Dynamic Translation as a System Service

Marc L. Corliss Vlad Petric E Christopher Lewis

Department of Computer and Information Science
University of Pennsylvania
{mcorliss,vladp,lewis}@cis.upenn.edu

Abstract

Dynamic translation is a well-known and powerful technique for transforming programs as they run. Dynamic translators have many uses including profiling, security assurance, dynamic optimization, and bug patching. However, the utility of dynamic translation is severely limited by a lack of integration with the system in which it is used, instead, requiring individual users to initiate and program the translator. As a result, translation is not transparent, cannot be used to protect system-level security, and can only be programmed by a single party (the user).

In this work, we propose integrating dynamic translation with the operating system, providing dynamic translation as an operating system service (*DTSS*). With DTSS, the OS dynamically translates all applications according to translation rules provided by any entity (with sufficient access rights) in the system. In addition, this paper describes a number of challenges in implementing DTSS and presents solutions to address them. Performance overhead poses the greatest challenge, but code caching can dramatically reduce overhead. That multiple parties may transform a single program is also challenging. We describe how transformations may be composed and how they are isolated from one another. DTSS has great utility beyond traditional dynamic translation systems, but new implementation strategies are required.

1. Introduction

Dynamic translation is a technique for transforming programs as they run. Many dynamic translation systems have been constructed and shown to be efficient (introduce little overhead), flexible (can be used for a variety of purposes), and useful (can be used to solve problems more effectively than with other techniques) [6, 10, 17, 18, 22]. The main utility of dynamic translation comes from the fact that it is performed at runtime, which has the obvious benefits of supporting whole-program translation (including dynamically generated/loaded code) and allowing for program optimization based on runtime program behavior. Because dynamic translators operate on the runtime (*i.e.*, binary) representation of a program, they are also independent of the languages or tools used to generate the program. In summary, dynamic translation provides a single, well-positioned facility for controlling, managing, or monitoring executing code. As such, dynamic translation is a perfect fit for a host of uses, including profiling, security assurance, dynamic optimization, and bug patching.

Although dynamic translation is obviously an important new technique, the utility of existing translators is severely limited by a lack of integration with the systems in which they are used. Currently, the user is responsible for initiating and programming the translator. This approach is problematic because (i) bug-patching translations should be transparent to all running programs and (ii) security-related translations should not be voluntary. In addition, only a single entity can program a translator, but it is easy to imagine, for example, a scenario in which the operating system wants to patch a hardware bug, the system administrator wants to detect and thwart malware, the user wants to perform profiling, and the applications wants to isolate dynamically loaded modules to protect itself from buggy third-party code. All these parties would benefit from dynamic translation, but existing translators provide no practical way to serve them all.

In this paper we explore the integration of dynamic translation with the operating system to address the above limitations. We describe the architecture and interface for a dynamic translation operating system service (*DTSS*). Via DTSS, the operating system dynamically translates all application programs according to translation specifications (called *transformations*) provided by any entity in the system. Naturally, application code may only transform itself, users may only transform their own applications, and the OS may transform any application in the system. Although the expectation is that dynamic translation will be frequently used in a DTSS system, translation can be disabled when performance is critical.

DTSS is a powerful addition to the operating system. It enables simple implementations of logical OS extensions such as security verifiers, malware detectors, or bug patchers. Because the OS manages translation for the whole system, cross-application optimizations are enabled. For example, if two applications are translated in the same way, the code they share (*e.g.*, shared libraries) need only be translated once. Most importantly, DTSS increases the overall utility of dynamic translation. Operating systems and administrators can leverage system-wide dynamic translation (*e.g.*, for bug patching). They can also use translation to transparently enhance security in potentially vulnerable applications. Finally, DTSS allows multiple parties (*e.g.*, user and OS) to simultaneously transform an application.

Although DTSS is promising, its realization presents a number of challenges beyond those of traditional dynamic translation systems. Adverse performance impact is a significant problem. Although previous research [6, 10, 17, 18, 22] has shown that dynamic translation systems have small overhead, this paper shows that existing techniques have intolerably high overhead in some contexts. DTSS must also manage translations on behalf of multiple entities with different privileges, presenting both semantic design and implementation difficulties. Finally, DTSS must isolate the translations of all parties, so that they may not observe or corrupt each other, while keeping the performance impact at reasonable levels.

This work makes a number of contributions toward solving the above challenges and designing an effective DTSS system. It qualitatively and quantitatively evaluates the limitations of existing dynamic translation systems. It presents an abstract DTSS architecture and discusses some implementation issues. It motivates and presents optimizations that allow DTSS to be more practical. Finally, it evaluates the performance implications of the architecture and optimizations. In future work, we will implement DTSS within the linux operating system and evaluate our implementation.

The outline for the remainder of this paper is as follows. Section 2 gives a brief background on dynamic translation. Section 3 motivates widespread use of dynamic translation as a system service. Section 4 presents the DTSS architecture and Section 5 dis-

cusses some DTSS implementation issues. Finally, Section 6 evaluates the performance of many aspects of a DTSS system.

2. Dynamic Translation

This section gives a brief background on dynamic translation techniques and discusses other system-wide dynamic translation proposals.

2.1 Dynamic Translation Techniques

Dynamic translation techniques can be broken into two general categories: *in-place* techniques and *code-cache* techniques. We discuss both below.

In-place techniques. As the name suggests, in-place techniques maintain the original layout of the program, and patch it locally. Because it is costly to insert a code sequence in the middle of a program, in-place techniques generally use *trampolining*. The translator replaces one or more instructions with a jump to the appropriate code sequence. When the code sequence finishes executing, it jumps to the next instruction in the original program. DynInst [13] is an example of an in-place dynamic translator.

Unfortunately, in-place modification limits the scope of possible translations. While in-place translators can instrument applications (*instrumentation transformations*), they cannot do more advanced types of transformations such as ISA conversions (*ISA transformations*). Furthermore, the runtime overhead associated with the extra control flow can be very high even if the amount of extra work is small. In addition, in-place techniques transform the code *eagerly* rather than *lazily*, meaning they translate all of the code at once rather than when the code is first executed. Code that never gets executed is translated, which adds overhead to the total cost of translation. Because of these drawbacks, we use only code-cache translators, discussed below, in DTSS.

Code-cache techniques. Code-cache dynamic translators fully reconstruct the program layout. They translate the dynamically-executed parts of the program into a code cache, and execute them from there, as opposed to the original binary image. Code-cache techniques also operate lazily. When an untranslated block of code is encountered, the application relinquishes control to the translator, which builds the appropriate block of code, loads it into the code cache, and returns control back to the application at the beginning of the newly translated block.

Most code-cache techniques translate code at the granularity of dynamic traces [6, 10, 17, 22] resulting in a translation unit that encodes dynamic program behavior. Most code-cache translators also link the traces within the code cache (*i.e.*, transform trace exits from jumps to the runtime system into jumps to other entries in the code cache) to minimize transitions between the application and the translator. Any additional functionality that is added to the translated code can often be inlined into the trace, making it easier to optimize. With good code reuse, the cost of translation is amortized over the total execution of the program, *i.e.*, the cost of translation is outweighed by the high performance of the translated code. Under the right circumstances, performance can actually improve over that of the untransformed program. Many code-cache dynamic translators exist today, including DELI [10], DynamoRIO [6], Strata [22], Valgrind [18] (Valgrind compiles basic blocks rather than traces and does not do any linking), and Pin [17].

2.2 System-wide dynamic translation

In addition to describing their dynamic translation infrastructure, Desoli *et al.* [10] also suggest the utility of system-wide dynamic translation. They propose placing the dynamic translator underneath the OS (and all other software), rather than within the OS as a system service. Unfortunately, the OS does not control translation under this configuration. Therefore, the OS cannot leverage translation to enhance security or to patch bugs. Of course, DELI could perform these transformations on its own, but operating systems are generally better equipped to handle security and bug patching. In addition, it is also much more difficult to *selectively* translate applications when the translator is in a separate layer below the OS. The OS naturally knows which applications require translation and how they should be translated. In the DELI approach, it is difficult to distinguish between various executing programs. Consequently, the overall utility of system-wide dynamic translation is limited. In addition, since the translator resides below the OS, the translator must manage hardware virtualization. In any case, the DELI project did not implement, evaluate, or discuss in detail system-wide dynamic translation.

3. Motivation

The virtues of dynamic translation have been discussed in a number of papers [6, 10, 17, 18, 22]. This section does not reiterate them. Instead, it motivates the use of dynamic translation as a system service.

System-controlled translation. The most important virtue of DTSS is that the operating system controls translation. Thus, the OS can apply its own transformations on any program and provide many important services to all applications that require them. For example, one potential use for DTSS is to perform bug patching to compensate for chip design errors (*e.g.*, the infamous Pentium fdiv bug [5]). In this case, the OS adds a transformation to replace the faulty instruction or instruction sequence with a working sequence. With system-controlled translation, bug patching is transparent to the user.

Security hardening is another important application for system-controlled translation. There are a number of transformation techniques that can help prevent buffer overflow and format strings attacks [4, 7, 8, 9, 11, 16, 23]. The problem with the existing dynamic translators is that they require users to decide which applications should be security-hardened. Alternatively, with DTSS, it is the system (or the system administrator) who makes the decision. For example, the administrator could set system-wide policies requiring all application running as root or connected to the network to be buffer-overflow protected.

A third potential application of system-controlled translation is supporting legacy code. In the well-publicized Apple shift from PowerPC to x86 [3], Apple is using dynamic translation (*i.e.*, Rosetta) to support legacy programs. We leave this application of DTSS for future work, as our current proposal does not support ISA transformations.

Hardware abstraction. DTSS provides a mechanism for abstracting aspects of the underlying hardware. For instance, DTSS can provide fine-grained distributed shared memory (*e.g.*, Shasta [21]) without user or application support. Another use of DTSS is as an address translator. For instance, extensible applications often load multiple modules within the same address space. These modules may share data with one another and DTSS can provide address translation to facilitate this sharing.

In addition, many recent processor enhancements ([15, 20, 27]) necessitate profiling. Developer profiling is in many cases inadequate as it is likely performed on a machine with slightly different micro-architectural characteristics and has limited coverage. With DTSS, the operating system can profile applications, transparently and on-the-fly, overcoming these limitations.

Extensibility. An important virtue of DTSS is that developers can implement services on top of it, leveraging DTSS to implement features not provided by the OS. For example, consider dynamic anti-virus or anti-malware detection. The developer simply defines transformations that monitor system calls in search of anomalous behavior. Similarly, DTSS could be used to dramatically simplify the implementation of dynamic software updating [12]. A complex, tedious, and machine-specific aspect of performing dynamic soft-

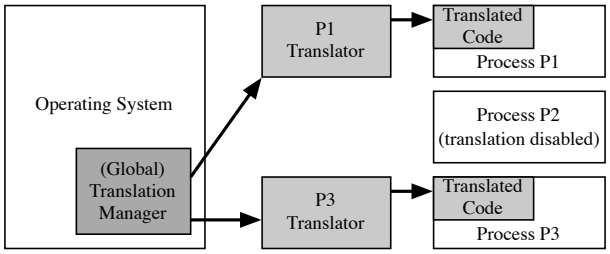

Figure 1. A diagram of the DTSS system architecture.

ware updates arises from transforming live program binaries. With DTSS, all of this aspect of the problem is shifted to the system service.

Optimization opportunities. Another motivation for DTSS is that it provides opportunities for optimizing translated code. One important optimization with dynamic translation is caching the translated code. However, with conventional dynamic translators, translated code cannot be shared across multiple users. In contrast, with DTSS, users can share dynamically-linked library code as well as whole programs, since translation is managed by a trusted entity (*i.e.*, OS). As shown in Section 6, caching significantly improves the performance, especially for short-running programs with poor code reuse.

There are other optimization opportunities as well in DTSS. For instance, DTSS makes it possible to inline a customized version of certain system calls [19]. DTSS systems can also profile applications and feed this information back to the dynamic translator in order to further optimize translation. Per-application dynamic translators could also perform this optimization, however, with DTSS it can be implemented across multiple users.

Centralized transformations. Another virtue of DTSS systems is that they have a centralized repository for commonly-used transformations. These transformations are accessible by users or applications on request. For example, many extensible applications require fault isolation [26] to prevent untrusted code from writing or jumping to arbitrary regions of memory. In DTSS, the application can request fault isolation via a system call for a certain region of code (*i.e.*, the untrusted code) without needing to build a transformation. Centralized transformations have an additional benefit: changes to them are localized to one place in the system. This benefit is especially important for fast-changing areas such as security. Instrumentation transformations designed to enhance security, may need to adapt as new attacks are discovered. With centralized security transformations, if new security vulnerabilities are announced, only one set of transformations needs to be updated.

4. Architecture

This section describes the DTSS architecture. First, it shows the basic design, then it looks at the DTSS system components, and finally, it discusses the API for programming DTSS.

4.1 Overview

In DTSS, translation is integrated within the operating system and provided as a service to users. DTSS exports an API through which users can program the translator to transform programs on their behalf, in arbitrary ways. Moreover, multiple parties can transform a single application in DTSS. For instance, a user and the OS can simultaneously transform a single application. The DTSS architecture ensures that all transformations are correctly applied to the application. At the same time, it prevents less-privileged transformations (*e.g.*, those submitted by a user) from hijacking higher-privileged transformations (*e.g.*, those submitted by the OS).

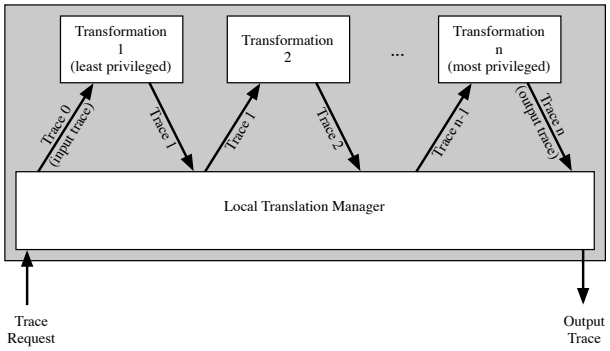

Figure 2. DTSS Translator.

Transforming programs. The unit of translation in DTSS is a trace. At runtime the DTSS translator generates traces for all executed code. Trace construction is not programmable by DTSS users. However, DTSS users can transform these traces in arbitrary ways. A *transformation* is effectively, a C program with DTSS API support, which takes as input a trace and outputs a newly transformed trace. DTSS supports instrumentation transformations. In future work, we will explore support for ISA transformations.

In this work, we only consider transformations on user-level applications. As future work, we will investigate dynamically translating some of the extensible components of the operating system such as device drivers. If device drivers, which run in kernel mode, contain bugs they can crash the entire system. In the future, we will explore fault isolating them using techniques such as those employed in Nooks [25].

DTSS users. In DTSS, multiple parties (four in total) can define transformations for the same application. Clearly, the user is one party. The application, itself, can also define a transformation. For instance, an extensible application can fault isolate untrusted code on its own without user support. An administrator can also define a transformation. An administrator might define a security transformation to be applied to certain applications (*e.g.*, webserver software). Finally, an OS can also add transformations. An OS might add a transformation to patch a hardware bug. Both an administrator and an OS can define a transformation that is applied across all applications. For instance, an administrator might define a profiling transformation that is applied to all applications running on the system.

Composition. When multiple transformations are defined for the same application, transformations are *composed* as follows. Transformations submitted by various parties are *ranked* in strict order, based on the privilege level of the party that submitted the transformation. The OS is the highest privileged party, followed by the administrator, followed by the user, followed by the application. Higher-privileged transformations, such as those submitted by the OS, are applied after lower-privileged transformations, such as those submitted by the user. Otherwise, a user could thwart the operating system by submitting a transformation to undo an OS transformation. Higher-privileged transformations are also applied to lower-privileged transformation code. This requirement prevents users from circumventing OS transformations by placing the offending code in a transformation.

4.2 System Components

Figure 1 shows a diagram of the DTSS system architecture. At the highest level there are two main components: the *translation manager* and a *per-application translator* (although the translator can be broken-down further into finer components).

Translation manager. In the DTSS architecture, a privileged translation manager (sometimes called the global translation manager)

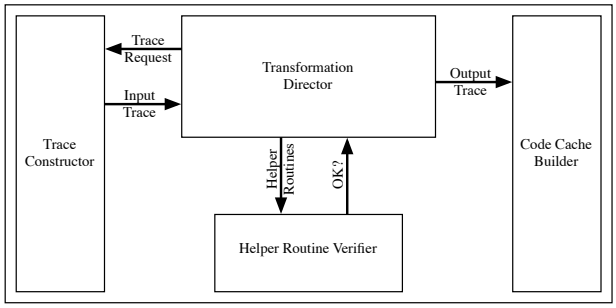

Figure 3. DTSS local translation manager.

within the operating system oversees all translation. Essentially, the translation manager is the component that provides dynamic translation as a system service. The translation manager sets up translation for any application that needs it (*e.g.*, process P1 in Figure 1). The translation manager is also responsible for transferring control from a process to its translator (and back) whenever the process requests an untranslated code cache entry. The translation manager also needs to be notified at a call to fork and exec[1]. If a translated process calls fork or exec, the new process must also be translated using the exact same transformations.

In addition, the translation manager has one other responsibility: protecting the translated code from corruption by the application. For instance, a malicious application could corrupt the code cache. To protect the translated code, the translation manager sets all code cache pages to execute-only. Applications could also attempt to change the permissions of these pages via the mprotect system call. Therefore, the translation manager must also observe all mprotect calls and verify that all requests are for pages that do not contain translated code.

Per-application translator. In a DTSS architecture, translation is not performed directly by the translation manager, because a malicious user-defined transformation could hijack the translation manager, and thus, the OS. Instead, translation is done in a separate process running at the privilege level of the application. A translator is created for each application in which translation is enabled. Unlike other dynamic translators [6, 10, 17, 18, 22], the translation mechanism and the application are isolated from one another. This separation prevents the application from intentionally or accidentally corrupting translation, some of which may have been defined by a privileged party (*e.g.*, OS). The translator and application may run in separate processes or can even be isolated within the same address space (see the next section), depending on the implementation.

Figure 2 shows a closer view of the per-application translator mechanism. The translator, itself, is composed of multiple components. First, the translator contains *transformations*, one for each submitted transformation. For instance, if a user submitted a security transformation, one component would be responsible for performing this transformation. In addition, the translator contains a *local translation manager*, which manages translation for that particular application. Less privileged transformations should not be able to corrupt higher privileged transformations. As a result, each must be separated from one another as well as from the local translation manager.

Transformations. A transformation takes as input a trace and outputs the corresponding transformed trace. As depicted in Figure 2, multiple transformations are applied to application traces in order of ranking (discussed above). The input trace to transformation x is the output trace from transformation $x - 1$. The implication of this configuration is that the later transformations observe the translated code from earlier transformations, but not vice versa. For this reason, the least privileged transformations should be applied first, while the most privileged should be applied last.

[1] Although this paper is mainly OS independent, it assumes unix-based system calls (*e.g.*, fork and exec).

```
// Called before untrusted memory writes.
void mfi(unsigned int addr) {
  if (addr < segment_start_addr || addr >= segment_end_addr) {
    fprintf(stderr, "Error: write to address outside of allotted segment.");
    exit(-1);
  }
}

// Called whenever a trace is translated
void instrument_trace(trace_t trace) {
  int i;
  // Loop through trace instructions
  for (i = 0; i < trace.inst_num; i++) {
    // Is this instruction an untrusted store?
    if (is_mem_write(trace.inst_list[i]) &&
        get_pc(trace.inst_list[i]) >= untrusted_start_pc &&
        get_pc(trace.inst_list[i]) < untrusted_end_pc) {
      // Add predicate and body calls
      add_analysis(&mfi, INSTRUMENT_BEFORE,
                   ARGTYPE_ADDR, INST_MEM_ADDR);
    }
  }
}

// Called once during initialization
void transform() {
  // Register callback function instrument_trace to be called
  // whenever a new trace is translated.
  reg_callback_trace(&instrument_trace);
  // Disallow library calls to the translator while executing
  // untrusted code.
  disallow_transformation_calls(untrusted_start_pc, untrusted_end_pc);
}
```

Figure 4. Instrumenting a program with memory fault isolation.

Transformations run at the privilege level of the party that submitted the transformation, allowing transformations to, for example, manipulate files managed by the submitter. OS and administrator transformations run at the root privilege level, while user transformations run at that particular user's privilege level. The only exception to this rule are transformations submitted by the application. The application's transformations should not be able to hijack the user who is running them. These transformations run at the privilege level of the user running the application, however, they are not allowed to make any system calls. Any system call that is made is caught by the global translation manager.

Transformations often insert function calls into the transformed program, allowing for the insertion of significant computation without unduly impacting the program's memory footprint. These helper functions are treated differently than application code in order to reduce transitions to the local translation manager. First, calls to helper routines are not treated as trace exit points unlike other program calls. As a result, the output of a transformation is generally still a valid trace, but we must ensure the helper routines are already translated. To achieve this, helper routines themselves are statically translated (by transformations with higher privilege) at load time before they are first called. Since runtime information is unavailable at this point, the trace constructor simply generates a new trace for each basic block and links them together. In order to ensure that helper routines do not transfer control to untranslated code, all control flow must be statically apparent (enabling complete linking). As a result, indirect jumps and function calls (the destination of returns are not statically apparent in general) are not allowed in helper routines.

Local translation manager. The local translation manager supervises translation for one particular application. The components of the local translation manager (Figure 3) do not need to be isolated from one another since each is a trusted part of the translation infrastructure.

Trace transformations are facilitated by the *transformation director*. The transformation director requests traces from the *trace constructor*. As is apparent in Figures 2 and 3, the transformation director then takes each trace and passes it to the first transformation component, which performs its transformation. Next, the transformation director takes this newly transformed trace and passes it to the second transformation component, and so forth. After the last transformation, the transformation director gives the final trace to the *code cache builder*. The code cache builder instructs the global translation manager to insert the trace into the translated application's code cache (linking as necessary).

The *helper routine verifier* confirms that all helper routines, supplied by each transformation, does not subvert the translator. Basically, the helper routine verifier inspects all control flow within a helper routine. Control flow within these routines could potentially circumvent other higher-privileged helper routines. The helper routine verifier first checks that all control flow within a helper routine is statically apparent. The verifier then checks that all control flow branches to an appropriate target.

4.3 API

The DTSS application programming interface (API) is mainly borrowed from previous translators such as Pin [17] and Atom [24]. To program DTSS, a user writes a C program, which when compiled is linked with the DTSS library. Within the program, the DTSS user defines a **transform** function, where the user can register callback functions to be triggered on particular events, *e.g.*, any time a new trace is translated. Applications can also directly program DTSS. Such applications are compiled using a new library, which allows them to call into the global translation manager at runtime. Applications can either pass a pointer to transformation code contained within the application's address space or it can specify a separate file containing the code.

Figure 4 shows an example (instrumenting) transformation. The code in Figure 4 fault isolates [26] some untrusted region of code (*e.g.*, a module downloaded from the Internet) in an extensible application. First, a function **instrument_trace** is registered, which is called whenever a new trace is translated. **instrument_trace** inspects each translated trace, looking for memory writes. A function call is inserted before (hence **INSTRUMENT_BEFORE**) each write. The call is to a function, **mfi**, that triggers an error on all writes outside of some pre-defined segment. Unlike most dynamic translation APIs, DTSS allows applications to add transformation rules. To disable parts of the application (in this case the untrusted module) from altering transformation rules, a more privileged entity (*e.g.*, user, administer, OS) can call **disallow_transformation_calls**.

5. Implementation Issues

Although we save implementing DTSS for future work, this section describes two important implementation issues: caching translated code and isolating transformations.

5.1 Caching and Reusing Translated Code

Although a number of papers [6, 10, 17, 18, 22] have shown that dynamic translation has low overhead, these papers have ignored a large class of applications. They have evaluated only long-running applications, which have good code reuse. However, as is shown in the next section, the overhead of translation on short-running programs is much higher, because execution time is dominated by the time it takes to warm up the code cache. Of course, users are less sensitive to overhead on a short-running programs; but this overhead can harm interactive programs. For example, if the performance of the **ls** command goes from a few milliseconds to a few seconds, DTSS becomes far less useful.

The expectation in DTSS is that all (or nearly all) programs are translated. For this reason, the overhead of translation on short-running programs in DTSS is important. Fortunately, caching and reusing translated code significantly improves the performance of dynamic translation for short-running programs. Moreover, the DTSS architecture enables caching across users, since translation is managed by the OS. Of course, translated code cannot be shared when using different transformations. A copy of the translated code needs to be cached for each distinct set of transformations. However, in practice only a few distinct sets of transformations are used for any single application. In addition, DTSS does not allow translated code to be cached and reused when transformations are dynamically submitted.

5.2 Isolating Transformations

As discussed in the Section 4, many of the DTSS components need to be isolated from one another (see the previous section). For example, the translator must be isolated from the (untrusted) application and transformation code. Below, two isolation techniques are described.

Straightforward approach: separate address spaces. The natural approach to providing isolation is to put the application, the local translation manager, and each transformation in separate processes. In order to minimize scheduling latencies between these processes, the process manager treats them all as one scheduling unit. Communication from these separate processes occurs through the OS. Components communicate with one another via system calls, which are serviced by the global translation manager. For instance, a jump to untranslated code in an application is implemented as a trap to the OS. The translation manager is then notified and immediately wakes up the local translation manager. The local translation manager executes a system call when it needs a transformation process to translate some trace. The global translation manager then immediately wakes up the appropriate transformation process. When the transformation process is done, it also makes a system call to notify the global translation manager.

The virtue of this implementation is that isolation is ensured through conventional address space protection. However, process crossings (*e.g.*, when control is transferred from the main process to the local translation manager in order to add a new trace to the code cache) become very expensive. The performance of applications that require frequent process crossing will suffer dramatically. Fortunately, most applications do not require frequent crossings. As we will see in Section 6, the additional overhead of isolation via separate processes is surprisingly small.

Optimized approach: single address space. In some contexts, applications may require frequent transitions to and from the translator. To this end, the second approach houses the application and the translator (including the transformations) in a single address space. This optimization eliminates context switches and reduces the number of processes in the system.

To isolate the various components from one another, DTSS leverages hardware support. Protection is achieved via virtual memory segmentation, which exists in many of today's processors (*e.g.*, x86, PowerPC, and PA-RISC) [14]. Virtual memory segmentation provides a level of indirection in converting addresses to physical addresses. In a segmentation processor, instructions reference memory via effective addresses. Effective addresses are then converted to virtual addresses using segments, which are located either in registers or memory. These virtual addresses are then converted to physical addresses. By modifying the segments, portions of code are allowed and disallowed from referencing particular regions of memory. DTSS requires user-level updates of the segments (available only in PA-RISC). Essentially, the translator can update the segments to manipulate which components can reference which virtual pages. In addition to user-level updates of the segments, updates should be statically apparent. Otherwise malicious application or transformations could corrupt the segments and thus higher privileged transformations.

Input Type	Benchmark	Baseline Runtime (seconds)	Instructions Executed (billions)	Traces Executed (billions)	Instructions Translated	Traces Translated	Transitions to the Translator
Test	bzip2	10.97	18.32	1.86	23,727 (1294.7)	1743 (95.1)	2569 (140.2)
	gcc	1.35	1.48	.17	366,981 (247,692.7)	26,610 (17,960.3)	35,998 (24,296.7)
	mcf	.36	.26	.03	28,143 (108,002.3)	1699 (6520.1)	2571 (9866.5)
	twolf	.52	.37	.03	63,162 (170,521.3)	4651 (12,556.5)	6140 (16,576.4)
Train	bzip2	88.67	105.70	10.29	24,455 (231.4)	1804 (17.1)	2646 (25.0)
	gcc	3.00	3.36	.38	358,267 (106,656.3)	25,856 (7697.3)	35,089 (10,446.0)
	mcf	39.05	15.51	2.31	28,267 (1822.9)	1714 (110.5)	2605 (168.0)
	twolf	21.51	18.80	2.09	66,477 (3536.0)	4920 (261.7)	6461 (343.7)
Ref	bzip2	140.31	147.21	14.95	24,211 (164.5)	1789 (12.1)	2625 (17.8)
	gcc	32.69	20.94	2.51	374,066 (17,859.0)	27,011 (1289.6)	36,633 (1749.0)
	mcf	362.50	108.87	17.58	28,245 (259.4)	1713 (15.7)	2604 (23.9)
	twolf	757.29	510.47	56.32	66,174 (129.6)	4911 (9.6)	6444 (12.6)

Table 1. Benchmark characteristics. The values in parentheses are normalized to instructions executed times 10^9.

6. Evaluation

This section presents performance characteristics of a DTSS system. It shows that the overhead of dynamic translation is low in many scenarios, confirming previous studies [6, 10, 17, 18, 22]. This section also shows that in some contexts the overhead is intolerably high and it demonstrates that this overhead can be significantly reduced via caching. Finally, this section explores the impact of dynamic translation on system throughput and latency in server applications.

6.1 Methodology

Dynamic translator. In this work, we use the Pin toolset [17] as our dynamic translator in DTSS. To define a transformation in Pin, a user writes a program in C++ called a Pintool (which is effectively a transformation). Pin is a single-program translator, which runs in the same address space as the process.

To model caching, we run the program back-to-back in Pin (in the same process). The second run represents the cached version. To model a separate process implementation (see the previous section), we run Pin normally, conservatively adding the overhead from switching between multiple processes. To obtain this overhead we multiply the number of necessary context switches (computed from Pin statistics) by the context switch overhead. For the context switch overhead, we use a conservative approximation of 50,000 cycles. Our default system uses caching and the separate process implementation.

Machine and operating system. We ran our experiments on a Intel Xeon Pentium IV running at 2 GHz with 512 KB L2 cache. The operating system is Linux, kernel version 2.6.8.

Benchmarks. To evaluate translation overhead (Sections 6.2 and 6.3) we use a subset of the SPEC2000 integer benchmark suite. We use all three data input sizes: test, train, and ref. Table 1 shows the benchmarks as well as some important characteristics such as the number of traces translated. The baseline runtime (in seconds) is the execution time of the benchmark running natively. The other characteristics were extracted from a run of Pin without any additional instrumentation.

To evaluate system throughput and latency (Section 6.4) we use two webserver benchmarks, based on Apache 2.0.54 [1] and Postgres 7.4 [2].

6.2 Translation overhead

Figure 5 shows the overhead of running Pin with no transformation for the SPEC2000 benchmarks. Each bar represents the execution time of Pin versus the native execution time running the same benchmark and the same input. In most cases, the overhead is quite low, with overheads less than 15%. No slowdown was greater than 3.1. Because Pin constructs highly efficient traces, the cost of translation is amortized by the higher performance of the generated code. For *bzip2*, *mcf*, and *twolf* there is almost no overhead. *gcc* has a higher overhead than the other three benchmarks due primarily to poor code reuse. From Table 1, the ratio of traces (as well as instructions) translated per instructions executed is higher for *gcc* than for the other three benchmarks. As the input size increases on *gcc*, the ratio decreases and as a result the overhead drops significantly.

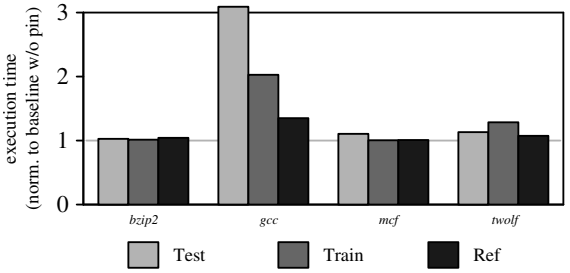

Figure 5. Overhead of dynamic translation using Pin.

Transformations. Figure 6 demonstrates the utility of dynamic translation, showing the performance of Pin for various transformations. The first set of bars in Figure 6 plots the performance of Pin with no transformation (for comparison purposes). This is followed by a bug patching transformation (*bp*) to prevent Intel Pentium fdiv bug [5]. The transformation represented by the third set of bars prevents stack smashing attacks by using a shadow stack (*ss*) [4, 7, 11] to verify that return addresses are not corrupted. The transformation represented by the fourth set of bars performs memory fault isolation (*mfi*) [26] for memory writes only (as shown in Section 4). The next set of bars shows a transformation that implements a debugging watchpoint (*db*). Memory writes to a particular address are monitored. The next set of bars shows a transformation performing profiling, counting how often each basic block is executed (*bb*). The final set of bars (*all*) represents the composition of all the transformations.

Figure 6 shows that DTSS has many uses. Although overheads get high for some transformations and some benchmarks, it is important to remember that, in these cases, significantly more work is being done. When performance is critical, DTSS users can disable the more expensive transformations (*e.g.*, *bb*).

6.3 Sensitivity Analysis

Here we examine the impact of two aspects of DTSS implementation: caching and reusing translated code, and hardware support for isolating transformations.

Caching and reusing translated code. The previous graphs have all leveraged caching, *i.e.*, the code cache is filled at load time with traces generated from a previous run. Figure 7 shows the impact

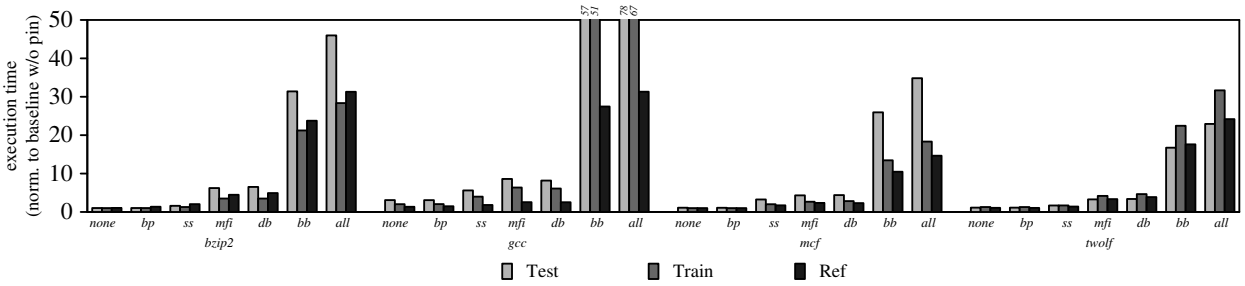

Figure 6. Overhead of various transformations in Pin.

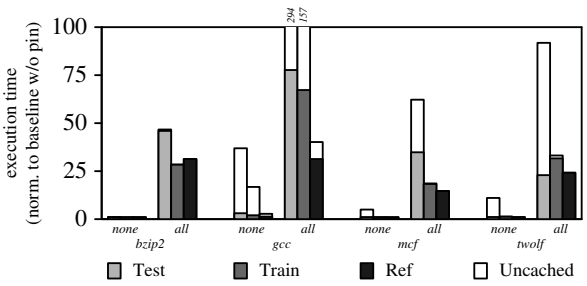

Figure 7. Performance impact due to caching.

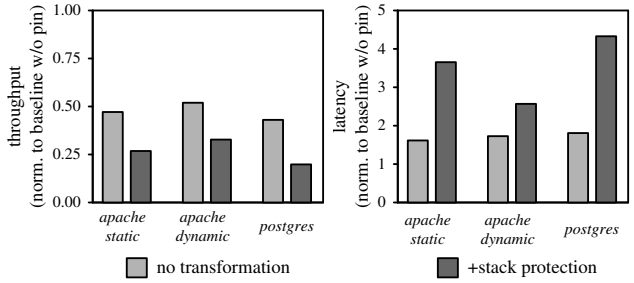

Figure 8. Impact of translation on throughput and latency for Apache and Postgres.

of caching with no transformation and with all the transformations (*none* and *all* from Figure 6). As is evident in Figure 7, caching is critical in DTSS. Without caching, benchmarks with shorter execution times (*i.e.*, test inputs) tend to have much higher overheads. For short-running programs there is less opportunity to amortize the cost of translation with the efficiently generated code. Although the performance of short-running programs is less critical than long-running programs, in some contexts short-running programs are performance sensitive. For example, we have observed the running time of *ls* degrade from a few milliseconds (without Pin) to a few seconds (with Pin). Furthermore, in DTSS the expectation is that translation is always on. Therefore, optimizing the performance for short-running programs is important.

Fortunately, caching significantly improves performance, especially for short-running programs. With no transformation, caching reduces the overhead by as much as a factor of 12 (*e.g.*, *gcc-test*). On *all*, caching reduces the overhead by as much as a factor of 4 (*e.g.*, *twolf-test*). Notice that the gap between the cached and uncached bars in Figure 7 shrinks as the input size increases. However, for programs with larger code sizes (*e.g.*, *gcc* and *twolf*), caching can still greatly improve performance even for the *ref* inputs.

Hardware support for isolation. In DTSS, since some transformations are submitted by unprivileged parties (*e.g.*, a user), transformations are isolated from one another as well as from the local translation manager. Transformation are either housed within their own process or are isolated using hardware support. Thus far, we have assumed a single address space approach. Surprisingly, in our experiments (not shown), we find that the additional overhead of multi-process isolating is minimal. Transitions to and from the local translation manager are relatively rare in our benchmarks, and as a result the cost in additional context switches is small. With caching the difference in overhead is only a few percent. Without caching the cost is slightly higher, but still rarely more than 5% for most benchmarks. Therefore, efficient isolation techniques are not critical.

6.4 Server Throughput and Latency

Thus far, we have examined translation overhead only on single-threaded interactive benchmarks, but DTSS is also desirable for multi-threaded server applications. We investigate how Pin affects the average throughput and latencies for three such workloads. The first two are based on the Apache webserver. Apache is configured to run in pre-fork mode and to serve either a static html file or a simple PHP script. The third benchmark is based on the Postgres database. Postgres is configured with a 100K-entry database. The benchmark performs three-way join queries with randomized inputs. For all three benchmarks the number of clients was determined by finding the saturation point for the baseline configuration (without Pin).

Figure 8 shows the degradation in average throughput and latency when using Pin. For both webserver configurations, Pin with no transformation degrades throughput by a factor of two. The main cause is the overhead of translation. While server workloads can hide these overheads through warm-up, unfortunately, Apache periodically kills and respawns worker threads, reducing this benefit. Tweaking this behavior will result in lower overhead. Finally, the database benchmark incurs a throughput reduction by a factor of 2.5. The reason is, however, *not* the overhead of translation (as there is no killing/respawning effect), but instead a very high number of indirect branches that Pin cannot handle efficiently, causing a significant increase in the dynamic instruction count.

In all three server benchmarks, adding the shadow stack transformation reduces the throughput by another factor of two. This result is similar to the slowdowns observed in Figure 6.

Figure 8 demonstrates that existing dynamic translation techniques are inadequate for server workloads where processes are often killed and respawned and applications frequently execute indirect jumps. Caching can help mitigate this cost, but challenges still remain in lowering this overhead.

7. Conclusions

We have proposed providing dynamic translation as a system service (DTSS) managed by the operating system. We have shown DTSS to have considerable value versus user-managed dynamic translation, but its implementation and performance implications present a number of challenges and unknowns. We have presented a DTSS architecture that supports the specification of transformations by multiple parties of varying privileges, and we have described implementation alternatives to achieve transformation isolation. In

our experimental evaluation, we have found that short running programs can suffer from intolerable overhead, but cross-invocation code caching assuages the problem. In addition, we found that server applications perform slightly worse in DTSS than interactive workloads, due to more frequent transitions to the transformation manager. Nevertheless, we conclude that DTSS is functionally very useful and practically promising when code caching is performed. In future work, we will implement DTSS within the linux operating system.

Acknowledgments

We thank the anonymous reviewers for their suggestions. This work was funded in part by NSF grant CCR-0311199. E Christopher Lewis is supported by NSF Career Award CCF-0347290.

References

[1] Apache http server project. http://httpd.apache.org.

[2] Postgresql database server. http://www.postgresql.org.

[3] Apple Corporation. *Universal Binary Programming Guidelines, Second Edition: Rosetta*. http://developer.apple.com/documentation/MacOSX/Conceptual/universal_binary/universal_binary_exec_a/chapter_7_section_1.html.

[4] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proc. of USENIX Annual Technical Conference*, Jun. 2000.

[5] M. Blum and H. Wasserman. Reflections on the pentium division bug. *IEEE Transactions on Computers*, 45(4):385–393, 1996.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. of Intl. Symp. on Code Generation and Optimization*, pages 265–275, Mar. 2003.

[7] T.-C. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proc. of 21st Intl. Conf. on Distributed Computing Systems*, Apr. 2001.

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proc. of 12th USENIX Security Symposium*, pages 91–104, Aug. 2003.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention buffer overflow attacks. In *Proc. of 7th USENIX Security Conference*, pages 63–78, Jan. 1998.

[10] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: a new run-time control point. In *Proc. of 35th Intl. Symp. on Microarchitecture*, pages 257–268, Nov. 2002.

[11] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proc. of 10th USENIX Security Symposium*, pages 55–66, Aug. 2001.

[12] M. Hicks and S. M. Nettles. Dynamic software updating. *ACM Trans. on Programming Languages and Systems*, Sep. 2005.

[13] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proc. of Scalable High Performance Computing Conf.*, pages 841–850, May 1994.

[14] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, 1998.

[15] I. Kim and M. H. Lipasti. Macro-op scheduling: Relaxing scheduling loop constraints. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 277, Washington, DC, USA, 2003. IEEE Computer Society.

[16] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proc. of 11th USENIX Security Symposium*, pages 191–206, Aug. 2002.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of Conf. on Programming Language Design and Implementation*, Jun. 2005.

[18] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proc. of Workshop on Runtime Verification*, Jul. 2003.

[19] C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[20] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proc. of 7th Intl. Symp. on High-Performance Computer Architecture*, Jan. 2001.

[21] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[22] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. In *Proc. of Workshop on Binary Translation*, Jul. 2001.

[23] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Proc. of Annual Computer Security Application Conf.*, pages 209–218, Dec. 2002.

[24] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proc. of 1994 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Jun. 1994.

[25] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. of 19th ACM Symp. on Operating Systems Principles*, Oct. 2003.

[26] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of 14th ACM Symp. on Operating Systems Principles*, Dec. 1993.

[27] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2001. ACM Press.