HDTrans: An Open Source, Low-Level Dynamic Instrumentation System

Swaroop Sridhar Jonathan S. Shapiro
Eric Northup
Systems Research Laboratory
Department of Computer Science
Johns Hopkins University
swaroop@cs.jhu.edu / shap@cs.jhu.edu / eric@digitaleric.net

Prashanth P. Bungale
Division of Engineering and Applied Sciences
Harvard University
prash@eecs.harvard.edu

Abstract

Dynamic translation is a general purpose tool used for instrumenting programs at run time. Performance of translated execution relies on balancing the cost of translation against the benefits of any optimizations achieved, and many current translators perform substantial rewriting during translation in an attempt to reduce execution time. Our results show that these optimizations offer no significant benefit even when the translated program has a small, hot working set. When used in a broader range of applications, such as ubiquitous policy enforcement or penetration detection, translator performance cannot rely on the presence of a hot working set to amortize the cost of translation. A simpler, more maintainable, adaptable, and smaller translator appears preferable to more complicated designs in most cases.

HDTrans is a light-weight dynamic instrumentation system for the IA-32 architecture that uses some simple and effective translation techniques in combination with established trace linearization and code caching optimizations. We present an evaluation of translation overhead under both benchmark and less idealized conditions, showing that conventional benchmarks do not provide a good prediction of translation overhead when used pervasively.

A further contribution of this paper is an analysis of the effectiveness of post-link static pre-translation techniques for overhead reduction. Our results indicate that static pre-translation is effective only when expensive instrumentation or optimization is performed.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Languages, Performance, Security

Keywords Dynamic instrumentation, Dynamic translation, Binary translation

1. Introduction

One of the notable developments over the last few years has been the use of dynamic binary translation to address numerous run time instrumentation, compatibility, and security challenges. Dynamo [3] and Mojo [12] perform run time optimization to improve the performance of native binaries. Valgrind [36] uses sophisticated dynamic translation methods to perform heavy-weight dynamic binary analysis which can be used for comprehensive performance measurements, profiling, memory analysis, and debugging. Shade [15] uses dynamic translation for high-performance instruction set simulation. Daisy [20, 2] uses dynamic compilation for instruction set emulation and evaluation. VMWare [17] uses selective dynamic translation to achieve full machine virtualization. UQBT [13], Walkabout [14] and Strata [41] provide a retargetable dynamic translation infrastructure. DynamoRIO [8] and Pin [34] are dynamic instrumentation systems that export a high level API for run time instrumentation of and optimization of programs. Program Shepherding [32] uses dynamic translation to monitor control flow transfers in order to enforce security policies on program execution.

Execution performance under dynamic translation is achieved by balancing the cost of translation against the performance gains from translations. Many current translators implement translation-time trace optimizations. The expectation is that by improving the performance of re-used code, the overhead of instrumentation is reduced and in some cases application performance may be improved. This expectation is violated in programs that have low percentages of dynamic code re-use, high frequencies of indirect control transfer, or short execution runs. If ubiquitous dynamic translation is intended (as proposed, for example, in program shepherding [32]), such "unfriendly" programs need to be efficiently instrumentable, and the cost of run-time optimization becomes difficult to amortize.

Instrumentation applications can be broadly divided into three categories: (1) those that are too complicated to benefit from the techniques that a run-time optimizer can apply, (2) those that benefit from repeated re-use of register(s) or non-trivial code scheduling, and therefore may benefit from code re-synthesis, and (3) those that can be accomplished efficiently without run-time optimization, using only register liveness analysis. Most of the motivating examples for run-time binary instrumentation that appear in the literature fall into the last category. The second category is significantly complicated if precise signal and exception handling is required, because the translator must be prepared to restore registers to their "official" state at any sequence point.

In this paper, we describe and evaluate HDTrans — a simple, high-performance, light-weight dynamic instrumentation infrastructure for the IA-32 architecture, that is optimized for simplicity and modifiability. Our original motivating application for HDTrans was supervisor-mode instrumentation. This prompted us

VEE'06 June 14–16, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-332-6/06/0006. . . $5.00.

to prioritize our implementation for simplicity rather than optimization. Code rewriting is complex and fragile, and trace optimization makes it *more* complicated. It is challenging to debug a dynamic translator at user level where relatively rich debugging tools are available. When running within a kernel, where the usual sign of failure is a system-wide reset, "simple" and "good" are closely correlated. For this reason, we chose to focus our attention on the simplicity and efficiency of the decoding and translation system rather than the efficiency of translated code. Surprisingly, given the sequence of prior results favoring translation-time optimization, the performance of HDTrans is competitive with the best current translation systems.

HDTrans is entirely open source and easily modifiable. It provides a basic framework for instrumentation at a per-trace, per-basic-block or per-instruction granularity. The instruction translation policy is table-driven, and can be revised on an instruction by instruction basis. The goal of the translator design is to facilitate modification of the translation strategy itself to provide inline instrumentation. The SYRAH group at Harvard University has used HDTrans to dynamically instrument programs for two applications: run-time security policy enforcement to ensure system integrity while running untrusted code, and providing fine-grain reverse execution for debugging.

The rest of the paper is organized as follows. In section 2, we provide a brief review of the state of the art in dynamic translation. In section 3, we introduce the basic translation methodology and the design choices adopted in HDTrans. We later present the special techniques we used to address some performance critical issues in dynamic translation. In section 4, we present comparative performance results with existing systems using both standard benchmarks and everyday programs. We provide a detailed evaluation of the overheads involved, and comment on the utility of static-pretranslation techniques for overhead reduction in dynamic translators. We present related work in section 5, and conclude in section 6.

2. Review of the Art

The essential techniques of binary translation date back to Deutsch's early work on Smalltalk-80 [16] and May's work on System/370 emulation [35]. Deutsch in turn was inspired by earlier work on universal host machines [38] and threaded code [4, 18]. May's work on MIMIC, in particular, appears to deserve credit for the idea of trace-based translation, and many of the associated optimizations, including run-time register allocation.

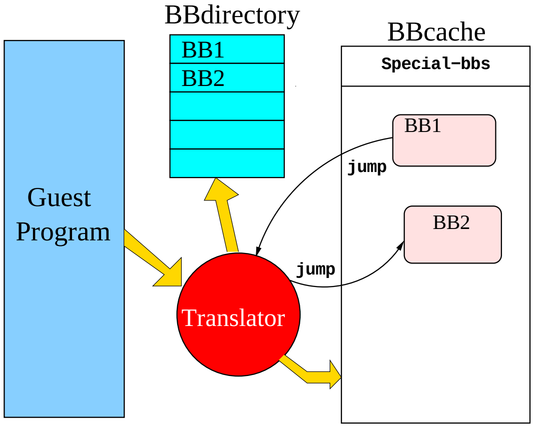

Figure 1. The dynamic translator

A binary translator proceeds by alternating its execution between "translation mode" and "target mode," as shown in Figure 1. In translation mode, the instructions of the subject (or "guest") program are translated into a "basic block cache" (in HDTrans: the BBcache). Depending on the specific translation strategy, the unit of translation may be a basic block, an extended basic block, or a trace [35] that spans multiple basic blocks. As translation is performed, a directory of translated units (in HDTrans: the BBdirectory) is maintained, indexed by the guest program counter. When translation is completed, the translator enters target mode by branching to the newly formed translation unit.

2.1 From Code Blocks to Traces

The earliest binary translators translated instructions one at a time. May [35] introduced translation in units of "code blocks." Flow analysis is used to discover all code sequences that are reachable from some initial entry point, which is compiled as a unit using an aggressive translation algorithm that back-tracks as branches into previously translated blocks are discovered. Eventually this method stabilizes and a translated code sequence is emitted along with a record of all source to target branch relationships. In May's design, an "intermediate memory" is used to record the translated location of source code blocks, allowing even indirect branches to proceed without re-entering the translator as long as no new translations are required. The key advantage to this approach is that all statically computable branch destinations are discovered at translation time. Two key disadvantages are the need for a comparatively large intermediate memory and a code block discovery strategy that yields variable translation delays and is likely to translate unused code unnecessarily.

Later translators refined this strategy by translating more selectively. Shade [15] translates instructions in units of (simple) basic blocks. It also *chains* the basic blocks ending in a direct branch to a known target in order to bypass translator intervention in control transfer. Dynamo [3] dynamically constructs optimized traces for instruction sequences that are identified to be in the "hot path". Pin [34] translates in units of static traces, proceeding forward across conditional branches and stopping at the first unconditional branch. Both Dynamo and Pin redirect direct branches whose destinations are unknown to "exit stubs". When an exit stub is encountered run time, it causes the translator to enter translation mode to translate the target of the branch, and then patch the address of the translated branch destination into the original code sequence. The end result of these patches is that traces and fragments in the translation cache come to be linked more and more directly over time, with fewer and fewer switches to translation mode required as execution progresses.

2.2 Indirect Branches

Indirect branches are more difficult to deal with, because they cannot be patched into the original location, and the "intermediate memory" technique requires prohibitive amounts of memory on modern machines. Instead, other implementations rely on some combination of hash tables and chained basic blocks. The Smalltalk-80 [16] optimizes its method dispatch by inlining the address of the most recently used method into the call site along with a comparison to check for correctness of dispatch. Embra [49] extends this idea to perform *speculative chaining* of basic blocks across indirect branches in general. Dynamo [3] uses lightweight runtime profiling information to construct traces across indirect branches. These techniques impose the cost of saving and restoring the hardware condition codes and / or several registers in order to perform comparisons and hash table lookup. Pin attempts to reduce overhead by jumping to a candidate destination block that begins with a compare to determine whether it is the correct destination, and on failure branches to the next candidate [34]. When all candidates are exhausted, Pin falls back to a hash table that is specific to the source instruction. If the destination cannot ultimately be found all methods proceed by translating the destination block.

Because RETURN instructions are a performance-critical special case, several strategies have been adopted to optimize them. Dynamo simply treats returns like other indirect branches. FX!32 [30] uses a "shadow stack" that holds the translated return addresses. Pin uses a form of polyinstantiation known as "function cloning," creating separate copies of a function for each call site so that the return destination is known at translation time [34].

2.3 Instrumentation and Optimization

Recent dynamic translation systems have focused on instrumentation, run-time optimization, profiling, debugging, or sandboxing [8, 32, 34, 36]. In order to reduce the overhead of instrumentation or improve run-time performance, some of these translators translate to a (low-level) intermediate form, insert instrumentation at the intermediate form level, and then re-generate code into the basic block cache. In some applications, translate-time optimization is known to be extremely important to reduce demultiplexing overheads [23]. Its use for general-purpose code optimization has (to date) yielded mixed results [3, 12].

For low-level instrumentation, liveness analysis of the condition codes [31] is often sufficient to introduce instrumentation with minimal performance overhead. General register liveness is better, and both condition code and register liveness over a trace can be obtained with minimal extra work during instruction decode. The advantage of code regeneration is that more complex instrumentation strategies can be optimized to reduce their overhead. Pragmatically, this advantage ends at the procedure call boundary: once an instrumentation strategy is obliged to insert procedure calls with any great frequency, the current trace construction strategies are usually not able to usefully optimize the instrumentation.

2.4 Other Issues

The original approach of flushing the translation cache in its entirety is commonly credited to Deutsch [16], though it seems likely that the technique was used by earlier emulators within IBM. Several recent systems have explored strategies for flushing the translation cache more selectively [3, 25, 26, 27, 28, 29]. In order to support multithreaded programs, systems like Mojo [12] use thread local code caches, while others [9, 34] have and examined translation caches that is shared across threads in multithreaded applications. Bruening *et al* have reported between 50% and 70% reuse of cache content [9] across threads for server applications, but much lower sharing (1% to 10%) for desktop applications [7]. In practice, the performance benefit of retaining such shared code is highly dependent on the throughput of the translator and the complexity of the instrumentation.

3. HDTrans

HDTrans was initially designed as a supervisor-mode translator for use in virtual machine emulation [10, 11]. Our original goal was to build a faster, open-source version of VMWare [17], and to explore the possibility that paravirtualization [19] might avoid the need for recompilation through a hybrid combination of static and dynamic translation techniques. That work remains incomplete, but our initial performance measurements on *user* mode code led us to conclude that HDTrans had value for user-mode instrumentation, and could provide a useful base for more advanced instrumentation systems. For a variety of reasons, we also we wanted to ensure that the state of the art was captured in open form. While several similar instrumentation systems exist, none of the machine-level translation or instrumentation systems are openly inspectable. This impedes research advance by making them hard to study, and introduces the need for redundant implementation of complex and delicate systems.

3.1 A Heretical Proposition

Because HDTrans was intended for kernel use where debugging would be difficult, we eliminated code re-generation from our design options immediately. Examination of the binary code size of existing translators revealed that they were substantially larger (and presumably more complex) than the microkernel systems that we most wished to emulate and instrument. The version of DynamoRIO reported here is approximately 382KB of code and 70KB data. The version of Pin reported here is over 3 megabytes of code and 45KB of data. For calibration, the EROS kernel [44] is approximately 65KB of code, and its successor, Coyotos [43], is expected to be significantly smaller. Our challenge was to achieve performance comparable to existing instrumentation systems without comparable complexity.

As we considered the very variable optimization results achieved by DynamoRIO, a heretical idea emerged: maybe the achieved performance on modern translators was primarily due to trace linearization, and run-time optimization was only achieving enough benefit to amortize the cost of the optimization. If so, and if we could come up with a way to implement the translation phase more efficiently, it might turn out that we didn't need to re-generate code to achieve comparable results. As far as trace linearization is concerned, we wanted to examine *static* trace linearization alternatives to the *dynamic* profile driven approach taken by Dynamo. Where instrumentation is concerned, matters clearly aren't quite this simple, but for kernel instrumentation purposes we were prepared to accept that fancy instrumentation might demand a dynamically supported *static* rewriting strategy, and for virtual machine translation simplicity was an overriding objective.

The version of HDTrans benchmarked here is 97KB of code, but 30% of this is due to aggressive inlining. When inlining is disabled and the disassembler (part of the debugging support) is discounted, the code size drops to 56KB. Essential function is embodied in a 27KB decode table that is stored as data. Much of the function of the translator can be validated by using it as a disassembler and comparing the output to the output of `objdump` utility. The balance of this section describes how we achieved an instrumentation system that is competitive with DynamoRIO and Pin in 1/6th and 1/50th of the code, respectively.

3.2 Basic Translator

The basic structure of HDTrans is similar to that of Dynamo [3], Pin [34], or Mojo [12]. The translation phase builds traces and accumulates a directory of mappings from source basic blocks to target basic blocks. As each translation phase finishes, execution of the guest program resumes with the newly translated basic block. HDTrans translates direct branches eagerly when the destination is known and uses exit stubs to patch them when the destination is not known. To lower the overhead of indirect branches and returns, HDTrans employs two new optimization techniques: the *return cache* and the *sieve*. Beyond these, HDTrans achieves its performance through four basic techniques:

- Through a carefully structured, table-driven decoder, HDTrans reduces the total number of cache line fetches required to translate each instruction.
- HDTrans optimizes for reuse of existing translation, and adds "extra" entries to the BBdirectory for instructions that are likely to be destinations of currently unseen branch instructions.

```
/* opCode, instr,    op1,     op2,     op3,  emitterFunc,    attribs */
...
{ 0xe8u, "callL",   Jv,      NONE,    NONE, EMIT(call_disp), XO, N },
{ 0xe9u, "jmpL",    Jv,      NONE,    NONE, EMIT(jmp),       XO, N },
{ 0xeau, "ljmp",    Ap,      NONE,    NONE, EMIT(normal),    XO, N },
{ 0xebu, "jmp",     Jb,      NONE,    NONE, EMIT(jmp),       XO, N },
{ 0xecu, "inB",     AL,      indirDX, NONE, EMIT(normal),    XX, N },
...
```

Figure 2. Translation table fragment

- HDTrans performs implicit trace construction through the Least Redundant Effort heuristic.
- HDTrans uses code sequences that carefully avoid modifying condition codes (as suggested in [8]). This optimization is IA-32 specific, but the IA-32's combination of sensitive and non-sensitive state in the EFLAGS register makes restoring this register extremely expensive. Fortunately, RISC architectures do not penalize condition code restore quite so effectively.

The sizes of the BBcache and BBdirectory are selected statically at HDTrans compile time. The default translation cache size, which is used for all measurements reported in this paper, is 4MB. If either the BBcache or the BBdirectory become full, we flush the translation cache and start over.

3.3 Table-Driven Translator

HDTrans performs basic block translation one instruction at a time. The translator is table-driven. The translation table (Figure 2) embodies rules for decoding all instructions in the current architecture. Each entry in the table occupies a single cache line, and a maximum of three table entries are visited in order to decode an instruction. The result of an instruction decode is a decode-structure that is passed to the emission or instrumentation routine corresponding to the instruction.

The table also identifies the back-end emit-routine that should be used to emit each instruction into the BBcache. The emitter routine also controls the translation process by deciding whether the current instruction terminates a trace, and what instruction pointer should be translated next following the current instruction. In order to support customized instrumentation, we only need to change the corresponding entry in the translation table to point to a function that calls user-supplied code in addition to its respective emit-routine. In the basic translator that does not perform any instrumentation, most instructions are translated by copying them verbatim into the BBcache, and only those involving a control transfer need special attention.

HDTrans works very hard to leave application registers undisturbed. Translator state, including the BBdirectory, is stored in a per-thread data structure called M-state, which is referenced through memory-absolute addressing modes. Translation of primarily indirect control transfer instructions requires that scratch registers be spilled. At present, HDTrans spills these registers to the application stack. This is safe for well-behaved UNIX applications, but is insufficient to support emulation of Windows applications which write beyond the current stack pointer. Note that this state is transient: an ill-behaved application may observe that state beyond the stack pointer has not been preserved, but HDTrans uses the stack only *between* guest instructions, and does *not* rely on these values at any other time. Late in the process of writing this paper, we realized that UNIX applications making use of `sigprocmask` may reliably use storage beyond the current stack pointer, and we are currently modifying the implementation to spill temporary registers to the M-state instead of the stack.

While the current translator does not support precise signal contexts in the case of exceptions or interrupts, all translator-emitted code sequences have been carefully designed so that they can be rolled back. This allows the implementation to restore the exact user register state at any architected sequence point. The missing feature in the current implementation is emitting the necessary "undo" information for register spills.

3.4 Unconditional Direct Branches: Trace Linearization

HDTrans performs lazy trace linearization using a Least Redundant Effort heuristic. Translation proceeds *straight* through conditional branches and `call` instructions, and terminates at any unconditional `jump` to a destination that is statically unknown or previously translated. Instructions following a branch or call are added to the BBdirectory as likely targets of future branches. When a direct `jump` to a previously untranslated basic block is encountered, we elide the `jump`, add a BBdirectory entry for the destination, and continue translating at the destination instruction. Pin terminates its traces when an unconditional branch is encountered. DynamoRIO [8] maintains a separate trace cache in addition to the basic block cache, where hot traces are maintained. Trace formation is aggressive, and is done even across indirect jumps at the cost of tail duplication. In contrast, HDTrans optimizes for maximum reuse of translation effort.

The above translation scheme is illustrated using the following example. We use AT&T syntax for the assembly fragments illustrated in this paper. All variables beginning with 'G' correspond to guest (original) values and those beginning with 'T' correspond to translated values. If the source instructions of the guest are:

```
        add $20, %ecx
        jmp $G_dest
        ...

G_dest: mov $30, %edx
        call $G_proc
G_next: add $4, %esp
        jmp $G_dest
        ...
```

If `G_proc` is already translated and `G_dest` is not, the corresponding translated instructions in the BBcache will be:

```
         add  $20, %ecx
T_dest:  mov  $30, %edx   ; new BB here
         push $G_next
         jmp  $T_proc
         <call-postamble> ; See section 3.7
T_next:  add  $4, %esp    ; new BB here
         Jmp $T_dest      ; end of trace
         ...
```

Average trace lengths in our scheme was about three basic blocks, or 10-15 instructions. The longest measured trace was 256 basic blocks with over 1,100 instructions in the case of `gcc`.

3.5 Conditional Branches

Translation at a conditional branch uses the technique used in Dynamo [3]. If the destination of the branch has already been translated, we emit a conditional jump to the existing translated basic block. Otherwise, we conditionally branch to an exit stub. On entry, the exit stub calls the translator, providing the original destination and the address of the conditional jump instruction in the basic block cache. The translator performs translation as needed at the jump target, and patches the destination into the translated jump instruction so that further jumps can go directly to the destination block. Exit stubs are also emitted for call instructions whose destination has not yet been translated.

The instruction following a conditional branch is noted as the start of a new basic block in the BBdirectory. Exit stubs do not sub-divide source traces; their emission is deferred till the end of a trace in order to preserve the sequentiality of the trace. Pin inserts exit stubs at the end of the code cache, in order to improve I-cache locality among the traces in the BBcache [26].

3.6 Indirect Branches

Since the dynamic translator cannot know the destination of the jump at translation time, it is necessary to emit code that performs a run time lookup to determine the translated destination of the branch, which is a potentially expensive operation. Computing the branch destination requires that a mapping from guest address to translated address be implemented. DynamoRIO attempts to avoid some of this overhead by inlining a small number of "guesses" at trace construction time. If these fail, it falls back to a global hash-table lookup. Pin emits a (back-patched) branch to a candidate basic block, and checks at the destination whether it is indeed the target. These guessed blocks are chained together. If the chain does not discover a translation, a source-specific hash table of destinations is consulted. In all schemes, the BBdirectory is consulted as the ultimate fallback and used to revise the optimized strategies for later use.

HDTrans proceeds by constructing a global hash table at run time. This table is hashed on the destination address. Each table entry contains a jump instruction to the start of a destination-specific chain of comparison blocks. Comparison blocks are added *only* for those basic blocks that are dynamically observed at run time to be indirect destinations. Collectively, the hash table and its comparison blocks are known as the "sieve" (Figure 3). The mechanism differs from the strategy of Pin [34] in that it hashes *first* and chains *second*. In the benchmarks reported here, and in a variety of other programs we have tested, the length of the sieve chains are observed to be 1 or 2 on an average, and are never more than 4. HDTrans currently uses separate seives for indirect jump and call instructions, but this appears to make no significant difference (Figure 7), and we expect to remove it in future implementations.

3.7 Return Caching

The return instruction is by far the most important form of indirect branch in terms of dynamic frequency. Although the return instruction can be handled by a generalized indirect branching scheme, we can exploit the symmetry between the call and return instructions to optimize this case. However, a key constraint on any implementation is that the call/return sequence should not alter the activation stack in a way that is observable by the subject program.

Some dynamic translators [40] have proposed a scheme in which the *translated* return address, rather than original code address is pushed on the stack. This approach is incompatible with C++ exception handling, garbage collection, or longjmp() without extensive and complicated fix-ups or built-in support for the exception handling control transfer mechanism. It also presents challenges when the BBdirectory is flushed. Pin uses function cloning (a.k.a polyinstantiation) to specialize functions that are called from multiple locations [34]. This allows the return instruction to return directly, at the cost of emitting redundant traces into the translation cache. FX!32 implements a "shadow stack" for translated addresses.

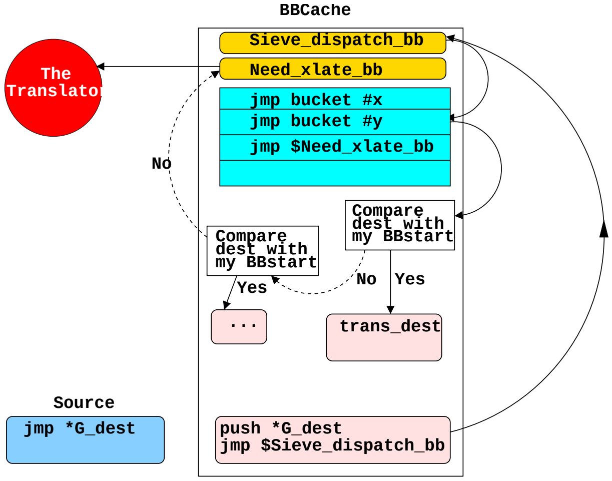

Figure 3. The sieve.

HDTrans uses a new technique known as the "return cache," which is built on a co-operative protocol implemented between call and return instruction emitters. The return cache is a single-entry hash table that is indexed by a hash of the called procedure's start address. The translation of a call instruction pushes the *untranslated* return address on the stack, and stores the *translated* return address into the appropriate return cache entry. If the call in question is a direct call, the return cache bucket calculation can be done at dynamic compile time.

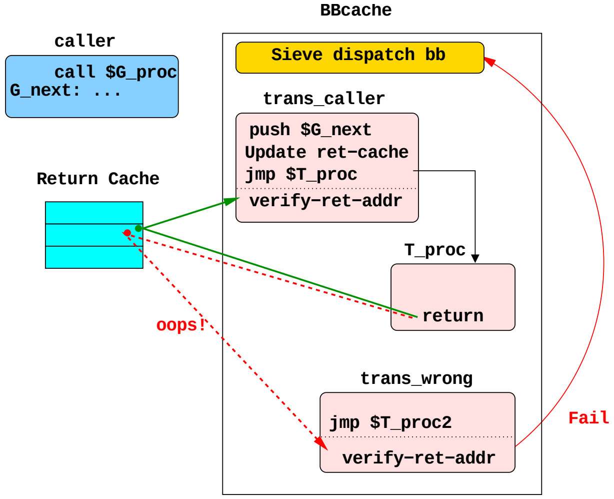

Figure 4. Return cache control-flow.

The translation of a return instruction leaves the original return address on the stack and *blindly* performs an indirect jump through the return cache entry indexed by the procedure entry point that dominates that return instruction (Figure 4).

This is an optimistic control transfer. If all goes well, control reaches the correct caller of this function (as shown by the solid arrow in the figure). However, due to return cache collisions (e.g. due to recursion), or failed return dominance tracking due to indirect control flow, this method can result in misdirected returns (as shown by the dashed arrow). We rely on the fact that every return ends up at the postamble of *some* call, and every postamble performs a check to see if the intended destination has been reached. If this postamble check fails, control is transferred to the sieve to locate the intended destination.

Between 15% and 22% of returns fall back into the sieve. The majority of these occur because of failures of return dominance tracking. Return cache entries are initialized with the address of the `sieve-dispatch-bb` at startup. This ensures that (perverse) code performing a `return` before `call` works correctly.

The return cache differs from the FX!32 shadow stack in three regards. First, the shadow stack is definitive. It contains (guest program counter, guest stack pointer, translated return address) triples that must be preserved if execution of return instructions is to proceed successfully. Second, it is precise: the shadow stack technique is more attractive for certain recursion patterns. Third, the shadow stack approach requires special handling to support `longjmp()` and exception handling. The return cache is somewhat easier to implement and introduces less register pressure.

3.8 Translation Startup

In order to take control over the the guest's execution, HDTrans uses the LD_PRELOAD environment variable to load itself before the guest program [37]. When our startup code is called by the dynamic loader, it hijacks the control flow, and never returns to the loader. Instead, it uses the return address pushed on the stack to start dynamic translation. The startup code passes this address to an initialization routine that initializes the M-state and branches into the translator, which then starts a co-routine like execution along with the guest program. The method can be straightforwardly adapted for debugger-directed injection, allowing more complete instrumentation of early startup code. Related techniques were used by the `Debug` debugger [42] and an unreleased incremental compilation environment developed at AT&T Bell Laboratories in 1989.

3.9 Multithreading Support

Previously reported versions of HDTrans did not support multithreaded execution. The version reported here has added multithreading support, and all of the performance results reported (including single-threaded benchmarks) are obtained from the multithreaded implementation. There is no measurable single-thread overhead incurred by the presence of support for multithreading.

In implementing multithreading, we considered several designs that would support sharing – or at least reuse – of translated code across the threads. The simplest approach is to copy the existing BBcache at the time of thread creation. The problem with this is that the emitted code in the BBcache contains absolute references to trampoline basic blocks and to the M-state structure. We considered moving the M-state pointer to thread-local storage, but the necessary run-time support is implemented by the `pthreads` library, and not all multithreaded programs use `pthreads`. The same problem is shared by various lazy cloning methods. While it would be possible to keep enough relocation information to be able to relocate these addresses, or to steal a register as is done in Pin [34]. Both of these require significant new effort and complexity in the translator. On register-starved architectures like the IA-32, stealing a register is not a thing to be undertaken lightly.

While translation cache reuse may be important in some applications, the degree of reuse varies widely. Bruening has reported between 50% and 70% reuse of cache content [9] across threads for server applications, but much lower sharing (1% to 10%) for desktop applications [7]. Ultimately, the benefit of reuse is a function of the cost of regeneration. In keeping with the rest of the HDTrans design, we ended up adopting a brute force approach. The M-state and BBcache structures have been made thread-local, and each newly-created thread begins with a cold translation cache.

Similar to the approach followed in DynamoRIO [8], when the *clone* system call is (successfully) invoked with the CLONE_VM flag set, the translator arranges for the child to resume execution in a dedicated initialization routine that allocates and initializes a new M-state and BBcache for the thread. The initialization routine then branches to the translator to begin translation of the new thread. No special handling is required for the *vfork* system call, because the parent is blocked until the child executes an `execve()` or an `_exit()`, and all signals to the parent are delivered after the child has exited.

3.10 Signal Handling

HDTrans currently provides support for signal handling, though it does not yet support introspective signal handlers. HDTrans treats signals as a separate thread of execution that happen to get "scheduled" on signal arrival. When the guest attempts to register a signal handler, we hijack the system call and set up our own "master" signal handler, after noting the guest signal handler information in our signal handler table. Upon signal arrival, our master signal handler — keeping with the brute force philosophy of HDTrans — allocates a new M-state[1] and BBcache, and starts the translation of the corresponding signal handler. When the signal handler eventually executes a `sigreturn` or a `rt_sigreturn`, we release the memory allocated for the signal handler's execution. This method side-steps many of the complications in translating signal handlers like BBcache flush within the signal handler, signals arriving on an alternate stack, special handling for one-shot signals, signal queuing and deferred delivery of signals that arrive while the translator itself is executing [8], etc.

Signal arrival is a very rare event when measured on the scale of the CPU clock cycle. Therefore, there is practically little performance impact due to the fact that we use a new code cache for every signal. We were able to run programs like emacs-x and openoffice with no interactively noticeable overhead as compared to the version of HDTrans that executes signal handlers natively.

Because of our interest in kernel-level translation, care has been taken in designing the HDTrans emitted code sequences to preserve the possibility of support for introspective signals and exceptions. At every point where guest registers have been spilled, it is possible to emit "undo" information that would allow them to be restored and a correct guest sequence point re-established. This would allow HDTrans to present a fully accurate `sigcontext` structure to signal handlers. HDTrans does not currently emit the necessary undo information. We also do not support signal handler sharing between parent and child processes.[2]

4. Performance Evaluation

In evaluating HDTrans, we are interested both in comparative performance and in understanding which optimizations used by HDTrans are significant. We also want to understand the overhead of instrumentation using the respective systems. Finally, we would like to understand the degree to which dynamic translation overhead is sensitive to particular processor implementations.

[1] There is a small amount of state that must be unique per thread, like the signal handler table, thread-wide profiling counters, etc. These are held in a separate structure and all M-states within a thread store a pointer to it.

[2] When the *clone* system call is invoked with the CLONE_SIGHAND flag set, the calling process and the child processes share the same table of signal handlers.

4.1 Experimental Setup

Hardware: The following machines are used to the collect the performance measurements reported in this section:

- *Machine-0:* Dual processor, hyperthreaded Intel(R) Xeon(TM) CPU 2.80GHz system with 512 KB cache and 6GB main memory.
- *Machine-1:* AMD Athlon(TM) 64 Processor 3200+, 2043.352 MHz system with 512 KB cache and 3 GB main memory.
- *Machine-2:* Dual AMD Athlon(TM) Processor 1526.7 MHz system with 256 KB cache and 3 GB main memory
- *Machine-3:* Intel(R) Pentium III (TM) CPU 931.2 MHz system with 256 KB cache and 512 MB main memory.

Except where noted in the processor comparisons, benchmarks are executed on Machine-0. In all cases, benchmarks are compiled on the machine where they are executed.

Operating System and Compiler: All benchmarks presented are executed on Linux Fedora Core 4 (2.6.15-1.1833_FC4smp kernel). Single-threaded performance is evaluated using SPEC INT2000 version 1.3 compiled with gcc version 4.0.2. Multi-threaded performance is evaluated using SPEC OMP2001 version 3.0. Due to the absence of OpenMP support in the GNU compiler chain, multithreading benchmarks are compiled using Intel's version 9.0 FORTRAN and C++ compilers.

Translators: The following dynamic translators are used to present comparative performance results:

- HDTrans version 0.3
- Pin Kit 3077, built for `gcc` 4.0
- Pin-PLDI – the version of Pin that was used to report performance numbers in the PLDI paper [34].
- DynamoRIO version 0.9.4
- Valgrind version 2.4.0, with null instrumentation (Nulgrind)

Except in the case of multithreaded benchmarks, we show the performance of Pin *without* the `-mt` option which is used to enable the execution of multi-threaded programs.

4.2 Single-Threaded Comparative Performance

Figure 5 shows the performance of HDTrans on the SPEC INT2000 version 1.3 benchmarks [48] in comparison to DynamoRIO and Pin. HDTrans compares favorably with the leading dynamic translation systems in terms of baseline execution translation speed.

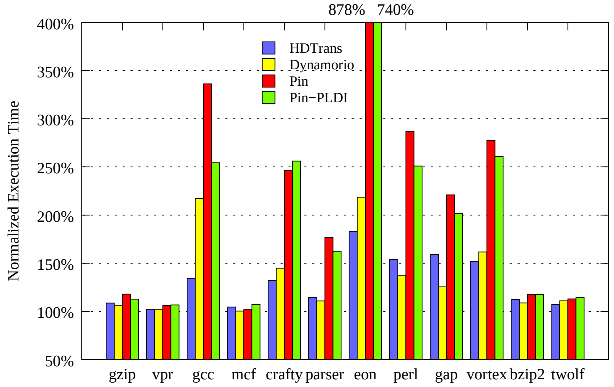

Figure 5. SPEC INT2000 benchmarks.

Benchmarks such as SPEC INT2000 are designed to measure the performance of relatively small codes with hot working sets, and therefore tend to minimize translation overheads. Pervasive instrumentation applications, such as Program Shepherding [32], run in environments where a significant proportion of programs may be dominated by translation startup costs.

In consequence, it is doubtful that these results accurately predict the performance of machine-level dynamic translators in production use.

4.3 Multi-Threaded Comparative Performance

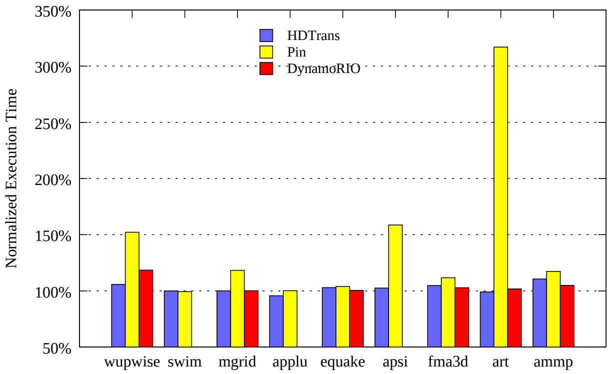

Figure 6. Multi threaded performance using SPECOMP medium benchmarks. Pin-PLDI does not support multi-threading. DynamoRIO failed to run `swim`, `applu` and `apsi`

Figure 6 shows the performance of HDTrans on SPEC OMP2001 version 3.00 benchmarks [47] in comparison to Pin (executed with the `-mt` option). We were unable to get two of the benchmarks – `318.galgel_m` and `326.gafort_m` to build and to run correctly, and they are not reported in Figure 6. This also meant that we could not run the benchmarks with the `--reportable` flag, but had to instead use `--ignore_errors` flag. Discounting this, everything else was compatible with a reportable run.

4.4 Evaluation of HDTrans Optimizations

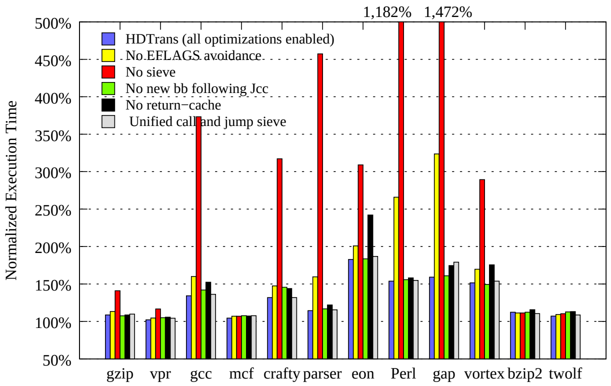

Figure 7. SPEC INT2000 benchmarks with optimizations selectively disabled.

Figure 7 shows the performance of HDTrans on SPEC INT2000 benchmark with individual optimizations disabled one at a time. These measurements demonstrate that the sieve, return cache and using code sequences that do not modify eflags are significant optimizations, and that maximizing basic block reuse is an effective choice for indirection intensive programs. The reuse result suggests that previously published arguments favoring trace construction may not be compelling in environments where simplicity and maintainability are paramount concerns, and may not be necessary for simpler instrumentation applications.

4.5 Instrumentation Overhead

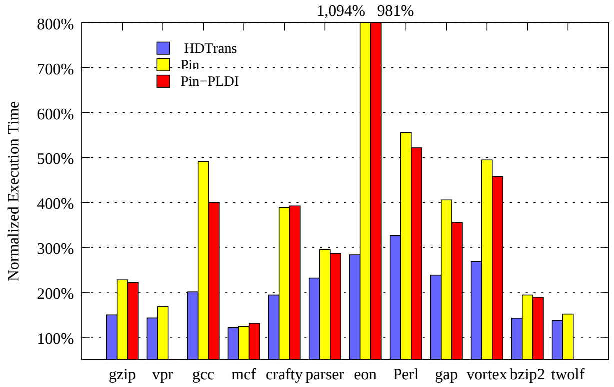

Figure 8. Performance with basic block counting. Pin-PLDI failed to run 175.vpr and 300.twolf with basic block counting

Figure 8 shows the performance of HDTrans on basic block counting. As the number of saves and restores of the `eflags` is the dominating factor in the performance of instrumented code, our basic block counting scheme performs a condition code liveness analysis using the information readily stored in our decode-table. If we encounter an instruction that modifies condition codes *anywhere* in the basic, we emit the increment before that instruction. If no such instruction is encountered until the end of the basic block, we emit the increment just before the branch (direct, conditional or indirect `jmp`s, `call`s and `ret`s) bracketed by code that saves and restores the condition codes. This emission policy is implemented explicitly by the instrumentation code. In the case of Pin, we used the instrumentation code that was used in the PLDI paper [34], which we obtained from the Pin group.

HDTrans performs favorably when compared to current leading dynamic instrumentation systems. The average overhead of instrumentation in HDTrans is 103% as opposed to 282% in the case of Pin.

4.6 CPU Sensitivity

Dynamic translation can be sensitive to particular processor implementations. In particular, differences in branch prediction, branch caching, and return address caching can interact with the trace construction strategy. Moreover, some systems may use documented or undocumented features peculiar to a processor and / or compiler implementation. For example, Pin does not support AMD processors.

Figure 9 shows the overhead of HDTrans measured on several CPU implementations using the SPEC INT2000 benchmark suite. Interestingly, there is no conclusive difference in performance between these platforms.

4.7 Cold Cache Performance

It has become common practice to evaluate dynamic translation systems using benchmarks such as SPEC INT2000 [14, 8, 34, 41, 32, 46]. Instrumentation using HDTrans shows that most of these benchmarks converge rapidly on a stable state that runs entirely out of the translation cache. In consequence, this approach evaluates translators under ideal conditions, and does *not* effectively reveal the impact of translator overhead. Especially in ubiquitous application, evaluation of "cold cache" overheads is important.

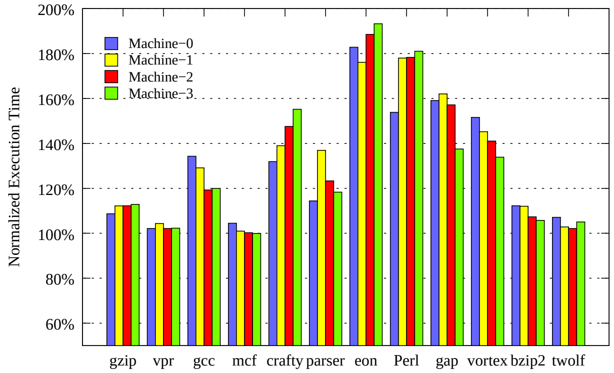

Figure 9. Overhead of HDTrans for the SPEC INT2000 benchmarks measured on different machines.

To evaluate the cold cache performance of HDTrans, we measured the performance of a number of short-running programs that are dominated by startup initialization costs or interpretation:

- `cc1` (v 4.0.2) compiling a 390 line Huffman encoder,
- `bzip2 -t` on a 4KB bzip file,
- the `clear` command,
- the `ls` command on `/bin`,
- `emacs` in batch mode directed to load a file, enter a highlighting mode, and quit, and
- `perl` (v 5.8.6) run on a 200 line script that generates random passwords

Figure 10 shows the comparative performance of HDTrans for these benchmarks. It should be noted that recent versions of `gcc` exhibit dramatically lower code reuse than the older version used in SPEC INT2000, and consequently stress dynamic translators much harder.

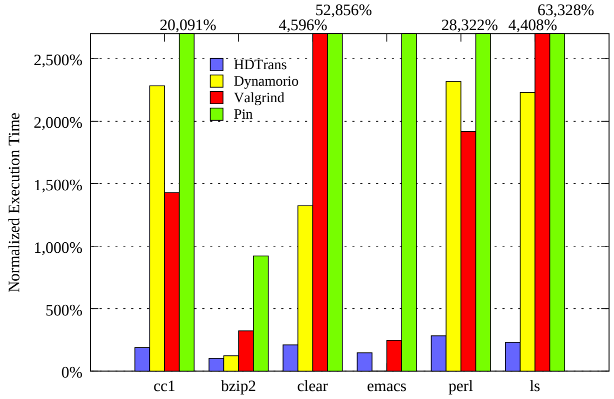

Figure 10. Overhead for some cold cache benchmarks. DynamoRIO failed to run `emacs` for this test.

4.8 Translation vs. Execution Overhead

The overhead of dynamic translation can be divided into the cost of the translation process itself, and the overhead introduced by the translated code. To isolate these effects, we modified HDTrans

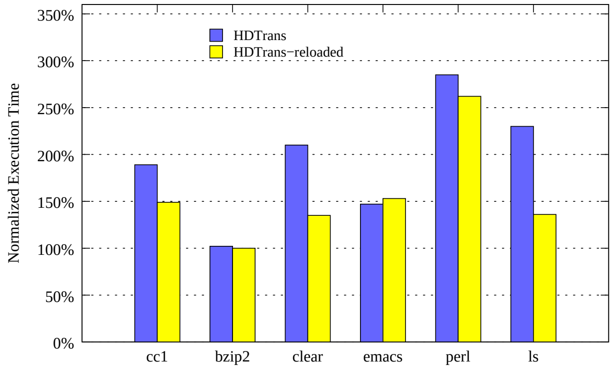

Figure 11. Overhead of HDTrans-pure and HDTrans-reloaded for some cold cache benchmarks.

to dump and reload its translation cache and associated metadata. To perform this comparison, Linux address space randomization is disabled. Figure 11 shows the performance of the purely dynamic version of HDTrans and the reloaded version for programs evaluated in the previous section. With the translation cache reloaded, very few basic blocks are dynamically translated in the second execution. Measurement shows that the overhead of the reload itself is negligible. As expected, higher execution overhead is incurred in programs having a high dynamic frequency of indirect control transfers, and the translation overhead is highest for programs exhibiting the least dynamic code reuse. A detailed analysis of the translation and execution time overheads in the case of Pin can be found in [39].

4.9 Utility of Static Pretranslation

The idea of using a semi-static approach to binary translation dates back to May's work on MIMIC [35]. FX!32 [30] uses a combination of emulation and profile driven binary translation for translating x86 binaries to Alpha systems. In HDTrans we considered employing a hybrid approach to translation, in which we statically pre-warm the basic block cache using a best-effort static disassembly and code-emission loop. While static disassembly of x86 code is imprecise, falsely identified basic blocks are dynamically unreached (therefore harmless), and missing basic blocks can be generated at runtime by the dynamic translator. For some programs, static pretranslation might provide substantial performance gains by recognizing common compiler idioms (e.g. `switch` statements or `vtable` dispatch) and eliminating the need for most exit stubs.

Using a minor variant of Kruegel *et al.*'s obfuscated disassembly techniques [33] (we assume that a call is followed by instruction bytes), we have confirmed that over 98% (often more than 99%) of the dynamically executed basic blocks can be statically identified and pretranslated. Therefore, the *reloaded* bar in Figure 11 is a reliable estimate of the performance of a hybrid translator *provided* that no substantial overhead is incurred when loading the statically generated precache in an unconstrained operating environment. The HDTrans source tree includes an implementation of this pretranslation strategy. Support for relocating reloading is currently unimplemented, because a substantial intrinsic overhead seems to exist in reloading.

The difficulty lies in the widespread use of address space randomization, which implies that absolute addresses embedded in the load image must be relocated when the image is loaded. Unfortunately, each page modified during reload incurs a demand copy-on-write (COW) overhead. Similar overheads are well known in the garbage collection literature, and have led to the abandonment of MMU-based guard pages in modern garbage collectors.

If a PC-relative addressing mode was available it would be possible to emit position independent code in the BBcache, and thus facilitate reuse without regard to address space randomization. Unfortunately, the IA-32 architecture (along with most other architectures) does not provide such an addressing mode. In this case, it is necessary to embed absolute addresses in the BBcache. In HDTrans, emitting such absolute addresses is necessary for optimizing the call/return sequence as described in section 3.7. More importantly, instrumentation code that is inlined into the BBcache also relies on the emission of absolute addresses (ex: basic block counting).

Our results suggest that *any* static reuse strategy will substantially exceed the cost of re-running HDTrans in most cases. We therefore believe that static pretranslation is effective only for optimization or instrumentation strategies where the cost of translation is a dominating factor and repeated reuse is anticipated.

5. Related Work

We have discussed DynamoRIO and Pin extensively, and compared their techniques and performance with HDTrans throughout the paper. In this section we address other related systems. Readers interested in further information about dynamic translation systems and other binary rewriting tools should refer Chapter 10 of Breuning's dissertation [6]. The papers on Walkabout [14] and Dynamo[3] give particularly clear descriptions of how earlier, high-performance translators were constructed.

Valgrind Valgrind [36] is a dynamic binary analysis tool for profiling and debugging applications. It constructs a full intermediate representation of traces using an IA32-specialized intermediate form. This form is instrumented according to the requirements of the user-selected tracing "skin." Following instrumentation, the intermediate form is optimized and native code is re-emitted. In comparison to tools providing a per-instruction instrumentation API, the Valgrind intermediate form is both rich and complex, but enables more invasive profiling to be performed with tolerable overhead. The Valgrind intermediate form is particularly well suited for tracing of memory references, cache behavior, and related dynamic performance characteristics that depend on deterministic but statically unpredictable attributes of the execution. Valgrind can also be used to perform use-before-store checking.

VMWare VMWare [17] is a full-system virtual machine emulator that uses a combination of native execution for non-privileged code and dynamic translation to emulate privileged-mode behavior. Prior to the arrival of Intel's "Vanderpool" [45] and AMD's "Pacifica" [1] technologies, VMWare was the highest performance full-system emulator for IA32 in widespread use. VMWare, Inc. asserts that systems emulated by VMWare run at up to 95% of the speed of the underlying system, which implies an extremely low translation overhead for supervisor-mode code. This is consistent with the performance results reported in Figure 9. Lightweight, same-machine translation is ideally suited to codes (such as operating systems) that do not make intensive use of indirect control flow. When the added facts that (a) operating system execution accounts for less than 50% of total instructions on a normal system, and (b) operating systems make extensive reuse of code, it is conceivable that the VMWare-asserted overheads might be achievable.

While the VMWare license precludes reporting performance figures, our experiences and the experiences of the Xen [19] group have been much less favorable in practice. The most likely explanation for this is that translation for supervisor-mode code requires additional checks, and therefore requires a somewhat heavier translation mechanism than the one used by HDTrans.[3]

QEMU Bellard's QEMU [5] provides *cross-machine* full system emulation using dynamic translation. The current implementation can, for example, emulate a SPARC guest system running on an IA32 host. This is achieved by precompiling native code to emulate common target instruction sequences and "stitching" these sequences together to translate instructions into the QEMU basic block cache. A particularly clever implementation technique in QEMU is taking advantage of the native compiler to construct the target code sequences automatically, but this technique relies on assumptions about compiler register usage, and has recently proven to be fragile.

'C The 'C (pronounced "tick see") system [22] builds on Engler's previous work on low-overhead code generation [24, 21, 23] to allow compiler-generated dynamic code generation. For example, the 'C system defers decisions about loop unrolling until run-time when loop bounds are available. Instead of generating code to execute the unrolled loop, the 'C compiler may alternatively emit code that *generates* the loop code at run time and then executes that code. Because the technique is fully compiler directed, it is not truly a dynamic *translation* strategy.

Strata The Strata system [41] explores "continuous compilation," an approach in which the compiler and the dynamic translator collaborate to generate code at the most appropriate time. The translator performs runtime optimization, but may do so using compiler-generated hints or directives. As a concrete example of one place where this approach can significantly reduce run-time performance overheads, the Strata runtime translator can be given direct knowledge of many dynamic branch targets, and therefore should not exhibit the types of performance overhead seen for `eon` or `perlbmk` in Figure 5.

6. Conclusion

The key to dynamic translator performance is balancing the overhead of translation against the performance improvement in translated code. HDTrans shows that satisfactory performance can be achieved using a much simpler translation strategy than has previously been assumed. HDTrans emits code that is competitive with the best existing translators, but has significantly lower startup and translation overheads.

If the dynamic translator will be used in a ubiquitous translation application, cold cache performance must be considered. Most of the benchmarks in conventional benchmark suites such as SPEC INT2000 are designed to evaluate hot cache performance of statically optimized code. In consequence, they provide an unrealistically favorable assessment of dynamic translator performance – the case where translation costs are effectively irrelevant. Because of its lightweight translation approach, HDTrans demonstrates significantly better cold cache performance than DynamoRIO or Pin.

The success of the SYRAH group in adapting HDTrans for reverse execution and run-time security policy enforcement tends to support our view that exposing a lower-level translator interface facilitates instrumentation. The reverse execution work, in particular, requires changes in the low-level code generation strategy that would be difficult in a "closed" translation infrastructure.

Several authors have speculated on the possible benefit of static pre-warming of the dynamic translation cache. Our examination of translation overheads vs. execution overheads and the comparative cost of reloading the translation cache suggest that cache prewarm is unlikely to improve the performance of a lightweight instrumentation infrastructure.

Source code for the HDTrans translator may be downloaded from `http://srl.cs.jhu.edu`. The version reported here is version 0.3.

Acknowledgments

Kim Hazelwood, our shepherd, provided many useful suggestions which significantly improved the paper you now read. Christopher Kruegel graciously allowed us to use his obfuscated binary disassembler implementation as a starting point for our static translator. Vijay Janapa Reddi provided a generous amount of time confirming our understanding of Pin, and helping us ensure that we achieved a fair comparative measurement. Robert Cohn made available the latest version of Pin to help us resolve compatibility issues in Fedora Core 4. Jack Davidson inspired us to complete the multithreading support and look further into signal handler instrumentation. Derek Bruening clarified certain issues about DynamoRIO. Michael Scott Doerrie provided useful feedback and comments about this paper.

Chi-Keung Luk of Intel was kind enough to supply a copy of the Pin version used for their PLDI paper [34], and assist us in benchmarking it accurately. Sandeep Sarat assisted in setting up experiments for profiling and measurement. Harish Patil of Intel clarified some issues about SPECOMP.

References

[1] Advanced Micro Devices, Inc. AMD64 Architecture Tech Docs, 2005. `http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_7044,00.html`.

[2] ALTMAN, E., GSCHWIND, M., AND SATHAYE, S. BOA: The architecture of a binary translation processor. In *Research Report RC21665 IBM T.J. Watson Research Center* (2000).

[3] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2000), pp. 1–12.

[4] BELL, J. R. Threaded Code. In *Communications of the ACM* (June 1973), no. 6, pp. 370–372.

[5] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *Proc. 2005 USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[6] BREUNING, D. L. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, September 2004.

[7] BRUENING, D., AND AMARASINGHE, S. Maintaining Consistency and Bounding Capacity of Software Code Caches. In *Proc.3rd International Symposium on Code Generation and Optimization (CGO 2006)* (Mar. 2005), pp. 74–85.

[8] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An Infrastructure for Adaptive Dynamic Optimizations. In *Proc. International Symposium on Code Generation and Optimization* (2003), pp. 265–275.

[9] BRUENING, D., KIRIANSKY, V., GARNETT, T., AND BANERJIA, S. Thread-Shared Software Code Caches. In *Proc.4th International Symposium on Code Generation and Optimization (CGO 2006)* (Mar. 2006).

[10] BUNGALE, P., SRIDHAR, S., AND SHAPIRO, J. S. Low-Complexity Dynamic Translation in VDebug. Tech. Rep. SRL2004-02, Johns Hopkins University Systems Research Laboratory, May 2004.

[11] BUNGALE, P., SRIDHAR, S., AND SHAPIRO, J. S. Supervisor-Mode Virtualization for x86 in VDebug. Tech. Rep. SRL2004-01, Johns Hopkins University Systems Research Laboratory, May 2004.

[3] This assessment considers only 32-bit supervisor code. When executing 16-bit code or code with active segmentation, the VMWare translator must emit code to emulate the translation subsystem, which introduces noticeable degradation. As a practical matter this has little relevance to the overall performance of VMWare, because such code is dynamically rare and occurs primarily at boot time.

[12] Chen, W. K., Lerner, S., Chaiken, R., and Gillies, D. M. Mojo: A Dynamic Optimization System. In *ACM Workshop on Feedback-directed and Dynamic Optimization (FDDO-3)* (Dec 2000).

[13] Cifuentes, C., and Emmerik, M. V. UQBT: Adaptable binary translation at low cost. In *IEEE Computer, 33(3)*.

[14] Cifuentes, C., Lewis, B., and Ung, D. Walkabout–A Retargetable Dynamic Binary Translation Framework. In *Technical report 2002-106, Sun Microsystems Laboratories* (January 2002).

[15] Cmelik, B., and Keppel, D. Shade: A fast instruction- set simulator for execution profiling. In *ACM SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems* (1994), pp. 128–137.

[16] Deutsch, L. P., and Schiffman, A. M. Efficient Implementation of the Smalltalk-80 System. In *Proc. ACM Symposium on Principles of Programming Languages* (Jan. 1984), pp. 297–302.

[17] Devine, S., Bugnion, E., and Rosenblum, M. Virtualization System Including a Virtual Machine Monitor for a Computer with a Segmented Architecture. In *United States Patent 6,397,242* (May 2002).

[18] Dewar, R. B. Indirect Threaded Code. In *Communications of the ACM* (June 1975), no. 6, pp. 330–331.

[19] Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., and Neugebauer, R. Xen and the Art of Virtualization. In *Proc. 2003 ACM Symposium on Operating Systems Principles* (Oct. 2003), pp. 164–177.

[20] Ebcioglu, K., and Altman, E. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *In Proc. 24th International Symposium on Computer Architecture* (June 1997), pp. 26–38.

[21] Engler, D. VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. In *Proc. 23rd Annual ACM Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA, May 1996).

[22] Engler, D., Hsieh, W. C., and Kaashoek, M. F. 'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation. In *Proc. 22nd Annual Symposium on Principles of Programming Languages* (Dec. 1995), pp. 131–144.

[23] Engler, D., and Kaashoek, M. F. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proc. SIGCOMM '96 Conference* (Stanford, CA, USA, Aug. 1992), pp. 53–59.

[24] Engler, D., and Proebsting, T. A. DCG: An Efficient, Retargable Dynamic Code Generation System. In *Proc. ASPLOS-VI* (Oct. 1994), pp. 238–245.

[25] Hazelwood, K. *Code Cache Management in Dynamic Optimization Systems*. PhD thesis, Harvard University, Cambridge, MA, May 2004.

[26] Hazelwood, K., and Cohn, R. A Cross-Architectural Framework for Code Cache Manipulation. In *4th Annual International Symposium on Code Generation and Optimization* (March 2006).

[27] Hazelwood, K., and Smith, J. E. Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems. In *2nd Annual International Symposium on Code Generation and Optimization* (March 2004), pp. 89–99.

[28] Hazelwood, K., and Smith, M. D. Code Cache Management Schemes for Dynamic Optimizers. In *Proc. Sixth Annual Workshop on Interaction between Compilers and Computer Architectures* (Feb. 2002), pp. 102–110.

[29] Hazelwood, K., and Smith, M. D. Generational Cache Management of Code Traces in Dynamic Optimization Systems. In *36th Annual International Symposium on Microarchitecture* (San Diego, CA, December 2003), pp. 169–179.

[30] Hookway, R. J., and Herdeg, M. A. DIGITAL FX!32: Combining Emulation and Binary Translation. In *Digital Technical Journal, 9(1):3–12* (1997).

[31] Hunter, C., and Banning, J. DOS at RISC. In *Byte Magazine* (Nov. 1989), pp. 361–368.

[32] Kiriansky, V., Bruening, D., and Amarasinghe, S. Secure Execution via Program Shepherding. In *11th USENIX Security Symposium* (August 2002).

[33] Kruegel, C., Robertson, W., Valeur, F., and Vigna, G. Static Disassembly of Obfuscated Binaries. In *Proceedings of USENIX Security 2004* (August 2004).

[34] Luk, C. K., Cohn, R. S., Muth, R., Patil, H., Klauser, A., Lowney, P. G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation. In *Programming Languages Design and Implementation 2005* (June 2005), pp. 190–200.

[35] May, C. MIMIC: A fast System/370 simulator. In *Proc. SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques* (June 1987), pp. 1–13.

[36] Nethercote, N. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004.

[37] Operation, A. U. S. *System V Interface Definition*. 1989.

[38] Rau, B. R. Levels of Representation of Programs and the Architecture of Universal Host Machines. In *Proc. 11th Annual Workshop on Microprogramming* (1978), pp. 67–79.

[39] Reddi, V. J., Connors, D. A., and Cohn, R. S. Persistence in Dynamic Code Transformation Systems. In *Proc. 2005 Workshop on Binary Instrumentation and Analysis* (Sept. 2005).

[40] Scott, K., Kumar, N., Childers, B., Davidson, J., and Soffa, M. Overhead Reduction Techniques for Software Dynamic Translation. In *NSF Workshop on Next Generation Software* (April 2004).

[41] Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J., and Soffa, M. Retargetable and Reconfigurable Software Dynamic Translation. In *ACM SIGMICRO Int'l. Conf. on Code Generation and Optimization* (March 2003).

[42] Shapiro, J. Debug: The Next Generation UNIX Debugger, 1989.

[43] Shapiro, J. S., Northup, E., Doerrie, M. S., and Sridhar, S. Coyotos Microkernel Specification, 2006. `http://www.coyotos.org/`.

[44] Shapiro, J. S., Smith, J. M., and Farber, D. J. EROS: A fast capability system. In *In Proc. 17th ACM Symposium on Operating Systems Principles* (Dec. 1999), pp. 170–185.

[45] Shiveley, R. Enhanced Virtualization on Intel Architecture-based Servers. In *Technology@Intel Magazine* (april 2005).

[46] Sridhar, S., Shapiro, J. S., and Bungale, P. P. HDTrans: A Low-Overhead Dynamic Translator. In *Proc. 2005 Workshop on Binary Instrumentation and Analysis* (Sept. 2005).

[47] Standard Performance Evaluation Corporation. SPEC OMP OpenMP Benchmark Suite, version 3.0, Dec. 2003. `http://www.spec.org/omp`.

[48] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmark Suite, version 1.3, Nov. 2005. `http://www.spec.org/ osg/cpu2000`.

[49] Witchel, E., and Rosenblum, M. Embra: Fast and exible machine simulation. In *Measurement and Modeling of Computer Systems* (1996), pp. 68–79.