Persistence in Dynamic Code Transformation Systems

Vijay Janapa Reddi[†], Dan Connors[†], Robert S. Cohn[‡]
[†]Department of Electrical and Computer Engineering
University of Colorado, Boulder
{janapare, dconnors}@colorado.edu
[‡]Intel Corporation
robert.s.cohn@intel.com

Abstract

Dynamic code transformation systems (DCTS) can broadly be grouped into three distinct categories: optimization, translation and instrumentation. All of these face the critical challenge of minimizing the overhead incurred during transformation since their execution is interleaved with the execution of the application itself. The common DCTS tasks incurring overhead are the identification of frequently executed code sequences, costly analysis of program information, and run-time creation (writing) of new code sequences. The cost of such work is amortized by the repeated execution of the transformed code. However, as these steps are applied to all general code regions (regardless of their execution frequency and characteristics), there is substantial overhead that impacts the application's performance. As such, it is challenging to effectively deploy dynamic transformation under fixed performance constraints. This paper explores a technique for eliminating the overhead incurred by exploiting persistent application execution characteristics that are shared across different application invocations. This technique is implemented and evaluated in Pin, a dynamic instrumentation engine. This version of Pin is referred to as Persistent Pin (PPin). Initial PPin experimental results indicate that using information from prior runs can reduce dynamic instrumentation overhead of SPEC applications by as much as 25% and over 90% for everyday applications like web browsers, display rendering systems, and spreadsheet programs.

1 Introduction

Dynamic code transformation systems (DCTS) have the potential to impact the design and use of modern computer systems since they can perform a number of tasks at runtime, such as profiling, optimization, and translation. DCTS have an inherent advantage over static techniques because they can access and collect execution characteristics that, in turn, can be used to adapt the code execution of the target application. However, since DCTS execution is interleaved with the execution of an application, there is a substantial overhead penalty during code transformation. In the context of performance analysis and program behavior tools, the overhead of a DCTS may be acceptable. Nonetheless, the overhead of DCTS is a major barrier to runtime transformation in real systems and for use in large-scale application environments.

The execution overhead of an application running under any DCTS is determined primarily by two things: (1) transformation overhead and (2) performance of the translated code. The transformation overhead is determined by the level of complexity of run-time analysis and transformation components applied to the code. Likewise, the overhead of each DCTS component is not fixed, but depends on the run-time characteristics of the application. At startup, most programs have tremendous amounts of code that are executed only once or a few times, a DCTS will not be able to amortize the run-time overhead costs for these code sequences. The performance of the translated code is controlled by factors such as whether optimization was applied or whether instrumentation code was added during transformation.

Traditionally, DCTS designs have primarily focused on achieving a given transformation task on a single execution instance of a program. Repeated invocations do not exploit persistent characteristics such as frequent/identical code paths or other runtime information from previous executions. This is a significant drawback since applications tend to exhibit identical characteristics across executions as shown in [5]. Exploiting shared behavior can dramatically reduce the run-time overhead since the time spent doing the same analysis and optimization in a repeated invocation can be avoided.

In this paper, a solution that addresses the transformation overhead issue is proposed. This strategy aims at exploiting *persistent* characteristics of the application that are recurring across independent invocations. By caching information from prior executions, the cost of transformation is minimized since subsequent invocations of the application may only require limited amounts of new code generation. While persistence support can be directed to cache many types of information (optimization, program state, etc), the target of this paper is to illustrate the fundamental challenges in materializing persistent caching of instrumentation information in a DCTS.

An experimental framework for capturing persistent code transformations, PPin, has been implemented and evaluated in the Pin dynamic instrumentation system. PPin is capable of caching instrumented executions, reusing previously cached runs and increasing the size of the cache execution as new paths are generated due to a new input set. PPin is also capable of supporting large, multithreaded applications. Persistence in Pin reduces the instrumentation overhead of SPEC programs by as much as 25% and as much as 90% for the start-up cost of everyday applications such as web browsers, graphics rendering systems, spreadsheets and text editors.

The remainder of the paper is organized as follows: Section 2 presents a detailed argument that supports the caching of dynamic program executions. Section 3 presents an overview of the solution along with an elab-

oration of the challenges in deploying persistence in a real run-time system. Section 4 is an evaluation of the persistence in a run-time binary instrumentation system. Finally, Section 5 summarizes the contributions of this paper.

2 Dynamic code transformation

Dynamic code transformation is a well known topic in the domain of instrumentation, debugging, performance analysis, and optimization. Jalapeno [1], Dynamo [2], DynamoRIO [3], Mojo [4] and Adore [6] are examples of native-to-native performance optimizers that monitor the application execution behavior and apply optimizations as best suited to improve the native binary performance. IA32EL, HP-Aries, Daisy, BOA and Transmeta are binary translators that execute binaries complied for one architecture on another. Pin [7] and Valgrind [8] are systems that use run-time compilation to perform application introspection via the use of instrumentation.

All of the above systems focus on accomplishing their task by monitoring a single instance of the program. They do not attempt to leverage information from past executions to reduce the overhead in subsequent executions or to further improve the application performance. The challenges faced by dynamic code transformation systems executing in a single execution instance are presented in the next section.

2.1 DCTS challenges

DCTS experience the primary challenge of overhead since their execution is interleaved with the execution of the application itself. The overhead is a summation of the time spent executing the generated code (translated code overhead) and the overhead involved in generating the code (transformation overhead).

Translated code overhead is determined by a number of factors that include the original code, modification to the code (optimization or added instrumentation), and transformation artifacts for transparently maintaining the original application's functionality. In the absence of code optimization and instrumentation, the dynamic execution count of translated code can still have significant overhead. Code that includes branches with varying targets requires special DCTS handling that amounts to overhead from run-time checking of branch targets, as well as directing the application execution to the DCTS system for new code compilation. Overhead can also be added based on the layout of the translated code in memory since this determines the I-cache, TLB, and memory system performance. DCTS overhead of applications such as *253.perlbmk, 252.eon and 255.vortex* is primarily dominated by translated code execution costs.

Transformation overhead is determined by the run-time compilation steps and the amount of code that requires run-time compilation. The cost varies based on the execution characteristics of the application. Programs with large footprints, such as *176.gcc*, tend to stress the infrastructure more heavily than their counterparts like, *181.mcf, 164.gzip and 256.bzip*, all of which have relatively smaller code sizes (small footprint) and fewer executed unique paths through code (less control intensive).

Start-up (initialization) transformation overhead: Figure 1 shows the usage of Pin's Just-In-Time (JIT) compiler infrastructure by the *176.gcc* benchmark for all five of the SPEC reference input sets. The vertical axis is time and the horizontal axis is the number of times the Pin virtual machine was entered across the entire run of the program. A straight line without a data point indicates that time is being spent in the code generated by the JIT (translated code), where as a point signifies the JIT is compiling a new program path that has not previously been seen (code transformation). From the figure it is evident that a lot of time is spent compiling new code paths. Only near the end is the majority of the time spent executing the translated application. This data corroborates a 3x slowdown for *176.gcc* in the Pin system. Repeated code transformation degrades performance since the effects of invoking the JIT on the architecture are similar to a process context switch. The insight gained from this data is that a benchmark with a large footprint will consume a substantial transformation time and that a significant amount of transformation time is spent on infrequently executed code. Another interesting characteristic evident from the figure is that the compiled application exhibits identical transformation behavior in the usage of the JIT compiler for the different inputs. Only at around JIT invocation 23000 do the different inputs to

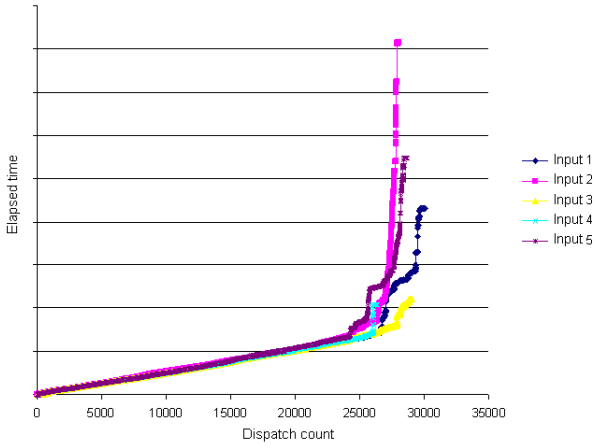

Fig. 1: Time spent in the Pin system for *176.gcc* benchmark for the various reference SPEC input sets. Every point represents the compilation of a new trace. A straight line reflects the time spent running translated code.

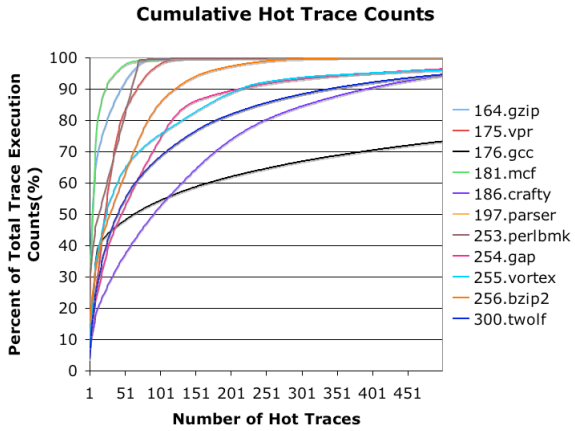

Fig. 2: Cumulative percentage of the number of traces that dominate program execution.

Graphic Applications	*Translated code execution time (%)*	*Transformation time (%)*
Gnumeric	4%	96%
Emacs	7%	93%
Xpdf	26%	76%
Wts (MT)	42%	58%
Gtv (MT)	87%	13%
Xlogo	14%	86%
Gv	10%	90%

Table 1: Distribution of translated code execution time versus transformation overhead for the startup phase of graphic applications

176.gcc deviate and cause different code transformation behavior.

Infrequently executed code transformation overhead: Another source of transformation overhead for run-time systems is the result of a trait common to most applications: cold code - code executed once or a few times. Cold code execution is a challenging problem for run-time systems because such systems rely on amortizing the cost of the transformation overhead by repeated executions of the translated code sequences. Figure 2 shows the sorted distribution of code sequences generated by Pin's JIT compiler to cover the complete execution of each program. It is evident from the data that most of the execution time is spent in just a few traces, except in a few applications. On average, benchmarks spend the majority of execution time in fewer than 150 traces. The *176.gcc* application has the largest code footprint, and therefore requires the most individual traces (greater than 500) to cover its entire execution.

Table 1 shows the distribution of the time spent executing the translated code versus generating it for the startup phase of graphic applications. A significant amount of time is spent transforming the code; even a faster JIT compiler will not be able to overcome this transformation overhead because the code consists of paths that are executed infrequently.

From Table 1 and Figure 2 it can be concluded that the transformation overhead for Pin is due to a large number of infrequently executed traces requiring transformation. Addressing this problem is therefore critical to improving the performance of code transformation systems.

The work in this paper aims at reducing the overhead of the transformation cost in a run-time code transformation system. The suggested strategy to minimize the overhead is to cache the program executions at the end of the run to a database maintained by the system. This methodology is referred to as *Persistence*. In [5] it has been shown that applications tend to share common code paths across multiple invocations. Thus it is believed that our approach is a practical and effective solution to minimize the incurred transformation overheads.

To evaluate persistence and its benefits, a working solution has been implemented in a binary instrumentation system. Persistent caching of instrumentation is particularly beneficial for large software systems under development. Complex software systems are usually put through some form of daily regression tests to ensure that development changes to the source do not break the application. They are fed with multiple input sets to ensure enough of the code is touched to ensure robustness. Such applications, due to their complexity, tend to stress the instrumentation systems aggressively. For instance, a database program under a memory checking Pin Tool experiences a 1.4x slowdown simply due to transformation overhead. Separate invocations of the program with different input sets would all therefore result in an identical transformation slowdown. By using persistence, only the first invocation of the application needs to incur the performance penalty. The rest of the invocations may reuse a cache generated from the first invocation. This results in incurring minimal amounts of transformation overhead as a result of any new paths taken by the application due to the input variation.

3 Persistence

3.1 A persistent run-time system

Figure 3 illustrates the control flow of a persistent run-time system. When the system is started, it may proceed with some basic initialization steps. Past the initialization phase, it checks to see if a prior execution exists in the persistence cache database. If a prior execution cache does not exist it proceeds to initialize modules that are persistent-specific, such as memory allocators that allocate space for run-time data structures and the translated code from memory pools that are live across application invocations. These memory pools are cached in the persistence database and are made available for reuse when the same binary is re-invoked.

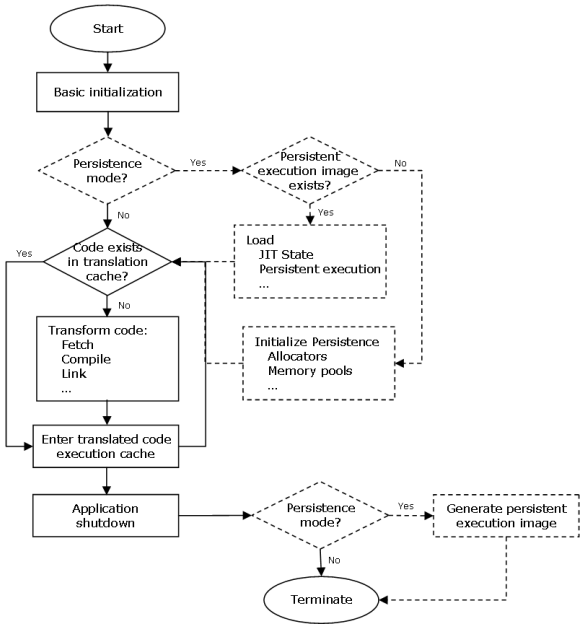

Fig. 3: Overview of a persistent code transformation system. Dotted lines indicate Persistence specific states.

Past the initialization phase, the system defaults to steps normally taken by any dynamic binary code transformer such as those taken by systems explained in Section 2. At the end of the execution, the translated code sequences and their relevant datastructures are stored in the persistence database.

Subsequent invocations of the system on the same application trigger the system to check the database for a prior execution instance that has been cached. If one exists, the system initializes itself with the required state

that allows it to reuse the cached execution and proceeds. Program paths previously seen demand no code transformation and translation since the paths already exist in memory. The system gets invoked only for new execution sequences. At the end of the program, all new code sequences generated may be appended to the already cached execution in the database or a new persistent cache file may be created.

3.2 Challenges

Realizing persistence in a dynamic system lends itself to many complex problems that need to be addressed to make it a practical solution. In this section, some of the fundamental challenges that are believed to be generic to any DCTS are elaborated.

Consistency: A cached execution cannot be reused if the application binary has been modified since its last invocation. If the application has been modified then all the cached executions for that particular application have to be invalidated and a new persistent cache file has to be generated. Identifying changes requires a signature for the current execution instance. This signature has to be generated and verified prior to executing the first instruction of the cached execution. Signatures may be generated using generators such as *md5sum* and *sha1sum*.

Randomized address space (RAS) [9]*:* Operating systems that support randomized address space are capable of loading shared libraries at different addresses across executions. This is a problem since all run-time systems maintain a translation mapping between the original and translated instructions. A scenario where this is problematic for a system is when two shared libraries, A and B, originally loaded at addresses X and Y, are swapped and loaded at addresses Y and X respectively in the second run. This interchange will break the application in the second run if a cached execution is reused. This is because the mappings will be incorrect - if the first instruction address of A is looked up in the cached execution, it will incorrectly return the translated instruction address as X when it is really Y during the second run.

A possible solution is to update the mappings at program start-up time so that all mapping lookups return valid results for the current execution instance.

Absloute addresses in translated code: The loading of executables at different addresses across executions creates yet another problem that is specific to translated instructions. For example, a run-time system may translate a `CALL 0x8048494` instruction into a `(PUSH 0x8048499, JUMP 0x8048494)` pair (the `PUSH` instruction is placing the return address onto the stack) to maintain transparency. If the `CALL` instruction is relocated in a subsequent run due to RAS then the literal in the `PUSH` instruction needs to be updated to reflect the new return address after the `CALL` instruction.

A possible solution is to generate translated code that is of the position-independent-code form, so that regardless of where the code is loaded, the translated code will work correctly. Another solution is to generate relocation entries for the translated code and to fix-up the code prior to execution. This technique is similar to what the loader does at program start-up.

Memory constraints for the run-time system: Translated code is cached in a special area of memory in the address space. Applications that have a very large footprint tend to exhaust the allocated space quickly. Most systems respond to this by reclaiming the space allocated for all the code translations generated in the current instance. If the execution is cached only at program termination, it will limit the performance of persistence because all the paths initially seen will not be in the cached version. Therefore, in the subsequent runs the system will have to regenerate the lost paths which results in transformation overhead again.

Rather than losing the paths, prior to space reclamation, it is better to generate a persistent cache every time the allocated space is being reclaimed. These multiple caches can be reused individually by the system in later executions.

Self modifying code: Code that dynamically modifies itself cannot be cached in the persistent database if the cache is generated only at the end. This is because it only contains the final version of the code which may have been modified through the life-time of the program.

Generations of code may have to be maintained in the persistent executions so that they may be swapped in when SMC is detected.

Optimization: Certain optimizations performed by the code transformation system during one execution instance cannot be propagated across executions since they might be dependent on the inputs. For example, constant propagation often tends to be input dependent. Therefore, in order to reuse an already existing execution the inputs may have to match, in case the system applied that optimization.

Persistent run-time system design: While the above are all important challenges a persistent run-time system must handle, the design of the system itself cannot be overlooked. Persistence relies not only on the state of the application but also on state that corresponds to the run-time system. Most systems are developed simultaneously by multiple coders. Requiring all coders to comprehend persistence and to cater for it is likely to diminish the rate of development and increase system complexity. Therefore it is important to design the system in a manner that its presence is constrained in the codebase.

Object oriented programming features have proven to be an important concept to exploit. They ease the implementation of persistence in a system that is being actively developed. Developers only had to register their classes with a persistent memory manager which ensured that run-time objects were cached properly and available for accesses/modifications in subsequent runs.

4 Evaluation of Persistence

4.1 Persistence in a binary instrumentation system

Pin is a JIT based instrumentation engine that supports instrumentation on the IA32, EM64T, IPF and XScale platforms via the use of Pin Tools that export a rich user interface for performing application introspection. Pin performs various optimizations such as code caching, trace linking, inlining, register allocation and liveness analysis on the generated code to minimize the overhead incurred at run-time.

Persistent Pin (PPin) is specifically designed to reduce the overhead of dynamic instrumentation. The overhead

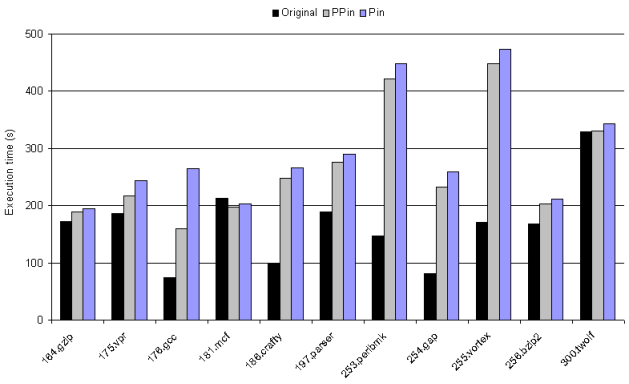

Fig. 4: Persistent Pin's performance in comparison to native and Pin's performance

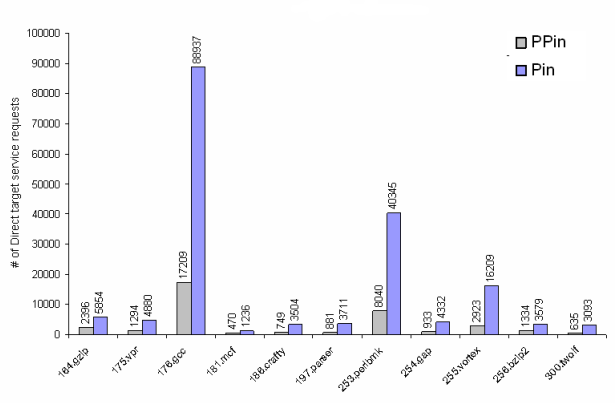

Fig. 5: Pin service requests from the translated code.

is the result of Pin generating new traces for paths being executed by the application that have not been seen yet and for generating the required instrumentation introspection code. The impact of the overhead on the architecture may be viewed as being similar to the effect of a regular operating system context switch. The overhead reduction is a two-step process that involves a warm-up phase of generating a cache file on disk that is later made available for reuse in subsequent invocations of the application under Pin, with either an identical or varying input. A cache file consists of the translated code in memory and the necessary data structures to support code reuse across executions. Thus by reusing as much of the code as possible from a prior run, PPin guarantees smaller overhead because paths that have been seen before will not require Pin's compilation.

PPin was evaluated on SPEC and everyday applications. The latter is a suite of interactive graphics applications comprising of a spreadsheet, text-editors, ps/pdf viewers, a virtual desktop manager and a media player. The last two are multi-threaded applications. They were chosen to characterize and reflect their more aggressive and demanding start-up behavior in comparison to SPEC programs, which are poor indicators of everyday application characteristics on DCTS. The results were gathered on an Intel 3.0GHz machine with 2GB main memory running RedHat 7.0 operating system. While Pin supports various platforms we have evaluated our preliminary design only on the IA32 platform.

The experiments performed are divided into two groups to characterize the (1) effectiveness of persistence in reducing the overhead over the lifetime of programs and its (2) effectiveness in minimizing start-up costs. The motivation for doing this is to clearly present the benefits of persistence.

4.2 Overhead reduction over program lifetime

Traditional dynamic code transformation systems rely on amortizing their overhead by repeated executions of the code already translated in the current execution instance. Therefore, once the system covers enough of the program footprint, its overhead becomes negligible. Thus, it is essential to carefully analyze how effective persistence is over the length of the program execution.

The first evaluation of persistence in Pin was to run Pin without instrumentation. Without instrumentation, Pin is highly representative of a generic DCTS. This can be seen as Pin functioning as a native-to-native translator with Pin's default optimizations and transformations. Some transformations are for transparency and incur overhead.

Figure 4 shows the execution times of the SPEC benchmarks running the original binary, under Pin, and using an already cached uninstrumented execution (PPin). Only SPEC Integer benchmark performance is reported since it has been shown in [7] that Pin does affect SPEC Floating point benchmarks performance significantly. The data shows two things. First, PPin is effective across all benchmarks in reducing the code transformation overhead. Second, the performance improvement of PPin is limited by the performance of the translated code.

The code transformation overhead reduction is confirmed by the reduced number of Pin service requests evident in Figure 5. A service request occurs when Pin is called to generate new code paths and handle system calls. In PPin, these services are still likely to occur based on the execution characteristics of the application and the environment. *176.gcc* benefits most from persistence with an improvement of 30% in execution time. This shows that Persistence is most effective on benchmarks that are complex in control flow with large footprints since that tends to stress the infrastructure heavily during the code transformation phase as shown in Figure 1. Other benchmarks do not exhibit dramatic gains because they have relatively smaller footprints. Small footprints are easy to work with for code transformation systems because once the code is translated and cached in memory, the system is not frequently invoked for code transformation. Therefore, it can easily amortize the initial cost over repeated executions of the cached code sequences.

253.perlbmk and *255.vortex* benchmarks also show dramatic reductions of 80% in service requests. However Figure 4 does not confirm execution time improvement. This is because the applications are affected by the performance of the translated code. The two benchmarks have a very high dynamic execution of indirect branches. Indirect branches are challenging to handle in a dynamic code transformation system and contribute a significant amount of overhead. The challenges in handling such branches are elaborated in detail in [2] and [7]. This is not the task of the proposed solution and thus is not

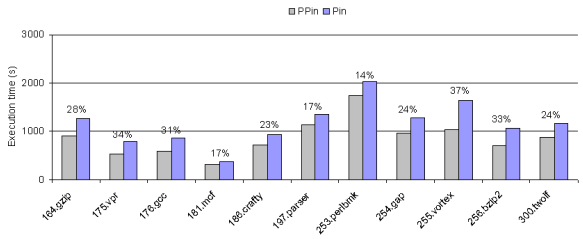

Fig. 6: Execution time for BBL instrumentation using Persistent Pin vs. Pin. Percentage improvements of PPin over Pin are shown.

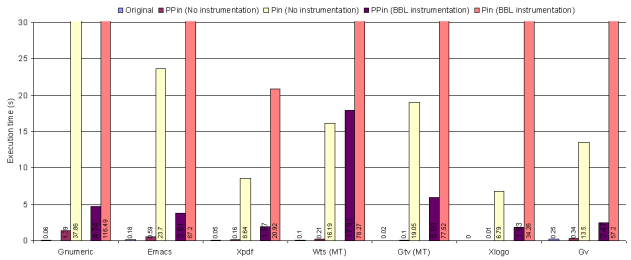

Fig. 7: Execution time of Persistent Pin vs. Pin for BBL instrumentation of everyday applications. Applications were started and immediately shutdown, this is the worst case scenario and illustrates the benefits of caching cold code.

discussed further.

Proceeding to the evaluation of persistence in instrumentation, basic block (BBL) profile instrumentation was applied to the SPEC Integer benchmarks. Figure 6 indicates an average saving of 26% execution time over the regular Pin by re-using cached executions for subsequent invocations of the program. This proves that the caching of instrumented program executions is beneficial and worthy of further investigation.

4.3 Start-up cost reduction

The start-up phase is the most difficult part for any DCTS because the system is constantly invoked to generate new code paths that have not been seen yet. Applying DCTS to interactive programs is difficult because users do not tolerate especially long pauses at start-up. This slowdown is a result of a lot of shared library initialization, cold code that is bad for DCTS because there is no way to amortize the overhead incurred during the start-up phase.

Figure 7 shows the execution time to start the graphic programs and to shut them down once they are completely ready for user interaction for the cases when no instrumentation is applied as well as when basic block profile is collected. The run-times show that persistence is highly effective in handling the start-up overhead with an average improvement of 90% execution time savings for both instrumented and uninstrumented executions. This dramatic improvement is possible because graphics programs exhibit tremendous amounts of initialization sequences. For instance, Gnumeric is a Linux spreadsheet program that relies on interacting with over 50 shared libraries and Emacs with 18 libraries. All of these libraries have to be loaded into memory and initialized to be ready for user interaction. This initialization code is rarely reused and thus, this overhead cannot be amortized by any DCTS that does not use persistence-like services.

5 Conclusion

In this paper, Persistence is proposed as a solution to reduce the overhead incurred by dynamic binary code transformation systems. Persistence is the process of caching executions in a database to be re-used in subsequent invocations.

Persistence was evaluated in the Pin run-time binary instrumentation engine, to create a system known as Persistent Pin (PPin). PPin is capable of caching instrumented program executions and reusing the cached instrumented executions in separate invocations. It is also capable of caching new code paths in subsequent executions if the application changes behavior from prior executions and supports multi-threaded applications. The Persistence model has proven to be very effective for Pin. Experimental data shows code transformation overhead reductions of up to 90% for interative graphic program start-up costs and up to 25% for SPEC benchmarks for instrumented executions.

References

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno JVM. In *Conference on Object-Oriented*, pages 47–65, 2000.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[3] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275. IEEE Computer Society, 2003.

[4] 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.

[5] 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*, pages 51–58, San Francisco, CA, June 2003.

[6] 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 *Proceedings of the 36th International Symposium on Microarchitecture(MICRO-36)*, December 2003.

[7] 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 *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, June 2005.

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

[9] PAX. Web site: http://pax.grsecurity.net/.