*International Journal of Parallel Programming, Vol. 33, Nos. 2/3, June 2005 (© 2005)*
DOI: 10.1007/s10766-005-3573-7

# Compile-Time Planning for Overhead Reduction in Software Dynamic Translators

Naveen Kumar,[1] Bruce R. Childers,[1,3] Daniel Williams,[2] Jack W. Davidson,[2] and Mary Lou Soffa[2]

Software dynamic translation (SDT) is a technology for modifying programs as they are running. The overhead of monitoring and modifying a running program's instructions is often substantial in SDT systems. As a result, SDT can be impractically slow, especially in SDT systems that do not or can not employ dynamic optimization to offset overhead. This is unfortunate since SDT has many advantages in modern computing environments and interesting uses of SDT continue to emerge. In this paper, we describe techniques to reduce the overhead of SDT. In particular, we present a compile-time planning technique to reduce the overhead due to indirect branch handling. Our results show that this technique is very effective and can improve SDT performance by up to 36%, with an average of 20%.

**KEY WORDS:** Dynamic translation; dynamic optimization; instruction traces.

## 1. INTRODUCTION

Software dynamic translation (SDT) is a technology that allows programs to be modified as they are running. SDT systems virtualize aspects of the host execution environment by interposing a layer of software between

[1] Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA. E-mail: {naveen, childers}@cs.pitt.edu

[2] Department of Computer Science, University of Virginia, Charlottesville, VA 22904, USA. E-mail: {dww4s, jwd, soffa}@cs.virginia.edu

[3] To whom correspondence should be addressed.

program and CPU. This software layer mediates program execution by dynamically examining and translating a program's instructions before they are run on the host CPU. Recent trends in research and commercial product deployment strongly indicate that SDT is a viable technique for delivering adaptable, high-performance software into today's rapidly changing, heterogeneous, networked computing environment.

SDT is used to achieve distinct goals in a variety of systems. One of these goals is binary translation. Cross-platform SDT allows binaries to execute on non-native platforms. This allows existing applications to run on different hardware than originally intended. Binary translation makes introduction of new architectures practical and economically viable. Some popular SDT systems that fall into this category are FX!32,(1) DAISY,(2) UQDBT,(3) and Transmeta's Code Morphing technology.(4)

Another goal of certain SDT systems is improved performance. Dynamic optimization of a running program offers several advantages over compile-time optimization. Dynamic optimizers use light-weight execution profile feedback to optimize frequently executed (hot) paths in the running program. Also, dynamic optimizers can continually monitor execution and reoptimize if the program makes a phase transition that creates new hot paths. Finally, dynamic optimizers can perform profitable optimizations such as partial inlining of functions and conditional branch elimination that would be too expensive to perform statically. SDT systems that perform dynamic optimization include Dynamo,(5) DBT,(6) and Voss and Eigenmann's remote dynamic program Optimization system.(7) Some of the binary translators previously described also perform some dynamic optimization (e,g., DAISY, FX!32, and Transmeta's Code Morphing).

SDT is also useful for virtualized execution environments. Such environments provide a framework for architecture and operating systems experimentation as well as migration of applications to different operating environments. The advantage of SDT in this area is that the simulation of the virtual machine is fast – sequences of virtual machine instructions are dynamically translated to sequences of host machine instructions. Examples of this type of SDT are Embra,(8) Shade,(9) and VMware.(10)

All uses of SDT can benefit from reductions of dynamic translation overhead. Reducing overhead improves overall application performance, allows SDT systems to implement additional functionality (e.g., additional optimizations, more detailed profiling, etc.), and enables uses of SDT in new application areas. In this paper, we describe a new technique for reducing the overhead of SDT with *compile-time planning*. Using Strata, a framework we designed for building SDT applications (for uni-processor and multi-processor machines), we performed experiments to identify and measure the sources of overhead in the basic translation actions done by

SDT. We observed that SDT overhead stems from just a few sources, including conditional and indirect branches. In particular, we implemented a technique for reducing SDT overhead associated with indirect control transfers. The resulting improvement in overhead ranges from 0% to 39% and demonstrates that compile-time knowledge can be used to improve the performance of dynamic translators.

## 2. SOFTWARE DYNAMIC TRANSLATION

SDT can affect an executing program by injecting new code, modifying some existing code, or controlling the execution of the program in some way. We have developed a reconfigurable and retargetable SDT system,(11) called Strata, which supports many SDT applications, such as dynamic optimization and safe execution of untrusted binaries.(12) It is available for many platforms, including SPARC/Solaris 9, x86/Linux, and MIPS/IRIX.

To realize a specific dynamic translator, Strata's basic services are extended to provide the desired functionality. Strata's default services implement a dynamic translator that mediates execution of native application binaries with no visible changes to application semantics and no aggressive attempts to optimize application performance. The basic services include memory management, fragment cache management, application context management, multithreading support, a dynamic linker, and a fetch/translate engine. All of these services are retargetable and extensible.

Execution of an application under Strata begins by invoking the Strata component known as the fragment builder with the starting program counter (PC) of the application. The fragment builder takes the PC, and if the application instruction at that PC is not cached, the fragment builder forms a sequence of instructions called a fragment. Strata attempts to make these fragments as long as possible. To this end, Strata inlines unconditional PC-relative control transfers into the fragment being constructed. In this mode of operation, each fragment is terminated by a conditional or indirect control transfer instruction. However, since Strata needs to maintain control of program execution, the control transfer instruction is replaced with a code trampoline that arranges to return control to the Strata fragment builder. Once a fragment is fully formed, it is placed in the fragment cache.

The transfers of control from Strata to the application and from the application back to Strata are *context switches*. On a context switch into Strata via a trampoline, a hash table is searched to determine if there is a cached fragment corresponding to the current PC. If a fragment is found, Strata immediately context switches back to the application; otherwise it builds a corresponding fragment before context switching back to the program. As discussed next, context switches are a large part of SDT overhead.

## 3. SOURCES OF OVERHEAD

SDT overhead can degrade overall system performance substantially. This is particularly true of dynamic translators which do not perform code optimizations to offset dynamic translation overhead. Overhead in software dynamic translators can come from time spent executing instructions not in the original program, from time lost due to the dynamic translator undoing static optimizations, or from time spent mediating program execution.

### 3.1. Methodology

To characterize overhead in such an SDT, we did experiments to measure where SDT systems spend their time. The experiments were done with Strata for the SPARC (called "Strata-SPARC") on a Sun Blade 100 (500 MHz and 256 MB RAM). In the experiments, no optimizations were done and a 2 MB fragment cache was used. Benchmarks from SPEC2K were compiled with the gcc C compiler (version 3.1) with "-O3" The train inputs were used for profiling and reference inputs for actual runs.

### 3.2. Non-indirect Branch Handling

In Strata's basic mode of operation, a context switch occurs after each fragment executes. A large portion of these context switches due to non-indirect branches can be eliminated by "linking" fragments together as they materialize into the fragment cache. For instance, when one or both of the destinations of a PC-relative conditional branch materialize in the fragment cache, the conditional branch trampoline can be rewritten to transfer control directly to the appropriate fragment cache locations rather than performing a context switch and control transfer to the fragment builder.

When fragment linking is used, there are 3.5–166.5*x* (average 26.4*x*) fewer builder re-enters than without fragment linking. This reduction in re-enters leads to a large performance improvement. Without fragment linking, there is a large slowdown over native execution – an average of 13*x* (23.4–2.4*x*) across all benchmarks. With fragment linking, the resulting slowdowns are much lower, but still high – an average of 3.2*x* across all benchmarks – and require other mechanisms to further reduce overhead.

### 3.3. Indirect Branch Handling

After applying fragment linking, the majority of the remaining overhead is due to indirect control transfer instructions. Because the target of an indirect control transfer is only known when the branch executes, Strata cannot link fragments ending in indirect control transfers to their targets. As a consequence, such fragments must save the application context and call the fragment builder with the computed branch-target address. The likelihood is very high that the requested branch target is already in the fragment cache, so the builder can immediately restore the application context and begin executing the target fragment. On the Sun Blade 100, the time between reaching the end of the indirect control transfer and beginning execution at the branch target averages about 250 cycles. For programs that execute many indirect control transfers, the overhead of handling the indirect branches can be substantial. To improve Strata's overhead further, we must either reduce the latency of individual context switches or the overall number of switches due to indirect control transfers. The code which manages a context switch is highly tuned assembler and it is unlikely this code can be improved any further. Thus, the challenge is how to reduce the number of context switches due to indirect branches.

## 4. COMPILE-TIME PLANNING FOR OVERHEAD REDUCTION

While fragment linking is effective for non-indirect branches, it does not handle indirect branches because the target address is unknown during fragment translation. However, profile information can be used, with *compile-time planning*, to predict the likely target of an indirect branch. In this approach, "trace plans" are generated by the compiler to determine fragment sequences, called *instruction traces*, which are on a hot path. The traces are used by the dynamic translator to place fragments contiguously in the fragment cache and to generate efficient inline code that jumps to the next fragment in the trace or invokes a separate translation process when the indirect branch target address is off trace.

### 4.1. Determining Instruction Traces

To determine the instruction traces, we developed an algorithm, called "next heaviest edge" (NHE), that uses an execution frequency profile. The profile is a set of CFG edges and their execution counts. NHE constructs a weighted control flow graph (WCFG) from the profile that is traversed to identify traces. In the WCFG, a vertex is a fragment from the profile and an edge is a pair of fragments with a weight. The WCFG is not the same as the CFG due to the presence of fragments and the absence of edges that were not executed.

Table I shows pseudocode for NHE. The algorithm begins by initializing a set of traces, `T`, to the empty set, a list of edge counts, `count []`, to 0, and constructs the WCFG `G` (line 2), Function `nextHead ()` finds a hot edge, seed, to serve as a start point for a trace (line 3). A check is made (also line 3) to ensure that the number of traces does not grow too large. The entry count `[seed]` is incremented to indicate how many duplicates there are of that edge in all traces (line 5). If seed is in too many traces, it is removed from `G` to avoid adding it to any subsequent traces (line 6).

The algorithm attempts to expand the trace in both forward and backward directions by adding edges from both the successor and predecessor edges of the trace head or tail. A trace is first expanded in the forward direction (lines 9–13). In this case, the successor to the trace tail with

**Table I. Next Heaviest Edge Algorithm**

```
Line   Description

 1     Traces nextHeaviestEdge(Profile P) begin
 2       int count[]← ∅; Traces T← ∅; WCFG G–constructWCFG(P);
 3       while ((Edge seed ← nextHead(G))≠ ∅ and |T| < MAXTRACES)
 4           Trace t ← seed;
 5           count[seed] ← count[seed] + 1; //number of instances
 6           if (count[seed] > MAXDUP) then removeEdge(G,seed);
 7           boolean expsucc ← true, exppred ← true;
 8           repeat
 9             if (expsucc) then
10               Edge s ← heaviestEdge(G, SUCC, t); expsucc ← false;
11               if (s ≠ ∅ and s.weight > HOTNESS) then
12                 expsucc ← addEdge(G,SUCC,t,s,seed,count);
13               end if;
14               if (exppred) then
15                 Edge p← heaviestEdge(G,PRED,t); exppred←false;
16                 if (p ≠ ∅ and p.weight > HOTNESS) then
17                   exppred←addEdge(G,PRED,t,p,seed,count);
18             end if;
19           until (¬(expandsuc or exppred))
20           T ← T ∪ t; // add new trace t to trace set T
21       end while;
22       return T;
23     end
```

the largest weight is selected. The selected successor, s, is added to the trace only if it meets a hotness criterion, the length of t has not exceeded a threshold, and the relative weight of s to the trace tail is sufficiently large (the latter two conditions are checked by function add Edge()). If any of these conditions are not satisfied, t is terminated in that direction (i.e., expandsucc is set to false). The same process is repeated for the predecessor edge of the trace head (lines 14–18). The expansion of t is terminated when the trace cannot be expanded in either the backward and forward directions (line 19). Finally, the newly created trace is added to T (line 20).

The function addEdge( ) is shown in Table II. This function adds an edge, e, to a trace, t, if three checks are satisfied. The first check is whether the length of t is below a threshold (line 3). If the trace is short enough, the second and third checks based on the relative weight of e are made (lines 11–19). The second check determines whether e accounts for a large portion of the execution flow associated with edge s (the seed). The

**Table II. Add an Edge to a Trace**

```
Line    Description

 1      Bool addEdge(WCFG G, Direction d, Trace t, Edge e, Edge s, int count[])
 2      begin
 3        if (|t| < MAXSIZE) then
 4          Edges K ← ∅;
 5          if (d = SUCC) then K ← successorEdges(G,t);
 6          else K ← predecessorEdges(G,t); // predecessor
 7          int total_weight ← 0;
 8          forall (Edge k ∈ K) do
 9            total_weight ← total_weight + k.weight;
10          end forall;
11          float flow ← FLOWFRACT × s.weight;
12          float dom ← DOMFRACT × total_weight;
13          if (e.weight > flow and e.weight > dom) then
14            if (d = SUCC) then t ← t . e; // add to trace end
15            else t ← e , t; // add to trace beginning
16            e.weight ← e.weight - flow;
17            count[e] ← count[e] + 1;
18            if (count[e] > MAXDUP) then removeEdge(G, e);
19            return true;
20          end if;
21        end if;
22        return false;
23        end
```

threshold for the amount of flow is computed on line 11. If e accounts for a portion of the flow that is equivalent to the constant FLOWFRACT, the edge can be added to the trace. The third condition determines whether e is the dominant edge among all successors (or predecessors), Lines 8–10 compute the total weight among all successors (or predecessors) for the trace tail (or head). Based on this weight, a threshold is computed on line 12 for the dominance of e. If e's weight is above this threshold, it can be added to t. The edge is appended to t's tail when expanding forwards (line 14) and to t's head when expanding backwards (line 15).

When e is added, its weight is decreased by the amount of flow associated with the edge (line 16). Decreasing the weight ensures that the priority of e is reduced based on the amount of flow "accounted for" by the current trace. Finally, the function returns true when an edge is successfully added to the trace to indicate that further expansion is possible.

### 4.2. Generating and Using the Trace Plan

Once the traces have been determined, they are used to generate a trace plan, which consists of the sequence of fragments associated with each trace. To generate the plan, the traces are converted from their edge representation to a fragment representation by replacing each edge with the addresses of its fragment vertices. A fragment address is recorded as an offset from the start of the program binary that can be used to determine the location of the fragment when the program is loaded at run-time.

The trace plan is read by Strata when the program is executed. Because the plan is a list of addresses (i.e., not actual code), a code generation step is done to emit the trace code into the fragment cache. The code generator processes each trace and traverses the instructions in the fragments in the trace. For every instruction in a trace, it is fetched, translated and emitted into the fragment cache. Indirect branches and conditional branches need special attention. For indirect branches, the target address should be on the trace. However, because an indirect branch changes control flow based on a register value, a check is inserted to ensure that the target is the next fragment. If the target is not on the trace, then control is transferred to a "trace exit" that translates the indirect's target address by context switching to Strata. Conditional branches are handled by rewriting the branch to make its not-taken direction be the next fragment in the trace. This step is essentially the same as fragment linking, except the branch condition may be inverted to ensure that the hot branch direction is on the trace. Also, unlike fragment linking, the code emitted for handling a conditional branch is simplified because a trampoline is unnecessary for the on-trace target, After the code has been generated for all fragments, an exit fragment is emitted at the end of the trace.

### 4.3. Experiments

To investigate the overhead reduction with traces, we profiled the benchmarks with their training data sets. The profile was used to determine instruction traces with NHE. These traces were saved to preload the fragment cache whenever Strata-SPARC is invoked. In the subsequent run, we used the reference input set for each benchmark.

Figure 1 shows how static trace formation can reduce the number of builder re-enters due to indirect branches. The figure shows that static trace formation has 1.01–1.8*x* (average 1.35*x*) fewer builder re-enters than fragment linking alone. Figure 2 shows how the reduction in builder re-enters affects performance. The figure shows the slowdown of preloading traces over not preloading traces. The improvement over fragment linking ranges from 0% to 36%, with an average of 20%. From our experiments, the improvement is due to both a reduction in context switches and instruction cache misses. These results are encouraging because they show that compile-time information can be used to reduce SDT overhead.

## 5. RELATED WORK

Software dynamic translation has been used for a number of purposes (see Section 1), including dynamic binary translation of one machine

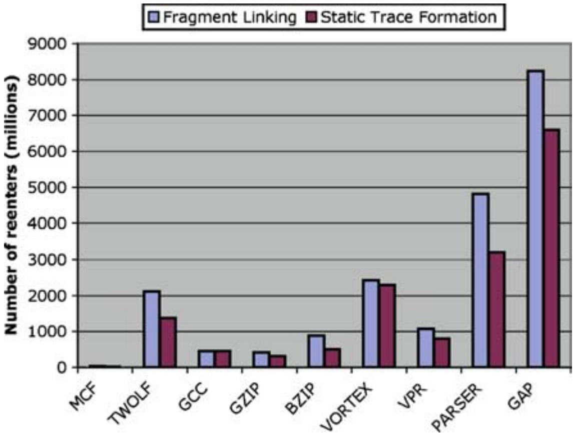

Fig. 1. Improvement in number of re-enters with trace plans.

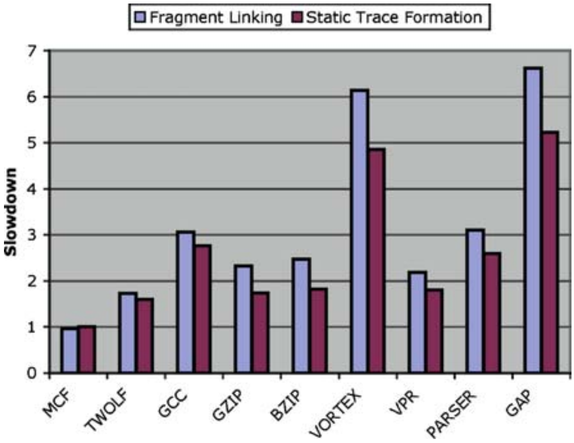

Fig. 2. Improvement in slowdown with trace plans.

instruction set to another,[1–4] emulation of operating systems (e.g., VMWare), and machine simulation.[8,9] There has also been work on general infrastructures for SDT that are similar to Strata. Walkabout is a retargetable framework that uses a machine-dependent intermediate representation to translate and execute binary code from a source machine on a host machine.[13] Another framework is DynamoRIO,[14] which is a library and set of API calls for building SDTs on the x86. Unlike Strata, DynamoRIO was not designed with retargetability in mind.

To achieve good performance in a SDT system, it is important to reduce the overhead of conditional and indirect branches. Shade[9] and the Embra[8] emulator use a technique called chaining to link cache-resident code fragments to bypass translation lookups. This technique is similar to fragment linking in Strata, which avoids context switches. Dynamo[5] and DAISY[2] convert indirect branches to chains of conditional branches to improve performance. Rather than eliminate context switches, the conditional branch chains remove indirect branch penalties. Since the conditional branch chains must be kept short to maintain any increases in performance, an indirect branch typically terminates the conditional branch chain to handle the case when none of the conditional branch comparisons actually match the branch-target address. For programs with many switch statements and large numbers of frequently executed cases, e.g., gcc, the conditional branch comparisons will frequently not match the branch-target address resulting in a context switch. However, with our approach, as long as the compile-time traces are representative of program behavior, the branch-target address should match inlined address translation.

## 6. SUMMARY

Reducing the overhead of software dynamic translation (SDT) is critical for making SDT systems practical. Using SPEC2K, we performed detailed measurements to determine major sources of SDT overhead. Our measurements revealed that the major source of overhead comes from conditional and indirect branches. To reduce the overhead of conditional branches, a technique called fragment linking is used to rewrite the trampoline code to transfer control directly to the appropriate fragment rather than doing a context switch. This technique reduced overhead by a factor of 4. For indirect branches, we described a compile-time planning approach that generates instruction traces to inline target address translation. This technique improved performance by up to 36% over fragment linking.

The results in this paper demonstrate that static information can be successfully used to reduce SDT overhead. We are continuing to develop other offline and online techniques to reduce overhead. Preliminary results indicate that by applying the techniques described here along with dataflow analysis of the executable, it may be possible to eliminate SDT overhead entirely. If achieved, this would make SDT a powerful tool for helping software developers achieve a variety of important goals including better security, portability, and better performance.

## ACKNOWLEDGMENTS

Kevin Scott did much of the original work on the design and implementation of Strata. This work was supported by National Science Foundation grants CNS-0305198, CNS-0305144, CNS-0203945 and CNS-0203956.

## REFERENCES

1. A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, FX!32: A Profile-Directed Binary Translator, *IEEE Micro*, **18**(2):56–64 (April 1998).
2. K. Ebcioglu and E. Altman, DAISY: Dynamic Compilation for 100% Architecture Compatibility, *International Symposium on Computer Architecture*, pp. 26–37 (1997).
3. D. Ung and C. Cifuentes, Machine-Adaptable Dynamic Binary Translation, *ACM Workshop on Dynamic Optimization*, pp. 41–51 (2000).

4. J, Dehnert, B. K, Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges, *International Symposium on Code Generation and Optimization*, pp. 15–24 (March 2003).
5. V. Bala, E. Duesterwald, and S. Banerjia, Dynamo: A Transparent Dynamic Optimization System, *ACM Conference on Programming Language Design and Implementation*, pp. 1–12 (2000).
6. K. Ebcioglu, E. Altman, S. Sathaye, and M. Gschwind, Optimizations and Oracle Parallelism with Dynamic Translation, *International Symposium on Microarchitecture*, pp. 284–295 (1999).
7. M. Voss and R. Eigenmann, A Framework for Remote Dynamic Program Optimization, *ACM Workshop on Dynamic Optimization*, pp. 32–40 (2000).
8. E. Witchel and M. Rosenblum, Embra: Fast and Flexible Machine Simulation, *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 68–79 (1996).
9. B. Cmelik and D. Keppel, Shade: A Fast Instruction-set Simulator for Execution Profiling, *ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pp. 128–137 (1994).
10. M. Rosenblum, Virtual Platform: A Virtual Machine Monitor for Commodity PCs, *Hot Chips 11* (1999) (http://www.hotchips.org/archives/hc11/3_Tue/hc99.s6.1.Rosenblum.pdf).
11. K. Scott, N. Kumar, S. Velusamy, B. R. Childers, J. W. Davidson, and M. L. Soffa, Retargetable and Reconfigurable Software Dynamic Translation, *International Symposium on Code Generation and Optimization*, pp. 36–47 (March 2003).
12. K. Scott and J. Davidson, Safe Virtual Execution using Software Dynamic Translation, *Annual Computer Security Application Conference*, pp. 209–218 (2002).
13. C. Cifuentes, B. Lewis, and D. Ung, Walkabout: A Retargetable Dynamic Binary Translation Framework, *Workshop on Binary Translation* (2002).
14. D. Bruening, T. Garnett, and S. Amarasinghe, An Infrastructure for Adaptive Dynamic Optimization, *International Symposium on Code Generation and Optimization*, pp. 265–275 (March 2003).