registers in x86 Architecture
EAX AX AH AL
EBX BX BH BL
ECX CX CH CL
EDX DX DH DL
EBP BP
ESI SI
EDI DI
ESP SP
MM0 MM1 MM2 MM3 MM4 MM5 MM6 MM7
XMM0 XMM1 XMM2 XMM3 XMM4 XMM5 XMM6 XMM7 XMM8 XMM9 XMM10 XMM11 XMM12 XMM13 XMM14 XMM15

Table 1: x86 registers

Available Registers	
Level 1	975,767
Level 2	1,650,727
Level 3	2,062,975
Level 4	2,296,119
Level 5	2,431,091
Level 6	2,511,738
Level 7	2,563,247
Level 8	2,601,352

Table 2: Available registers per level

Precision Increase	
Level 1	–
Level 2	69.17%
Level 3	24.97%
Level 4	11.30%
Level 5	5.88%
Level 6	3.32%
Level 7	2.05%
Level 8	1.49%

Table 3: Precision increase per level

timize or not. Notice that the registers that are considered available, are those that we can ensure that were assigned a value on the side exit. If a register never is used in a program, so this register will be considered as not available, because we will never find a instruction defining it.

Figure 9 and Table 3 give us an idea on how deep the analysis should dive. The percentage numbers are related to the previous level. From level 1 to level 2 the precision increase in the number of available registers was 69.17%, while from level 2 to level 3 the precision increase was 24.97%. This diminishing return behavior continues until we reach the last level (level 8). As shown, after level 4 or 5 the increase in the number of available registers is not large, meaning that for most of the purposes, diving until level 4 or 5 is enough to determine the dead/live status of most registers.

Table 4 and Figure 10 show the time execution overhead caused by our cold code analysis. The overhead corresponds to the amount of time spent by the analysis, with respect to the whole program execution time. As shown, until level 3 the overhead is below 1%, in 21 out of the 26 programs available. As expected, Figure 10 shows that the cost is exponential, and thus the overhead for the final levels is very high. This probably happens because some basic block may be analyzed several times, as for example, in the case of a loop. An implementation which considers code re-analysis could certainly decrease this overhead.

6 Conclusion

In this paper we examined a way of increasing data flow information available on traces constructed by DBTs. The process consists in analyzing cold code, beyond program hot traces, to obtain the desired information. As far as we know, no DBT employs this technique to analyse code outside traces.

We implemented a data flow analysis to determine the set of available registers at each side exit of a trace. To increase the precision of the data flow information, we performed cold code analysis, running the benchmark several times and analyzing a large amount of cold code at each run.

The tests were performed using the whole SPEC CPU 2000 benchmark and the results allowed us to conclude that almost 1/4 of the x86 architecture registers are dead on trace exits. Moreover, the results also showed that there is no need to go far into cold code in order to achieve this information: a level 4/5 dive is enough.

Regarding the example presented in Figure 1, we were able to determine that register `esi` was dead on the side exits at instructions 8, 14, 24 enabling us to perform the optimization.

For the future we plan to change our code analysis technique so as to avoid the overhead of repeating basic block analysis. We also intend to test it in other types of data flow analysis, like for example register flag detection, which is a very relevant information in other code optimizations.

References

[1] *Standard Performance Evaluation Corporation (SPEC).* http://www.spec.org.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[3] E.R. Altman, D. Kaeli, and Y. Sheffer. Welcome to the opportunities of binary translation. *IEEE Computer*, 33(3):40–45, Mar 2000.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[5] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigal Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems, 2003.

[6] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, 2000.

[9] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Trans. Comput.*, 50(6):529–548, 2001.

[10] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[11] David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving region selection in dynamic optimization systems. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–154, Washington, DC, USA, 2005. IEEE Computer Society.

[12] J.D. Hiser, N. Kumar, Min Zhao, Shukang Zhou, B.R. Childers, J.W. Davidson, and M.L. Soffa. Techniques and tools for dynamic optimization. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, 25-29 April 2006.

[13] Bich C. Le. An out-of-order execution technique for runtime binary translators. *SIGOPS Oper. Syst. Rev.*, 32(5):151–158, 1998.

[14] J. Lu, H. Chen, P. Yew, and W. Hsu. Design and implementation of a lightweight dynamic optimization system, 2004.

[15] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 1997.

[16] M. Probst, A. Krall, and B. Scholz. Register liveness analysis for optimizing dynamic binary translation. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 35, Washington, DC, USA, 2002. IEEE Computer Society.

Impact of Dynamic Binary Translators on Security

Yu-Yuan Chen[*], Youfeng Wu[+], Shiliang Hu[+] and Ruby B. Lee[*]
[*]Princeton University and [+]Intel

Abstract: *Dynamic Binary Translators (DBTs) allow programs written for a specific platform to be run on other platforms without the need for recompilation. They allow legacy software to be run on newer hardware architectures, they can perform dynamic optimization of software, and virtualization. Other benefits include providing enhanced security by dynamically adding checking code around possible software security vulnerabilities. However, before this is even considered, there are two aspects of DBTs that must first be addressed. First, are software protections provided by the application preserved under the runtime translation and optimizations done by a DBT? Will they be optimized out? We study a range of software protection techniques including Stackshield, Propolice and Stackguard, Libsafe, address space randomization, checksumming, watermarking, system call sandboxing, authenticated system calls, code obsfucation and morphing, anti-debugging, instruction-set randomization, and proof carrying code. Second, how is the DBT itself protected? How is its code cache protected? Without adequate protection, a DBT can be exploited by an attacker to cause disastrous system consequences. We propose three solutions. One solution adds a small set of hardware features to the microprocessor, as defined by the Secret Protection (SP) architecture, to protect the DBT and its code cache.*

1. Introduction

Dynamic Binary Translation (DBT) is a software technology that allows programs written for a specific platform to be run on other platforms without the need for recompilation. This not only introduces the opportunity for legacy software to be run on newer hardware architectures, but also enables dynamic optimization of software. The key capability of DBT is that it can transform and optimize any binary before it is executed on the processor, thus enabling many new features on existing binaries, such as virtualization [1], redundant execution for reliability [2], information flow tracking for security [3], dynamic voltage-frequency scaling for power management [4], etc. In this paper, we study StarDBT [5], an application-level DBT (running in user space), to see whether the translation and optimizations it performs affect the security of the original programs or not, and to examine ways of protecting the DBT itself.

When security protection is provided by software, and if this code undergoes standard DBT translation and optimizations, what happens to the security protections? Do optimizations done by a DBT preserve the security protections provided in the original code? Will they be optimized out? We study a range of software security protections to check if the translation and optimizations performed by a DBT affect the security provided to the original program.

A DBT's resources can be used to increase system security. For example, it could add checking code around potential software security vulnerabilities, such as locations where buffer overflow attacks might occur. A DBT could also add some randomization techniques (e.g., address or instruction set randomization) to thwart attacks. However, as the DBT is only another layer of software, if the attacker knows the existence of the DBT, it would not be difficult to attack the DBT itself and the DBT itself becomes a target.

Especially for an application-level DBT which runs in user space, we need to study how the DBT itself can be protected. How is its code cache protected? Without adequate protection, a DBT can be exploited by an attacker to cause serious system consequences. We discuss solutions based on moving the DBT to the system level, or alternatively using a small set of hardware features (as in the Secret Protection architecture **[6,7]** to protect the DBT and its code cache.

In section 2, we discuss different types of software security techniques that can be provided to an application program. In section 3, we describe our experimental setup and methodology for testing whether the security provided by these software security techniques is preserved under DBT translation and optimizations. In section 4, we provide the results of our tests and explain why the StarDBT we tested preserved the security properties provided, or not. We also suggest why other types of DBT might fail to preserve security properties, and also how to improve the DBT to not perturb security properties provided by application or system software. In section 5, we propose solutions for protecting the DBT, including a hardware technique that can protect the DBT itself, while leaving it in application space. In section 6, we present our conclusions and suggestions for future work.

2. Software Security Techniques

In order to see if a Dynamic Binary Translator will affect the security protection provided by different software products and techniques, we need to first understand what *types* of protection are provided and *how* they are provided. Hence, we illustrate the variety of software techniques that may be used to enhance security by preventing software attacks. These include preventing buffer overflow attacks, detecting hostile code modification attacks, checking system calls, mitigating reverse engineering, and checking security assertions. We describe a dozen different software techniques under these five types of protection. (Note that some of these techniques could be classified under more than one of these five categories.)

2.1. Preventing Buffer Overflow attacks

Perhaps the most common attack is the buffer overflow attack. There are many different software solutions for this, but it still remains one of the most prevalent software vulnerabilities in both legacy and new code. Buffer overflow attacks can cause return address corruption or function pointer corruption, leading to some forms of dynamic hostile code insertion or illegitimate library calls.

A buffer overflow attack occurs when data is written into a variable without checking if the length of this data exceeds the size of the variable. This can over-write other data, for example, the return address of a procedure call. This is illustrated in Figure 1, which shows a buffer overflow attack with ***return address corruption*** in the stack.

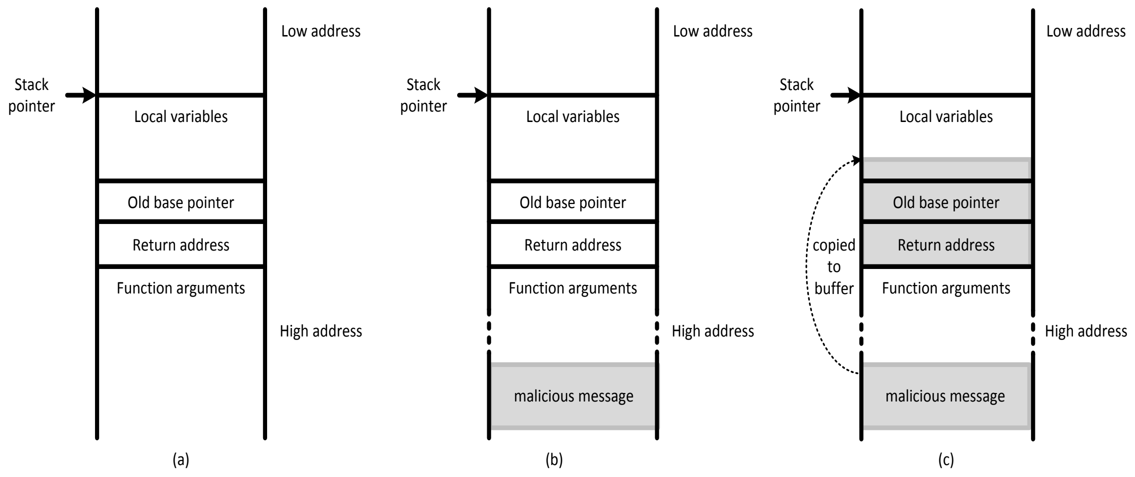


Figure 1: (a) a typical stack frame, (b) malicious message received from the network, (c) stack buffer overflow when message copied to local buffer without checking the size of the message.

By overwriting the return address, the attacker can alter the program control flow on a procedure return to jump to an address in the stack where hostile code has also been inserted by the buffer overflow. This hostile code, to which the program jumps, can gain root shell access. After this hostile code is executed, another jump instruction is executed to the original return address.

Alternately, a ***return to libc attack*** can also be launched by a buffer overflow attack, where the return address is changed to a legitimate library call entry-point. This causes a library call – that should not otherwise be called at that point in the program – before returning to the original return address. The return-to-libc attack is also based on overwriting return addresses on the stack; however, instead of transferring control to malicious inserted code, return-to-libc returns to existing functions in the system while overwriting the contents on the stack to feed bogus parameters for the existing functions to perform malicious operations for the attacker. Since return-to-libc does not require injecting new code into the system, even if the overflow buffer is not large enough for malicious shell code, it can still perform a successful attack. However, the attacker needs to figure out the address of the desired libc function such as `system()` or `printf()` to perform the attack.

a. Stackshield

Stackshield [8] is a software mechanism to prevent return address corruption by a buffer overflow attack. It is a gcc extension that maintains a (software) shadow stack for return addresses. It copies the return address to the beginning of the DATA segment, a location that cannot be overflowed, on function prologs, and checks if the two values of the return address are different on function epilogs.

b. Propolice and Stackguard

Propolice [9] is a patch to gcc. It inserts canaries to check for buffer overflow, as does StackGuard [10], to protect the return address on the stack. A canary is a bird used in mines to detect toxic air in the mine – the canary will die first from toxic air. Here, a "canary" is a random value inserted in the stack, such that any buffer overflow will change the value of the canary before damaging the return address; hence on a procedure return, the value of the canary is first checked to see to see if it has changed - if not changed, then the return address can be trusted. Additionally, Propolice reorders the declaration of local variables to place buffers before pointers to avoid the corruption of pointers that could be used to point to arbitrary memory locations.

c. Libsafe

Libsafe [11] intercepts known-vulnerable library calls and substitutes them with a safe version to ensure no that no buffer overflows can occur. This prevents buffer overflow vulnerabilities and attacks in these "safe" library calls. Libsafe does not require source code or recompilation and checks for violations at runtime. It is loaded as a linked library during runtime and checks a few "unsafe" functions, e.g. `strcpy(), strcat(), scanf(), memcpy()` etc. Libsafe intercepts these known-vulnerable library calls and substitutes them with a safe version to ensure no overflowing can occur.

d. Address space randomization

Address space randomization [12] is a technique devised to thwart buffer overflow attacks by introducing randomness into the memory locations of certain system components: stack, heap, BSS, etc. For example, the return-to-libc attack, discussed above, needs to know the virtual address of the libc function to be written into a function pointer or return address. If the base address of the memory segment containing libc is randomized, then the success rate of such an attack significantly decreases. Figure 2 (adapted from [12]) shows an example of the address space layout before and after randomization is applied. Linux application program code usually starts from 0x08048000, followed by the data segment that stores all initialized data and the BSS segment that stores uninitialized data. Dynamic Shared Objects (DSO) segments are where the shared libraries are placed.

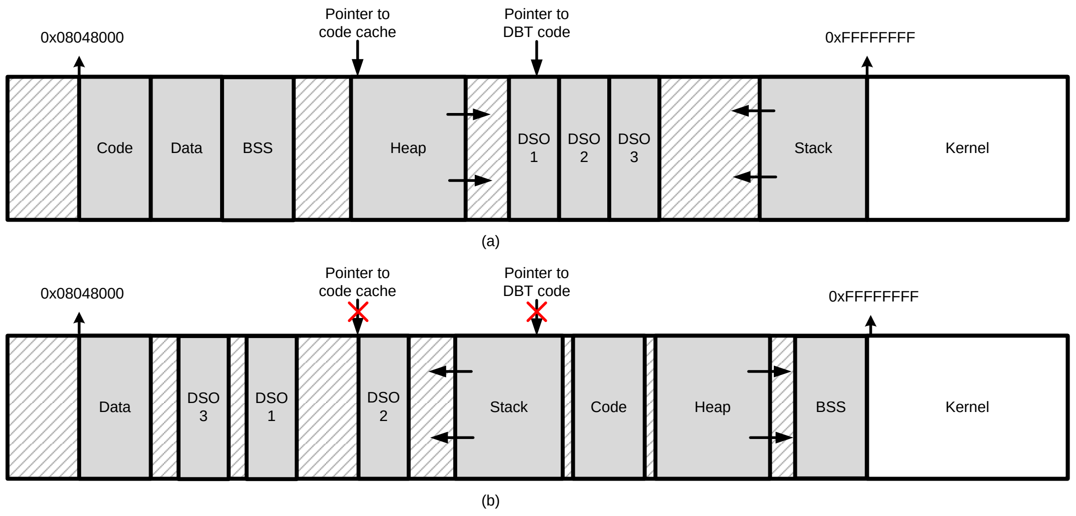


Figure 2: Address space randomization (a) typical address space layout. (b) layout after randomization.

For the StarDBT that we test, the StarDBT program is loaded as a dynamic library, residing in one of the DSO segments. The code cache of the application-level StarDBT is allocated in the user heap space.

2.2. Detecting hostile code modification attacks

Hostile code injection into a program, or code modification, can be detected by checking that the code has not changed from a known good version, We call this "static" code modification if it occurs when the program is not active, e.g., while the program was residing on disk. Checking that the code has not been maliciously modified is typically done only on program launch. Dynamic code modification occurs when the program is active – during runtime. The buffer overflow attacks, described earlier, enable dynamic hostile code insertion and modification.

The software Code Integrity Checking techniques described here are typically used to detect static code modification. However, they could potentially be adapted to detect some forms of dynamic code modification. We describe two different software techniques which add software to re-compute and check a checksum (or hash) on the code, or check for an inserted watermark.

e. Self-checksum

Self-checksum [13] is a technique where the program checksums itself to make sure the integrity of the protected part of the program is not compromised. A code producer would first compute a checksum (or hash) over the critical parts of the program that is to be protected, and insert that information into the program itself. In addition, the code producer inserts hash calculation and verification code throughout the program execution, to calculate the hash of the critical parts and compare with the inserted known "good" hash of those critical parts of the program. Therefore, any modification to the critical parts of the program will be detected.

Note that checksumming is not cryptographically strong, and can easily be defeated by an attacker, who can change the code then change the checksum value which is kept in the program itself. A better technique is to use a keyed-hash, or Message Authentication Code (MAC) rather than a checksum. Such a cryptographic hash function has strong collision resistance (unlike a checksum). Further, a keyed hash function (MAC) prevents an attacker from generating a new MAC value to correspond to the changes he makes in the code. This is because he does not have the key, which is kept securely elsewhere, and not with the program itself.

f. Watermarking

Watermarking [14] exploits the inherent redundancy in how instructions may be specified to encode a secret message in the program's binary. The exploitation is based on the fact that a given operation can be represented in many ways. For example, adding the value 50 to register `eax` could be represented as either "`add %eax, $50`" or "`sub %eax, &-50`". By replacing selected original machine instructions with functionally-equivalent instructions, we can encode a secret message into the program binary and later decode it to extract the message. The protection given by using this technique is that, if an attacker modified the disk image of the binary by inserting malicious code, the watermarked message would be destroyed and we would be able to detect that the binary had been modified. The watermarking technique works as follows: a checking program will first read in another program binary in which the watermark is going to be inserted, and change the instructions with functionally-equivalent instructions to encode a secret message. Afterwards, the checking program could read in the watermarked program's binary to decode the secret message. The watermarked program is not affected in terms of program execution result and the checking program only reads the watermarked program as static input.

2.3. Protecting System Calls

A common characteristic of many attacks is to exploit the system call interface to perform malicious actions. Incorrectly invoking a system call can do a great deal of damage to the system. For example, if a system call is invoked at the wrong time or in the wrong sequence, the system might be compromised. Forrest et al [15] confirmed that due to the different system call behavior when the system is attacked by the `sunsendmailcp` script [16], a local user may gain root access to the system. Another way of compromising a system call is by modifying system call parameters [17]. For example, the actions taken by a system call that normally retrieves the current user's file could be changed to access another user's file by including path traversal "../" characters as part of a filename request.

g. Sandboxing System calls

All sensitive system resources are usually accessed through the OS system call interface, e.g. file operations, network sockets, etc. System call sandboxing monitors all system calls made by an application by inserting checking instructions before and after the monitored system call and blocking the call if it violates some predefined security policy.

h. Authenticated system call

One technique proposed to protect system calls is the authenticated system call [18]. This transforms a system call to include extra arguments that specify the policy of that call and a MAC to guarantee the integrity of that policy. The policy specifies the allowed system call name, call site, control flow, and constant parameter values to be constrained for a system call. The policy is encoded into a string, and a Message Authentication Code (MAC), or keyed hash, is computed over the string with a cryptographic key that is available only to the kernel. This allows detection of any tampering of the string. The encoded string and its MAC are added to the system call as extra arguments. Before the kernel executes the system call, the MAC is re-calculated and verified to guarantee the integrity of the system call.

Before any program in the system is installed, it has to go through a trusted installer program, which extracts the allowed behavior for each system call by static analysis and then rewrites the binary to insert the policy and MAC. At runtime, each system call is intercepted and the MAC is checked to verify the integrity of the system call. If the behavior of the system call matches the policy, it is allowed; otherwise it is rejected and the process is terminated. Therefore, only trusted programs that have the authenticated system calls and pass the integrity check can execute on the system; regular programs cannot run on the system.

2.4. Mitigating reverse engineering

An attacker often first performs some reverse engineering to see if the software has exploitable security vulnerabilities, what and where they are, and where certain information is stored. Reverse engineering may also be performed to understand how secret or proprietary software works.

i. Code Obfuscation and morphing

Obfuscated code [19,20,21] is source code that has been transformed to make it harder to read and analyze, in order to prevent reverse-engineering of the code. Obfuscated code has the same function as normal code, just that the code is hard to understand without some processing. A simple code example of C that prints out prime numbers less than 100 is given in obfuscated form in Figure 3. Detailed directions on how to transform the original source code to obfuscated code are available, e.g., in [22].

```
_(__,___,____){___/__<=1?_(__,___+1,____):!(___%__)?_(__,___+1,0):___%__==___/
__&&!____?(printf("%d\t",___/__),_(__,___+1,0)):___%__>1&&___%__<___/__?_(__,1+
___,____+!(___/__%(___%__))):___<__*__?_(__,___+1,____):0;}main(){_(100,0,0);}
```

Figure 3: code obfuscation example for a C program

Some code obfuscation techniques simply add unrelated code into the binary. Other obfuscation techniques include the following:

- Lexical Transformations scramble the identifiers used in the programs, making it difficult to get semantic context from the identifiers, e.g., deposit (amount) is transformed to a (b).
- Control flow obfuscations [23] introduce redundant predicates whose outcome are known at the obfuscation time, but are difficult to deduce. The resilience of these obfuscations is directly related to the resilience of the opaque predicates on which they rely. Opaque predictors based on the intractability of static pointer analysis problem have been proposed.
- Data obfuscation [24] is achieved by modifying data structures to obscure their semantic meaning, e.g., variable splitting. Bool A -> int a1, a2. A = TRUE -> a1 = 0, a2 = 1, A = FALSE -> a1 = 1, a2 = 0.
- Inter-module call relation obfuscation [20] makes it a very complex problem to determine the address a function pointer points to, in the presence of arrays of function pointers.

Code morphing is code obfuscation done to the object code to deter disassembly and reverse engineering and analysis of object code. This technique breaks up the protected code into several processor commands and replaces them by others, while maintaining the original function. It turns binary code into an undecipherable mess that is not like normal compiled code. An example of code morphing is given in [25].

j. Anti-debugging techniques

Some hackers use debugging software to gain knowledge of how programs work, thereby allowing the hacker to find vulnerabilities that he/she can exploit. Therefore, commercial software often employ some anti-debugging techniques to prevent hackers from gaining this knowledge. There are various anti-debugging techniques, e.g. detecting SoftICE (a kernel-mode debugger for Windows), detecting VMware [26], searching for names installed by SoftICE in memory, etc. In general, these anti-debugging methods are simple instructions and do not depend on any dynamic behavior. However, one particular anti-debugging technique needs special attention: it checks execution timing to determine if being debugged.

```
xor ecx,ecx
rdtsc
add ecx,eax
…
rdtsc
sub eax,ecx
cmp eax,0FFF
jb short _FAST_ENOUGH
…
_FAST_ENOUGH: -> Code continues here
```

Figure 4: Timing-based anti-debugging detection

In Figure 4 the `rdtsc` instruction reads the time stamp counter and returns a 64-bit value in register `eax` that represents the count of ticks from processor reset. When being debugged, i.e. stepping through the program, the distance between the values in `eax` will be higher when the program is being debugged than when it is not being debugged. For emulators or interpreters, the overhead could be great enough to cause the program to always exit.

k. Instruction set randomizations

Randomizing the instruction set (i.e., the instruction encodings) [27] can deter remote attacks assuming that the attacker cannot obtain the encryption key that is used for randomizing. The idea is to randomize the entire instruction by XOR'ing the instruction with a key, as illustrated in Figure 5 The decoding (de-randomizing) to get the plaintext instruction from the encoded (randomized) instruction can be done in hardware or in software.

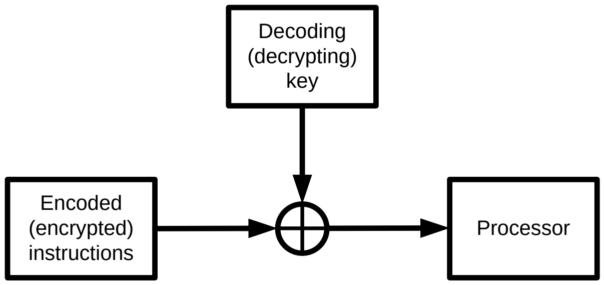


Figure 5: Decoding instructions in instruction set randomization (from [27])

Note that this form of instruction-set randomization is a very insecure form of One-Time Pad (OTP) encryption, since the encoding key is repeated over and over in the key pad of the OTP cipher. It would be rather easy for an attacker to break this encoding and discover the key. However, the purpose here is randomization to introduce diversity to thwart attacks and deter reverse engineering, rather than strong encryption.

2.5. Checking Security Assertions

l. Proof-Carrying Code

Proof-carrying code (PCC) [28] is code that carries an attached "safety proof" to prove that the code abides by certain security policies. The program binary is just a normal binary. The attached proof is written in Logical Framework (LF) and is encoded to be later checked by a "proof checker". All of the proof generation, encoding and proof checking are done statically—before the program is actually executed. Once the program passes the proof checker, the program will be executed as "trusted".

3. Experimental Setup and Methodology

We investigated several of these software protection techniques with and without using StarDBT. StarDBT is an application-level dynamic binary translator, and like most other sophisticated DBT systems [5, 29,30], it adopts the two-phase translation strategy. It uses a simple fast translator for cold code translation and once a workload hotspot is detected, it applies optimizations. The simple and fast translation tries to generate target code with minimal runtime overhead. Control transfer instructions such as branches, function calls and returns need to be rewritten. This involves some translation table lookup and dispatch code. The simply translated code can also be instrumented with profiling code to collect block frequency information to recognize program hotspots for optimizations.

For program hotspots, StarDBT forms straight-line traces (superblocks) to optimize, such as code layout and partial redundancy elimination. The translators place generated code in the code cache for later reuse.

The runtime dispatcher maintains a translation lookup table. The execution of the native IA32 code is achieved via the execution of the translated code and dispatching in the runtime system.

To verify that StarDBT does not compromise the security enhancement in the applications, we run the security-enhanced applications by themselves and also with StarDBT, under security attacks, such as the "attack benchmark" suite **[**31**]**. If the application, with and without DBT, can detect the same attacks, we consider that StarDBT does not compromise the security. Otherwise, we analyze the compromised security situation and determine how we may enhance StarDBT to prevent the situation from happening.

Since StarDBT is just a particular implementation of dynamic binary translation, some other types of DBT may compromise a security feature that is preserved by StarDBT, or vice versa. We identify this possibility by detailed analysis of the software security enhancement techniques (discussed in section 2), and point out the possible DBT translation and optimizations that may compromise the security provided.

4. Experimental results

We present the results of our experiments, described in the previous section, under three categories. Table 1 summarizes our overall findings. In section 4.1, we list the software protection techniques (DBT friendly techniques) that are not affected by DBT optimizations. In section 4.2, we list those software protections (DBT-tolerable techniques) that are preserved under certain conditions, and describe these conditions. In section 4.3, we list those software protections (DBT-troublesome techniques) that are not preserved by an application-level DBT, but can be preserved if the DBT is moved to the system level. In each case, we explain why the translation and optimizations done by the DBT we tested either preserved the software protections or not. We also discuss what might happen with other types of DBTs. We base our explanations on the following fundamental requirements for a DBT.

1) DBT must translate a program into a semantically equivalent version, in terms of all visible system states, including the detection of self-modifying code and retranslate them when necessary.
2) DBT must preserve the relative code and data layout of the original program in each of the data, bss, stack, and heap segments. This is required as some program may have built the relative offset in the program logic, although the runtime system is allowed to load each segment in a statically unknown location.
3) DBT must preserve self-referencing, i.e. the original program must be kept untouched and any self-reference to the original program during the translated execution must be faithfully provided.
4) DBT must preserve precise exceptions of the original program, i.e. all the executions must occur in the same order and the same places as in the original execution.

4.1 DBT-friendly techniques

For **Stackshield,** the shadow return stack value and the checking code are inserted statically. Their operations are part of the required behavior of the transformed program, and DBT will preserve the correct execution of these statically-inserted transformations. Specifically, the values in the shadow return stack cannot be determined as redundant by DBT if there is any code that may overrun the original stack, and therefore DBT will preserve the checking code, and will translate and execute it correctly.

Similarly, for **Propolice and Stackguard**, the special "canary" data value inserted onto the stack and checking code are inserted statically. If there is any chance of stack overrun, DBT cannot determine that the checking of the canary value is redundant computation, and thus is required to guarantee the correct execution of the input binary with its static Propolice or Stackguard additions.

Libsafe may substitute one library call with another dynamically at program load time. In this case, DBT will only translate the library calls after the substitution and thus a program under Libsafe will work correctly. If Libsafe substitutes library calls during program execution, DBT will detect the substitution as self-modifying code, and will still be able to translate and execute the newly substituted library code correctly.

For **address space randomization**, since the randomization is done at program loading time, DBT is required to *not change* the original memory layout within each segment, so the randomized address space will not be affected by DBT operations. For an application-level 32-bit DBT, the DBT needs to place its code cache in the same address space as the user application. There is a chance that the code cache may conflict with some of the user code/data. When this happens, DBT will relocate itself to another unused space and thus there will be no problem.

For **watermarking,** a secret message is embedded in the binary's disk image. Since the watermark checking program reads the image to decode the secret message and DBT translation will not modify the disk image at all, watermarking will not be affected by DBT.

Sandboxing system calls involves inserting checking instructions before and after the monitored system calls. Since the checking code is inserted for a valid purpose, DBT will not remove the checking code and so it will not compromise the protection.

For **Proof-Carrying Code**, if this proof-carrying code remains static, it will not be affected at all by DBT. Since all of the proof generation, encoding and proof checking are done statically -- before the program is actually executed, a DBT would not interfere with any step within the PCC framework.

Table 1: Summary of Preservation of Software Security Technique's protections under DBT

Software protection technique	Execution with DBT
Preventing Bufffer Overflow attacks	
a. Stackshield	OK
b. Propolice and Stackguard	OK
c. Libsafe	OK
d. Address space randomization	OK
Detecting Hostile Code Modification attacks	
e. Self-checksum (and self-MAC)	OK if self-referencing property guaranteed
f. Watermarking	OK
Protecting System Calls	
g. System call sandboxing	OK
h. Authenticated system call	This could be affected
Mitigating Reverse Engineering	
i. Code obfuscation and code morphing	May make the code less obfuscated
j. Anti-debugging	OK, except for some timing-based methods
k. Instruction set randomizations	DBT may not work with HW decryption of instructions; OK with SW decryption.
Checking Security Assertions	
l. Proof-carrying code	OK

4.2. DBT-tolerable techniques

Self-checksum should not be affected by DBT as the checksum is validated by the program itself and DBT ensures that all self-referencing refer to the original binary, not the translated code. However, for some DBTs that patch the input binary to link with the optimized code [32], the checksum may change because the patched code is different from the original code, so it will change the checksum. However, product quality DBT is required to support correct self-referencing at all times so this should not be a problem for a product quality DBT.

4.4. DBT-troublesome techniques

Authenticated system call replaces original system calls at program installation time with new system calls carrying additional parameters for authentication by the OS syscall checker. For binaries running under DBT, the system call is made from the code cache instead of the original code address. Furthermore, DBT may perform branch optimization which may result in a different control flow. If the policy string takes runtime call sites and control flow into consideration, this could be affected by DBT.

For **code obsfucation**, DBT nay affect some methods, especially the ones that simply add unrelated, redundant code into the binary. Since a DBT may perform sophisticated optimizations, it may optimize such code away and thus undo the obfuscation effect. The good news is that the recent code obfuscation techniques also try to make the code hard to optimize, and hence the obfuscation performed by the tools cannot be easily reversed by StarDBT.

In particular, the four types of obfuscations discussed in section 2.4 will not be compromised by DBT. Lexical transformations make it difficult to get semantic context from the identifiers -- but since this does not change the binary code, DBT will not affect it. Control flow obfuscations introduce redundant or opaque predicates -- but since these are based on the intractability of the static pointer analysis problem, DBT will not be able to compromise this. Data obfuscation is achieved by modifying data structures to obscure their semantic meaning -- but since DBT does not change data layout, it will not affect this transformation. (The variables are split statically in the example in section 2.4, and hence DBT will not impact this.) Inter-module call relation obfuscation makes it hard to determine the address a function pointer points to -- but since “point-to” analysis is difficult to perform at runtime by DBT, this technique will also not be affected by DBT.

For **code morphing**, DBT could, in theory, optimize the morphed code and revert it back to more readable binary instruction sequences. Therefore, this technique is not fundamentally secure under DBT code transformation that is not aware of the morphing. However, in practice; the binary codes are typically morphed in such a way that it is not easily optimized away by a fast runtime DBT optimizer.

For **anti-debugging**, since these methods consist of simple instructions and do not depend on any dynamic behavior, they should not be affected by DBT translation and optimizations. For anti-debugging techniques that check execution timing, it is possible that DBT may translate the code into a code sequence with a different timing. However, since the timing constraints for detecting a debugging event is usually quite large and DBT does not change the execution timing much, this technique should work with DBT. Note that timing-based anti-debugging techniques may result in false positives anyway -- if events like interrupts or context switching occur. Hence, DBT will not make this situation worse.

Instruction set randomization can deter remote attacks assuming that the attacker cannot obtain the encryption key that is used to randomize the instructions. Whether DBT will affect the security of this technique depends on how the randomization system is designed. If the system’s decryption is done in hardware, the encrypted code will be fed into the decoder of the DBT, causing the DBT to fail completely, and the program will not work normally. The DBT can just leave the code un-translated and un-optimized – which may be OK in some situations, but not in cases where translation is necessary for correct execution of legacy code on a new machine. On the other hand, if the decryption is done in software, DBT can use the already decrypted instructions and the execution will not be affected. However, the DBT must be trusted to see the decrypted code, or even to have access to the secret key for decryption.

If a DBT is to co-exist with a system enhanced with instruction set randomization, both DBT and the decryption engine of the randomization should be incorporated into the hardware (or firmware) to be truly protected and, furthermore, to allow the DBT to use the decrypted machine code for its translation and optimizations.

4.6. Summary of Results

Although the 12 cases examined above are not exhaustive, they are representative and do include some of the most recent software protection techniques. With the results and discussion given in these cases, it seems that, as long as the implementation of the DBT is transparent and correct, most of the software protection mechanisms are preserved by DBT, except for a few cases where side-channel information or hardware is used for protection.

We make the following general observations based on our experimental results and analysis:

1) If the software protection is done statically, or at load time, then the DBT does not affect it – unless redundant information is added that the DBT can easily detect as redundant and may optimize away. (This is true for some simple, and probably less effective, types of code obsfucation and morphing.)
2) The software protections that operate at runtime should not be affected by DBTs if they are executed by the protected application program itself – this is due to the guarantee of correct self-referencing behavior that all product-quality DBTs should provide.
3) However, if the checking of the application program to be protected is done by a different program at runtime, based on the runtime behavior of the protected application code, then this can be problematic for the DBT since the dynamic code executed may be different from the application's static code. (This is the case for the time-dependent anti-debugging technique and for some types of authenticated system calls.)
4) If the instructions are transformed into a form not recognizable by the DBT, it would not work. Otherwise, the DBT has to be disabled, or it has to be trusted to see the decrypted instructions, or the key for decryption, before it can apply its translations and optimizations.

5. Protecting the DBT itself from attackers

An application-level (or user-level) DBT is as vulnerable as the application it protects, since it is just another level of application itself. It could potentially cause the application to be compromised without being detected if the DBT does not implement any protection to its code cache, or if the DBT's code itself is changed by an attacker. For example [33] reports a vulnerability with DBT via interprocess communications.

Alternative solutions to protecting the DBT's code are:

- Move the entire DBT to the kernel to get a system-level DBT
- Integrate DBT as a part of the hardware or firmware component
- Use Secret-Protection (SP) hardware mechanisms [6,7] to protect the DBT itself as a Trusted Software Module.

5.1. System-level DBT

The DBT can be integrated as part of the OS kernel, i.e., the DBT runs as a kernel module and the code cache is only written in kernel mode by the DBT module. The code cache can be mapped read-only and/or execute-only into user space. Thus, both the DBT module and code cache are protected by the OS kernel. However, if the kernel itself is not protected, then it can also be compromised by an attacker. Especially as commodity OS are getting increasingly larger and more complex, it is hard to guarantee that the OS is free from bugs and software security vulnerabilities that can be exploited by attackers.

Alternatively, if a trusted Virtual Machine Monitor (VMM) layer is present in the system, a system-level DBT can reside there. Since this VMM will run at the highest privilege level, we assume that the enhanced hierarchical ring protection mechanism present in the processor ISA (e.g., using VT) can be used to protect both the VMM and the DBT. Since VMM is typically much smaller than OS kernels, it should, in theory, be less vulnerable to attacks.

5.2. Integrate DBT as part of the hardware or firmware component

A DBT system can also be integrated as part of the processor or platform microarchitecture, such as the Transmeta code morphing software [34]. In this case the DBT is coded in internal implementation ISA and the translated code is placed in memory hidden inside the processor. Thus DBT will be as trustable as the processor itself. .

5.3. Use SP architecture to protect the DBT

As an application layer, DBT is as vulnerable to attacks as any application. We propose that a small set of hardware-software mechanisms, called the Secret Protection (SP) architecture [6,7], can be used to protect the DBT as a trusted software module (TSM).

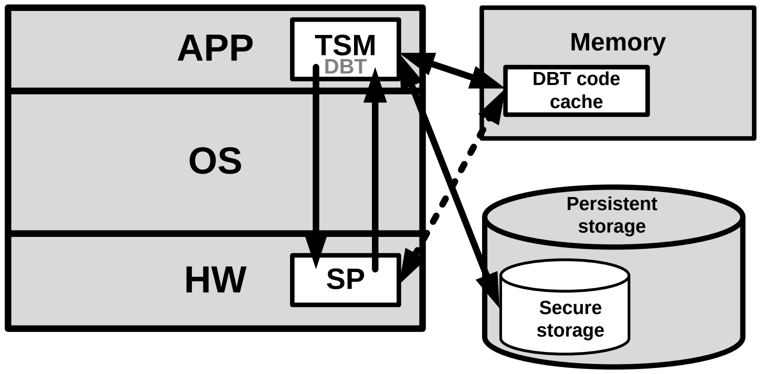


Figure 6: SP protection of DBT and code cache (white = the trusted parts.).

Figure 6 shows a conceptual view of the SP architecture consisting of a Trusted Software Module (TSM) in application space directly protected by hardware (SP) mechanisms in the microprocessor, even if the operating system has been compromised. With the size and complexity of commodity operating systems, this is not unlikely. SP architecture provides 3 main protections: (1) The TSM has access to pervasive secure storage. (2) SP hardware provides a secure execution environment for the TSM by protecting the intermediate data the TSM generates during its execution. (3) The TSM's code is integrity protected, so it cannot be changed by an attacker without detection. We describe each of these SP protections in the following paragraphs, after we first describe the SP hardware roots-of-trust added to the microprocessor.

SP contains two hardware roots-of-trust, a Device Root Key (DRK) and a Storage Root Hash (SRH), that protect the TSM and can only be accessed by the TSM (see Figure 7). Each SP device has a different DRK value (installed at deployment in its DRK register – not burned in at manufacture) that cannot be accessed by any software, except the TSM which can use an SP instruction, DRK_derive, to derive new keys for encryption or decryption from the DRK. No other computer can derive these keys without knowledge of the DRK.

The TSM has access to pervasive secure storage whose integrity is protected by an encrypted Merkle hash tree with the root of this tree stored in the SRH register in the microprocessor chip. The SRH can only be accessed by the TSM. Each level of the hash tree encrypts and hashes its sons using an encryption key and a hash key derived from the DRK. This pervasive storage structure can be stored in the untrusted disk, and no one, not even the OS, can access it except the TSM with the correct DRK. This pervasive secure storage is used by the TSM to store critical secrets, such as long-term public and private keys, certificates, licenses, encryption keys and security policies.

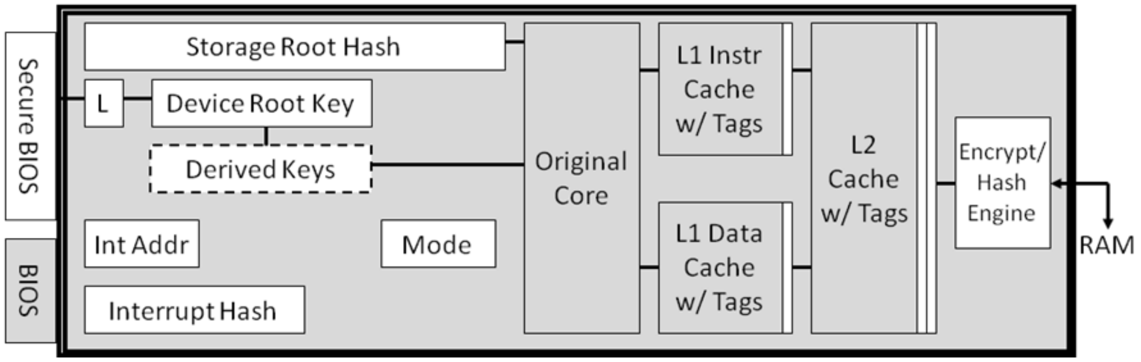


Figure 7: New SP components in microprocessor (white areas).

SP hardware provides a secure execution environment for the TSM so that no secret information accessed by the TSM is leaked out of the microprocessor chip. Since the security provided by SP depends so heavily on the DRK and SRH roots-of-trust, they must not be leaked out during TSM execution. Hence, SP provides a Concealed Execution Mode for TSM execution. SP considers only the microprocessor chip to be the physical security perimeter, i.e., only information inside the microprocessor chip is considered trusted and safe from physical probing and eavesdropping attacks – all information outside the microprocessor chip are considered untrusted, including information in the external main memory. (This is stricter security than TPM which considers the whole box to be the physical security perimeter.) Hence, SP provides automatic encryption and hashing of all temporary data belonging to the TSM if its cache lines get evicted from the microprocessor chip. Within the microprocessor chip, the TSM's secure cache lines are decrypted, so that there is no performance penalty with respect to the processor's pipelined execution. These secure lines are tagged (as shown in Figure 7). Further, SP hardware protects the general register values of the microprocessor, on interrupts, by automatic encryption and hashing, before allowing the OS to handle register saving. (This GR hash value and the interrupt return address is securely stored in new SP registers on-chip, as shown in Figure 7) Hence, the attacker can not get any access to plaintext information of the TSM during its execution, or while the TSM process is suspended or resumed. This protects leaking out bits of the DRK and SRH, as well as the secrets in the pervasive storage structure.

The TSM's code is integrity checked. SP provides dynamic Code Integrity Checking to TSM code on a cacheline granularity during runtime. Since this need only be done on the rare cache misses of the last level of on-chip caches, performance degradation is not noticeable. Prior to TSM installation on a given SP-enhanced machine, each instruction cache line chunk of TSM code is "signed" by generating a keyed-hash of that cacheline (minus the space to insert a 128-bit hash value) of instructions, keyed by the DRK. On a cachemiss, as the instruction cacheline for a TSM is brought into the microprocessor chip, its hash is re-computed. If it does not match the keyed-hash value stored at the end of the instruction cacheline, an exception is raised and that corrupted TSM instruction cacheline is not stored into an on-chip cache (which is typically the Level-2 or Level-3 cache). Hence, SP automatically provides protection of the TSM code from dynamic hostile code insertion, including those from buffer overflow attacks. It also provides detection of static hostile code modification while on the disk, by both the method just described and by verifying the integrity of the static image of the TSM code signed by the private key of the trusted software vendor (in this case the vendor of the trusted DBT).

We protect the entire DBT as a TSM in application space protected by SP hardware (Figure 6). We protect the code cache of the DBT as secure temporary data of the TSM that is automatically protected by encryption and hashing if any of it is evicted from the on-chip caches as the DBT translates or optimizes instructions. This means that attackers cannot see unencrypted contents of the code cache, and cannot modify the code cache without detection. Also, the integrity of the DBT code itself will be protected by SP's Code Integrity Checking mechanism. Hence, the application-level DBT itself, and its code cache, can be protected by SP from software and hardware attacks. In addition, the SP pervasive secure storage allows the DBT to keep secret information safe from attackers, across power off and on events.

Since the hardware additions for SP represent a very small area in a microprocessor chip, it is not difficult to add. (SP consists of only a few registers, 1-2 tag bits on a cacheline, and an encryption and hashing engine at the last on-chip cache miss handling mechanism.) These few hardware additions for security of the DBT can be done together with the hardware additions to speed up the dynamic translations done by a DBT.

6. Conclusions and Future Work

We have surveyed a range of software protection techniques and commercial software programs that can be used to enhance the security of programs, and described how they work. While many of the software techniques can fall into multiple categories, we have categorized them into 5 classes of techniques that: (1) prevent buffer overflow attacks, (2) prevent code modification attacks, (3) protect system calls, (4) mitigate reverse engineering and (5) check security assertions. Within these 5 classes, we described 12 representative software protection techniques.

We examined whether a DBT, in particular the StarDBT, inadvertently removes or lessens these software protections when the DBT applies its standard translation and optimizations to the application programs. Our results show that almost all of the software security protections are preserved, except for 4 cases where care has to be taken and certain changes in the DBT may be necessitated (see Table 1 and section 4).

Since DBTs are often application-level programs, they can themselves be attacked. Hence, we also proposed three solutions to protect the DBT itself. In particular, we show that the Secret Protection (SP) processor features can protect the application-level DBT and its code cache.

Future work can examine more software protection techniques, different types of DBT translation and optimizations, and DBT changes or features that can preserve the security of all essential software protection techniques. Some software protections may be superfluous – especially if the DBT is enhanced to provide these protections itself. Future work will also explore the many ways a trusted DBT can provide security enhancements for all programs – even legacy programs with no software security protection at all. In addition, future work can also perform a detailed security analysis of the mapping of the DBT and its code cache to a Trusted Software Module protected by the SP hardware.

References:

1. Adams, K. and Agesen, O. 2006. “A comparison of software and hardware techniques for x86 virtualization.” In *Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems* (San Jose, California, USA, October 21 - 25, 2006). ASPLOS-XII. ACM, New York, NY, 2-13.
2. Reis, G.A.; Chang, J.; Vachharajani, N.; Rangan, R.; August, D.I., "SWIFT: software implemented fault tolerance,", *Proceedings of International Symposium on Code Generation and Optimization*, vol., no., pp. 243-254, 20-23 March 2005
3. Feng Qin; Cheng Wang; Zhenmin Li; Ho-seop Kim; Yuanyuan Zhou; Youfeng Wu, "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on* , vol., no., pp.135-148, Dec. 2006
4. Wu, Q., Martonosi, M., Clark, D. W., Reddi, V. J., Connors, D., Wu, Y., Lee, J., and Brooks, D. 2005. “A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance.” In *Proceedings of the 38th Annual IEEE/ACM international Symposium on Microarchitecture* (Barcelona, Spain, November 12 - 16, 2005). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 271-282.
5. Wang, C., Hu, S., Kim, H.-s., Nair, S., Breternitz, M., Ying, Z., and Wu, Y. “StarDBT: An Efficient Multi-platform Dynamic Binary Translation System.” *ACSAC’07: Asia-Pacific Computer Systems Architecture Conference*. pp. 4—15. 2007, Seoul, Korea.

6. Dwoskin, J. S. and Lee, R. B. 2007. “Hardware-rooted trust for secure key management and transient trust.” In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA, October 28 - 31, 2007). CCS '07. ACM, New York, NY, 389-400.
7. Lee, R. B., Kwan, P. C., McGregor, J. P., Dwoskin, J., and Wang, Z. 2005. “Architecture for Protecting Critical Secrets in Microprocessors.” In *Proceedings of the 32nd Annual international Symposium on Computer Architecture* (June 04 - 08, 2005). International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, 2-13.
8. Vendicator, Stackshield, available at http://www.angelfire.com/sk/stackshield/
9. Propolice, “GCC extension for protecting applications from stack-smashing attacks”, available at http://www.trl.ibm.com/projects/security/ssp/
10. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.” In *Proceedings of the 7th USENIX Security Symposium*, pages 63--78, San Antonio, TX, January 1998.
11. Libsafe, Avaya Labs Research, available at http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails
12. C. Kil, J. Jun, C. Bookholt, J. Xu and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, 2006, pp. 339-348.
13. Horne, B., Matheson, L. R., Sheehan, C., and Tarjan, R. E. 2002. “Dynamic Self-Checking Techniques for Improved Tamper Resistance.” In *Revised Papers From the ACM Ccs-8 Workshop on Security and Privacy in Digital Rights Management*
14. R. El-Khalil and A.D. Keromytis, "Hydan: Hiding information in program binaries," *Proc. 6th Int'l Conf. Information and Communications Security (ICICS'04)*, LNCS 3269, pp.187–199, Springer, 2004.
15. Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A., “A Sense of Self for Unix Processes”, In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (May 06 - 08, 1996)
16. [8LGM]. [8lgm]-advisory-16.unix.sendmail-6-dec-1994. http://www.8lgm.org/advisories.html.
17. “Injection flaws”, Open Web Application Security Project (OWASP), available at http://www.owasp.org/index.php/Injection_Flaws
18. Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, Richard D. Schlichting, "System Call Monitoring Using Authenticated System Calls," *IEEE Transactions on Dependable and Secure Computing* ,vol. 3, no. 3, pp. 216-229, July-September, 2006.
19. Christian S. Collberg and Clark Thomborson. “Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection.” In *University of Arizona Technical Report 2000-03*, Feb 2000.
20. Ogiso T, Sakabe Y, Soshi M, and Miyaji A. “Software Obfuscation on a Theoretical Basis and Its Implementation.” In *IEICE Trans Fundam Electron Commun Comput Sci (Inst Electron Inf Commun Eng),* Vol.E86-A;No.1;pp..176-186(2003).
21. Dotfuscator. PreEmptive Solutions. http://msdn.microsoft.com/en-us/library/ms227240%28VS.80%29.aspx
22. “Obfuscated code”, Wikipedia, available at http://en.wikipedia.org/wiki/Obfuscated_code.
23. Christian Collberg, Clark Thomborson, and Douglas Low. “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs.” In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL98)*, January 1998.
24. Christian Collberg, Clark Thomborson, and Douglas Low. “Breaking abstractions and unstructing data structures.” In *IEEE International Conference on Computer Languages (ICCL'98)* , May 1998.
25. EXECryptor, available at http://www.strongbit.com/execryptor_inside.asp
26. Lallous, “Detect if your program is running inside a Virtual Machine”, available at http://www.codeproject.com/KB/system/VmDetect.aspx
27. Kc, G. S., Keromytis, A. D., and Prevelakis, V. 2003. “Countering code-injection attacks with instruction-set randomization.” *In Proceedings of the 10th ACM Conference on Computer and Communications Security* (Washington D.C., USA, October 27 - 30, 2003). CCS '03. ACM, New York, NY, 272-280.
28. George Necula, “Proof-carrying code”, In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

29. Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skalesky, A., Wang, Y., and Zemach, Y. “IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems.” In *MICRO’36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. pp. 191—201. 2003, San Diego, CA, USA.
30. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser. A., Lowney. G., Wallace, S., Reddi, V., and Hazelwood, K. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation.” In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. pp. 190—200. 2005, Chicago, IL, USA.
31. J.Wilander and M. Kamkar. “A comparison of publicly available tools for dynamic buffer overflow prevention.” In NDSS, Feb 2003.
32. Lu, J., Chen, H., Fu, R., Hsu, W., Othmer, B., Yew, P., and Chen, D. 2003. “The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System.” In *Proceedings of the 36th Annual IEEE/ACM international Symposium on Microarchitecture* (December 03 - 05, 2003). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 180.
33. Bruening, D. “Efficient, Transparent, and Comprehensive Runtime Code Manipulation.” Ph.D thesis, Massachusetts Institute of Technology, 2004.
34. Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., and Mattson, J. “The Transmeta Code Morphing$_{TM}$ Software: Using Speculation, recovery, and Adaptive Retranslation to Address Real-Life Challenges.” *Proceedings of the International Symposium on Code Generation and Optimization*. pp. 15—24. 2003, San Francisco, CA, USA.

A hardware/software co-designed virtual machine to support multiple ISAs

Tingtao Li , Alei Liang, Bo Liu, Ling Lin, Haibing Guan [*]
Shanghai Jiao Tong University, Shanghai, 200240
litingtao@gmail.com

ABSTRACT

This paper employs hardware/software co-designed methodology to design a virtual machine. A mathematic model is dedicated to analyze the Dynamic Binary Translation overhead. The co-designed virtual machine consists of a coprocessor and software running on existing processor. The coprocessor applies Dynamic Binary Translation to translate blocks of source instructions to target instructions on the fly, whereas target processor executes translated instructions. In this way, a single processor supports multiple ISAs(Instruction Set Architecture). With hardware's support, the context switch of virtual machine is eliminated and the overhead of Tcache lookup is reduced. Compared to pure software virtual machine, experimental results show that there is significant performance improvement.

Categories and Subject Descriptors

D.3.4 [Programming Languages] Processors -code generation, optimization and retargetable compilers. B.5.0 [Register Transfer Level Implementation] General.

General Terms

Algorithms, Performance, Design, Experimentation, Languages

Keywords

Virtual Machine, Dynamic Binary Translation, hardware/software co-design, Instruction Set Architecture

1. Introduction

Instruction Set Architecture (ISA) is the interface between hardware and software. Therefore software closely depends on the underlying machine platform, which means program compiled for one machine cannot run on another. For example, software compiled for x86 cannot run on PowerPC. This dependency means software cannot migrate between different machine platforms, and affects the interoperability and portability. It also makes hardware designers unable to change the hardware/software interface for further innovation because of the importance of backward compatibility.

There are all kinds of computing platforms such as x86, Power, ARM, MIPS and so on. Unfortunately, they are not compatible with each other, which makes software can't be compiled once and run anywhere. Compared with other ISA platforms, x86 is the most popularly used computing platform and has plenty of programs. In recent years, the prominence of the Internet and networked computing require providing support for heterogeneous computing platforms. Sometimes it is not convenient to port software developed for one platform to other platforms, especially when there is no source code. Sometimes programs may migrate from one machine to another machine at running time. These problems require a single processor to support multiple ISAs. Much research employs virtual machines [1] to solve these problems. The binary code of software doesn't directly run on the target processor. The Virtual machine interprets or dynamically translates the source binary code to target binary code that can run on target processor. But traditional software virtual machine results in unsatisfying performance and transparency. For example, although many optimization methods are adopted, QEMU [2] is sill 4-10 times slower than native execution. For many computational intensive real-time applications such as audio and video encoding/decoding applications, the performance is not satisfying to user. Therefore, pure software virtual machine cannot provide the "near native" performance.

With the development of microelectronics, it becomes convenient to add a coprocessor or custom IP core to existing processor. This paper employs hardware/software co-designed methodology to design a virtual machine. Application-specific hardware is used to accelerate the critical parts of virtual machine. A coprocessor, implemented using an FPGA, combines the upper software to form the virtual machine with better performance. The virtual machine makes system provide multiple ISAs to user transparently. The close interaction of software and hardware provides opportunities for great flexibility, satisfying performance and cost.

The rest of the paper is organized as the follows. Section 2 introduces background and related research about virtual machine and binary translation. Section 3 introduces a mathematic model to analyze the dynamic binary

*Corresponding Author: hbguan@sjtu.edu.cn translation overhead. In section 4, hardware support for Process Virtual Machine [1] is proposed. Section 5 and section 6 introduce the software part and hardware part of this co-designed virtual machine respectively. Evaluation of the performance is presented in section 7. Section 8 concludes this paper.

2 Background and related research

According to the interface layer of virtual machine, we can classify the virtual machine to two main categories [1], Process Virtual Machine and System Virtual Machine. Process Virtual Machine provides user applications with a virtual ABI (Application Binary Interface) environment. System Virtual Machine provides a complete system environment to support multiple operating systems. This paper will focus on the Process Virtual Machine to provide x86 ISA on PowerPC platform.

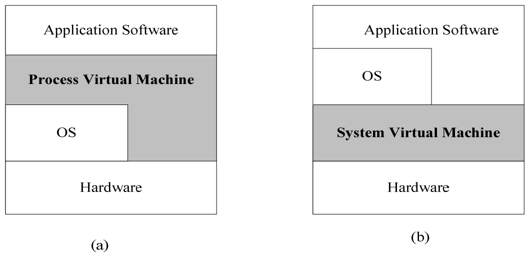


Figure 1 Process Virtual Machine and System Virtual Machine

Dynamic Binary Translation (DBT) [3] [16] is an important technology to implement Process Virtual Machine. Dynamic Binary Translation translates blocks of source binary code to target binary code, and places the target code into a region of memory for reuse. Therefore, the repeated execution of cached target code doesn't require translation.

There are various process virtual machines that employ dynamic binary translation as the key technology. IA32EL [4] is a pure software solution that enables x86 software to run on Intel Itanium processor family systems. It first translates cold code with minimal optimizations and overhead. When some trace becomes "hot", it will retranslate and further optimize those hotspots. Digital FX!32 [5] is a process virtual machine for running x86 windows applications on DEC Alpha platforms. UQBT[6],developed by Queensland University, is a retargetable and resourceable dynamic binary translator. It supports multiple source platforms and target platforms.

In recent years, virtual machines have been a new avenue to implement a processor compatible to conventional ISA. In this field, the most famous paradigms are IBM's DAISY [7] and Transmeta's Crusoe [8] [9]. The upper layer of DAISY is a binary translator to emulate the PowerPC ISA. The underlying layer is a VLIW architecture. All existing software for an old architecture runs without changes on the new VLIW architecture processor. Each time a new fragment of code is executed for the first time, the code is translated to VLIW primitives, and saved in a portion of main memory for reuse. Subsequent executions of the same fragment do not require translation (unless cast out).The Transmeta's Crosue and its successor Efficeon are similar to DAISY. The significant difference is that Crusoe emulates the x86 ISA. The Code Morphing Software [8] with some hardware support translates x86 instructions to the underlying VILW instructions. Hardware support for DBT based virtual machines is less widespread, but we argue that it will be an inevitable trend [12].

There has been some other work in the field of binary translation in converting binary for one ISA to another ISA. Gaurav Mittal's [13] work automatically translates binaries targeted for general DSP processors into Register Transfer Level(RTL) VHDL or Verilog code to be mapped onto commercial FPGAs. The focus of their work is on automated synthesis of software binaries onto hardware. In some sense, the reconfiguring hardware executes source instructions. Our work combines dynamic binary translation with co-designed methodology to implement a virtual machine. The coprocessor is a virtual machine acceleration unit, whereas target processor is execution unit.

3. **Performance Modeling of Dynamic Binary Translation**

In common case, DBT uses basic block as the translation unit. Figure 2 is the entire process of DBT. DBT first uses a Source Program Counter (SPC) to look up the SPC-TPC (Target Program Counter) map table. If the corresponding SPC value exists in the map table, this basic block has been translated. The corresponding target code is stored in the Target Code Cache (Tcache), so processor can directly jump to TPC to execute instructions. Otherwise, if the lookup result shows that there is a code cache miss, the processor will translate the source code block to target code block, then places it in the target code cache. Dynamic binary translation uses basic block as the translation unit.

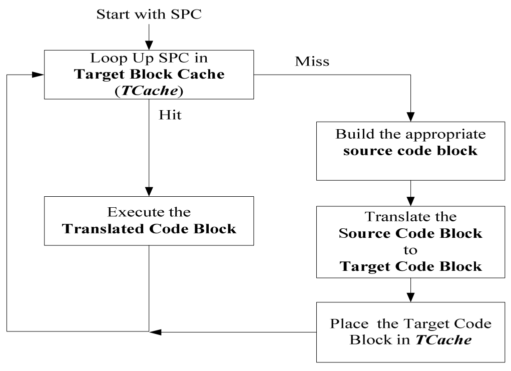

Start with SPC
Loop Up SPC in Target Block Cache (TCache)
Miss
Hit
Build the appropriate source code block
Execute the Translated Code Block
Translate the Source Code Block to Target Code Block
Place the Target Code Block in TCache

Figure 2 Structure of DBT

According to figure 2, we propose a mathematical model to analyze the DBT overhead. Eq.1 is the total overhead of a basic block.

T_total = (1-F_linked) * (T_lookup + T_context_switch) + T_execution + F_cache_miss_rate * T_translation

Eq.1 where F_linked = fraction of time that basic blocks are linked together in Tcache
T_lookup= the overhead of looking up SPC-TPC map table.
T_context_switch= overhead of context switch between execution and translation
T_execution= the time of executing translated code
F_cache_miss_rate = fraction of basic block accesses that miss the Tcache
T_translation = the overhead of translating a source basic block to target block.

According to the SPEC2000 benchmark results and workflow [Figure 2] of process virtual machine, there are three main bottlenecks in pure a software virtual machine.

(1)The first bottleneck is the Tcache (Target code cache) manager [14]. The corresponding overhead is represented by symbol "T_lookup" in Eq.1. If the basic block is not linked, processor must first check if the basic block has been translated before execution. The lookup of SPC-TPC map table is time-consuming and frequent. It's inefficient to implement it by software.

(2)The second bottleneck is the context switch time (Eq.1) between execution and translation. Because the virtual machine itself and the emulated program have different contexts. For most of basic blocks the virtual machine must save and restore the execution context. Given a basic block has only about ten instructions, the cost is even more than execution overhead, especially when the processor is RISC architecture and has many registers.

(3)The third bottleneck is the dynamic translation unit (Translation time in Eq.1) [15]. Because it takes hundreds of instructions to translate a basic block in miss case, the performance will fluctuate distinctly between hit and miss case. This bottleneck affects the startup time and miss penalty, especially for computational intensive real-time applications.

4. Hardware support for Binary Translation

Adding hardware support for virtual machine is an obvious approach to improve performance. Based on the discussions above, a subset of the virtual machine will be implemented by hardware. We employ the following approaches to alleviate the three bottlenecks.

We apply hardware to implement the Tcache manager. The Tcache manager maintains a SPC-TPC map table. This is similar with TLB to implement the map of virtual address to physical address. In this way, the lookup overhead is only about three to five instructions. At the same time, the linked rate of blocks can be improved by the branch prediction translation unit. In the lookup process, Tcache manager can sample the basic block execution information. If some basic blocks become "hot", it will interrupt software to optimize the hotspot of the program.

We also employ hardware to implement the dynamic translation unit. There are three main advantages for performance improvement. 1) Because target processor needn't switch to translate source instructions, the context switch time can be eliminated. 2) Application-specific hardware replaces the software's translation process, so the startup time can be reduced and miss penalty can be improved. 3) If the hardware translation unit translates basic blocks before the processor executes corresponding instructions, the miss rate of Tcache can be reduced significantly.

In this paper, we implement the experiments on Xilinx Virtex FPGA board. The platform uses PowerPC processor combined with a coprocessor, which contains the Tcache manager and translation unit of virtual machine. The source and target ISA is x86 [18] [19] and PowerPC [20] respectively. Based the software-hardware partition principle, we draw a line between software and hardware. The software layer contains the x86 ELF file loader, virtual machine IP core driver and Linux OS. The hardware layer contains two parts. One part is the PowerPC processor and memory. The other part is virtual machine IP core. The virtual machine IP core consists of the dynamic binary translation unit and Tcache manager unit. The system architecture is illustrated in figure 3.

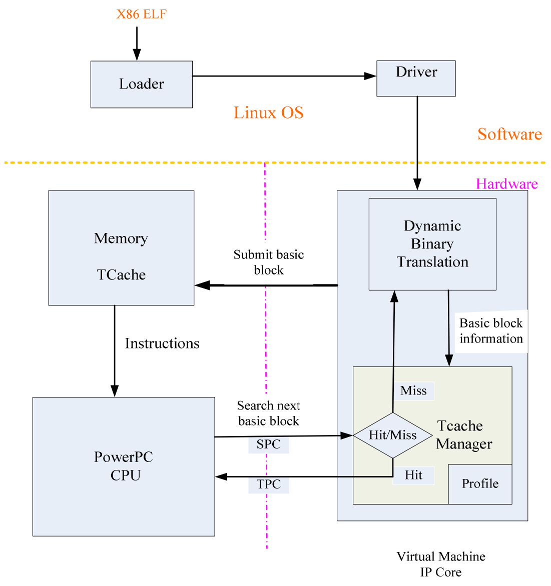


Figure 3 System Architecture

5. Software design

The software layer of virtual machine consists of x86 ELF file loader, the driver for virtual machine IP core, Linux 2.6.23 kernel and file system.

Loader loads x86 binary image and data into appropriate region of memory. Loader analyzes elf file section and maps the text and data sections to the loader's process space. The stack and heap of the running target program will be placed in the process space of loader.

As showed in figure 4, the loader will jump to the stub code after the loader reads the entry address of the source code section. The stub code is fixed code in memory to lookup the SPC-TPC map table. The stub code uses the SPC (Source Program Counter) to search the TPC (Target Program Counter). If there is a hit, then the PowerPC processor will jump to the TPC place to execute the basic block. If there is a miss, the processor will wait here until the hit_miss_reg has been changed. The change of the hit_miss_reg means that the basic block has been translated in the wait time and the PowerPC processor can execute the basic block. In this way, there is no context switch between the PowerPC execute engine and the translation unit. Therefore the second bottleneck discussed in section 3 is eliminated implicitly. If some basic blocks have been linked together, the previous basic block will skip stub code and directly jump to the following basic block.

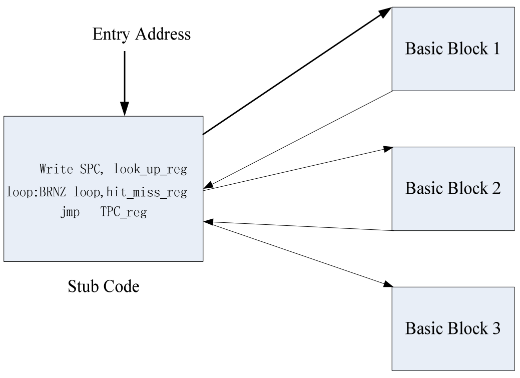


Figure 4 Basic blocks execution process

6. Hardware design

The hardware consists of PowerPC processor and our virtual machine IP core. The virtual machine IP core contains two parts: the dynamic binary translation unit and the Tcache manager. As illustrated in figure 5, Virtual machine IP core connects with PowerPC by Processor Local Bus. The Sbuffer is a buffer for the source x86 instructions. The Tbuffer is used as the target PowerPC instruction buffer.

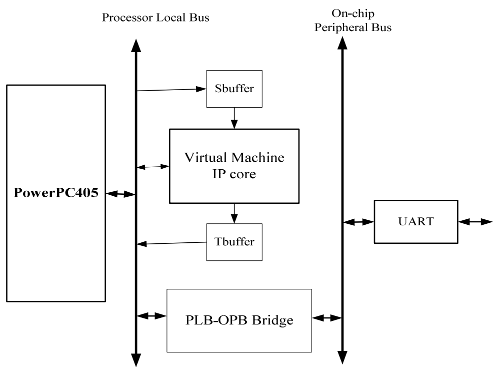


Figure 5 Hardware architecture

6.1 Dynamic Translation Unit

We employ state machine to implement binary translation process. Dynamic translation unit translates blocks of x86 instructions to PowerPC instructions. BOCHS [18] is used to analyze x86 decode and instruction emulation. The state machine is illustrated in Figure 6 (some states can be further divided into subset sates). The translation process contains two phases: translation of operand address and translation of semantics. The first phase will translate complex x86 operand address mode and place the operand in temporary register (in yellow circle state). The second phase will finish the semantic translation (in green circle state). In this phase, operand is register or immediate. Finally, target instructions will be generated.

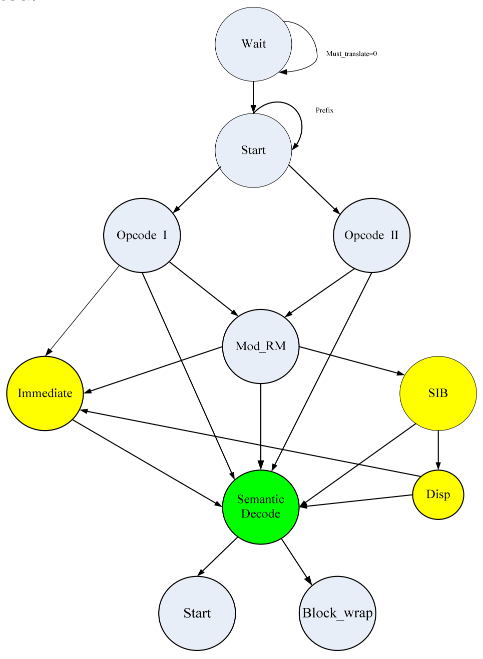


Figure 6 State machine of translation unit

6.2 Tcache Manager

Tcache manager maintains a SPC-TPC map table. Code cache performance is utmost importance. System accesses the map table frequently. We employ hardware to implement the Tcache lookup and management. The lookup process is illustrated in figure 7. The lookup begins with hashing the SPC value to access the map table, reading the corresponding SPC from the map table, comparing the two SPC values to determine if the basic block has been translated. The Tcache has limited store space, we use LRU algorithm to implement the Tcache replacement strategy. If a new basic block has been translated, the SPC-TPC map table will be updated. Tcache manager can sample basic block execution information, if some trace of the program reaches 'hot', optimization can be done in software layer.

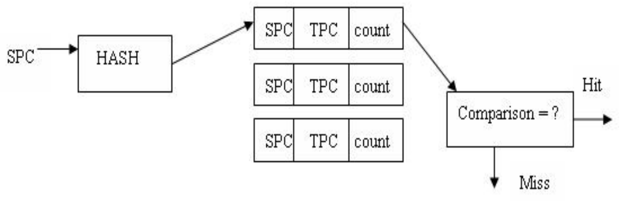


Figure 7 SPC-TPC mapping by hardware

We can use the special instructions to lookup the map table in software layer. On the other hand, we can employ the APU (Auxiliary Processor Unit) to extend native PowerPC instruction set with custom instructions for lookup operation. So the lookup overhead will be only few instructions. These instructions called Stub Code in figure 4 reside in fixed place in memory. The stub code is showed by the following pseudocode.

```
write   SPC, look_up_register
loop_here: branch_not_zero   loop_here, hit_miss_register
jmp    TPC_register
```

7. Performance evaluation and analysis

To evaluate the performance of co-designed virtual machine, we will compare the overhead of bottlenecks, the basic block execution time in hit and miss case with pure software virtual machine. SPEC2000 integer benchmarks are used to evaluate the performance. We will compare the performance of codesigned virtual machine with QEMU and native execution.

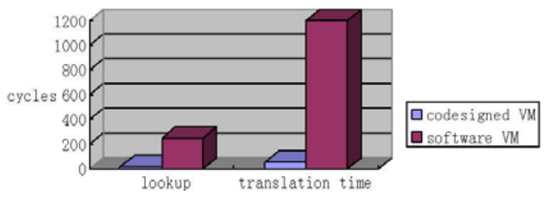


Figure 8 lookup and translation overhead comparison

Figure 8 is the performance comparison of the bottlenecks (lookup and translation time in Eq.1) between co-designed virtual machine and software virtual machine running on the PowerPC. As shown in figure 8, co-designed virtual machine only requires about twenty cycles to finish the lookup operation and less than sixty cycles to translate a basic block. There is 10-25 times speedup. Hardware plays an important role in the performance improvement of bottleneck.

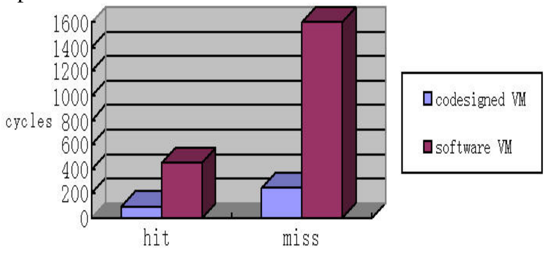


Figure 9 execution time in hit and miss case

Figure 9 shows the comparison of a basic block execution time in hit and miss case between co-designed virtual machine and software virtual machine. There is a distinct performance improvement. In addition, the performance of co-designed virtual machine doesn't fluctuate distinctly between hit and miss case. This character is important to software's real-time ability and startup performance. However, software virtual machine has a bad miss penalty.

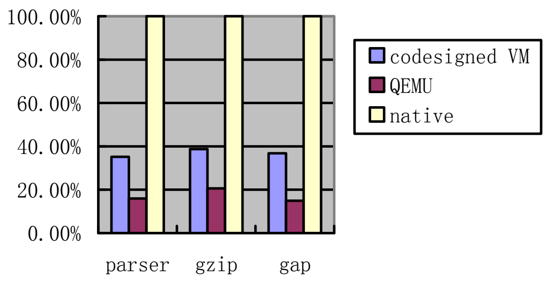


Figure 10 performance comparison

Figure 10 shows the performance comparison between codesigned virtual machine, software virtual machine (QEMU) and native execution (native code running on the PowerPC). The performance of co-designed virtual machine nearly reaches to 40% of native performance. In addition, the performance of co-designed virtual machine is currently evaluated without further optimizations such as profile and super block [17]. Adding some optimization methods, the rate may be more than 50%. The performance is satisfying and acceptable to user. Co-designed virtual machine provides a "near native" performance.

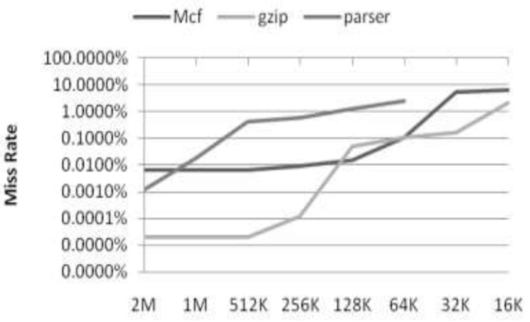


Figure 11 miss rate at different Tcache size

As showed in Figure 11, x-axis represents the Tcache size. The miss rate of the SPEC 2000 program is extremely low if Tcache size is big enough. Most of the time, the translation unit of coprocessor is idle. The low miss rate makes the virtual machine coprocessor have capability to provide service for multiple processors. Moreover, we can turn off the translation unit to reduce power consumption if it is idle.

8. Conclusion and future directions

In this paper, we construct a software/hardware co-designed virtual machine. Compared to pure software virtual machine, the performance is significantly improved. So x86 software can run on PowerPC processor at an acceptable speed. PowerPC processor can provide multiple ISAs to user due to our virtual machine IP core. The design is meaningful to solve the software's portability and compatibility in heterogeneous computing platforms.

For future research, we will do the following improvements. First, we will profile the program in hardware and optimize the program in software. Second, thanks to the low Tcache miss rate, the virtual machine coprocessor will be applied in multi-cores system and provide hardware assist for multiple processor.

9. ACKNOWLEDGMENTS

We appreciate all members in virtual machine group of Shanghai Jiao Tong University. This work was supported by the National Science Foundation, Project Number: 60773093, the "973" National Basic Research Program of China, Project Number: 2007CB316506, the Ministry Of Education and Intel joint research foundation, Project Number: MOE-INTEL-08-11. We also thank IBM Corporation and Xilinx Corporation for the donation of development board and software.

10. REFERENCES

[1] J.E. Smith, and R. Nair, Virtual Machines: Versatile Platforms for Systems and Process, Morgan Kaufman, 2005.

[2] F Bellard, QEMU, a Fast and Portable Dynamic Translator, Proceedings of the USENIX Annual Technical Conference, 2005

[3] Kemal Ebcioglu, Erik R. Altman, et al., Dynamic Binary Translation and Optimization, IEEE Transactions on Computers, Vol. 50, No. 6, pp. 529-548. June 2001.

[4] Leonid Baraz, et al. IA-32 Execution Layer: a two phase dynamic translator designed to support IA-32 applications on Itanium-based systems, Proceedings of the 36th International Symposium on Microarchitecture pp. 191-202 Dec. 2003.

[5] R.J. Hookway, and M.A. Herdeg, DIGITAL FX!32: combining emulation and binary translation.Digital Tech. J. 9 (1997) 3-12.

[6] C. Cifuentes, M.V. Emmerik, and N. Ramsey, The Design of a Resourceable and Retargetable Binary Translator, WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering, IEEE Computer Society, 1999.

[7] K. Ebcioglu, E.. Altman,DAISY: Dynamic Compilation for 100% Architectural Compatibility, Proc. of the 24th Int'l Symp. on Computer Architecture, pp.26-37, Jun. 1997.

[8] J.C. Dehnert, et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges , Proc. of the 1st Int'l Symp. on Code Generation and Optimizations, pp. 15-24, Mar. 2003.

[9]A. Klaiber, The Technology Behind Crusoe Processors ,Transmeta Technical Brief, 2000.

[10] Keith Adams, A Comparison of Software and Hardware Techniques for x86 Virtualization, ACM ASPLOS 06.

[11]UHLIG, et al, Intel virtualization technology,Computer 38.5 (2005), 48–56.

[12] Leendertvan Doorn, Hardware Virtualization Trends, ACM VEE 2006.

[13] Gaurav Mittal, Automatic Translation of Software Binaries onto FPGA, ACM Design Automation Conference 2004.

[14] H-S. Kim and J. E. Smith, Hardware Support for Control Transfers in Code Caches , 36th Int. Symp. on Microarchitecture, pp.253-264, Dec. 2003.

[15] Shiliang Hu, and James E. Smith, Reducing Startup Time in Co-Designed Virtual Machines, Proceedings of the 33rd International Symposium on Computer Architecture. June, 2006.

[16] Erik R. Altman, Advances and Future Challenges in Binary Translation and Optimization , Proceedings Of

the IEEE, Special Issue on Microprocessor Architecture and Compiler Technology, Nov. 2001, pp. 1710-1722.

[17]Huihui Shi, Yi Wang, Haibing Guan, Alei Liang, An Intermediate Language Level Optimization Framework for Dynamic Binary Translation , ACM SIGPLAN, 2007

[18] K.Lawton,The BOCHS IA32 Emulation Project . http://bochs.sourceforge.net

[19] Intel Corp., IA-32 Intel Architecture Software Developer’s Manual, vol.2: Instruction Set Reference, Intel Corp., 2003

[20]IBM Corp., PowerPC User Instruction Set Architecture Book I : Instruction Set Reference，IBM Corp., 2003

A New Replacement Algorithm on Content Associative Memory for Binary Translation System*

Jing Li, Chenggang Wu

Key Laboratory of Computer System and Architecture,

Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190

wucg@ict.ac.cn

Abstract

Binary translation and dynamic optimization is important for porting legacy code and improving program performance. Co-designing is a very useful way for these kinds of systems. Indirect–jump-mapping is one of the key factors which affect system performance greatly. Usually, a co-designed system has a hardware lookup table named CAM to reduce the overhead of access memory during indirect–jump-mapping. A good replacement algorithm will lower the CAM miss rate and get better performance.

In this paper, we analyze the characters of indirect–jump-mapping and existing replacement algorithms. Based on this we proposed a PCBPC algorithm. Compared to LRU, FIFO and RANDOM algorithms, the PCBPC algorithm reduced the miss rate by 32.09%、37.58% and 84.86% in SPEC CPU2000 benchmarks.

1 Introduction

Binary translation and dynamic optimization is important for porting legacy code and improving program performance. How to improve system performance in the field is one of the focal points. The improvement is limited if we only consider the software optimization. Co-designing results in optimal performance by using hardware and software together. [1][2][3].

Since programs have large amounts of indirect *jump*, indirect *call* and *return* instructions, indirect-jump-mapping is one of the key factors which affect the performance of binary translation and dynamic optimization systems[4][5][6]. The systems always store the source and target indirect-jump-mapping addresses in memory as hash table. Dynamic mapping is required when program meets an indirect jump instruction. It will always be translated to dozens of search instructions which include several memory access instructions. The performance will be bad if we only use software to do this work. In order to reduce the overhead in memory look-up table, the co-designed system will add a hardware look-up table[7], which named Content Associative Memory. Using an appropriate replacement algorithm to reduce the miss rate is crucial to accelerate process of indirect-jump mapping.

In this paper, based on the detailed analysis of the indirect-jump-mapping features and existing replacement algorithms for CAM, we proposed a new replacement algorithm, PCBPC—Partitioning CAM Basing on Programs’ Character, to reduce the miss rates of CAM table. SPEC CPU2000 benchmarks is used as our test case[8]. In our designed binary translation system, named DigiatlBridge, the data obtained show that the PCBPC algorithm achieves good performance compared with the LRU, FIFO and RANDOM replacement algorithms.

The rest of the paper is organized as follows: the next section will present the indirect–jump-mapping hardware support method. The third section will analyze the indirect–jump-mapping characteristics. The forth section will introduce the PCBPC algorithm. Section 5, 6 and 7 will include performance, related work, as well as conclusions.

2 Hardware support for indirect-jump-mapping in Loongson-3

A Content Associated Memory (CAM) is designed in

* This paper is funded by the project of National Nature Science of China (No. 60736012) and the project of Hi-Tech Research and Development Program of China (No. 2007AA01Z110).

Loongson-3 processors to realize functions similar to JTLB[4]. It can work with a hash table to realize indirect–jump mapping. CAM has three columns: ASID, Source Address and Target Address. It provides fully associative mapping. ASID is used to distinguish processes. Source and target addresses are stored in the second and third columns respectively, as shown in Figure 1. When we want to map a source address to its target address, we can let the table make a fully-associative search. If the address is hit in this table, the DigitalBridge assigns the hit subroutine. If not, it calls the miss subroutine, which will search the software hash table.

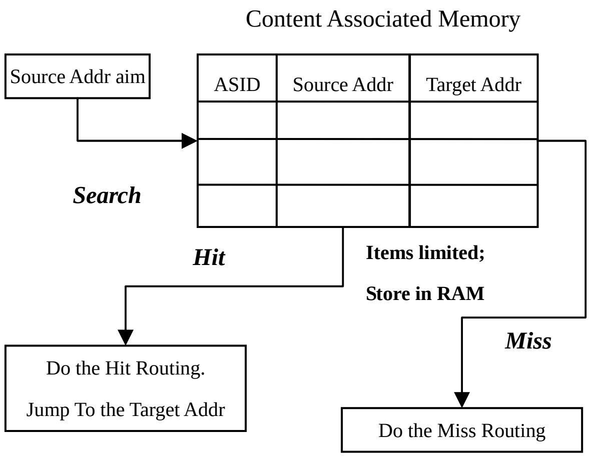


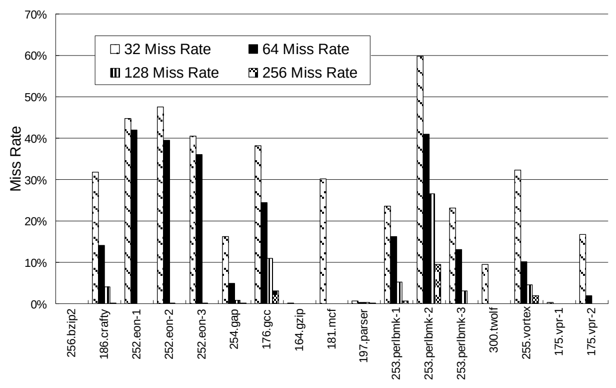
Figure 1. Hardware support for mapping an indirect-jump-address by using CAM

2.1 Experimental Environment

We realize the replacement algorithms in our binary translation System — DigitalBridge. DigitalBridge can migrate the X86/Linux binary codes into the Loongson-3/Linux platform.

The test case is SPEC CPU2000 benchmarks. We compiler them by GNU CC 3.4.6 compiler with the peak options from the SPEC website. To reduce the simulating time, we use the train data as inputs.

The size of the CAM table can not be too large so as to limit the increasing of hardware overheads. To get the proper size, we simulate the results when we set the size at 32, 64, 128 and 256 respectively, using the LRU replacement algorithm. Figure 2 illustrates the results. After comparing the miss rate of each, we found 64 is the optimal choice.



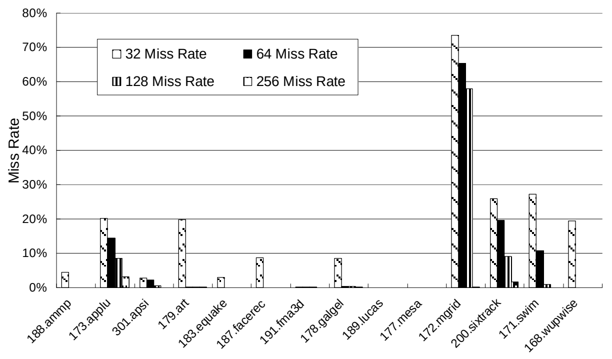


Figure 2. The CAM miss rate with different sizes by using LRU replacement algorithm

3 The characteristics of indirect-jump-mapping when executing program

CAM is similar with TLB, Cache, and other traditional hardware tables, which are used to accelerate memory access. However there are some differences between them. (1) Mapping relationships are different. (2) There are special characteristics for indirect jump patterns, which will be discussed bellow. As a result, CAM needs a different appropriate replacement algorithm.

3.1 The First Characteristic of test examples

To express the information clearly in the following part of the paper, we use 2 special terms. One is *static-number*, which counts every indirect-jump-mapping address once no matter how many times it repeats. In the contrast, the other one is *dynamic-amount*, which stands for the amount of indirect-jump-mapping addresses emerged in runtime.

The test examples have few *static-number* of addresses, which is about 10^3 in average. However their *dynamic-amount* is over 250 million times in the runtime. Their addresses also have good locality. Table 1 provides the data in detail.

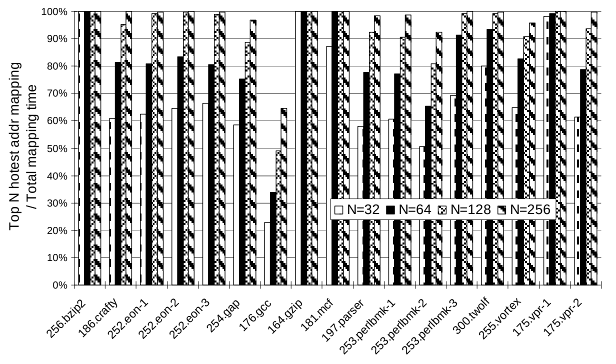
Benchmark	Dynamic Address Mapping Total Time	Static Address Mapping Number	% of Static Number of Dynamic Mapping Time
256.bzip2	153576631	480	0.0003%
186.crafty	412339165	1542	0.0004%
252.eon-1	36324705	3207	0.0088%
252.eon-2	190584896	3228	0.0017%
252.eon-3	55438644	3232	0.0058%
254.gap	255733408	2443	0.0010%
176.gcc	55686182	9039	0.0162%
164.gzip	261848957	479	0.0002%
181.mcf	11423023	539	0.0047%
197.parser	120188465	1434	0.0012%
253.perlbmk-1	879136897	3485	0.0004%
253.perlbmk-2	516872137	2712	0.0005%
253.perlbmk-3	862786050	1781	0.0002%
300.twolf	67277513	1315	0.0020%
255.vortex	324603949	3792	0.0012%
175.vpr-1	45078166	1004	0.0022%
175.vpr-2	8476339	1303	0.0154%
Average	**250433831**	**2413**	**0.0037%**

Benchmark	Dynamic Address Mapping Total Time	Static Address Mapping Number	% of Static Number of Dynamic Mapping Time
188.ammp	46718132	699	0.0015%
173.applu	38385	1009	2.6286%
301.apsi	20325857	1725	0.0085%
179.art	237705	441	0.1855%
183.equake	8864531	573	0.0065%
187.facerec	949380575	1517	0.0002%
191.fma3d	1004670381	2507	0.0002%
178.galgel	10092056	1920	0.0190%
189.lucas	1843657192	817	0.0000%
177.mesa	255359338	739	0.0003%
172.mgrid	1545514	1038	0.0672%
200.sixtrack	128609267	3414	0.0027%
171.swim	3535402	782	0.0221%
168.wupwise	839437681	1231	0.0001%
Average	365176573	**1316**	**0.21%**

Table 1. The Relationship of *Static-number* and *Dynamic-amount*

Although the *static-number* of addresses is much smaller than their *dynamic-amount*, it still exceeds the size of the CAM greatly, which is 64. Let us do a deeper analysis. We sort the indirect addresses according to their frequency, select the hottest N addresses and add their times together. Figure 3 shows the ratio of the total top N addresses to the total of all addresses.

The average *static-number* of addresses is 2,413. Although the top 64 only occupy 2.78%, they can be 82.4% of the dynamic amount for integer benchmarks. Floating point benchmarks have 1,316 *static-number* of addresses on average. The top 64 occupy 4.9% of *static-number* and 90.6% of *dynamic-amount*.



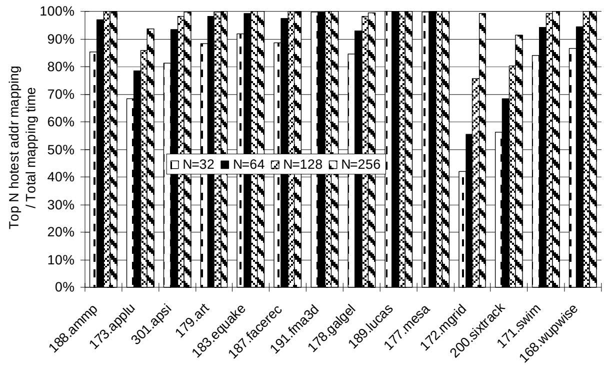
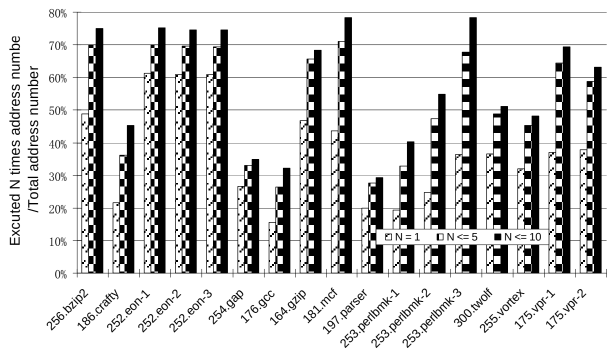


Figure 3. The ratio of the total top N addresses to the total of all addresses

Let us analyze the benchmarks from another perspective. We select the addresses whose dynamic amount is less than N, count their static number, and divide them by the total *static-number* of addresses. The bars in Figure 4 give the ratios, whose N are 1, 5, and 10 respectively. When N is 10, the ratios are 53.9% and 63.6% for integer and floating-point benchmarks in average.

The above data show that a few static addresses occupy most of the dynamic indirect-jump-mapping addresses.



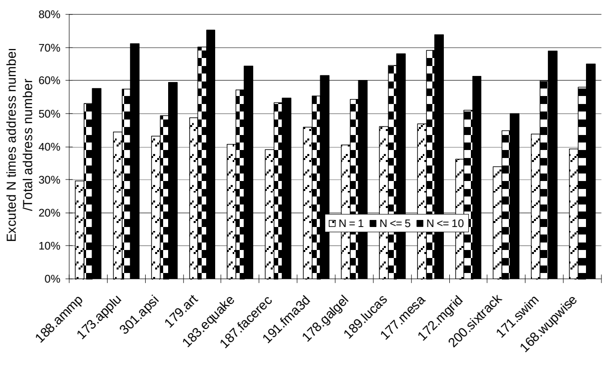


Figure 4. The ratio of the coldest N addresses to the total of all addresses

3.2 The Second Character of test examples

A great proportion of the dynamic indirect-jump-mapping addresses are from *return* instructions. Figure 5 gives the data about the proportion. For integer and floating-point benchmarks, the proportions are 82.9% and 76.0% in average.

The target address of a *return* instruction is different from other indirect-jump-mapping addresses[22][23][24], which have great relationship with the call point. So, it should be handled specially.

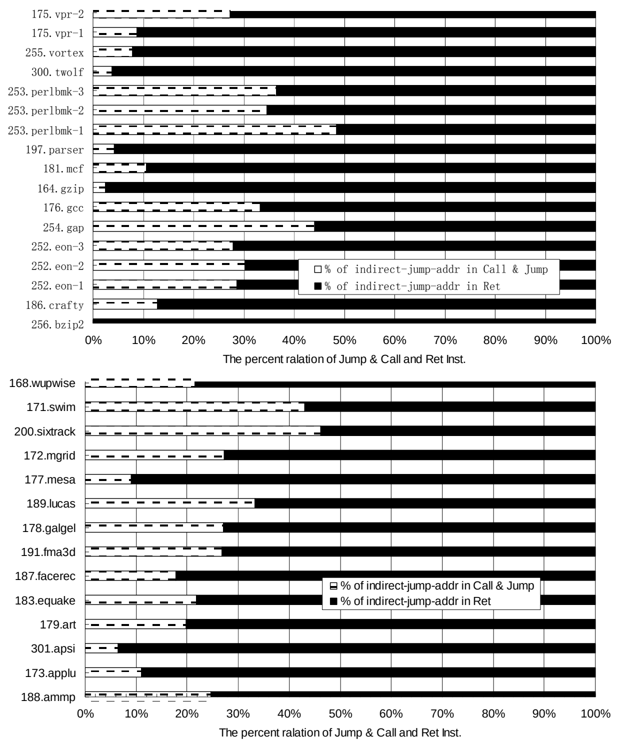


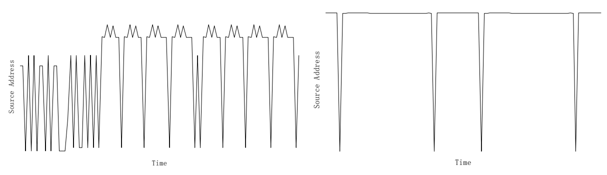
Figure 5. The proportion which return instructions occupy the dynamic indirect-jump-mapping addresses

3.3 The Third Character of test examples

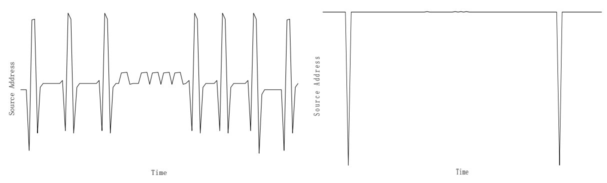
The dynamic sequences of source addresses show phase and cyclic characteristic.

In the execution process of a program, several indirect –jump-mapping addresses have a stable order, which is named a *stable group* in this paper. The *stable group* repeats constantly in a given stage. If its repeat times exceed a threshold, we call it *hot stable group*.

We slice the sequences of indirect-jump-mapping address for 176.gcc, 181.mcf, 252.eon-1 and 172.mgrid randomly. The lengths of slices are 100. Figure 6 gives the results. The y axis is value of address and x axis is the sequence. Usually the length of a *hot stable group* is less than 30.



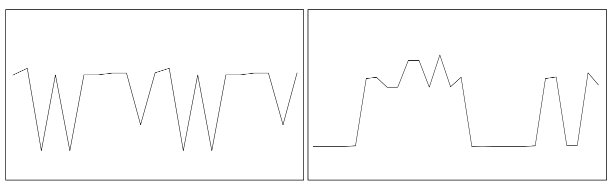
176.gcc 181.mcf



252.eon-1 172.mgrid

Figure 6. The source addresses changing by time

In Figure 6, some addresses sequence look like a straight line. We enlarge the graphs of 181.mcf and 172.mgrid to get Figure 7. Figure 7 shows 181.mcf also has good phase behavior in the straight line, however, 172.mgrid does not has the same character.



181.mcf 172.mgrid

Figure 7. The enlargement of part graphs in 181.mcf and 172.mgrid

4 PCBPC Algorithm

4.1 The existing replacement algorithms

As we know, LRU, FIFO and RANDOM are three most popular replacement algorithms. Let us analyze them now.

LRU: When a miss occurs, it evicts the data that was accessed least recently.

FIFO: When a miss occurs, it evicts the data that was brought into the CAM least recently.

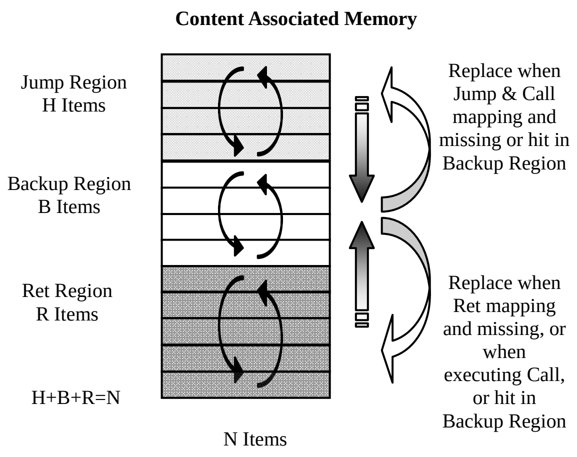
RANDOM: When a miss occurs, it evicts the data randomly. [25]

RANDOM algorithms can be realized easily. However, it does not consider the usage and special feature of data [26].

FIFO replacement algorithm is ideal for applications with linear sequence addresses, but for those with few linear sequence addresses it is not efficient. For example, some addresses are frequently accessed, so they stay very long and become old ones in CAM, which causes them replaced by new items. Another flaw of FIFO replacement algorithm is that in some extreme cases, the increase of CAM size causes the rising of miss rate.

For most applications, comparing with FIFO and RANDOM replacement algorithms, LRU has the lowest CAM miss rate. However, it has high time overhead, so it needs hardware support. Because of the differences between the executing frequency and characteristics of instructions, the frequency and characteristics vary from one address need to be mapped to another. When we use LRU replacement algorithm to manage CAM directly, the addresses of *jump*, *call* and *return* instructions compete for CAM each other, which worsens the miss rate.

4.2 Design ideas



Content Associated Memory
Jump Region H Items
Backup Region B Items
Ret Region R Items
H+B+R=N
N Items
Replace when Jump & Call mapping and missing or hit in Backup Region
Replace when Ret mapping and missing, or when executing Call, or hit in Backup Region

Figure 8. The regions partitioned in CAM for PCBPC algorithm

According to runtime indirect-jump-mapping characteristics concluded from large amount of data in previous sections, the PCBPC algorithm was proposed to make better use of CAM hardware resources.

CAM is logically divided into Jump Region, Backup Region and Return Region, as Figure 8 shows. The sizes of the three regions are H, B and R respectively and H+B+R=N. Their functions are as follows, as Figure 9 shows. The function of Insert(X,Y) is inserting item X into Y by corresponding replacement algorithm.

Hit Routing

```
if ( Hit (BackupRegion) ){
  if ( Ret Inst. Search ){
    Insert(HitItem,ReturnRegion);
    Shift the Backup Region;
    Insert(ReplacedItemInReturnR
egion,BackupRegion);
  }else{
    Insert(HitItem,JumpRegion);
    Shift the Backup Region;
    Insert(ReplacedItemInJumpRe
gion,BackupRegion);
  }
}else
    Do nothing;
Jump to the Target Address;
```

Miss Routing

```
Search in the Hash Table;
if ( Hit (HashTable) ){
  if ( Ret Inst. Search ){
    Insert(HitItem,ReturnRegion);
    Insert(ReplacedItemInReturnRe
    gion,BackupRegion);
}else{
      Insert(HitItem,JumpRegion);
      Insert(ReplacedItemInJumpRe
      gion,BackupRegion);
  }
  Jump to the Target Address;
}else{
  Translate the Block will be
executed;
  Insert(Item,HashTable);
  if ( Ret Inst. Search ){
    Insert(Item,ReturnRegion);
    Insert(ReplacedItemInReturnRe
    gion,BackupRegion);
  }else{
    Insert(Item,JumpRegion);
    Insert(ReplacedItemInJumpReg
ion,BackupRegion);
}
```

Figure 9. The functions for Hit Routing and Miss Routing in PCBPC algorithm

Jump Region stores mapped addresses of *jump* and *call* instructions. Return Region stores mapped addresses of *return* instructions. Each region can be set to certain sizes according to the characteristics of indirect-jump-mapping addresses, and FIFO policy is used as replacement algorithm in each region separately, which has lower cost than LRU. The sizes of regions are determined by the quantity of *jump*, *call* and *return* instructions, so that high miss rate caused by the competition for CAM resource can be avoided. For example, if *jump* and *call* instructions are more than *return*, the size of Return Region is constrained. Although the miss rate of *return* addresses may grow, the total miss rate is reduced. When a call instruction is executed, the source and target return addresses must be added to the Return Region, and the victim item will be moved to Backup Region.

Backup Region stores victim addresses from Jump Region and Return Region. If an address is hit in this region, it will be moved back to Jump Region or Return

Region according to the type of current instruction, so that advantages of LRU replacement are preserved while flaws of FIFO replacement are overcome, and frequently used addresses can stay longer in CAM. What is more, Backup Region can store all types of addresses, so it provides flexibility for changing sizes of *jump* and *return* addresses.

5 Performance Results

Since *return* instructions dominate great part of indirect-jump-mapping addresses, we added a related optimization for LRU, FIFO, RANDOM algorithms. We named it *return optimization*. When the program meets a *call* instruction in runtime, we insert its return address and relative target address into CAM. It can reduce the miss rates noticeably, as shown in Figure 10.

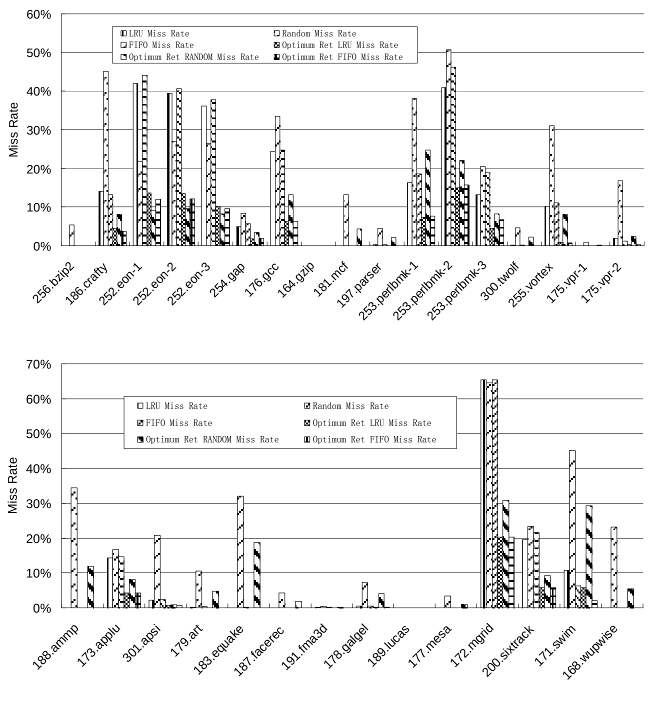


Figure 10. The miss rates of LRU, FIFO, RANDOM with and without return optimization

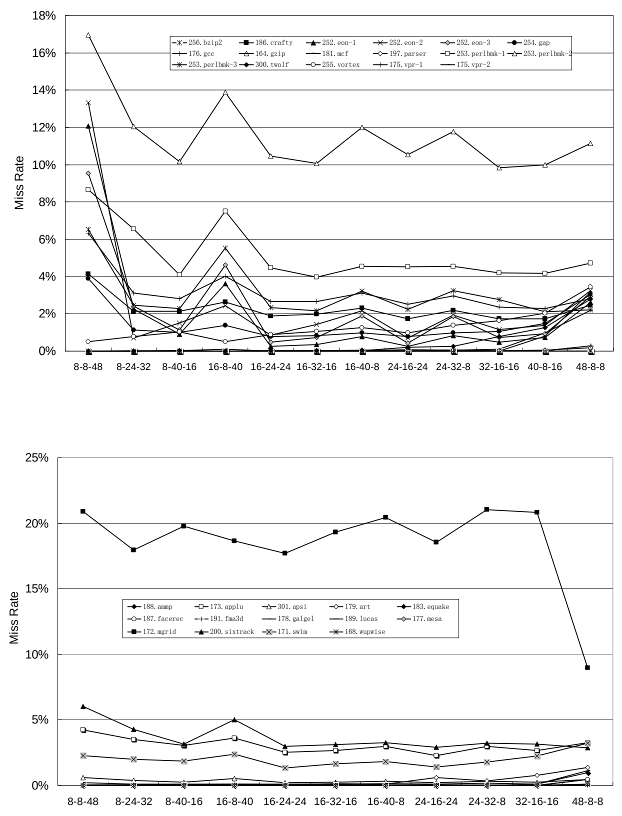


Figure 11. The miss rates of PCBPC algorithm with various values of H, B and R

To analyze the influence of H, B and R, we do some experiments by selecting various sizes of them. The miss rates of every example of SPEC CPU2000 in various selections of H, B and R are shown in Figure 11, in which the x-axis represents values of H, B and R. We can see that the miss rates are not changed evidently when H, B and R have different values. Given a H value, when we increases the size of R, the CAM Miss rate decreases for the programs with a large amount of *return* instructions, while it increases for another kind of programs with a large amount of *jump* and *call* instructions. The reason is that the more items serving for *return* instructions can bring benefit for the programs with large amount of *return* instructions.

Benchmark	Optimum Ret LRU Miss Rate	Optimum Ret RANDOM Miss Rate	Optimum Ret FIFO Miss Rate	Only Ret Miss Rate	PCBPC Miss Rate
256.bzip2	0.0001%	0.0002%	0.0001%	0.0002%	0.0001%
186.crafty	4.5373%	8.0331%	3.7822%	13.0182%	1.8731%
252.eon-1	13.6372%	9.1110%	12.0649%	28.6585%	0.2527%
252.eon-2	13.5144%	9.5812%	12.1627%	30.4033%	0.8446%
252.eon-3	10.1358%	8.1782%	9.6146%	27.9804%	0.4701%
254.gap	1.7658%	3.4641%	1.9716%	44.1090%	0.7753%
176.gcc	6.2648%	13.1817%	6.3034%	33.5304%	2.6510%
164.gzip	0.0001%	0.0044%	0.0001%	2.4190%	0.0002%

181.mcf	0.0015%	4.2753%	0.0015%	8.2124%	0.0016%
197.parser	0.0321%	2.0660%	0.0227%	4.3022%	0.0182%
253.perlbmk-1	7.4217%	24.7887%	7.7025%	48.4966%	4.4676%
253.perlbmk-2	15.0741%	22.0547%	15.7969%	34.9814%	10.4607%
253.perlbmk-3	4.5113%	8.3238%	6.6810%	36.4940%	2.3274%
300.twolf	0.0065%	2.2099%	0.0192%	3.7321%	0.0194%
255.vortex	0.8881%	8.0288%	0.7612%	8.0620%	0.8768%
175.vpr-1	0.0025%	0.0774%	0.0023%	7.3065%	0.0012%
175.vpr-2	0.2366%	2.3301%	0.2662%	25.9178%	0.0068%
188.ammp	0.0037%	11.9718%	0.0042%	24.5858%	0.0013%
173.applu	4.2282%	8.1412%	4.2230%	11.1684%	2.5322%
301.apsi	0.6319%	0.7743%	0.6144%	6.4679%	0.2169%
179.art	0.0509%	4.7058%	0.0623%	5.8463%	0.0825%
183.equake	0.0174%	18.7452%	0.0167%	21.7102%	0.0031%
187.facerec	0.0048%	1.8622%	0.0047%	17.7334%	0.0014%
191.fma3d	0.0466%	0.1073%	0.0493%	26.8824%	0.0375%
178.galgel	0.1160%	4.0472%	0.1225%	27.0676%	0.0990%
189.lucas	0.0002%	0.0004%	0.0002%	33.3302%	0.0002%
177.mesa	0.0056%	1.0358%	0.0056%	8.8704%	0.0001%
172.mgrid	20.2491%	30.7963%	20.2890%	28.2435%	17.7021%
200.sixtrack	5.7133%	9.1183%	5.7585%	45.8722%	2.9975%
171.swim	5.8117%	29.2146%	1.9716%	13.3228%	1.3308%
168.wupwise	0.0002%	5.3840%	0.0002%	21.1040%	0.0002%
Average	**3.7068%**	**8.1166%**	**3.5573%**	**20.9622%**	**1.6146%**

Table 2. The miss rates of PCBPC and other replacement algorithms

As to 172.mgrid, when H >= 40, its CAM miss rate has a big decrease. According to the experimental data, it is caused by the decrease of miss rate of *jump* and *call* instructions include indirect-jump-mapping. Because the indirect-jump-mapping ratio of *jump* and *call* instructions it contains is relatively large, and the hot sequence is very long.

We chose H=16, B=24, R=24, and compared PCBPC's CAM miss rate with return optimized LRU, FIFO, RANDOM replacement algorithms and R=64 PCBPC, as Table 2 shows. Experimental results showed that the average decrease of CAM miss rate is 32.09%, 37.58%, 84.86% and 92.21% respectively in SPEC CPU2000 benchmarks.

When H=16, B=24 and R=24, PCBPC's CAM miss rate is higher than LRU's in 164.gzip, 181.mcf, 255.virtex and 179.art, the reason is that the indirect-jump-mapping addresses concentrate in a small set and there are little hot addresses—as character 1 shown in Figure 3 and 4. But if the H, B and R are set appropriately according to the characteristics of the applications, PCBPC can achieve lower miss rate than LRU replacement in SPEC CPU2000.

The overhead of PCBPC is larger than FIFO replacement and smaller than LRU. If the addresses appear linearly and little repeated addresses; PCBPC has the similar effect and acting as FIFO replacement. For a fixed set of H, B and R values, PCBPC may not achieve better performance than other replacement algorithms in every application. However, as we tested in SPEC CPU2000 benchmarks, for a certain application, PCBPC can always achieve better performance by adjusting H, B and R to a proper value. We can see it from Figure 11.

6 Related Works

During the translation process, the source binary codes will be divided into basic blocks, corresponding to the superblocks of target codes. When the current superblock finishes execution, the next superblock's address needs to be determined through address mapping. In order to do so, a context switch from the local codes to controller of the translation system is needed. After getting the address, the context needs to be switched back to local codes. This process has a large overhead. In order to improve the efficiency, Jump instructions can be added at the end of each superblock to link them together so that switching overhead can be reduced. In Dynamo [9], Mojo [10], DELI [11] and DynamoRIO [12], etc, chaining (link) technology is used[4] [13].

In the binary translation system, the indirect-jump-mapping is usually implemented using hash tables. The initial address of every target program block needs to be mapped, which is heavy work. QEMU system[14] improved the efficiency through an extra relatively small but fast hash table. The address mapping is firstly looked up in the fast hash table. If it misses, the full table is then used for address mapping. Although the fast table doesn't include all elements, it stores the mostly used data, so the looking up process can be speeded up.

Software-Based Jump Target Prediction[4][5][15][16][12][17] uses a series of judging instructions to compare the source address in the registers with the source addresses of the judging instructions. If they are the same, the program jumps to the corresponding target address, otherwise, the forecast fails and the hash table approach is used.

Jump Target-address Lookup Table (JTLT) [4] [18] [19] [20] implemented mapping tables with hardware and one or two special instructions are adopted when looking up the table. If the target hit, the program jumps to the corresponding address. The detailed hardware implementation depends on different systems. For example, Transmeta[21] used a TLB with 256 items to implement the table.

Dual-address return address stack[4] is a dedicated hardware to handle *return* instructions, since more than 70% of indirect-mapping addresses are return addresses, it can bring great benefits.

Since the Loongson-3 provides a CAM and the instructions for software management, our goal is to find a way to let it bring optimal performance. In PCBPC algorithm, JTLB and Dual-address return address stack are embodied. In addition, PCBPC are more flexible than others.

7 Conclusions

Indirect-jump-mapping is the key performance factor of the binary translation and optimization system. In this paper, we use co-design method to seek a solution for this kind of systems' indirect-jump-mapping in order to improve its performance. In our designed binary system, software was used to simulate CAM's behavior, and gets its miss rate under different parameters for improving the hardware design. Then, based on large amount of experimental data, three runtime characteristics of indirect-jump-mapping were concluded: 1. The test examples have few *static-number*, and large *dynamic-amount* of addresses. Their addresses also have good locality. 2. *Return* instructions constitutes a large proportion in all instructions need address mapping. 3. The dynamic sequences of source addresses show phase and cyclic characteristic.

Based on the established hardware support architecture for indirect-jump-mapping and these runtime characteristics, we analyzed related replacement algorithms and proposed the PCBPC algorithm, which divides the CAM according to program behavior. The proposed replacement algorithm fully utilizes the indirect-jump-mapping of programs and outperforms LRU, FIFO, and RANDOM. Experimental results show that the proposed algorithm averagely reduced CAM miss rate by 32.09%、37.58% and 84.86% on SPEC CPU2000 benchmarks, compared with these three algorithms.

At present, the parameters of PCBPC algorithm are statically set. The values of each region cannot be changed during runtime. However, we found that for different benchmarks, the values of H, B and R to achieve the best performance are different. This indicates that the PCBPC algorithm can be improved by changing parameters dynamically according to characteristics of the application.

In addition, the Loongson-3 has only one CAM, but it may run more than one binary translation system at a time. The target programs supported by multiple binary translation systems need to share the only CAM, which leads to a new problem: how to share the CAM between many processes in order to get good performance? Several strategies can be used, such as empty CAM and a fixed CAM allocated to different processes. Which strategy is better needs further research.

8 Acknowledgements

The software platform of our experiments in this paper is DigitalBridge, which is developed by the Binary Translation Team of Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences. The hardware platform is based on Loongson CPU, which is developed by the Loongson Team.

Thanks Peng Li and Zhenjiang Wang for their help to revise the English expression errors in this paper.

References

1. Shiliang Hu, James E. Smith, "Reducing Startup Time in Co-Designed Virtual Machines", Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06), pp. 277-288, 2006.
2. A. Klaiber, "The Technology Behind Crusoe Processors", Transmeta Technical Brief, 2000.
3. K. Krewell, "Transmeta Gets More Efficeon" Microprocessor report. v.17, October 2003.
4. Ho-Seop Kim and James E. Smith, "Hardware Support for Control Transfers in Code Caches", Proceedings of the 36th International Symposium on Microarchitecture, 2003.
5. Ho-Seop Kim and James E. Smith, "Dynamic Binary Translation for Accumulator-oriented Architectures", The 1st International Symposium on Code Generation and Optimization, 2003.
6. Jose A. Joao, Onur Mutlu, Hyesoon Kim, Rishi Agarwal, Yale N. Patt, "Improving the performance of object-oriented languages with dynamic predication of indirect jumps", ASPLOS'08, pp. 80-90, March, 2008.
7. James E.Smith, Ravi Nair, "Virtual Machines——Versatile Platforms for Systems and Processes", Publishing House of Electronics Industry, pp. 339-347, July, 2006.
8. SPEC CPU2000, http://www.spec.org/cpu/.
9. Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System," Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1-12, Jun. 2000.
10. W. Chen, S. Lerner, R. Chaiken and D. Gillies, "Mojo: A Dynamic Optimization System", in Proceedings of the 3rd Workshop on Feedback-Directed and Dynamic Optimization, December 2000.
11. Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, Joseph A. Fisher, "DELI: A New RunTime Control Point," The 35th Int. Symp. Microarchitecture, pp. 257-268, Dec. 2002.
12. Derek Bruening, Timothy Garnett, Saman Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," Int. Symp. Code Generation and Optimization, pp. 265-275, Mar. 2003.
13. R. J. Hookway and M. A. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation," Digital Technical J., Vol. 9, No.1, pp. 3-12, 1997.
14. Fabrice Bellard, "QEMU, a Fast and Portable Dynamic Translator", 2005 USENIX Annual Technical Conference, 2005.
15. Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Transparent dynamic optimization: the design and implementation of Dynamo," Hewlett Packard Laboratories Technical Report HPL-1999-78, Jun. 1999.
16. Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System," Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1-12, Jun. 2000.
17. Kemal Ebcioglu, Erik R. Altman, Michael Gschwind, Sumedh Sathaye, "Dynamic Binary Translation and Optimization," IEEE Trans. Computers, Vol. 50, No. 6, pp. 529-548, Jun. 2001.
18. James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, Jim Mattson, "The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges", ACM International Conference Proceeding Series; Vol. 37, 2003.
19. Gabriel M. Silberman, Kemal Ebcioglu, "An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures," IEEE Computer, Vol. 26, No. 6, pp. 39-56, 1993.
20. Michael Gschwind, "Method and Apparatus for Determining Branch Addresses in Programs Generated by Binary Translation, IBM Disclosures YOR819980334, Jul. 1998.

21. Michael Gschwind, Eric R. Altman, “Optimizing and Precise Exceptions in Dynamic Compilation,” Second Workshop on Binary Translation Held in PACT 2000.
22. Order Number 243190, The Intel Architecture Software Developer’s Manual, Volume 1: Basic Architecture, 1997.
23. 245317-002, Intel IA-64 Arichitecture Software Developer’s Manual Volume 1: IA-64 Application Architecture, Revision 1.1, July 2000.
24. M Srinivasan, “Method and apparatus for emulating status flag,” US Patent 5774694, 1998.
25. Marek Chrobak, John Noga, “LRU is better than FIFO,” Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms, pp. 78-81, 1998.
26. Yannick Deville, “A low-cost usage-based replacement algorithm for cache memories”, ACM SIGARCH Computer Architecture News, Volume 18, Issue 4, pp. 52-58, Dec. 1990.

Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure

Darek Mihocka
Emulators
darekm@emulators.com

Stanislav Shwartsman
Intel Corp.
stanislav.shwartsman@intel.com

Abstract

A recent trend in x86 virtualization products from Microsoft, VMware, and XenSource has been the reliance on hardware virtualization features found in current 64-bit microprocessors. Hardware virtualization allows for direct execution of guest code and potentially simplifies the implementation of a Virtual Machine Monitor (or "hypervisor")[1]. Until recently, hypervisors on the PC platform have relied on a variety of techniques ranging from the slow but simple approach of pure interpretation, the memory intensive approach of dynamic recompilation of guest code into translated code cache, to a hardware assisted technique known as "ring compression" which relies on the host MMU for hardware memory protection. These techniques traditionally either deliver poor performance[2], or are not portable. This makes most virtualization products unsuitable for use on cell phones, on ultra-mobile notebooks such as ASUS EEE or OLPC XO, on game consoles such as Xbox 360 or Sony Playstation 3, or on older Windows 98/2000/XP class PCs.

This paper describes ongoing research into the design of portable virtual machines which can be written in C or C++, have reasonable performance characteristics, could be hosted on PowerPC, ARM, and legacy x86 based systems, and provide practical backward compatibility and security features not possible with hardware based virtualization.

Keywords

Bochs, Emulation, Gemulator, TLB, Trace Cache, Virtualization

1.0 Introduction

At its core, a virtual machine is an indirection engine which intercepts code and data accesses of a sandboxed "guest" application or operating system and remaps them to code sequences and data accesses on a "host" system. Guest code is remapped to functionally identical code on the host, using binary translation or sandboxed direct execution. Guest data accesses are remapped to host memory and checked for access privilege rights, using software or hardware supported address translation.

When a guest virtual machine and host share a common instruction set architecture (ISA) and memory model, this is commonly referred to as "virtualization". VMware Fusion [3] for running Windows applications inside of Mac OS X, Microsoft's Hyper-V [4] feature in Windows Server 2008, and Xen [5] are examples of virtualization products. These virtual machines give the illusion of full-speed direct execution of native guest code. However, the code and data accesses in the guest are strictly monitored and controlled by the host's memory management hardware. Any attempt to access a memory address not permitted to the guest results in an exception, typically an access violation page fault or a "VM exit event", which hands control over to the hypervisor on the host. The faulting instruction is then emulated and either aborted, re-executed, or skipped over. This technique of virtualization is also known as "trap-and-emulate" since certain guest instructions must be emulated instead of executed directly in order to maintain the sandbox.[6].

Virtualization products need to be fast since their goal is to provide hardware-assisted isolation with minimal runtime overhead, and therefore generally use very specific assembly language optimizations and hardware features of a given host architecture.

A more general class of virtual machine is able to handle guest architectures which differ from the host architecture by emulating each and every guest instruction. Using some form of binary translation, either bytecode interpretation or dynamic recompilation or both, such virtual machines are able to work around differences in ISA, memory models, endianness, and other differentiating factors that prevent direct execution.

Emulation techniques generally have a noticeable slowdown compared to virtualization, but have benefits such as being able to easily capture traces of the guest code or inject instrumentation. Dynamic instrumentation frameworks such as Intel's Pin [7] and PinOS [8], Microsoft's Nirvana[9], PTLsim[10], and DynamoRIO[11] programmatically intercept guest code and data accesses, allowing for the implementation of reverse execution debugging and performance analysis tools. These frameworks are powerful but can incur orders of magnitude slowdown due to the guest-to-host context switching on each and every guest instruction.

Emulation products also end up getting customized in host-specific ways for performance at the cost of portability. Apple's original 68020 emulator for PowerPC based Macs [12] and their more recent Rosetta engine in Mac OS X which run PowerPC code on x86 [13] are examples of much targeted pairings of guest and host architectures in an emulator.

As virtualization products increasingly rely on hardware-assisted acceleration for very specialized usage scenarios, their value diminishes for use on legacy and low-end PC systems, for use on new consumer electronics platforms such as game consoles and cell phones, and for instrumentation and analysis scenarios. As new devices appear, time-to-market may be reduced by avoiding such one-off emulation products that are optimized for a particular guest-host pairing.

1.1 Overview of a Portable Virtual Machine Infrastructure

For many use cases of virtualization we believe that it is a fundamental design flaw to merely target the maximization of best-case performance above all else, as has been the recent trend. Such an approach not only results in hardware-specific implementations which lock customers into limited hardware choices, but the potentially poor worst-case performance may result in unsatisfactory overall performance[14].

We are pursuing a virtual machine design that delivers fast CPU emulation performance but where portability and versatility are more important than simply maximizing peak performance.

We tackled numerous design issues, including:

- Maintaining portability across legacy x86 and non-x86 host systems, and thus eliminating the use of host-dependent optimizations,
- Bounding the memory overhead of a hypervisor to allow running in memory constrained environments such as cell phones and game consoles,
- Bounding worst-case performance, and thus allowing for efficient tracing and run-time analysis,
- Efficiently dispatching guest instructions on the host,
- Efficiently mapping guest memory to the host, and,
- Exploring simple and lightweight hardware acceleration alternatives.

Our research on two different virtual machines – the Bochs portable PC emulator which simulates both 32-bit x86 and x86-64 architectures, and the Gemulator[15] Atari ST and Apple Macintosh 68040 emulator on Windows – shows that in both cases it is possible to achieve full system guest simulation speeds in excess of 100 MIPS (millions of instructions per second) using purely interpreted execution which does not rely on hardware MMU capabilities or even on dynamic recompilation. This work is still in progress, and we believe that further performance improvements are possible using interpreted execution to where a portable virtual machine running on a modern Intel Core 2 or PowerPC system could achieve performance levels equal to what less than ten years ago would have been a top of the line Intel Pentium III based desktop.

A portable implementation offers other benefits over vendor specific implementations, such as deterministic execution, i.e. the ability to take a saved state of a virtual machine and re-execute the guest code with deterministic results regardless of the host. For example, it would be highly undesirable for a virtual machine to suddenly behave differently simply because the user chose to upgrade his host hardware.

Portability suggests that a virtual machine's hypervisor should be written in a high level language such as C or C++. A portable hypervisor needs to support differences in endianness between guest and host, and even differences in register width and address space sizes between guest and host should not be blocking issues. An implementation based on a high level language must be careful to try to maintain a bounded memory footprint, which is better suited for mobile and embedded devices.

Maintaining portability and bounding the memory footprint led to the realization that dynamic recompilation (also known as just-in-time compilation or "jitting") may not deliver beneficial speed gains over a purely interpreted approach. This is due to various factors, including the very long latencies incurred on today's microprocessors for executing freshly jitted code, the greater code cache and L2 cache pressure which jitting produces, and the greater cost of detecting and properly handling self-modifying code in the guest. Our approach therefore does not rely on jitting as its primary execution mechanism.

Supporting the purely-interpreted argument is an easily overlooked aspect of the Intel Core and Core 2 architectures: the stunning efficiency with which these processors execute interpreted code. In benchmarks first conducted in 2006 on similar Windows machines, we found that a 2.66 GHz Intel Core 2 based system will consistently deliver two to three times the performance of a 2.66 GHz Intel Pentium 4 based system when running interpretation based virtual machines such as SoftMac[16]. Similar results have been seen with Gemulator, Bochs, and other interpreters. In one hardware generation on Intel microprocessors, interpreted virtual machines make a lot more sense.

Another important design goal is to provide guest instrumentation functionality similar to Pin and Nirvana, but with less of a performance penalty when such instrumentation is active. This requires that the amount of context switching involved between guest state and host state be kept to a minimum, which once again points the design at an interpreted approach. Such a low-overhead instrumentation mechanism opens the possibilities to performing security checks and analysis on the guest code as it is running in a way that is less intrusive than trying to inject it into the guest machine itself. Imagine a virus detection or hot-patching mechanism that is built into the hypervisor which then protects otherwise vulnerable guest code. Such a proof-of-concept has already been demonstrated at Microsoft using a Nirvana based approach called Vigilante[17]. Most direct execution based hypervisors are not capable of this feat today.

Assumptions taken for granted by virtual machine designs of the past need to be re-evaluated for use on today's CPU designs. For example, with the popularity of low power portable devices one should not design a virtual machine that assumes that hardware FPU (floating point unit) is present, or even that a hardware memory management is available.

Recent research from Microsoft's Singularity project [18] shows that software-based memory translation and isolation is an efficient means to avoid costly hardware context switches. We will demonstrate a software-only memory translation mechanism which efficiently performs guest-to-host memory translation, enforces access privilege checks, detects and deals with self-modifying code, and performs byte swapping between guest and host as necessary.

Finally, we will identify those aspects of current micro-architectures which impede efficient virtual machine implementation and propose simple x86 ISA extensions which could provide lightweight hardware-assisted acceleration to an interpreted virtual machine.

In the long term such ISA extensions, combined with a BIOS-resident virtual machine, could allow future x86 microprocessor implementations to completely remove not only hardware related to 16-bit legacy support, but also hardware related to segmentation, paging, heavyweight virtualization, and rarely used x86 instructions. This would reduce die sizes and simplify the hardware verification. Much as was the approach of Transmeta in the late 1990's, the purpose of the microprocessor becomes that of being an efficient host for virtual machines[19].

1.2 Overview of This Paper

The premise of this paper is that an efficient and portable virtual machine can be developed using a high-level language that uses purely interpreted techniques. To show this we looked at two very different real-world virtual machines - Gemulator and Bochs - which were independently developed since the 1990s to emulate 68040 and x86 guest systems respectively. Since these emulators are both interpretation based and are still actively maintained by each of the authors of this paper, they served as excellent test cases to see if similar optimization and design techniques could be applied to both.

Section 2 discusses the design of Gemulator and looks at several past and present techniques used to implement its 68040 ISA interpreter. Gemulator was originally developed almost entirely in x86 assembly code that was very x86 MS-DOS and x86 Windows specific and not portable even to a 64-bit Windows platform.

Section 3 discusses the design of Bochs, and some of the many optimization and portability techniques used for its Bochs x86 interpreter. The work on Bochs focused on improving the performance of its existing portable C++ code as well as eliminating pieces of non-portable assembly code in an efficient manner.

Based on the common techniques that resulted from the work on both Gemulator and Bochs and the common problems encountered in both – guest-to-host address translation and guest flags emulation - Section 4 proposes simple ISA hardware extensions which we feel could aid the performance of any arbitrary interpreter based virtual machine.

2.0 Gemulator

Gemulator is an MS-DOS and Windows hosted emulator which runs Atari 800, Atari ST, and classic 680x0 Apple Macintosh software. The beginnings of Gemulator date back to 1987 as a tutorial on assembly language and emulation in the Atari magazine ST-LOG[20]. In 1991, emulation of the Motorola MC68000 microprocessor and the Atari ST chipset was added, and in 1997 a native 32-bit Windows version of Gemulator was developed which eventually added support for a 68040 guest running Mac OS 8.1 in a release called "SoftMac". Each release of Gemulator was based around a 68000/68040 bytecode interpreter written in 80386 assembly language and which was laboriously retuned every few years for 486, Pentium Pro "P6", and Pentium 4 "Netburst" cores.

In the summer of 2007 work began to start converting the Gemulator code to C for eventually hosting on both 32-bit and 64-bit host machines. Because of the endian difference between 68000/68040 and 80386 architectures, it was a goal to keep the new C code as byte agnostic as possible. And of course, the conversion from 80386 assembly language to C should incur as little performance penalty as possible.

The work so far on Gemulator 9.0 has focused on converting the guest data memory access logic to portable code, and examining the pros and cons of various guest-to-host address translation techniques which have been used over the years and selecting the one that best balances efficiency and portability.

2.1 Byte Swapping on the Intel 80386

A little-endian architecture such as Intel 80386 stores the least significant byte first, while a big-endian architecture such as Motorola 68000 stores the most significant byte first. Byte values, such as ASCII text characters, are stored identically, so an array of bytes, or a text string is stored in memory the same regardless of endianness.

Since the 80386 did not support a BSWAP instruction, the technique in Gemulator was to treat all of guest memory address space - all 16 megabytes of it for 68000 guests - as one giant integer stored in reverse order. Whereas a 68000 stores memory addresses G, G+1, G+2, etc. in ascending order, Gemulator maps guest address G to host address H, G+1 maps to H-1, G+2 maps to H-2, etc. G + H is constant, such that G = K – H, and H = K – G.

Multi-byte data types, such as a 32-bit integer can be accessed directly from guest space by applying a small adjustment to account for the size of the data access. For example, to read a 32-bit integer from guest address 100, calculate the guest address corresponding to the last byte of that access before negating, so in this case guest address 100 + sizeof(int) – 1 = guest address 103. The memory read *(int *)&K[-103] correctly returns the guest data.

The early-1990's releases of Gemulator were hosted on MS-DOS and on Windows 3.1, and thus did not have the benefit of Win32 or Linux style memory protection and mapping APIs. As such these interpreters also bounds checked each negated guest offset such that only guest RAM accesses (usually guest addresses 0 through 4 megabytes) used the direct access, while all other accesses, including to guest ROM space, video frame buffer RAM, and memory mapped I/O, took a slower path through a hardware emulation handler.

Instrumentation showed that only about 1 in 1000 memory accesses on average failed the bounds check, allowing roughly 99.9% of guest memory accesses to use the "adjust-and-negate" bounds checking scheme, and this allowed a 33 MHz 80386 based PC to efficiently emulate close to the full speed of the original 8 MHz 68000 Atari ST and Apple Macintosh computers.

2.2 Page Table using XOR Translation

A different technique must be used when mapping the entire 32-bit 4-gigabyte address space of the 68040 to the smaller than 2-gigabyte address of a Windows application. The answer relies on the observation that subtracting an integer value from 0xFFFFFFFF gives the same result as XOR-ing that same value to 0xFFFFFFFF. For example:

```
0xFFFFFFFF – 0x12345678 = 0xEDCBA987
0xFFFFFFFF XOR 0x12345678 = 0xEDCBA987
```

This observation allows for portions of the guest address space to be mapped to the host in power-of-2 sized power-of-2-aligned blocks. The XOR operation, instead of a subtraction, is used to perform the byte-swapping address translation. Every byte within each such block will have a unique XOR constant such that the H = K –G property is maintained.

For example, mapping 256 megabytes of Macintosh RAM from guest address 0x00000000..0x0FFFFFFF to a 256-megabyte aligned host block allocated at address 0x30000000 requires that the XOR constant be 0x3FFFFFFF, which is derived taking either the XOR of the address of that host block and the last byte of the guest range (0x30000000 XOR 0x0FFFFFFF) or the first address of the guest range and the last byte of the allocated host range (0x00000000 XOR 0x3FFFFFFF). Guest address 0x00012345 thus maps to host address 0x3FFFFFFF – 0x00012345 = 0x3FFEDCBA for this particular allocation.

To reduce fragmentation, Gemulator starts with the largest guest block to map and then allocates progressively smaller blocks, the order usually being guest RAM, then guest ROM, then guest video frame buffer. The algorithm used is as follows:

```
for each of the RAM ROM and video guest address ranges
 {
 calculate the size of that memory range rounded up to next power of 2
 for each megabyte-sized range of Windows host address space
 {
   calculate the XOR constant for the first and last byte of the block
   if the two XOR constants are identical
   {
     call VirtualAlloc() to request that specific host address range
     if successful record the address and break out of loop;
   }
 }
}
```

Listing 2.1: Pseudo code of Gemulator's allocator

This algorithm scans for host free blocks a megabyte at a time because it turns out the power-of-2 alignment need not match the block size. This helps to find large unused blocks of host address space when memory fragmentation is present.

For example, a gigabyte of Macintosh address space 0x00000000 through 0x3FFFFFFF can map to Windows host space 0x20000000 though 0x5FFFFFFF because there exists a consistent XOR constant:

```
  0x5FFFFFFF XOR 0x00000000 = 0x5FFFFFFF
  0x20000000 XOR 0x3FFFFFFF = 0x5FFFFFFF
```

This XOR-based translation is endian agnostic. When host and guest are of the same endianness, the XOR constant will have zeroes in its lower bits. When the host and guest are of opposite endianness, as is the case with 68040 and x86, the XOR constant has its lower bits set. How many bits are set or cleared depends on the page size granularity of mapping.

A granularity of 64K was decided upon based on the fact that the smallest Apple Macintosh ROM is 64K in size. Mapping 4 gigabytes of guest address space at 64K granularity generates 4GB/64K = 65536 different guest address ranges. A 65536-entry software page table is used, and the original address negation and bounds check from before is now a traditional table lookup which uses XOR to convert the input guest address in EBP to a host address in EDI::

```
; Convert 68000 address to host address in EDI
; Sign flag is SET if EA did not map.
    mov   edi,ebp
    shr   ebp,16
    xor   edi,dword ptr[pagetbl+ebp*4]
```

Listing 2.2: Guest-to-host mapping using flat page table

For unmappable guest addresses ranges such as memory mapped I/O, the XOR constant for that range is selected such that the resulting value in EDI maps to above 0x80000000. This can now be checked with an explicit JS (jump signed) conditional branch to the hardware emulation handler, or by the resulting access violation page fault which invokes the same hardware emulation handler via a trap-and-emulate.

This design suffers from a non-portable flaw – it assumes that 32-bit user mode addresses on Windows do not exceed address 0x80000000, an assumption that is outright invalid on 64-bit Windows and other operating systems.

The code also does not check for misaligned accesses or accesses across a page boundary, which prevents further sub-allocation of the guest address space into smaller regions. Reducing the granularity of the mapping also inversely grows the size of the lookup table. Using 4K mapping granularity for example requires 4GB/4K = 1048576 entries consuming 4 megabytes of host memory.

2.3 Fine-Grained Guest TLB

The approach now used by Gemulator 9 combines the two methods – range check using a lookup table of only 2048 entries - effectively implementing a software-based TLB for guest addresses. Each table entry still spans a specific guest address range but now holds two values: the XOR translation value for that range, and the corresponding base guest address of the mapped range. This code sequence is used to translate for a guest write access of a 16-bit integer using 128-byte granularity:

```
mov    edx,ebp
shr    edx,bitsSpan  ; bitsSpan = 7
and    edx,dwIndexMask ; dwIndexMask = 2047
mov    ecx,ebp       ; guest address
add    ecx,cb-1      ; high address of access
; XOR to compare with the cached address
xor     ecx,dword ptr [memtlbRW+edx*8]
; prefetch translation XOR value
mov   eax,dword ptr [memtlbRW+edx*8+4]
test    ecx,dwBaseMask
jne     emulate       ; if no match, go emulate
xor     eax,ebp      ; otherwise translate
```

Listing 2.3: Guest-to-host mapping using a software TLB

The first XOR operation takes the highest address of the access and compares it to the base of the address range translated by that entry. When the two numbers are in range, all but a few lower bits of the result will be zero. The TEST instruction is used to mask out the irrelevant low bits and check that the high bits did match. If the result is non-zero, indicating a TLB miss or a cross-block access, the JNE branch is taken to the slow emulation path. The second XOR performs the translation as in the page table scheme.

Various block translation granularities and TLB sizes were tested for performance and hit rates. The traditional 4K granularity was tried and then reduced by factors of two. Instrumentation counts showed that hit rates remained good for smaller granularities even of 128 bytes, 64 bytes, and 32 bytes, giving the fine grained TLB mechanism between 96% and 99% data access hit rate for a mixed set of Atari ST and Mac OS 8.1 test runs.

The key to hit rate is not in the size of the translation granularity, since data access patterns tend to be scattered, but rather the key is to have enough entries in the TLB table to handle the scattering of guest memory accesses. A value of at least 512 entries was found to provide acceptable performance, with 2048 entries giving the best hit rates. Beyond 2048 entries, performance improvement for the Mac and Atari ST workloads was negligible and merely consumed extra host memory.

It was found that certain large memory copy benchmarks did poorly with this scheme. This was due to two factors:

- 64K aliasing of source and destination addresses, and,
- Frequent TLB misses for sequential accesses in guest memory space.

The 64K aliasing problem occurs because a direct-mapped table of 2048 entries spanning 32-byte guest address ranges wraps around every 64K of guest address space. The 32-byte granularity also means that for sequential accesses, every 8th 32-bit access will "miss". For these two reasons, a block granularity of 128 bytes is used so as to increase the aliasing interval to 256K.

Also to better address aliasing, three translation tables are used – a TLB for write and read-modify-write guess accesses, a TLB for read-only guest accesses, and a TLB for code translation and dispatch. This allows guest code to execute a block memory copy without suffer from aliasing between the program counter, the source of the copy, or the destination of the copy.

The code TLB is kept at 32-byte granularity and contains extra entries holding a dispatch address for each of the 32 addresses in the range. When a code TLB entry is populated, the 32 dispatch addresses are initialized to point to a stub function, similar to how jitting schemes work. When an address is dispatched, the stub function calculates the true address of the handlers and updates the entry in the table.

To handle self-modifying code, when a code TLB entry is first populated, the corresponding entry (if present) is flushed from the write TLB. Similarly, when a write TLB entry misses, it flushes six code TLB entries – the four entries corresponding to the 128-byte data range covered by the write TLB entry, and one code "guard block" on either side are flushed. This serves two purposes:

- To ensure that an address range of guest memory is never cached as both writable data and as executable code, such that writes to code space are always noted by the virtual machine, and,
- To permit contiguous code TLB blocks to flow into each other, eliminating the need for an address check on each guest instruction dispatch.

Keeping code block granularity small along with relatively small data granularity means that code execution and data writes can be made to the same 4K page of guest memory with less chance of false detection of self-modifying code and eviction of TLB entries as can happen when using the standard 4K page granularity. Legacy 68000 code is known to place writeable data near code, as well as using back-patching and other self-modification to achieve better performance.

This three-TLB approach gives the best bounded behavior of any of the previous Gemulator implementations. Unlike the original MS-DOS implementation, guest ROM and video accesses are not penalized for failing a bounds check. Unlike the previous Windows implementations, all guest memory accesses are verified in software and require no "trap-and-emulate" fault handler.

The total host-side memory footprint of the three translation tables is:

- 2048 * 8 bytes = 16K for write TLB
- 2048 * 8 bytes = 16K for read TLB
- 2048 * 8 bytes = 16K for code TLB
- 65536*4 = 256K for code dispatch entries

This results in an overall memory footprint of just over 300 kilobytes for all of the data structures relating to address translation and cached instruction dispatch.

For portability to non-x86 host platforms, the 10-instruction assembly language sequence was converted to this inlined C function to perform the TLB lookup, while the actual memory dereference occurs at the call site within each guest memory write accessor:

```
void * pvGuestLogicalWrite(
  ULONG addr, unsigned cb)
{
  ULONG ispan;
  ispan = (((addr + cb - 1) >> bitsSpan)
         & dwIndexMask);

  void *p;
  p = ((addr ^ vpregs->memtlbRW[ispan*2+1])
         - (cb - 1));

  if (0 == (dwBaseMask &
         (addr ^ (vpregs->memtlbRW[ispan*2]))))
  {
    return p;
  }
  return NULL;
}
```

Listing 2.4: Software TLB lookup in C/C++

This code compiles into almost as efficient a code sequence as the original assembly code, except for a spill of ECX which the Microsoft Visual Studio compiler generates, mandated by the __fastcall calling convention of preserving the ECX register.

On a 2.66 GHz Intel Core 2 host computer, the 68000 and 68040 instruction dispatch rate is about 120 to 170 million guest instructions per second, or approximately one guest instruction dispatch every 15 to 22 host clock cycles, depending on the Atari ST or Mac OS workload.

The aggregate hit rate for the read TLB and write TLB is typically over 95% while the hit rate for the code TLB's dispatch entries exceeds 98%.

For example, a workload consisting of booting Mac OS 8.1, launching the Speedometer 3.23 benchmarking utility, and running a short suite of arithmetic, floating point, and graphics benchmarks dispatches a total of 3.216 billion guest instructions of which 43 million were not already cached, a 98.6% hit rate on instruction dispatch.

That same scenario generates 3.014 billion guest data read and write accesses of which 132 million failed to translate, for a 95.6% hit rate. The misses include accesses to memory mapped I/O that never maps directly to the host.

This latest implementation of Gemulator now has very favorable and portable characteristics:

- Runs on the minimal "least common denominator" IA32 instruction set of 80386 which performs efficient byte swapping without requiring a host to support a BSWAP instruction,
- Short and predictably low-latency code paths,
- No exceptions are thrown as all guest memory accesses are range checked,
- Less than 1 megabyte of scratch working memory.

These characteristics are applicable not just to running 68040 guest code, but for more modern byte-swapping scenarios such as running PowerPC guest code on x86, or running x86 guest code on PowerPC.

The high hit rate of guest instruction dispatch and guest memory translation means that the majority of 68000 and 68040 instructions are simulated using short code paths involving translation functions with excellent branch prediction characteristics. As is described in the following section, improving the branch prediction rates on the host is critical.

3.0 Bochs

Bochs is a highly portable open source IA-32 PC emulator written purely in C++ that runs on most popular platforms. It includes emulation of the CPU engine, common I/O devices, and custom BIOS. Bochs can be compiled to emulate any modern x86 CPU architecture, including most recent Core 2 Duo instruction extensions. Bochs is capable of running most operating systems including MS-DOS, Linux, Windows 9X/NT/2000/XP and Windows Vista. Bochs was written by Kevin Lawton and currently maintained by the Bochs open source project[21]. Unlike most of its competitors like QEMU, Xen or VMware, Bochs doesn't feature a dynamic translation or virtualization technologies but uses pure interpretation to emulate the CPU.

During our work we took the official Bochs 2.3.5 release sources tree and made it run over than three times faster using only **host independent and portable** optimization techniques without affecting emulation accuracy.

3.1 Quick introduction to Bochs internals

Our optimizations concentrated in the CPU module of the Bochs full system emulator and mainly dealt with the primary emulation loop optimization, called the CPU loop. According to Bochs 2.3.5 profiling data, the CPU loop took around 50% of total emulation time. It turned out that while every instruction emulated relatively efficiently, Bochs spent a lot of effort for routine operations like fetching, decoding and dispatching instructions.

The Bochs 2.3.5 CPU main emulation loop looks very similar to that of a physical non-pipelined micro-coded CPU like Motorola 68000 or Intel 8086 [22]. Every emulated instruction passes through six stages during the emulation:

1. At prefetch stage, the instruction pointer is checked for fetch permission according to current privilege level and code segment limits, and host instruction fetch pointer is calculated. The prefetch code also updates memory page timestamps used for self modifying code detection by memory accesses.

2. After prefetch stage is complete the specific instruction could be looked up in Bochs' internal cache or fetched from the memory and decoded.

3. When the emulator has obtained an instruction, it can be instrumented on-the-fly by internal or external debugger or instrumentation tools.

4. In case an instruction contains memory references, the effective address of an instruction is calculated using an indirect call to the resolve memory reference function.

5. The instruction is executed using an indirect call dispatch to the instruction's execution method, stored together with instruction decode information.

6. At instruction commit the internal CPU EIP state is updated. The previous state is used to return to the last executed instruction in case of an x86 architectural fault occurring during the current instruction's execution.

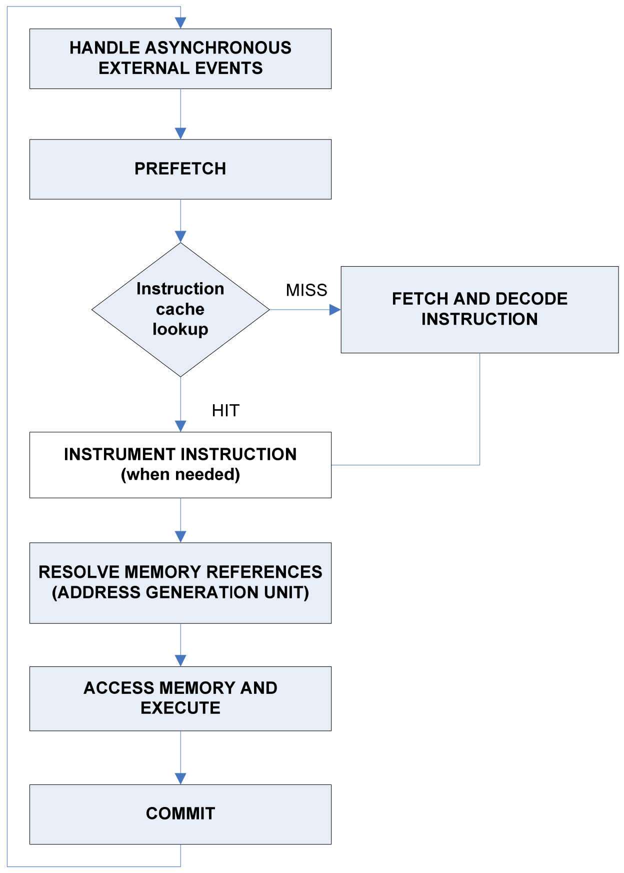

HANDLE ASYNCHRONOUS EXTERNAL EVENTS
PREFETCH
Instruction cache lookup
MISS
FETCH AND DECODE INSTRUCTION
HIT
INSTRUMENT INSTRUCTION (when needed)
RESOLVE MEMORY REFERENCES (ADDRESS GENERATION UNIT)
ACCESS MEMORY AND EXECUTE
COMMIT

Figure 3.1: Bochs CPU loop state diagram

As emulation speed is bounded by the latency of these six stages, shortening any and each of them immediately affects emulation performance.

3.2 Taking hardware ideas into emulation – using decoded instructions trace cache

Variable length x86 instructions, many different decoding templates, three possible operand and address sizes in x86-64 mode make instruction fetch-decode operations one of the heaviest parts of x86 emulation. The Bochs community realized this and introduced the decoded instruction cache to Bochs 2.0 at the end of 2002. The cache almost doubled the emulator performance.

The Pentium 4 processor stores decoded and executed instruction blocks into a trace cache[23] containing up to 12K of micro-ops. The next time when the execution engine needs the same block of instructions, it can be fetched from the trace cache instead of being loaded from the memory and decoded again. The Pentium 4 trace cache operates in two modes. In the "execute mode" the trace is feeding micro-ops stored in the trace to the execution engine. This is the mode that the trace cache normally runs in. Once a trace cache miss occurs the trace cache switches into the "build mode". In this mode the fetch-decode engine fetches and decodes x86 instructions from memory and builds a micro-ops trace which is stored in the cache.

The trace cache introduced into Bochs 2.3.6 is very similar to the Pentium 4 hardware implementation. Bochs maintains a decoded instruction trace cache organized as a 32768-entry direct mapped array with each entry holding a trace of up to 16 instructions. The tracing engine stops when it detects an x86 instruction able to affect control flow of the guest code, such as a branch taken, an undefined opcode, a page table invalidation or a write to control registers. Speculative tracing through potentially non-taken conditional branches is allowed. An early-out technique is used to stop trace execution when a taken branch occurs.

When the Bochs CPU loop is executing instructions from the trace cache, all front-end overhead of prefetch and decode is eliminated. Our experiments with a Windows XP guest show most traces to be less than 7 guest instructions in length and almost none longer than 16.

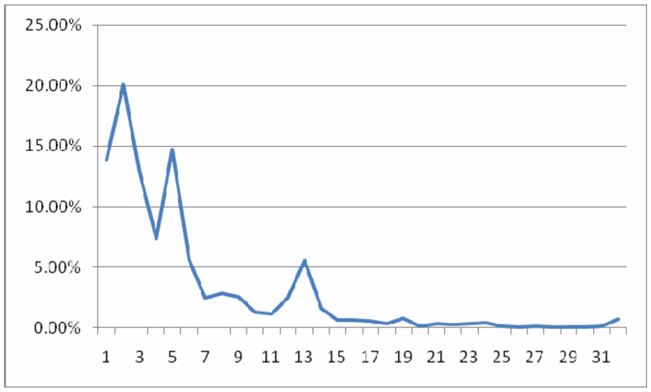


Figure 3.2: Trace length distribution for Windows XP boot

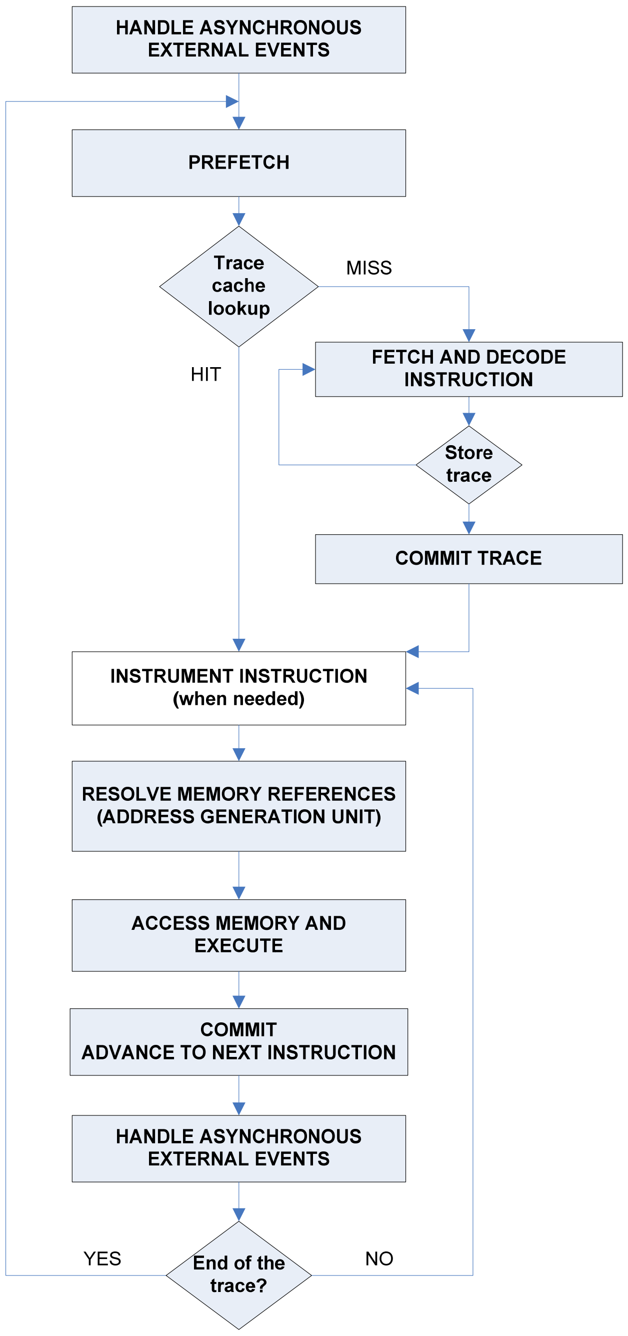

HANDLE ASYNCHRONOUS EXTERNAL EVENTS
PREFETCH
Trace cache lookup
MISS
HIT
FETCH AND DECODE INSTRUCTION
Store trace
COMMIT TRACE
INSTRUMENT INSTRUCTION (when needed)
RESOLVE MEMORY REFERENCES (ADDRESS GENERATION UNIT)
ACCESS MEMORY AND EXECUTE
COMMIT ADVANCE TO NEXT INSTRUCTION
HANDLE ASYNCHRONOUS EXTERNAL EVENTS
YES
End of the trace?
NO

Figure 3.3: Bochs CPU loop state diagram with trace cache

In addition to the over 20% speedup in Bochs emulation, the trace cache has great potential for the future. We are working on the following techniques which will help to double emulation speed again in a short term:

- Complicated x86 instructions could be decoded to several simpler micro-ops in the trace cache and handled more efficiently by the emulator.

- Compiler optimization techniques can be applied to the hot traces in the trace cache. Register move elimination, no-op elimination, combining memory accesses, replacing instruction dispatch handlers, and redundant x86 flags update elimination are only a few techniques that can be applied to make hot traces run faster.

The software trace cache's primary problem is direct mapped associativity. This can lead to frequent trace cache collisions due to aliasing of addresses at 32K and larger power-of-two intervals. Hardware caches use multi-way associativity to avoid aliasing issues. A software implementation of a two- or four-way associative cache and LRU management can potentially increase branch misprediction during lookup, reducing cache gain to a minimum.

What Bochs does instead today is use a 65536-entry table. A hash function calculates the trace cache index of guest address X using this formula:

- *index := (X + (X<<2) + (X>>6)) mod 65536*

We found that the best trace cache hashing function requires both a left shift and a right shift, providing the non-linearity so that two blocks of code separated by approximately a power-of-two interval will likely not conflict.

3.3 Host branch misprediction as biggest cause of slow emulation performance

Every pipelined processor features branch prediction logic used to predict whether a conditional branch in the instruction flow of a program is likely to be taken or not. Branch predictors are crucial in today's modern, superscalar processors for achieving high performance.

Modern CPU architectures implement a set of sophisticated branch predictions algorithms in order to achieve highest prediction rate, combining both static and dynamic prediction methods. When a branch instruction is executed, the branch history is stored inside the processor. Once branch history is available, the processor can predict branch outcome – whether the branch should be taken and the branch target.

The processor uses branch history tables and branch target buffers to predict the direction and target of branches based on the branch instruction's address.

The Core micro-architecture branch predictor makes the following types of predictions:

- Direct calls and jumps. The jump targets are read from the branch target array regardless of the taken/not taken prediction.

- Indirect calls and jumps. May either be predicted as having a monotonic target or as having targets that vary in accordance with recent program behavior.

- Conditional branches. Predicts branch target and whether or not the branch will be taken.

- Returns from procedure calls. The branch predictor contains a 16-entry return stack buffer. It enables accurate prediction for RET instructions.

Let's look closer into at the Bochs 2.3.5 main CPU emulation loop. As can be seen the CPU loop alone already gives enough work to the branch predictor due to two indirect calls right in the heart of the emulation loop, one for calculating the effective address of memory accessing instructions, and another for dispatching to the instruction execution method. In addition to these indirect calls many instruction methods contain conditional branches in order to distinguish different operand sizes or register/memory instruction format.

A typical Bochs instruction handler method:

```
void BX_CPU_C::SUB_EdGd(bxInstruction_c *i)
{
  Bit32u op2_32, op1_32, diff_32;

  op2_32 = BX_READ_32BIT_REG(i->nnn());

  if (i->modC0()) {    // reg/reg format
    op1_32 = BX_READ_32BIT_REG(i->rm());
    diff_32 = op1_32 - op2_32;
    BX_WRITE_32BIT_REGZ(i->rm(), diff_32);
  }
  else {               // mem/reg format
    read_RMW_virtual_dword(i->seg(),
        RMAddr(i), &op1_32);
    diff_32 = op1_32 - op2_32;
    Write_RMW_virtual_dword(diff_32);
  }
  SET_LAZY_FLAGS_SUB32(op1_32, op2_32,
        diff_32);
}
```

Listing 3.1: A typical Bochs instruction handler

Taking into account 20 cycles Core 2 Duo processor branch misprediction penalty [24] we might see that a cost of every branch misprediction during instruction emulation became huge. A typical instruction handler is short and simple enough such that even a single extra misprediction during every instruction execution could slow the emulation down by half.

3.3.1 Splitting common opcode handlers into many to reduce branch misprediction

All Bochs decoding tables were expanded to distinguish between register and memory instruction formats. At decode time, it is possible to determine whether an instruction is going to access memory during the execution stage. All common instruction execution methods were split into methods for register-register and register-memory cases separately, eliminating a conditional check and associated potential branch misprediction during instruction execution. The change alone brought a ~15% speedup.

3.3.2. Resolve memory references with no branch mispredictions

The x86 architecture has one of the most complicated instruction formats of any processor. Not only can almost every instruction perform an operation between register and memory but the address of the memory access might be computed in several ways.

In the most general case the effective address computation in the x86 architecture can be expressed by the formula:

- *Effective Address := (Base + Index * Scale + Displacement) mod* $2^{AddrSize}$

The arguments of effective address computation (*Base*, *Index*, *Scale* and *Displacement*) can be encoded in many different ways using ModRM and S-I-B instruction bytes. Every different encoding might introduce a different effective address computation method.

For example, when the *Index* field is not encoded in the instruction, it could be interpreted as *Index* being equal to zero in the general effective address (EA) calculation, or as simpler formula which would look like:

- *Effective Address := (Base + Displacement) mod* $2^{AddrSize}$

Straight forward interpretation of x86 instructions decoding forms already results in 6 different EA calculation methods only for 32-bit address size:

- *Effective Address := Base*
- *Effective Address := Displacement*
- *Effective Address := (Base + Displacement) mod* 2^{32}
- *Effective Address := (Base + Index * Scale) mod* 2^{32}
- *Effective Address := (Index * Scale + Displacement) mod* 2^{32}
- *Effective Address := (Base + Index * Scale + Displacement) mod* 2^{32}

The Bochs 2.3.5 code went even one step ahead and split every one of the above methods to eight methods according to which one of the eight x86 registers (EAX...EDI) used as a Base in the instruction. The heart of the CPU emulation loop dispatched to one of thirty EA calculation methods for every emulated x86 instruction accessing memory. This single point of indirection to so many possible targets results in almost a 100% chance for branch misprediction.

It is possible to improve branch prediction of indirect branches in two ways – reducing the number of possible indirect branch targets, and, replicating the indirect branch point around the code. Replicating indirect branches will allocate a separate branch target buffer (BTB) entry for each replica of the branch. We choose to implement both techniques.

As a first step the Bochs instruction decoder was modified to generate references to the most general EA calculation methods. In 32-bit mode only two EA calculation formulas are left:

- *Effective Address := (Base + Displacement) mod* 2^{32}
- *Effective Address := (Base + Index * Scale + Displacement) mod* 2^{32}

where *Base* or *Index* fields might be initialized to be a special NULL register which always contains a value of zero during all the emulation time.

The second step moved the EA calculation method call in the main CPU loop and replicated it inside the execution methods. With this approach every instruction now has its own EA calculation point and is seen as separate indirect call entity for host branch prediction hardware. When emulating a guest basic block loop, every instruction in the basic block might have its own EA form and could still be perfectly predicted.

Implementation of these two steps brought ~40% emulation speed total due elimination of branch misprediction penalties on memory accessing instructions.

3.4. Switching from the PUSHF/POP to improved lazy flags approach

One of the few places where Bochs used inline assembly code was to accelerate the simulation of x86 EFLAGS condition bits. This was a non-portable optimization, and as it turned out, no faster than the portable alternative.

Bochs 2.3.7 uses an improved “lazy flags” scheme whereby the guest EFLAGS bits are evaluated only as needed. To facilitate this, handlers of arithmetic instructions execute macros which store away the sign-extended result of the operation, and as needed, one or both of the operands going into the arithmetic operation.

Our measurements had shown that the greatest number of lazy flags evaluations is for the Zero Flag (ZF), mostly for Jump Equal and Jump Not Equal conditional branches. The lazy flags mechanism is faster because ZF can be derived entirely from looking at the cached arithmetic result. If the saved result is zero, ZF is set, and vice versa. Checking a value for zero is much faster than calling a piece of assembly code to execute a PUSHF instruction on the host on every emulated arithmetic instruction in order to update the emulated EFLAGS register.

Similarly by checking only the top bit of the saved result, the Sign Flag (SF) can be evaluated much more quickly than the PUSHF way. The Parity Flag (PF) is similarly arrived by looking at the lowest 8 bits of the cached result and using a 256-byte lookup table to read the parity for those 8 bits.

The Carry Flag (CF) is derived by checking the absolute magnitude of the first operand and the cached result. For example, if an unsigned addition operation caused the result to be smaller than the first operand, an arithmetic unsigned overflow (i.e. a Carry) occurred.

The more problematic flags to evaluate are Overflow Flag (OF) and Adjust Flag (AF). Observe that for any two integers A and B that (A + B) equals (A XOR B) when no bit positions receive a carry in. The XOR (Exclusive-Or) operation has the property that bits are set to 1 in the result only if the corresponding bits in the input values are different. Therefore when no carries are generated, (A + B) XOR (A XOR B) equals zero. If any bit position b is not zero, that indicates a carry-in from the next lower bit position b-1, thus causing bit b to toggle.

The Adjust Flag indicates a carry-out from the 4th least significant bit of the result (bit mask 0x08). A carry out from the 4th bit is really the carry-in input to the 5th bit (bit mask 0x10). Therefore to derive the Adjust Flag, perform an Exclusive-OR of the resulting sum with the two input operands, and check bit mask 0x10, as follows:

```
    AF = ((op1 ^ op2) ^ result) & 0x10;
```

Overflow uses this trick to check for changes in the high bit of the result, which indicates the sign. A signed overflow occurs when both input operands are of the same sign and yet the result is of the opposite sign. In other words, given input A and B with result D, if (A XOR B) is positive, then both (A XOR D) and (B XOR D) need to be positive, otherwise an overflow has occurred. Written in C:

```
    OF = ((op1 ^ op2) & (op1 ^ result)) < 0;
```

Further details of this XOR math are described online[25].

3.5. Benchmarking Bochs

The very stunning demonstration of how the design techniques we just described were effective shows up in the time it takes Bochs to boot a Windows XP guest on various host computers and how that time has dropped significantly from Bochs 2.3.5 to Bochs 2.3.6 to Bochs 2.3.7. The table below shows the elapsed time in seconds from the moment when Bochs starts the Windows XP boot process to the moment when Windows XP has rendered its desktop icons, Start menu, and task bar. Each Bochs version is compiled as a 32-bit Windows application and configured to simulate a Pentium 4 guest CPU.

	1000 MHz Pentium III	2533 MHz Pentium 4	2666 MHz Core 2 Duo
Bochs 2.3.5	882	595	180
Bochs 2.3.6	609	533	157
Bochs 2.3.7	457	236	81

Table 3.1: Windows XP boot time on different hosts

Booting Windows XP is not a pure test of guest CPU throughput due to tens of megabytes of disk I/O and the simulation of probing for and initialization of hardware devices. Using a Visual C++ compiled CPU-bound test program [26] one can get an idea of the peak throughput of the virtual machine's CPU loop.

```
#include "windows.h"
#include "stdio.h"

static int foo(int i)
{
    return(i+1);
}

int main(void)
{
    long tc = GetTickCount();
    int i;
    int t = 0;

    for(i = 0; i < 100000000; i++)
        t += foo(i);

    tc = GetTickCount() - tc;
    printf("tc=%ld, t=%d\n", tc, t, t);

    return t;
}
```

Listing 3.2: Win32 instruction mix test program

The test is compiled as two test executables, T1FAST and T1SLOW, which are the optimized and non-optimized compiles of this simple test code that incorporates arithmetic operations, function calls, and a loop. The difference between the two builds is that the optimized version (T1FAST) makes more use of x86 guest registers, while the unoptimized version (T1SLOW) performs more guest memory accesses.

On a modern Intel Core 2 Duo based system, this test code achieves similar performance on Bochs as it does on the dynamic recompilation based QEMU virtual machine:

Execution Mode	T1FAST.EXE time	T1SLOW.EXE time
Native	0.26	0.26
QEMU 0.9.0	10.5	12
Bochs 2.3.5	25	31
Bochs 2.3.7	8	10

Table 3.2: Execution time in seconds of Win32 test program

Instruction count instrumentation shows that T1FAST averages about 102 million guest instructions per second (MIPS). T1SLOW averages about 87 MIPS due to a greater mix of guest instructions that perform a guest-to-host memory translation using the software TLB mechanism similar to the one used in Gemulator.

This simple benchmark indicates that the average guest instruction requires approximately 26 to 30 host clock cycles. We tested some even finer grained micro-benchmarks written in assembly code, specifically breaking up the test code into:

- Simple register-register operations such as MOV and MOVSX which do not consume or update condition flags,
- Register-register arithmetic operations such as ADD, INC, SBB, and shifts which do consume and update condition flags,
- Simple floating point operations such as FMUL,
- Memory load, store, and read-modify-write operations,
- Indirect function calls using the guest instruction CALL EAX,
- The non-faulting Windows system call VirtualProtect(),
- Inducing page faults to measure round trip time of a __try/__except structured exception handler

The micro-benchmarks were performed on Bochs 2.3.5, the current Bochs 2.3.7, and on QEMU 0.9.0 on a 2.66 GHz Core 2 Duo test system running Windows Vista SP1 as host and Windows XP SP2 as guest operating system.

	Bochs 2.3.5	Bochs 2.3.7	QEMU 0.9.0
Register move (MOV, MOVSX)	43	15	6
Register arithmetic (ADD, SBB)	64	25	6
Floating point multiply	1054	351	27
Memory store of constant	99	59	5
Pairs of memory load and store operations	193	98	14
Non-atomic read-modify-write	112	75	10
Indirect call through guest EAX register	190	109	197
VirtualProtect system call	126952	63476	22593
Page fault and handler	888666	380857	156823
Best case peak guest execution rate in MIPS	62	177	444

Table 3.3: Approximate host cycle costs of guest operations

This data is representative of over 100 micro-benchmarks, and revealed that timings for similar guest instructions tended to cluster around the same number of clock cycles. For example, the timings for register-to-register move operations, whether byte moves, full register moves, or sign extended moves, were virtually identical on all four test systems. Changing the move to an arithmetic operation and thus introducing the overhead of updating guest flags similarly affects the clock cycle costs, and is mostly independent of the actual arithmetic operation (AND, ADD, XOR, SUB, etc) being performed. This is due to the relatively fixed and predictable cost of the Bochs lazy flags implementation.

Read-modify-write operations are implemented more efficiently than separate load and store operations due to the fact that a read-modify-write access requires one single guest-to-host address translation instead of two. Other micro-benchmarks not listed here show that unlike past Intel architectures, the Core 2 architecture also natively performs a read-modify-write more efficiently than a separate load and store sequence, thus allowing QEMU to benefit from this in its dynamically recompiled code. However, dynamic translation of code and the associated code cache management do show up as a higher cost for indirect function calls.

4.0 Proposed x86 ISA Extensions – Lightweight Alternatives to Hardware Virtualization

The fine-grained software TLB translation code listed in section 2.3 is nothing more than a hash table lookup which performs a “fuzzy compare” for the purposes of matching a range of addresses, and returns a value which is used to translate the matched address. This is exactly what TLB hardware in CPUs does today.

It would be of benefit to binary translation engines if the TLB functionality was programmatically exposed for general purpose use, using a pair of instructions to add a value to the hash table, and an instruction to look up a value in the hash table. This entire code sequence:

```
mov   edx,ebp
shr   edx,bitsSpan
and   edx,dwIndexMask
mov   ecx,ebp
add   ecx,cb-1
xor   ecx,dword ptr [memtlbRW+edx*8]
mov   eax,dword ptr [memtlbRW+edx*8+4]
test  ecx,dwBaseMask
jne   emulate
```

could be reduced to two instructions, based on the new “Hash LookUp” instruction HASHLU which takes a destination register (EAX), an r/m32/64 addressing mode which resolves to an address range to look up, and a “flags” immediate which determines the matching criteria.

```
hashlu eax,dword ptr [ebp],flags
jne emulate
```

Flags could be an imm32 value similar to the mask used in the TEST instruction of the original sequence, or an imm8 value in a more compact representation (4 bits to specify alignment requirements in lowest bits, and 4 bits to specify block size in bits). The data access size is also keyed as part of the lookup, as it represents the span of the address being looked up.

This instruction would potentially reduce the execution time of the TLB lookup and translation from about 8 clock cycles to potentially one cycle in the branch predicted case.

To add a range to the hash table, use the new “Hash Add” instruction HASHADD, which takes an effective address to use as the fuzzy hash key, the second parameter specifies the value to hash, and flags again is either an imm32 or imm8 value which specifies size of the range being hashed:

```
hashadd dword ptr [ebp],eax,flags
jne error
```

The instruction sets Zero flag on success, or clears it when there is conflict with another range already hashed or due to a capacity limitation such that the value could not be added.

The hardware would internally implement a TLB structure of implementation specific size and set associativity, and the hash table may or may not be local to the core or shared between cores. Internally the entries would be keyed with additional bits such as core ID or CR3 value or such and could possibly coalesce contiguous ranges into a single entry.

This programmable TLB would have nothing to do functionally with the MMU’s TLB. This one exists purely for user mode application use to accelerate table lookups and range checks in software. As with any hardware cache, it is subject to be flushed arbitrarily and return false misses, but never false positives.

4.1 Instructions to access EFLAGS efficiently

LAHF has the serious restriction of operating on a partial high register (AH) which is not optimal on some architectures (writing to it can cause a partial register stall as on Pentium III, and accessing it may be slower than AL as is the case on Pentium 4 and Athlon).

LAHF also only returns 5 of the 6 arithmetic flags, and does not return Overflow flag, or the Direction flag.

PUSHF is too heavyweight, necessitating both a stack memory write and stack memory read.

A new instruction is needed, SXF reg32/reg64/r/m32/64 (Store Extended Flags), which loads a full register with a zero extended representation of the 6 arithmetic flags plus the Direction flag. The bits are packed down to lowest 7 bits for easy masking with imm8 constants. For future expansion the data value is 32 bits or 64-bits, not just 8 bits.

SXF can find use in virtual machines which use binary translation and must save the guest state before calling glue code, and in functions which must preserve existing EFLAGS state. A complementary instruction LXF (Load Extended Flags) would restore the state.

A SXF/LXF sequence should have much lower latency than PUSHF/POPF, since it would not cause partial register stalls nor cause the serializing behavior of a full EFLAGS update as happens with POPF.

5.0 Conclusions and Further Research

Using two completely different virtual machines we have demonstrated techniques that allow a mainstream Core 2 hosted virtual machine to reach purely interpreted execution rates of over 100 MIPS, peaking at about 180 MIPS today.

Our results show that the key to interpreter performance is to focus on basic micro-architectural issues such as reducing branch mispredictions, using hashing to reduce trace cache collisions, and minimizing memory footprint. Counter-intuitive to conventional wisdom, it shows that it is irrelevant whether the virtual machine CPU interpreter is implemented in assembly language or C++, whether the guest and host memory endianness matches or not, or even whether one is running 1990's Macintosh code or more current Windows code. This is indicated by the fact that both Bochs and Gemulator exhibit nearly identical average and peak execution rates despite the very different guest environments which they are simulating.

This suggests that C or C++ can implement a portable virtual machine framework achieving performance up to hundreds of MIPS, independent of guest and host CPU architectures. Compared to an x86-to-x86 dynamic recompilation engine, the cost of portability today stands at less than three-fold performance slowdown. In some guest code sequences, the portable interpreted implementation is already faster. This further suggests that specialized x86 tracing frameworks such as Pin or Nirvana which need to minimize their impact on the guest environment they are tracing could be implemented using such an interpreted virtual machine framework.

To continue our research into the reduction of unpredictable branching we intend to explore macro-op fusion of guest code to reduce the total number of dispatches, as well as continuing to split out even more special cases of common opcode handlers. Either of these techniques would result in further elimination of explicit calls of EA calculation methods.

To confirm portability and performance on non-x86 host systems, we plan to benchmark Bochs on a PowerPC-based Macintosh G5 as well on Fedora Linux running on Sony Playstation 3.

We plan to benchmark flash drive based devices such as the ASUS EEE sub-notebook and Windows Mobile phones. An interesting area to explore on such memory constrained devices is to measure whether using fine-grained memory translation and per-block allocation of guest memory on the host can permit a virtual machine to require far less memory than the usual approach of allocating the entire guest RAM block up front whether it ever gets accessed or not.

This fine-grained approach could effectively yield a "negative footprint" virtual machine, allowing the virtualization of a guest operating system which otherwise could not even be natively booted on a memory constrained device. This in theory could allow for running Windows XP on a cell phone, or running Windows Vista on the 256-megabyte Sony Playstation 3 and on older PC systems.

Finally, using our proposed ISA extensions we believe that the performance gap between interpretation and direct execution can be minimized by eliminating much of the repeated computation involved in guest-to-host address translation and computation of guest conditional flags state. Such ISA extensions would be simpler to implement and verify than existing heavyweight hardware virtualization, making them more suitable for use on low-power devices where lower gate count is preferable.

5.1 Acknowledgment

We thank our shepherd, Mauricio Breternitz Jr., and our reviewers Avi Mendelson, Martin Taillefer, Jens Troeger, and Ignac Kolenko for their feedback and insight.

References

[1] **VMware and CPU Virtualization Technology**, VMware, http://download3.vmware.com/vmworld/2005/pac346.pdf

[2] **A Comparison of Software and Hardware Techniques for x86 Virtualization,** Keith Adams, Ole Agesen, ASPLOS 2006, http://www.vmware.com/pdf/asplos235_adams.pdf

[3] **VMware Fusion**, VMware, http://www.vmware.com/products/fusion/

[4] **Microsoft Hyper-V**, Microsoft, http://www.microsoft.com/windowsserver2008/en/us/hyperv-faq.aspx

[5] **Xen**, http://xen.xensource.com/

[6] **Trap-And-Emulate explained**, http://www.cs.usfca.edu/~cruse/cs686s07/lesson19.ppt

[7] **Pin**, http://rogue.colorado.edu/Pin/

[8] **PinOS: A Programmable Framework For Whole-System Dynamic Instrumentation**, http://portal.acm.org/citation.cfm?id=1254830

[9] **Framework for Instruction-level Tracing and Analysis of Program Executions,** http://www.usenix.org/events/vee06/full_papers/p154-bhansali.pdf

[10] **PTLSim cycle accurate x86 microprocessor simulator**, http://ptlsim.org/

[11] **DynamoRIO**, http://cag.lcs.mit.edu/dynamorio/

[12] **DR Emulator**, Apple Corp., http://developer.apple.com/qa/hw/hw28.html

[13] **Rosetta**, Apple Corp., http://www.apple.com/rosetta/

[14] **Accelerating two-dimensional page walks for virtualized systems**. Ravi Bhargava, Ben Serebrin, Francesco Spadini, Srilatha Manne: ASPLOS 2008

[15] **Gemulator**, Emulators, http://emulators.com/gemul8r.htm

[16] **SoftMac XP 8.20 Benchmarks (multi-core)**, http://emulators.com/benchmrk.htm#MultiCore

[17] **Vigilante: End-to-End Containment of Internet Worms,** http://research.microsoft.com/~manuelc/MS/VigilanteSOSP.pdf

[18] **Singularity: Rethinking the Software Stack**, http://research.microsoft.com/os/singularity/publications/OSR2007_RethinkingSoftwareStack.pdf

[19] **Transmeta Code Morphing Software**, http://www.ptlsim.org/papers/transmeta-cgo2003.pdf

[20] **Inside ST Xformer II**, http://www.atarimagazines.com/st-log/issue26/18_1_INSIDE_ST_XFORMER_II.php

[21] **Bochs**, http://bochs.sourceforge.net

[22] **Intel IA32 Optimization Manual**: (http://www.intel.com/design/processor/manuals/248966.pdf)

[23] **Overview of the P4's trace cache**, http://arstechnica.com/articles/paedia/cpu/p4andg4e.ars/5

[24] **Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters** http://www.complang.tuwien.ac.at/papers/ertl&gregg03.ps.gz

[25] **NO EXECUTE! Part 11**, Darek Mihocka, http://www.emulators.com/docs/nx11_flags.htm

[26] **Instruction Mix Test Program**, http://emulators.com/docs/nx11_t1.zip