Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications

Vijay Janapa Reddi
Harvard University
vj@eecs.harvard.edu

Dan Connors
University of Colorado
dconnors@colorado.edu

Robert Cohn
Intel Corporation
robert.s.cohn@intel.com

Michael D. Smith
Harvard University
smith@eecs.harvard.edu

Abstract

Run-time compilation systems are challenged with the task of translating a program's instruction stream while maintaining low overhead. While software managed code caches are utilized to amortize translation costs, they are ineffective for programs with short run times or large amounts of cold code. Such program characteristics are prevalent in real-life computing environments, ranging from Graphical User Interface (GUI) programs to large-scale applications such as database management systems. Persistent code caching addresses these issues. It is described and evaluated in an industry-strength dynamic binary instrumentation system – Pin. The proposed approach improves the intra-execution model of code reuse by storing and reusing translations across executions, thereby achieving inter-execution persistence. Dynamically linked programs leverage inter-application persistence by using persistent translations of library code generated by other programs. New translations discovered across executions are automatically accumulated into the persistent code caches, thereby improving performance over time. Inter-execution persistence improves the performance of GUI applications by nearly 90%, while inter-application persistence achieves a 59% improvement. In more specialized uses, the SPEC2K INT benchmark suite experiences a 26% improvement under dynamic binary instrumentation. Finally, a 400% speedup is achieved in translating the Oracle database in a regression testing environment.

1. Introduction

Run-time compilation systems alter the dynamic instruction stream of an application during execution. Functioning independently of tool-chain dependencies allows run-time compilers to provide a variety of powerful services while operating on existing, unmodified binaries. Dynamic optimization systems [1, 4, 6, 10] recognize optimization opportunities and apply them at run time. Dynamic binary translators [2, 9, 12, 35] facilitate the execution of binaries compiled for one instruction set architecture (ISA) on a different ISA, while dynamic instrumentation [21, 22, 25, 31] enables researchers and software developers to study applications without recompilation. New methods of employing security [11, 18] and fault tolerance [3] are also being implemented using these systems.

While run-time compilation systems open up many exciting possibilities, their full potential is governed by strict constraints on increasing program overhead. *Virtual Machine (VM) overhead* arises from spending time within the infrastructure either emulating system functions, or translating new application code. The latter is significantly more dominant. In addition, *translated code overhead* is observed while executing the dynamically compiled application code. Even in the absence of providing any new service, performance of translated code incurs execution overhead as the VM alters the dynamic instruction stream to maintain control of the program's execution. This paper focuses on reducing VM overhead, specifically on the costs associated with translating application code.

Run-time compilation systems are effective in minimizing the VM overhead of hot/frequently executed code, but the overhead remains significant for infrequently executed (or *cold*) code. Cold code, in the context of a run-time compiler, is code whose translation cost is not amortizable by repeated execution during a program's lifetime. Applications exhibiting cold code behavior are prevalent in everyday computing environments ranging from shell programs to *Graphical User Interface (GUI)* and enterprise-scale applications. Pin [21], a state-of-the-art dynamic instrumentation system, causes large slowdowns even prior to injecting instrumentation code for many small, as well as large, *GUI*

programs. Large enterprise-scale applications like the *Oracle* database incur several orders of slowdown, directly due to VM overhead associated with compiling cold code.

Performance degradation effects of cold code are important in run-time compilation systems providing translation, security, or instrumentation services. These services run programs completely under the system's control, requiring every instruction to be dynamically interpreted/translated. Thus, run-time compilers must overcome cold code overheads, unlike same-ISA dynamic optimizers resorting to original program execution in the presence of cold code [1, 20]. Furthermore, transformations applied towards providing a service (e.g. instrumentation) only increase the overhead associated with cold code.

To overcome translation costs, run-time compilation systems manage software-based code caches to avoid repeated translations of the same code. While effective for frequently executed code, the benefits are small for cold code due to its limited reuse. Some existing compilation systems resort to interpretation [4] or less aggressive translation techniques [2] in an attempt to reduce cold code translation overheads.

The approach discussed in this paper is based on the observation that cold code within an execution is often hot code across multiple executions. Run-time compilation systems can exploit this *inter-execution persistence* by generating *persistent code caches*, and using them across subsequent program invocations. Applications with high code sharing benefit from improved execution time by reusing a single persistent cache. Performance of applications with low code sharing is improved over time by accumulating new code discovered across executions into the persistent code cache. The proposed model of code reuse is further extendable by leveraging *inter-application persistence*, which exists in the form of common library dependencies between programs (i.e. translations of library code generated by one application are reusable by another application).

Persistent code caching is evaluated in Pin. Experimental results are discussed across different types of workloads: the *SPEC2K INT* benchmark suite, *GUI* applications, and the *Oracle* database. Overall, inter-execution persistence improves the performance of *GUI* applications by nearly 90%, and inter-application persistence improves their performance by 59%. Under dynamic instrumentation, performance of the *SPEC2K INT* suite is improved by 26%. Finally, exploiting the multi-process execution model of the *Oracle* database yields a 400% improvement in a regression test setting.

The paper is structured as follows: Section 2 introduces the evaluation framework, discusses VM overhead, and identifies when VM overhead is detrimental to performance. Section 3 explains the exploitation of persistent application characteristics to improve performance and presents a working system. Section 4 evaluates the benefits of persistent caching in Pin. Section 5 addresses prior work, and Section 6 summarizes the paper.

2. Run-time Compilation Infrastructure

Run-time compiler designs vary based upon the goals of a system. Nevertheless, certain components are fundamental to all. This section presents background into these components and discusses the associated VM overhead in the context of Pin [21], the evaluation framework.

2.1. Pin Overview

Pin is a dynamic binary instrumentation engine. It is supported on the IA32, EM64T, ARM, and IPF platforms under Windows, Linux, MacOS and FreeBSD operating systems. Pin's components are illustrated in Figure 1. Pin exports an instrumentation interface (*Client API*) to support the writing of *Pin Tools* (*Client*). Its core internal components are the *Emulation unit*, *Compilation unit*, and *Dispatcher*.

The compilation unit compiles/translates application code into code units called *traces*. A trace, in Pin's context, is a linear sequence of instructions fetched from a starting address until a fixed instruction count is reached or an unconditional branch instruction is encountered. Execution always enters a trace via its first instruction; no side-entrances are allowed. The fetched instruction layout of the trace is not altered, nor are any optimizations performed on it as Pin does not attempt original program optimization. Pin only optimizes the instrumentation code it generates.

Once a trace is compiled, with or without instrumentation, it is placed in the *code cache* and a *translation map* is updated. The translation map maintains information pertaining to code within the cache. For example, given an original instruction address, the map returns its code cache address. After updating the map, translated branch instructions with targets corresponding to the compiled trace are *linked* together. Hence, subsequent executions of the same code require no re-translation and control remains in the code cache. Control transfers back to the VM only when code needs to be generated, or the emulation of a system call is required. The emulation unit handles the latter to ensure proper program execution.

2.2. Motivation

VM overhead is the time spent within the virtual machine translating application code or emulating system functions. The former dominates the time spent within the VM. Through the rest of the paper, VM overhead measures only the cost of dynamically generating application code.

Current run-time compilation systems implement an *intra-execution* code cache to tackle VM overhead. To un-

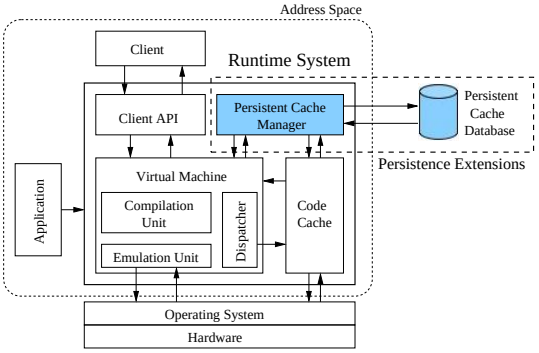
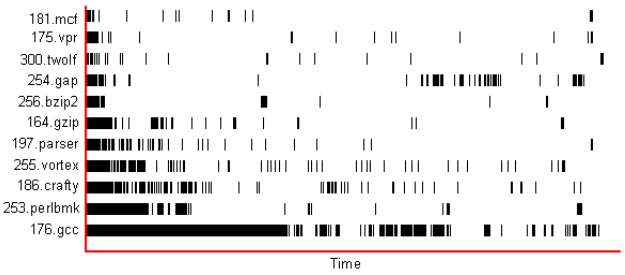
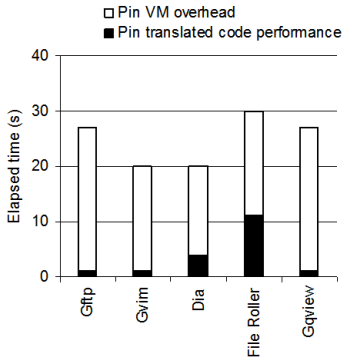


Figure 1. Pin's run-time compilation framework. Persistence extensions (shaded) are discussed in Section 3.2.



(a) SPEC2K behavior using Reference inputs under Pin.



(b) GUI overhead breakdown.

Figure 2. Behavior and performance observations under Pin (assuming an unbounded code cache).

derstand its effectiveness, Figure 2(a) shows the behavior of the *SPEC2K INT* benchmarks under Pin without instrumentation. Only the first *Reference* input is used for programs with multiple inputs. Vertical lines on the graph represent VM translation requests. The rightmost vertical line indicates the end of program execution. White space in-between the black lines indicates translated application execution within the code cache.

According to Figure 2(a), VM translation requests occur frequently at program startup as new code is discovered. Much of the translated code at the beginning of execution corresponds to program initialization and is typically cold code (e.g. initialization of the run-time loader). As the frequently executed code is generated, the number of translation requests drops because more time is spent executing the translated application code. All *SPEC2K INT* benchmarks, except *176.gcc*, fit this profile. Benchmark *176.gcc* is an outlier. Its footprint is not captured even towards program completion. Over 60% of its execution time (substantial number of vertical lines on the graph) is spent generating code, which is not reused enough to amortize VM overhead.

The issue of an application's footprint not being present in the code cache is mostly prevalent at the beginning of program execution. While this is observed in the *SPEC2K INT* suite, it is often insignificant relative to the overall execution time. However, everyday desktop applications prove more challenging. To illustrate this, *GUI* applications presented in Table 1 are discussed below.

The minimum wait time, under Pin, for a *GUI* program to be in full-effect (i.e. buttons, menus, etc.) for user interaction is shown in Figure 2(b). The startup times are between 20x-100x slower than without Pin. The large overheads are not due to any inefficiencies in the design or implementation of Pin, but are the result of significant cold code execution during program startup.

Time spent executing the translated application code (*Pin translated code performance*) is much smaller than the time spent generating traces (*Pin VM overhead*) for all *GUI* programs except *File-Roller*. *File-Roller* replaces the operating system's signal handlers with its own, which requires Pin to intercept and emulate signals on the program's behalf. Signal emulation is an expensive mechanism, thereby resulting in *File-Roller's* poor translated code performance.

Upon completion of the startup phase, VM overhead drops substantially (not shown). Thereafter, user interaction is tolerable for all programs. Program behavior begins to resemble that of *253.perlbmk* and *186.crafty* in Figure 2(a).

VM overhead arising from program initialization/startup is a real-life concern in applying run-time instrumentation to regression testing environments. Regression tests are *short* running instances of a program that exercise localized regions of code. This characteristic allows testing of specific program features. While instrumentation enables tasks like code coverage characterization and memory error detection to aid debugging [32], the translation cost cannot be amortized due to the short execution times of these tests.

Consider the *Gnu Gcc* compiler whose test cases are several hundred source code files. Across many tests, the compiler performs identical tasks of analysis, optimization and

Application	Description	% Lib code
Gftp	File transfer client	97%
Gvim	Graphical text editor	80%
Dia	Diagram creation tool	96%
File Roller	Archive manager	97%
Gqview	Image manager	95%

Table 1. Linux GUI applications used to evaluate startup performance. % Lib code is the amount of library code executed at startup.

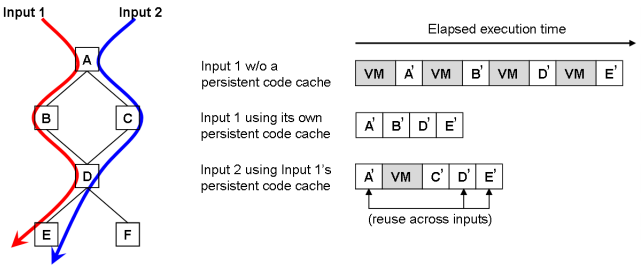


Figure 3. An example of translation reuse across inputs via persistent code caching.

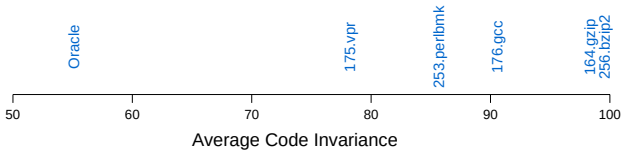


Figure 4. Code coverage percentage between phases within Oracle and Reference inputs of SPEC2K INT.

code generation to verify output (i.e. program binary or result). Prior discussion identifies that over 60% of the compiler's (*176.gcc*) time is spent in the VM translating code. Such slowdowns are unacceptable in the presence of so many tests. Much like *Gcc*, 100,000 tests are utilized in the development and verification of the *Oracle* database [23]. The large number of tests demonstrates a severe challenge in employing run-time instrumentation services for validation. Nevertheless, VM overhead can be substantially reduced by storing and reusing translations across executions, thereby enabling testing and debugging services even for large-scale domains.

3. Persistent Code Caching

Present run-time compilation systems discard the *intra-execution* code cache at the end of an execution. Subsequent executions of the same program start with an empty code cache, and translations are generated as required: possibly re-translating code already translated in previous runs. *Persistent code caching* eliminates these redundant translations, thereby reducing VM overhead. A *persistent run-time compilation system* extends the intra-execution code caching model by storing and reusing intra-execution code caches across executions.

3.1. Exploiting Code Reuse

The reuse of code across multiple executions of an application, including both its application and library code, is *inter-execution persistence*. Figure 3 shows an example of code reuse across two inputs. Original program control flow is shown on the left, while the right-half illustrates the same program under Pin. *Input 1 w/o a persistent code cache* illustrates VM requests to generate translations (e.g. *A'*) as code is executing for the first time. *Input 1 using its own persistent code cache* shows improved execution time when translations are stored and reused across executions of the same input. The VM is not invoked, even though code is executing for the first time, as the translations already exist within the persistent code cache. This is *same-input persistence*, and yields the most VM overhead savings as no new translations are generated.

In real computing environments, it is possible to encounter inputs for which persistent translations are non-existent. In such scenarios, persistent run-time compiler systems have the option of either starting with an empty code cache, or using a persistent cache generated by another input. The first option suffers the cost of *re*-translating all code. However, in the latter, common code need not be re-translated, as shown in Figure 3. *Input 2 using Input 1's persistent code cache* illustrates translation reuse of `A'`, `D'` and `E'` across inputs. Only `C` requires translation. While VM overhead savings may not match same-input persistence, using another input's cache is a more viable approach than complete re-translation. Using persistent caches across inputs is *cross-input persistence*.

The effectiveness of cross-input persistence is dependent upon the amount of code coverage between inputs (corroborated by data in Section 4.3). Figure 4 shows a scale of the average inter-execution code coverage for *Oracle*, and for the set of *SPEC2K INT* benchmarks having multiple *Reference* inputs. Benchmarks *164.gzip* and *256.bzip2* are clustered close to 100%, indicating all inputs exercise identical code. Benchmarks *176.gcc*, *253.perlbmk*, and *175.vpr* are further down the scale, indicating lower percentages of code coverage.

Oracle, a multi-process application experiences little code coverage (∼55%) between its different processes.

	Gftp	*Gvim*	*Dia*	*File Roller*	*Gqview*
Gftp	41				
Gvim	25	43			
Dia	29	22	63		
File Roller	23	22	43	62	
Gqview	28	26	23	23	29

Table 2. Number of common libraries between GUI applications.

Each process is a separate invocation of the program's binary to serve specific needs of the database during execution. Since the processes perform highly specialized tasks, there is little code coverage amongst them.

The proposed model of code reuse is further extendable via *inter-application persistence*, which leverages library usage of dynamically linked applications. Table 2 exposes significant library sharing between *GUI* applications. On average, at least a third of all libraries used by a *GUI* application are also used by other *GUI* applications discussed in Table 2. *GUI* applications execute up to 97% of their startup and initialization code from shared libraries (Table 1). The startup time of these applications is reducible by reusing library translations generated by one program for another.

3.2. A Persistent Run-time Compilation System

Persistent code caching is implemented in the Pin framework. While Pin is supported on multiple platforms, this work is evaluated only in the *Linux* environment on the IA32 platform. The system supports inter-execution, as well as inter-application persistence of single-threaded, multi-threaded, and multi-process applications.

Extensions required to persist dynamically generated translations, *persistent cache manager* and *persistent cache database*, are illustrated in Figure 1. The manager performs the fundamental tasks of generating persistent caches, verifying possible reuse, and storing them in the database. In the following paragraphs, components of a persistent cache are presented and a description of its generation and usage follows. This is initially presented in the context of an inter-execution persistent system, but is followed by a discussion of the changes required to facilitate inter-application persistence.

3.2.1. Persistent Code Caches. A persistent code cache is a file stored on disk containing traces and their associated data structures. The data structures contain information such as trace links and translation maps. Data structures are persisted to facilitate translations and optimizations of new code discovered across executions.

Persistent caches are generated assuming that application binaries remain unaltered inbetween executions. However, it is possible they are modified by a static compiler or optimizer. If modified, the translations become invalid, as reusing them results in erroneous execution. Translations are also invalid if the persistent system itself changes. Code and the data structures are specific to a version of the system and cannot be utilized across versions.

To prevent the use of invalid/inconsistent translations, persistent caches contain information pertaining to executable mappings present in memory at the time of their creation. The information is contained in *keys*. Keys are a hash of the base address, mapping size, binary path, program header, and modification timestamps. The number of keys generated varies based on the number of libraries present in memory at the point of persistent cache creation. At minimum, keys are generated for the application, Pin, and the Pin Tool specifying instrumentation. The application key ensures the original program has not been modified, while the Pin key ensures translations are not reused across versions of the persistent system. The Pin Tool key ensures instrumentation semantics are consistent across executions.

Keys cannot be created for code dynamically generated by the application (e.g. self-modifying code). Thus, persistent caches only contain traces backed by a file on disk. All others traces are invalidated by removing their information from the translation map.

3.2.2. Persistent Code Cache Generation. Information is written to a persistent code cache whenever the intra-execution code cache becomes full or the last thread of execution performs the `exit` system call. In particular, at program startup, the persistence manager allocates two large linear regions of memory from the application's virtual address space. These are the persistent memory pools, which together form a persistent cache. If the pools are unavailable, persistence is abandoned and execution continues normally (i.e. no persistent cache is generated at any point during execution).

One of the persistent pools is dedicated to contain traces; this is akin to Pin's intra-execution code cache. The other contains data structures associated with the persisted traces. Pin's data structures are `C++` programming objects allocated on an application's heap and destroyed upon program termination. The manager persists these objects by overloading the objects' default memory allocators (i.e. `new` and `free`) with custom versions that manage memory requests out of the persistent data structures pool.

Persistent memory pools for data structures and traces are maintained separately for performance reasons; intermixing code and data structures results in poor performance. Data structures are frequently accessed while an application's footprint is still being captured in the code cache.

Thereafter, control remains mostly in the code cache with little need for them. Separating code from data allows for code to be more tightly compacted. Otherwise, data intermixed with code results in increased cache misses/conflicts, page faults, and translation lookaside buffer misses.

3.2.3. Persistent Code Cache Reuse. The persistent cache manager facilitates persistent code cache reuse by invoking a cache lookup function at the beginning of execution. The function attempts to locate a persistent cache utilizing keys computed on the application being run, Pin, and its Pin Tool. If found, the cache is loaded into memory.

A persistent cache is loaded via two `mmap` system calls of the file on disk. One maps the code cache, while the other loads the data structures. Since the *Linux* kernel employs demand-based paging, disk I/O occurs based on the access pattern of the executing code.

Library loading occurs via the `mmap` system call. Libraries may load at different addresses across executions, as a result of changes in program behavior or host environment [24]. A persistent system must ensure the validity and reusability of the cached translations. To ensure proper execution, all library loads are intercepted and keys are computed on the loaded binary. If the computed key matches the key contained within the persistent cache, the translations are valid and reusable. Otherwise, a conflict has occurred. The implementation leverages Pin's existing mechanism to handle such conflicts, as they occur even in the intra-execution code cache model.

Even in the absence of conflicts, loading libraries at different addresses across executions is problematic. Absolute addresses embedded in the persisted translations cause incorrect execution. For example, a run-time compiler may translate a `CALL 0x8048494` instruction into a (`PUSH 0x8048499`, `JMP 0x78048494`) pair. The `PUSH` instruction places the return address on top of the stack, while the `JMP` instruction transfers control to a trace containing translated instructions of the called subroutine. Such a translation is performed to maintain transparency and ensure proper execution. Reusing this translation causes program failure if the library containing the `CALL` instruction is relocated to a different address during a subsequent run; the literal in the persisted `PUSH` instruction becomes invalid. The described implementation cannot use the persisted translations if library locations vary across executions. However, the run-time compiler can be adapted to generate position independent translations capable of coping with library relocation.

Thus far, the system is discussed in the context of inter-execution persistence. Enabling inter-application persistence requires minor changes: the application key used in the persistent cache lookup function is ignored, thereby allowing the function to return a cache corresponding to *any* application instrumented identically. As such, a check verifying the usability of the persisted application translations is mandatory to ensure execution of the proper program. If the check fails, the persisted application translations do not correspond to the running program, hence requiring their invalidation. The invalidation consequently triggers translations of the current binary during execution. As execution proceeds, persistent library translations common between programs are reused if identical libraries are loaded at the same address across programs. Otherwise, they too are invalidated and re-translated.

4. Results

This section discusses the benefits of persistent code caching. First and foremost, same-input persistence is evaluated to illustrate the benefits when all persisted translations are reused without requiring new code generation. Following that is an evaluation of the benefits of persistent caches across inputs. Lastly, code reuse across program boundaries is presented.

4.1. Methodology

Performance is evaluated on the *SPEC2K INT* suite, *GUI*, and *Oracle Database 10g Express Edition (XE)* applications. The *SPEC2K INT* suite is compiled using the *Gnu Gcc 3.2* compiler at level `-O2` optimization. Benchmark *252.eon* is omitted because its source code cannot be compiled in the experimental environment. *GUI* programs are evaluated only for their startup phase; the time it takes for the graphic interface (i.e. buttons, menus, etc.) to be ready for user interaction. Reproducible interactive behavior is achieved using *Gnu*'s *Xnee* [34] package. Experimental data for *SPEC2K INT* and *GUI* applications is gathered on an Intel (R) Pentium (R) 4 1700MHz machine with 1.5GB memory running the *RedHat 7.3 Linux* distribution.

Oracle is discussed in the context of regression testing in a setup representative of production testing environments. Every regression test is comprised of five phases. A phase is a new instance of the program, which performs highly specialized tasks. The start of a phase is identifiable via process creation and is treated as a separate execution. In addition, the phases are treated as unique inputs as they exercise significantly different code. The sequence of phases are as follows: instance preparation (*Start*), instance association with database (*Mount*), database enabling (*Open*), transaction/unit-test execution (*Work*), database deactivation (*Close*). The unit-test evaluated performs sixty transactions (i.e. additions, modifications and deletions) on ten database tables. *Oracle* is evaluated on a 8-way Intel (R)

Xeon (TM) clocked at 1700MHz with 4GB memory running the *RedHat Enterprise Linux 3*.

All benchmarks are run unmodified under Pin. 512MB of an application's address space (a tunable parameter) is reserved for Pin's use. The pre-allocated memory is equally divided between the code cache and its supporting data structures. If the reserved memory is exhausted, Pin reclaims the space by flushing the code cache. A code cache flush discards all translated code and data structures. Through the course of all experiments discussed in this section, none of the benchmarks triggered a code cache flush.

4.2. Same-input Persistence

Same-input persistence demonstrates the peak potential of the system. Figure 5(a) presents this peak potential by showing improvements relative to running base Pin. Figure 5(b) discusses benefits under instrumentation services.

SPEC2K INT's *Train* and *Reference* inputs have substantially different run times; execution is 6× longer when the *Reference* inputs are used. As expected, longer runs limit the benefits of using persistent code caches. Once an application's footprint is captured in the code cache, the fixed VM overhead becomes a smaller percentage of the execution time. This is specifically the case with the *SPEC2K INT* benchmarks, which exercise little new code over time (Figure 2(a)). As a result, performance gains are better for the *Train* inputs. Benchmarks *197.parser* and *254.gap* have insignificant VM overhead for the *Reference* inputs, but both experience ∼50% savings under the *Train* inputs. Large benefits (>10%) are not seen for the *Reference* inputs with the exception of *176.gcc* (>30%) and *253.perlbmk* (∼10%).

The variability in performance between the *Train* and *Reference* inputs indicates that cold code exists in programs. While the amount of cold code is dependent upon the input, it cannot be dismissed as a rare occurrence. A persistent system, as discussed in this paper, reduces VM overhead when cold code exists without penalizing performance when cold code is absent/negligible.

Figure 5(b) shows the breakdown of VM overhead and the performance of translated code. The leftmost bar in each cluster is the original program execution time. The bar in the middle of the cluster shows the execution time when Pin is performing native-to-native binary translation. The bar is split, with the lower section representing time spent executing traces, and the upper section representing VM overhead. A similar breakdown is presented in the last bar when basic block instrumentation is added.

Under native-to-native translation, the two benchmarks in the *SPEC2K INT* suite having significant VM overhead are *176.gcc* and *253.perlbmk*. The remaining benchmarks incur negligible overheads, or their slowdowns are primarily due to poor performance of the translated code. As such, there is little gained from using persistently cached translations. Benchmark *176.gcc* consistently executes new code throughout its lifetime, while the *253.perlbmk* has more startup cost relative to other benchmarks (Figure 2(a)). Using same-input persistent caches on these two benchmarks eliminates VM overhead completely as shown in Figure 5(a).

If additional services like instrumentation are added for long running programs, it is possible to disproportionally increase VM overhead, and worsen translated code performance. This is illustrated by the last bar within the clusters of Figure 5(b). Detailed basic block profiling increases VM overhead by as much as 25%. The increase is substantial in *255.vortex* and *256.bzip2* which in the absence of instrumentation incur ∼2% VM overhead.

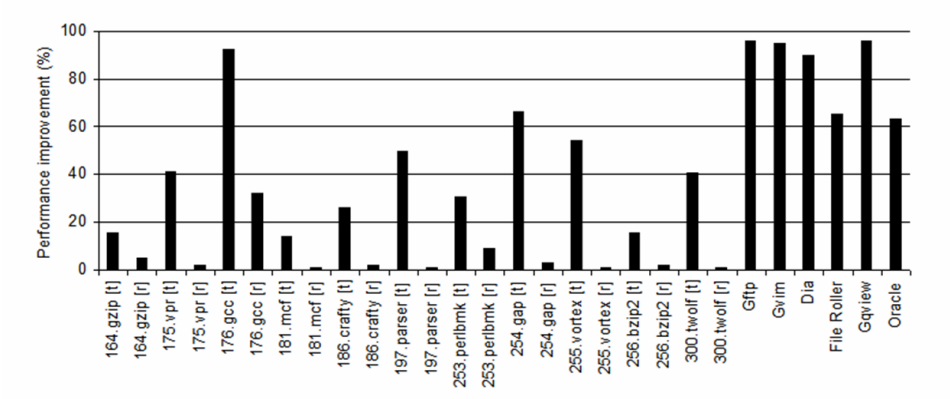
GUI programs highlight an interesting aspect of same-input persistence – the startup code of a *GUI* program remains consistent across different inputs. A substantial part of a *GUI* program's input is user interaction (e.g. mouse activity and button clicks). User input cannot be presented to the program or processed by the program unless initialization completes. The same initialization code (i.e. cold code) is consistently executed across all executions. Hence, caching the cold startup code of *GUI* programs is extremely beneficial. Figure 5(a) shows an average improvement of 90% in their execution time. Such improvements are highly desirable as *GUI* applications are interactive in nature, and slow startup times are not tolerable.

Lastly, the performance of the *Oracle* benchmark is shown. A single unit-test (all phases in sequence) without Pin takes approximately 80 seconds. Running the same test under Pin takes nearly 1300 seconds without instrumentation. Using persistence reduces execution time to just over 490 seconds, a 63% improvement in performance. Benefits under instrumentation are even larger. Instrumenting memory references without persistence extends execution by 4000 seconds, but with persistence it takes slightly over 1000 seconds (∼4× speedup).

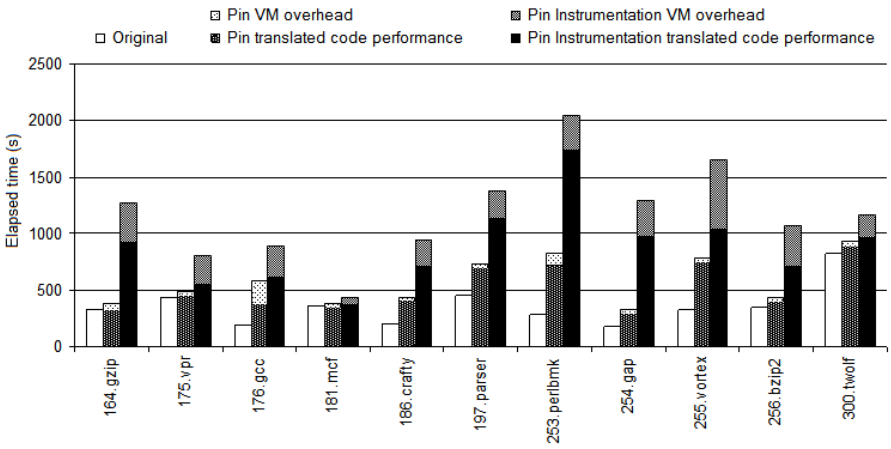
VM overhead is sensitive to the amount of instrumentation added, and to the analysis performed within the instrumentation routines. Instrumentation increases VM overhead due to additional code generation, but complex and time consuming analysis can diminish the relative significance of VM overhead. To avoid such potential biases, instrumentation results are not discussed in the remainder of the paper. Performance is compared to the minimum overhead Pin must overcome before applying any instrumentation: the cost of dynamically recompiling application code.

4.3. Cross-input Persistence

Persistent caches are capable of improving the performance of not only the input used in creating them, but



(a) Performance improvement of SPEC2K INT (Train and Reference inputs), GUI and Oracle benchmarks.



(b) SPEC2K INT Reference input overheads with and without instrumentation.

Figure 5. Evaluation of Same-input Persistence.

other inputs as well. This cross-input persistent cache usage model is most desirable when same-input persistent caches are unavailable. To evaluate this, two benchmarks illustrated in Figure 4 are discussed in detail as a case study. The criteria for choosing the benchmarks is the amount of VM overhead and the amount of code coverage between inputs.

The two interesting benchmarks worth investigating in detail are *176.gcc* and *Oracle*. Both experience large VM overheads while exhibiting different code coverage characteristics between inputs/phases. *176.gcc* suffers from ∼33% VM overhead with tight code coverage (∼90%) between its inputs. *Oracle* experiences ∼63% slowdown due to VM overhead, and its average code coverage is the lowest at 55%, compared to the other benchmarks in Figure 4.

While benchmarks *253.perlbmk* and *175.vpr* are interesting due to low code coverage between their respective inputs (Figure 4), they incur low VM overheads of only 2% and 8% (Figure 5(a) for *Reference* inputs). Cross-input persistence does improve their execution time relative to running Pin without persistence, but the improvements are not substantial as the headroom for VM overhead reduction is small. The same is observed with benchmarks *164.gzip* and *256.bzip2*.

Table 3(a) shows *176.gcc*'s code coverage across different combinations of the *Reference* inputs. The leftmost column indicates a run, and the percentage of its code covered by other inputs is listed in the columns following the same row. Code coverage is the amount of static code corresponding to an input also executed by other inputs. For example, 97% of *Input 5*'s code is also executed by *Input 2*. 100% code coverage corresponds to same-input persistence. The table shows fluctuation between 84% and 98% code coverage for *176.gcc*. *Input 4* shows the least code coverage for all inputs, suggesting lower benefits are likely compared to using other inputs' persistent caches.

Table 3(b) shows a similar code coverage table for the

	Input 1	*Input 2*	*Input 3*	*Input 4*	*Input 5*
Input 1	100%	87%	89%	84%	88%
Input 2	93%	100%	90%	85%	98%
Input 3	93%	88%	100%	91%	89%
Input 4	95%	90%	98%	100%	90%
Input 5	92%	97%	90%	84%	100%

(a) 176.gcc

	Start	*Mount*	*Open*	*Work*	*Close*
Start	100%	47%	47%	33%	46%
Mount	22%	100%	78%	66%	64%
Open	18%	66%	100%	68%	56%
Work	18%	66%	77%	100%	56%
Close	29%	89%	91%	74%	100%

(b) Oracle

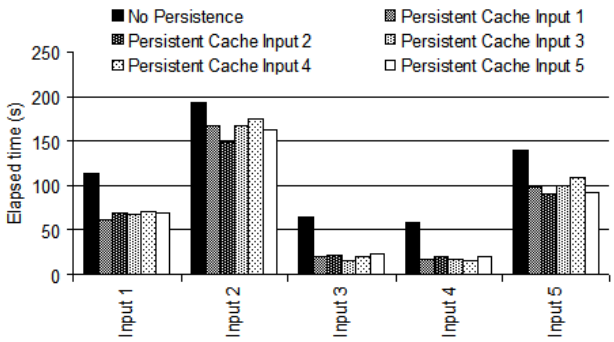
Table 3. Code coverage percentage.

Oracle database. The data indicates significant fluctuation in coverage across phases ranging between 18% (*Work*'s coverage by *Start*) to 91% (*Close*'s coverage by *Open*). A persistent cache created using the *Start* phase is least likely to help the other phases due to low code coverage (between 18% and 29%). The persistent cache from the *Open* phase is most likely to help the other phases since it encompasses large amounts of code corresponding to all inputs (between 47% and 91%).

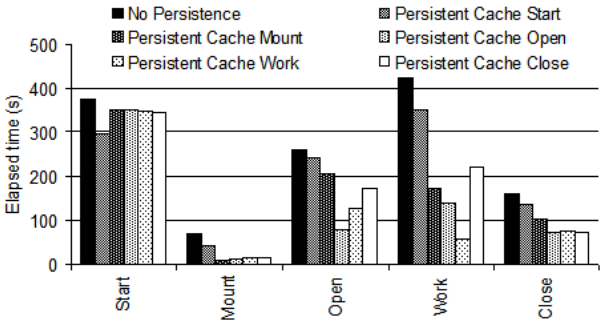
The benefits of cross-input persistence on *176.gcc* are shown in Figure 6(a). Within each cluster, the leftmost bar indicates the performance of an input executing without a persistent cache. The rest of the bars correspond to priming the code cache with a persistent cache generated using an input corresponding to the legend. For example, *Persistent Cache Input 2* refers to a persistent cache generated using *Input 2*.

In analyzing the performance of cross-input persistence for *176.gcc*, the most significant insight is that sizable performance improvements are achieved over running Pin without persistence. Performance is tied to the amount of code coverage between inputs, similar to results in profile-driven compilation systems. While best results are achieved with same-input persistence (100% coverage), the execution of *Input 5* using a persistent cache from *Input 4* results in higher execution time (84% coverage and 108 seconds execution time) than using a persistent cache from *Input 2* (97% coverage and 90 seconds execution time).

Performance of *Oracle*'s phases under cross-inputer persistence is shown in Figure 6(b). Even though *Oracle* has a lower percentage of code coverage between inputs, all of its phases benefit from using persistent code caches. Improvements range from 7% (*Start* phase with *Persistent Cache*



(a) 176.gcc



(b) Oracle

Figure 6. Time savings under Cross-input Persistence.

Mount) to 81% (*Mount* phase with *Persistent Cache Open*).

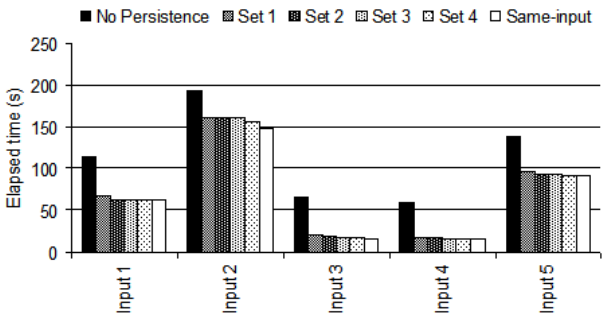
Unlike *176.gcc*'s inputs, *Oracle*'s phases behave substantially different based upon the cache being utilized. Consider the execution times of the *Close* phase using *Persistent Cache Start* and *Persistent Cache Open*. There is ∼60% difference in execution time. A correspondingly large difference of 70% exists in coverage of the *Close* phase between the *Open* and *Start* phases.

The input that benefits the least, even with same-input persistence, is the *Start* phase. Relative to the other phases, *Start* is covered the least by other phases. As a result, cross-input persistence improves performance by only 7%.

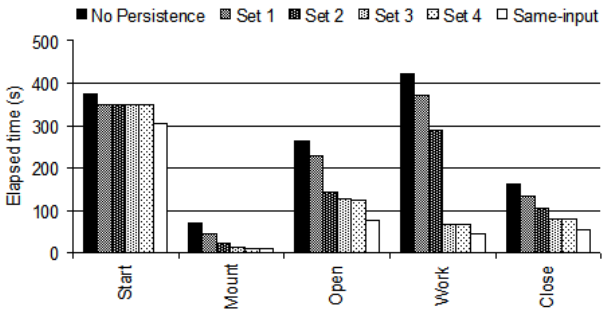
Overall, persistent caches are useful even when their inputs are run infrequently as they can improve the performance of other inputs. In addition, a persistent cache does not degrade performance when it is ineffective. As such, persistence improves the current code caching model.

4.4. Persistent Cache Accumulation

Using a persistent cache created from only one input limits the benefits to the amount of code coverage between



(a) 176.gcc



(b) Oracle

Figure 7. Time savings under Persistent Cache Accumulation.

inputs. With applications such as *Oracle* that experience low code coverage between inputs (i.e. phases), performance differs significantly depending on the persistent code cache being used. For instance, *Persistent Cache Start* yields the least improvement in performance across all the remaining phases. On average it executes only 22% of the code exercised by other inputs (Table 3).

New code is discovered across executions as inputs change, which is an opportunity to improve the performance of persistent caches over time. The code coverage of a persistent cache can be increased by repeatedly using it across executions of different inputs, and adding newly discovered translations into it. The run-time addition of new translations into a persistent code cache is *persistent cache accumulation*.

The effect of applying persistent cache accumulation on *176.gcc* is shown in Figure 7(a). Each cluster shows the execution time when an accumulated persistent cache is used for an input. The leftmost bar in each cluster is the performance of base Pin running without persistence. The last bar in every cluster is the performance of same-input persistence. It exists to compare the effectiveness of persistent cache accumulation. The bars inbetween indicate the performance of the accumulated caches.

Persistent code caches are accumulated in ascending input order, skipping the cache corresponding to the input being evaluated. For example, consider *Input 2* on the x-axis. *Set 1* in its cluster contains a persistent cache generated using *Input 1*. *Set 2* contains the accumulation of persistent caches generated using *Input 1* and *Input 3*. *Input 2* in *Set 2* is skipped because it corresponds to the input being evaluated. *Set 3* is made up of *Input 1*, *Input 2* and *Input 4*, and *Set 4* is comprised of *Input 1*, *Input 2*, *Input 3* and *Input 5*.

Across all inputs of *176.gcc*, accumulated persistent caches outperform Pin without persistence, while closely matching the performance of same-input persistence. The benefits from accumulating more than two persistent caches are not substantial/noticeable due to large amounts of code coverage between inputs of this benchmark. Therefore, additional accumulations do not add large amounts of new code to the persistent cache.

In contrast, accumulation largely benefits *Oracle*. As traces from more inputs are accumulated (increasing set number), performance improves with the exception of the *Start* phase. The gains are limited as it experiences the least code coverage by the other phases.

Across the *Mount*, *Work*, and *Close* phases, *Set 3* yields the most improvement in performance. It contains code accumulated from the *Start*, *Mount*, and *Open* phases. Of these, the *Open* phase is the most complex and executes large amounts of new code not present in the *Start* and *Mount* phases. As a result, its accumulation contributes a significant number of traces to the persistent cache resulting in improved execution time.

Set 4, the addition of the *Close* phase to *Set 3*, does not contribute much improvement. The *Close* phase is relatively small. Additionally, *Set 3* already contains much of the code *Close* exercises. This is due to the *Open* phase, which covers 91% of *Close*'s code footprint (Table 3(b)).

Overall, persistent cache accumulation is very effective. For applications like *176.gcc*, which exhibit high code sharing, accumulation maintains the persistent code to be used in later executions without decreasing performance. For applications like *Oracle* experiencing lower amounts of code sharing, persistent cache accumulation greatly improves performance. Aggregating the benchmark's traces from different phases into a single persistent cache narrows performance to within 22% of same-input persistence.

4.5. Inter-application Persistence

Startup time of real world applications is diminishable by leveraging library sharing amongst programs. Table 4 shows the amount of a *GUI* application's library code found in other *GUI* applications persistent caches. For example, 78% of *Gvim*'s library code is found in *Gftp*'s persistent

	Gftp	*Gvim*	*Dia*	*File Roller*	*Gqview*
Gftp	100%	71%	64%	78%	78%
Gvim	78%	100%	76%	62%	72%
Dia	64%	55%	100%	74%	78%
File Roller	62%	81%	74%	100%	84%
Gqview	79%	72%	78%	84%	100%

Table 4. Library code coverage percentage between GUI applications.

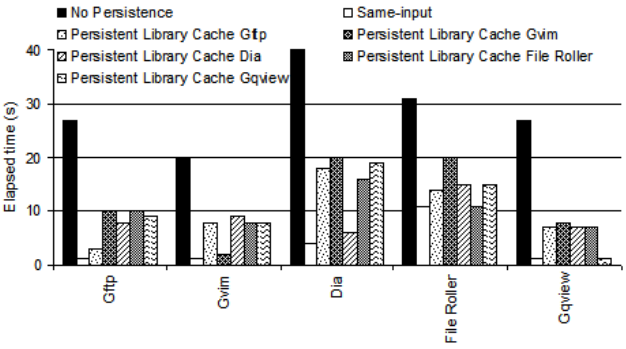


Figure 8. Time savings under Inter-application persistence.

cache. This data is a refinement of Table 2, which presents code sharing at the coarse granularity of entire libraries and does not account for the actual code coverage.

Execution time improvements under inter-application persistence (i.e. the use of one application's persistent code cache for another) are shown in Figure 8. The leftmost bar reflects the startup time of the application under Pin without persistence, which on average is over 20 seconds for all applications. *Same-input* persistence (second bar in cluster), previously discussed in Section 4.2 provides a basis for evaluating time savings under inter-application persistence.

Every cluster has a legend corresponding to itself (i.e. *Gftp* on the x-axis has a *Persistent Library Cache Gftp* legend entry). These bars isolate the maximum benefits achievable using only library code. It is a form of same-input persistence, but without traces corresponding to the primary application itself. Across all programs, this bar is within a second or two of same-input persistence, indicating that *GUI* applications indeed execute significant startup code from libraries as claimed in Table 1. Ergo, persisting translated library code in itself can offer large performance improvements in startup time.

Remaining bars within the clusters show time savings under inter-application persistence. Improvements in performance are significant (averaging around 59%) across all applications, but do not correspond closely with the code

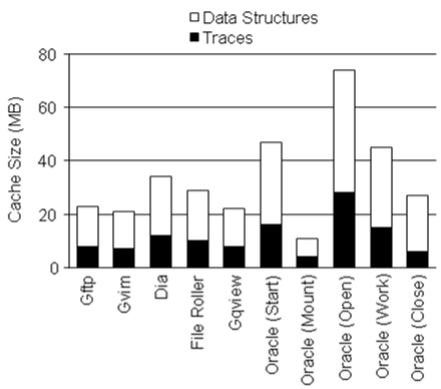


Figure 9. Persistent cache sizes.

coverage data in Table 4 (averaging 70% coverage). While applications exhibit code sharing, the implementation has inherent limitations. Traces corresponding to identical libraries loaded at different addresses across programs cannot be used because the system does not generate relocatable translated code. Instead, the system falls back to retranslation. Hence, potential benefits are lost.

4.6. Persistent Code Cache Sizes

Input code coverage (i.e. code footprint) determines the size of a persistent cache. Most *SPEC2K* benchmarks have small code footprints. As a result, their caches are less than 3MB in size. Benchmark *176.gcc*, due to its large code footprint, has a larger 14MB persistent cache. As per Figure 9, the *GUI* and *Oracle* benchmarks discussed in this paper have even larger persistent caches.

The stacked bars of Figure 9 illustrate the memory consumed by the persistent traces (i.e. code cache) and the persistent data structures. Interestingly, the data structures corresponding to the traces consume more memory than the traces themselves. Applications in the *SPEC2K INT* benchmark suite exhibit the same characteristic. Data structure memory consumption is more because trace management requires large amounts of information such as incoming/outgoing links, register liveness analysis and register bindings.

Memory management is an important aspect of run-time compilation systems. Prior work focuses on reducing the memory footprint of the translated code [14, 15, 29] assuming the code cache to be the largest consumer of memory. However, data in Figure 9 indicates there is more room for reducing memory footprint by targeting the runt-time data structures.

5. Related Work

Implementations of persistence have been explored in the domain of binary translation as a means of reducing VM overhead. Static pre-translators [7, 17, 28] support offline translation for online usage. However, static pre-translation has proven infeasible in production environments due to extensive code expansion. Even when adding small amounts of instrumentation, field experiments show a 10× increase in code size. Such expansion is impractical for large applications like *Oracle*, which are ∼100MB in static size and 1GB when statically pre-translated. These applications require the use of a dynamic system that persistently caches only executed code. Experiments using the implementation discussed in this paper yielded a manageable cache size of ∼256MB.

Persistence in dynamic binary translators is briefly touched upon by the authors of Strata [27] and HDTrans [30]. But neither focused entirely on persistence nor demonstrated significant benefits using it. In this paper, the potential of persistence is explored thoroughly across different classes of applications, as well as their corresponding inputs.

Hazelwood and Smith [16] characterizes code sharing in the context of a dynamic optimization system for the *SPEC2K INT* suite. This work is motivated by that initial study and builds upon it in three important ways. First, runtime compilation overhead is broken down and the overhead reducible via persistent caching is explained. Second, code coverage and the significance of persistent caching are discussed across a broader set of real-life applications, contrasting the *SPEC2K INT* suite with the *GUI* programs and the *Oracle* database. Lastly, performance results using a real system are presented.

Li et al. [19] improve the software-managed code cache model of the IA32EL binary translator by not discarding translations of modules unloaded from memory. Rather, the invalid translations are cached separately from the code cache in an attempt to reuse them if the module is later reloaded during execution. Persistence as investigated in this paper goes further by extending the model of code reuse across executions, as well as applications.

Conte et al. [8] use persistence to improve object-code dynamic rescheduling as binaries are used across VLIW machine generations. Object-code, once dynamically *re*-scheduled to a VLIW machine other than its native counterpart, is stored on disk and used across executions. Hence, reducing the number of rescheduling requests over the lifetime of a binary on an incompatible VLIW machine.

In evaluating the benefits of profile information, prior work [5, 13, 26, 33] explores the effectiveness of training inputs in predicting code coverage, and future program behavior under new inputs. Considerable work in profile-driven optimization leverages such information. At present, this work investigates the benefits of persistence in a system (Pin) affected only by code coverage, and not by run-time specifics of control flow and memory usage.

While persistence is evaluated in Pin, the fundamental approach of leveraging code reuse across executions and applications is exploitable by other dynamic compilation systems as well. These systems engage in complex optimizations and analysis, but prior to engaging in such complex tasks, they too must overcome cold code and/or startup penalties. Therefore, persistence can be exploited as a means of overcoming these performance bottlenecks.

6. Conclusion

This paper proposes extending the intra-execution model of code reuse, employed by present run-time compilation systems, to inter-execution and inter-application (by leveraging common library dependencies). Translations generated during executions are cached persistently on disk for reuse across runs.

The important contribution of this paper is that persistent translations are useful for several important application classes. In particular, persistence benefits programs with severe startup costs (e.g. GUI) and large code footprints (e.g. Gcc and Oracle). Performance improvements are even larger under more specialized uses of run-time compilation systems such as dynamic binary instrumentation.

Persistent code caching is implemented in Pin. The persistent system supports inter-execution, as well as inter-application persistence of single-threaded, multi-threaded, and multi-process applications. The performance benefits are evident from the results: *SPEC2K INT* benchmarks experience an average improvement of 26% with instrumentation. The startup phases of desktop GUI applications benefit by nearly 90%. Furthermore, a 400% speedup is achieved in translating the Oracle database in a regression testing environment. Aside from improvements in performance, the system does not degrade performance when persistence is ineffective.

7. Acknowledgements

We thank Alex Shye, Kelley Kyle, Bala Narasimhan and the anonymous reviewers for their detailed comments and suggestions on improving the quality of this paper. We are also extremely thankful to the Pin Team for granting us access to the Pin source code. This work was funded by Intel Corporation.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2000.

[2] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 Execution Layer. In *Proc. of the 36th International Symposium on Microarchitecture*, 2003.

[3] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In *Proc. of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2006.

[4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proc. of the international symposium on Code generation and optimization*, 2003.

[5] B. Calder, D. Grunwald, and A. Srivasta. The predictability of branches in libraries. In *Digital WRL Technical Report*, June 1995.

[6] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[7] R. S. Cohn, D. W. Goodwin, and P. G. Lowney. Optimizing alpha executables on windows nt with spike. *Digital Technical Journal*, 9(4):3–20, 1998.

[8] T. M. Conte and S. W. Sathaye. Dynamic rescheduling: A technique for object code compatibility in VLIW architectures. In *Proc. of the 28th Annual International Symposium on Microarchitecture*, 1995.

[9] J. C. Dehnert, B. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software. In *Proc. of the International Symposium on Code Generation and Optimization*, 2003.

[10] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: A new run-time control point. In *35th Annual International Symposium on Microarchitecture*, December 2003.

[11] Determina. http://www.determina.com/.

[12] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proc. of the 24th International Symposium on Computer Architecture*, June 1997.

[13] P. T. Feller. Value profiling for instructions and memory locations. Master's thesis, University of California at San Diego, 1998.

[14] A. Guha, K. Hazelwood, and M. L. Soffa. Reducing exit stub memory consumption in code caches. In *International Conf. on High-Performance Embedded Architectures and Compilers (HiPEAC)*, Ghent, Belgium, January 2007.

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

[16] K. Hazelwood and M. D. Smith. Characterizing inter-execution and inter-application optimization persistence. In *Workshop on Exploring the Trace Space for Dynamic Optimization Techniques*, San Francisco, CA, 2003.

[17] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1), August 1997.

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

[19] J. Li, P. Zhang, and O. Etzion. Module-aware translation for real-life desktop applications. In *Proc. of the 1st ACM/USENIX international conference on Virtual execution environments*, 2005.

[20] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization sytem. In *Proc. of the International Symposium on Microarchitecture*, 2003.

[21] 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 the ACM SIGPLAN 2005 Conf. on Programming Language Design and Implementation*, June 2005.

[22] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proc. of the 3rd Workshop on Runtime Verification*, July 2003.

[23] Oracle. The making of oracle database 10g. Website. URL:http://www.oracle.com/technology/oramag/oracle/03-sep/index.html.

[24] PaX. Web site: http://pax.grsecurity.net/.

[25] D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder. GILK: A dynamic instrumentation tool for the linux kernel. In *Proc. of the 12th International Conf. on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation (TOOLS '02)*, 2002.

[26] S. Savari and C. Young. Comparing and combining profiles. volume 2, 2000.

[27] K. Scott, J. Davidson, and K. Skadron. Low-overhead software dynamic translation. Technical Report CS-2001-18, University of Virgina, 2001.

[28] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: A quasi-static compiler for java. In *Proc. of the ACM SIGPLAN International Conf. on Object-Oriented Programming Languages, Systems, Languages, and Applications*, October 2000.

[29] S. Shogan and B. R. Childers. Compact binaries with code compression in a software dynamic translator. In *Proc. of the conference on Design, Automation and Test in Europe*, 2004.

[30] J. S. S. Swaroop Sridhar and P. P. Bungale. Hdtrans: A low-overhead dynamic translator. In *Proc. of the Workshop on Binary Instrumentation and Applications*, 2005.

[31] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, 1999.

[32] Valgrind. Project suggestions, 2003. Website. URL:http://valgrind.org/help/projects.html.

[33] D. W. Wall. Predicting program behavior using real and estimated profiles. In *Proc. of the ACM SIGPLAN 1991 Conf. on Programming Language Design and Implementation*, 1991.

[34] G. Xnee. We site: http://www.gnu.org/software/xnee/.

[35] C. Zheng and C. Thompson. Pa-risc to ia-64: Transparent execution, no recompilation. *Computer*, 33(3), 2000.