HDTrans: A Low-Overhead Dynamic Translator

Swaroop Sridhar swaroop@cs.jhu.edu
Jonathan S. Shapiro shap@cs.jhu.edu
Systems Research Laboratory
Department of Computer Science
Johns Hopkins University

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

Abstract

Dynamic translation is a general purpose tool used for instrumenting programs at run time. Many current translators perform substantial rewriting during translation in an attempt to reduce execution time. When dynamic translation is used as a ubiquitous policy enforcement mechanism, the majority of program executions have no dominating inner loop that can be used to amortize the cost of translation. Even under more favorable usage assumptions, our measurements show that such optimizations offer no significant benefit in most cases. A simpler, more maintainable, adaptable, and smaller translator may be preferable to more complicated designs.

In this paper, we present HDTrans, a light-weight IA-32 to IA-32 binary translation system that uses some simple and effective translation techniques in combination with established trace linearization and code caching optimizations. We also present an evaluation of translation overhead under non-ideal 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-compile static pre-translation techniques for overhead reduction. Our results indicate that static pre-translation is effective only when expensive instrumentation or optimization is performed, and that efficient reload of pre-translated code incurs a substantial execution-time penalty.

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 and compatibility challenges. Dynamo [1] and Mojo [3] perform run time optimization to improve the performance of native binaries. Valgrind [10] uses sophisticated dynamic translation methods to perform heavy-weight dynamic binary analysis which can be used for comprehensive performance measurements, profi ling, memory analysis, and debugging. Shade [5] uses dynamic translation for high-performance instruction set simulation. VMWare [6] uses selective dynamic translation to achieve full machine virtualization. DynamoRIO [2] and Strata [12] provide an infrastructure for customizable instrumentation and optimization. Pin [9] is a dynamic binary instrumentation tool that provides a high level API for run time instrumentation of programs that can be used towards computer architecture research and education. Program Shepherding [7] uses dynamic translation to monitor control flow transfers in order to enforce security policies on program execution.

Many current translator implementations involve complex translations to generate an optimized target code cache. The rationale for doing so is the expectation that most of the execution time is spent in the code cache, and hence, optimizing it will amortize the overhead of a complex translator. This expectation is violated in programs that have low percentages of dynamic code re-use.

The complexity of a translation system depends on the degree of instrumentation required. Our experience suggests that for most applications, a simple translator is more important than a clever optimization strategy. When the desired instrumentation can be achieved at instruction granularity, a simple translator suffi ces. When it cannot, the performance overhead of dynamic instrumentation is quickly dominated by increases in translator size and complexity. This makes ubiquitous use of dynamic-translation infrastructures (for example, ubiquitous program shepherding as a "drop in" mechanism for existing systems) infeasible. In this paper, we describe and evaluate HDTrans — a simple, high-performance light-weight IA32-to-IA32 dynamic translator that is optimized for simplicity.

The rest of the paper is organized as follows. In section 2, 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 3, we then present comparative performance results with existing systems using both standard benchmarks and everyday programs. Later, in section 4, we analyze the overheads involved in translation, and consider the use of static-pretranslation techniques for overhead reduction in dynamic-translators.

2 HDTrans

HDTrans is a dynamic translator for the IA-32 family that combines a small number of new mechanisms with well-established techniques used in existing translators.

2.1 Basic Translator

The basic mechanism of translation is well understood and explained in the literature. This section aims to present the distinctive design choices made in HDTrans, and readers interested in further explanation may refer to Dynamo [1], Walkabout [4], etc.

HDTrans executes in a coroutine fashion with its subject binary image. Source basic blocks (a sequence of straight-line instructions that is "bracketed" by branches) are translated into a basic block cache (BBCache), and a directory of all such translated basic blocks indexed by source program counter is

maintained. Directory entries are added *before* the corresponding block is translated to ensure that self-looping traces reuse existing basic blocks. Translated basic blocks are never removed from the BBCache. In the rare situation where the BBCache becomes full, both the BBCache and the translation directory are discarded and the translation process starts over.

HDTrans performs instruction-at-a-time IA32-to-IA32 binary translation. In order to minimize the overall cache footprint, the translator is table driven. The table embodies the rules for decoding each instruction. It also identifies the emitter function to be used in each case. Each entry in the table occupies a single cache line, and a maximum of three entries are visited for every instruction decoded. In the absence of any instrumentation, a majority of instructions are translated by copying them verbatim into the BBCache. Instructions dealing with control-flow need special handling in order for the translator to remain in control of the application. As execution proceeds, a steady state converges rapidly on a situation where all dynamically active basic blocks have been translated.

This approach does restrict instrumentation to *instruction-at-a-time* methods, but significantly reduces the overall cost of translation. No attempt is made to optimize the target code cache except for trace linearization. This approach should be seen in contrast to Valgrind [10], which builds a high level intermediate form to support a rich pool of instrumentation options, and Pin [9], which performs sophisticated run time optimizations including optimization of the instrumentation code.

2.2 Register State

One of the important design decisions in building dynamic translators is to decide where the definitive copy of the application's register state is maintained. On the Pentium, where registers are few, there is considerable incentive to preserve all available registers for the application's use. Otherwise, excessive register spills will effectively mandate a translation strategy with intermediate code generation and register reallocation. In HDTrans, since we are translating user-mode programs, we assume that the guest has a valid stack that is backed by a general purpose page fault handler. This constraint must be maintained in any case in order for signal handling to work. HDTrans saves the register state whenever necessary by pushing it on the application stack. We ensure that HDTrans is never observed to have changed the application stack during the execution of the application's instructions. Saving the register state on the stack is also an effective way to handle register-states of multi-threaded programs (we assume implicitly that two threads of control in the same application do not examine each other's stacks).

2.3 Direct Branches and Traces

When a basic blocks ends with an unconditional immediate `jump` to a previously untranslated basic block, we elide the `jump` and continue to translate destination basic block immediately, so that the execution can proceed in straight line. In the case of a `call` instruction, we proceed by translating the instructions past the `call` instruction and *not* the destination of the `call`. If the destination of the `call` has already been translated, a `jump` is emitted to that basic block. Otherwise, suitable arrangements are made (Section 2.4) so that the target of the `call` is translated lazily. The translation of `call` instructions is further explained in the section on return-caching.

The above translation scheme is illustrated using the following example. We use gcc syntax for the assembly fragments illustrated in this paper. If the source instructions of the application are:

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

dest:  mov $30, %ecx
       call $proc
next:  add $4, %esp
       ...
```

The corresponding translated instructions in the BBCache will be:

```
add  $20, %ecx
mov  $30, %ecx
push $next
jmp  $<translation of proc>
add  $4, %esp
...
```

Translation terminates when an indirect branch is encountered or when the destination of a direct `jump` has already been translated. No optimization is performed, source basic block boundaries are preserved, and the translator records a new basic block in the translation directory for mid-trace blocks. Our goal is to form traces passively, but to reuse existing translations in preference to trace duplication. This approach is in contrast to DynamoRIO [2], which has a separate trace cache in addition to the basic block cache and maintains hot traces based on the Next Execution Tail [1] scheme. Measurements show that the HDTrans passive trace construction strategy commonly emits trace lengths of 4-5 basic blocks, or between 10 and 15 instructions. The longest measured trace was 256 basic blocks with over 1,100 instructions in the case of `gcc`.

2.4 Conditional Branches, Backpatching

Translation at a conditional immediate branch proceeds by linearizing the trace in a straight line, on the assumption that the branch is not taken. If the destination of the branch is known, a conditional jump is emitted to that translated basic block. Otherwise, we conditionally branch to a custom emitted fixup trampoline called the "patch block" inside the BBCache. The patch block contains a `call` to the translator and notes the address of the conditional branch and its destination. After eventual translation of the destination, the original branch instruction is patched *in place*, so that further jumps can go directly to the destination block. Emission of patch blocks is deferred to the end of the current trace translation, so that it never subdivides source traces. The translator assumes that the instruction following a conditional branch is likely to be a

basic block start, and inserts a basic block directory entry for this instruction to suppress duplicate trace constructions.

As an example, consider the following source instruction sequence, where 'dest' is not yet translated.

```
       add $20, %ecx
this:  jcc $dest
       sub $20, %ecx
       ...
```

These instructions are translated as:

```
       add $20, %ecx
       jcc $pb-n
       sub $20, %ecx
       ...
       ... ;<end of trace>
pb-n:  call $translator
       $dest
       $this
```

After 'dest' is translated this sequence changes to:

```
       add $20, %ecx
       jcc $<translation-of-dest>
       sub $20, %ecx
       ...
```

2.5 Indirect Branches

A decisive factor in the performance of the translated code is the handling of indirect jumps. 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. Many techniques have been proposed in literature for handling this case, including inlining of the most frequently used or recently used destination into the trace, and then emitting a comparison to ensure correctness. HDTrans uses a simpler mechanism in which we implement a hash-table (called the "sieve") using a hash of the destination address. In order to reduce register pressure and cache pollution, the sieve is implemented using blocks of instructions rather than blocks of data.

An indirect jump instruction such as `jmp *dest` is translated as:

```
       push *dest
       jmp $sieve-dispatch-bb
```

The `sieve-dispatch-bb` is a special basic block that computes the hash of the destination and *jumps* to the proper hash bucket. We use a sieve having 2^{15} buckets. The hash computation is performed in a way that the condition codes are not affected, as suggested in [2]:

```
       push %ecx
       mov 4(%esp), %ecx
       leal 0(,%ecx,4), %ecx
       movzwl %cx, %ecx
       leal $sieve-table(,%ecx,2)
       jmp *%ecx
```

The buckets of the sieve are initialized with a `jump` to the translator. As the entries are added, the target of the `jump` is updated to go to the corresponding hash table entry. Each entry consists of code that checks for a possible destination and branches on mismatch to the next entry:

```
       mov 4(%esp), %ecx
       lea -curr-src-eip(%ecx), %ecx
       jecxz match
       jmp $next-bucket
match: pop %ecx
       leal 4(%esp), %esp
       jmp $curr-translated-block
```

Each hash chain terminates with a fall back case that invokes the translator to translate the destination. The sieve blocks are built *only* for those indirect branches that are taken at least once. The length of the sieve chains are observed to be 1 or 2 on an average, and are never more than 4. As a further optimization, we can divide the sieve table into two sieves – one for looking up the destination of indirect `jumps` and another for indirect `calls`, as these destinations are mutually exclusive in practice.

2.6 Return Caching

The `return` instruction is by far the most important form of indirect branch in terms of dynamic frequency. However, any attempt to optimize the call return sequence should not alter the activation stack in a way that is detectable by the subject program. For example, some dynamic translators [11] 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 `setjmp()`/`longjmp()`.

In order to efficiently implement the `return` instruction, HDTrans uses a *return cache*. The return cache is a fixed size D-space direct mapped hash table, indexed by a simple hash of the destination procedure start address. The translation of a `call` instruction pushes the original *untranslated* return address on the stack, computes the appropriate return cache bucket, and stores the *translated* return address into this return cache entry (as shown below). If the `call` in question is a direct call, the return cache bucket calculation can be done at dynamic compile time.

```
    ;; call $someProc:
     push $source-ret-address
     mov $post-call, ret-cache-entry
     jmp $someProcTrans
post-call:
     <call-postamble>
```

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 associated with the return instruction (Figure 1):

```
    ;; return from someProc:
     jmp *my-return-cache-bucket
```

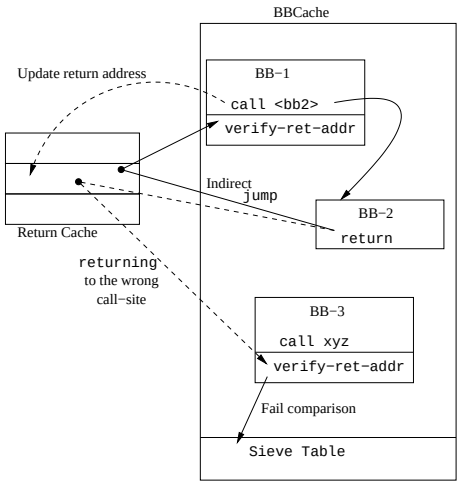

Figure 1: Return cache control-flow.

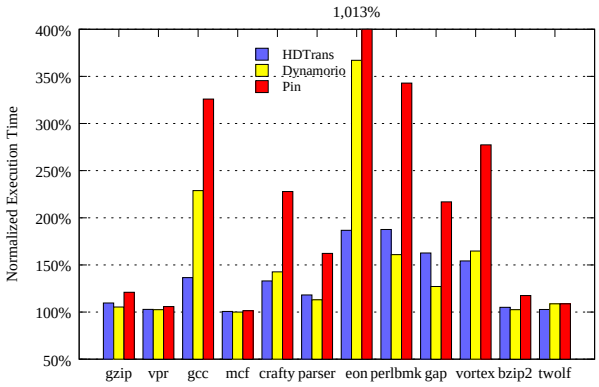

Figure 2: SPEC INT2000 benchmarks.

Due to return cache collisions (which may be caused due to recursion, for example), this method can result in mistranslated returns. We rely on the fact that the only way to insert an entry into the return cache is to execute *some* `call` instruction. The translator emits code following each `call` to check that control has returned to the intended destination. A comparison is done between the known static address of the instruction following the `call`, and the return address left on the stack by the `return` translation. If these addresses match, then the stack pointer is incremented past the return address, and the computation continues normally. Otherwise, the sieve is used to locate the correct return point, exploiting the fact that the "dispatch frame" left on the stack by the translated `return` instruction matches the one used for indirect branches. The use of this fallback method has been observed to be dynamically rare.

```
        ;; call-postamble
        push %ecx
        mov 4(%esp), %ecx
        leal -src_ret_addr(%ecx), %ecx
        jecxz equal
        jmp $sieve-dispatch-bb
equal:  pop %ecx
        leal 4(%esp), %esp
```

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.

3 Performance

3.1 Benchmark Performance

HDTrans compares favorably with the leading dynamic translation systems. Figure 2 shows the performance of HDTrans on the SPEC INT2000 [13] benchmarks in comparison to DynamoRIO and Pin. Benchmarks are compiled with gcc version 3.4.4 and run on a dual processor, hyperthreaded Intel(R) Xeon(TM) CPU 2.80GHz system with 512 KB L2 cache and 6GB main memory, running Linux Fedora Core 3 (2.6.12-1.1376_FC3smp kernel). All benchmarks are single-threaded.

HDTrans does not (yet) support signals or threads. The DynamoRIO website [14] notes that DynamoRIO does not support threads or signals reliably as well. We ran Pin *without* the `-mt` option which is used to enable the execution of multi-threaded programs. Also, we believe that supporting signals that *do not* examine thread context can be done without a signifi cant overhead.

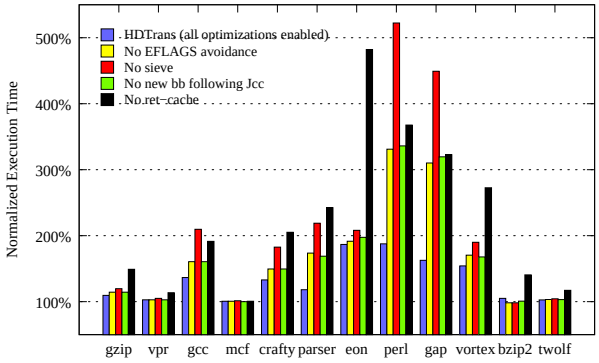

Figure 3: SPEC INT2000 benchmarks with optimizations selectively disabled.

Figure 3 shows the performance of HDTrans on SPEC INT2000 benchmark with individual optimizations disabled one at a time. These measurements demonstrate that the sieve and return cache optimizations are critical, and that maximizing basic block reuse is an effective choice for indirection intensive programs. The last result calls into question the previously published arguments in favor of explicit trace construction.

The performance of dynamic translation relies on Amdahl's law. Translator overhead can only be amortized if the majority of a program's execution time is spent in a small fraction of the code. This assumption is violated when dynamic translation is

applied pervasively (e.g. as proposed in Program Shepherding [7]), where the program executions are less likely to have a dominating inner loop. Benchmarks such as SPEC INT2000 are designed to measure the performance of inner loops, and therefore tend to minimize translation overheads. In consequence, it is doubtful that these results accurately predict the performance of machine-level dynamic translators in production use.

3.2 Cold Cache Translation Overheads

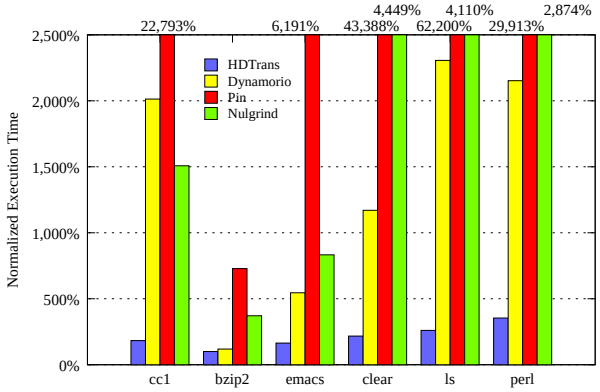

Figure 4: Overhead for some cold cache benchmarks.

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 3.4.4) 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 5.8.5) run on a 200 line script that generates random passwords

Figure 4 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.

3.3 Analysis of Overhead

The overhead of dynamic translation can be divided into the cost of translation *per se* and the execution overhead introduced by the translation process. To isolate these effects, we modified HDTrans to dump and reload its translation cache and associated metadata. To perform this comparison, Linux address space randomization is disabled. Figure 5 shows the performance of the purely dynamic version of HDTrans and the reloaded version for programs evaluated in the previous

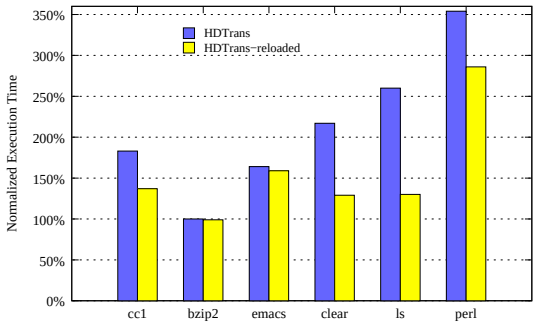

Figure 5: Overhead of HDTrans-pure and HDTrans-reloaded for some cold cache benchmarks.

section. With the translation cache reloaded, only one or two 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.

3.4 Utility of Static Pretranslation

We visited several performance plateaus during the development of HDTrans, and at each one we thought "This is it, doing any better will require static precaching." 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 patch blocks.

Using a minor variant of Kruegel *et al.*'s obfuscated disassembly techniques [8] (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 5 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 exists 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.

In most cases, absolute addresses used in HDTrans emitted code reside in trampoline basic blocks that are gathered at

the beginning of the basic block cache. If copy-on-write costs could be successfully restricted to this page, relocation might be cost effective. Unfortunately, emitting efficient position-independent code for the call/return sequence is difficult without a PC-relative memory addressing mode. The IA-32 (along with most other architectures) does not provide such an addressing mode. The "no ret-cache" bar of Figure 3 shows the performance impact of maintaining a precise stack without the return cache: a 200% to 350% slowdown in execution performance (i.e. in *addition* to the cost of relocation). Taken alone, this execution overhead is significantly higher than the cost of *de novo* dynamic retranslation.

Collectively, 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.

4 Conclusion

Conventional benchmarks such as SPEC INT2000 are designed to evaluate inner loop 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.

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.

Source code for the HDTrans translator may be downloaded from `http://srl.cs.jhu.edu`.

Acknowledgements

Eric Northup provided feedback and clarifications about many IA32 architecture issues. Sandeep Sarat assisted in setting up experiments for profiling and measurement. Christopher Kruegel graciously allowed us to use his obfuscated binary disassembler implementation as a starting point for our static translator. Vijay Janapareddi provided a generous amount of time confirming our understanding of Pin, and helping us ensure that we achieved a fair comparative measurement.

References

[1] BALA, V., DUESTERWALD, E., AND BANERJIA, S. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2000).

[2] 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.

[3] 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).

[4] CIFUENTES, C., LEWIS, B., AND UNG, D. Walkabout– A Retargetable Dynamic Binary Translation Framework. In *Technical report 2002-106, Sun Microsystems Laboratories* (January 2002).

[5] 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.

[6] 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).

[7] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure execution via program shepherding. In *11th USENIX Security Symposium* (August 2002).

[8] KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. Static Disassembly of Obfuscated Binaries. In *Proceedings of USENIX Security 2004* (August 2004).

[9] LUK, C., COHN, R. S., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, P. G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. M. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming Languages Design and Implementation 2005* (June 2005), pp. 190–200.

[10] NETHERCOTE, N. Dynamic Binary Analysis and Instrumentation. In *PhD Dissertation, University of Cambridge* (November 2004).

[11] 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, during the Int'l. Parallel and Distributed Processing Symposium* (April 2004).

[12] 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).

[13] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC CPU2000 Benchmark Suite, 2000. `http://www.spec.org/osg/cpu2000`.

[14] THE DYNAMORIO COLLABORATION. Using DynamoRIO – What Does and Doesn't Work, 2005. `http://www.cag.lcs.mit.edu/dynamorio/using.html/#Unsupported`.