Deploying Dynamic Code Transformation in Modern Computing Environments

by

Vijay Janapa Reddi

B.S., Santa Clara University, 2003

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

2006

This thesis entitled:
Deploying Dynamic Code Transformation in Modern Computing Environments
written by Vijay Janapa Reddi
has been approved for the Department of Electrical and Computer Engineering

_____

Professor Daniel A. Connors

_____

Dr. Robert S. Cohn

_____

Professor Andrew Pleszkun

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Janapa Reddi, Vijay (M.S., Computer Engineering)

Deploying Dynamic Code Transformation in Modern Computing Environments

Thesis directed by Professor Daniel A. Connors

Dynamic code transformation systems are steadily gaining acceptance in computing environments for services such as program optimization, translation, instrumentation and security. Code transformation systems are required to perform complex and time consuming tasks such as costly program analysis and apply transformations (i.e. instrumentation, translation etc.) As these steps are applied to all code regions (regardless of characteristics), the *transformation overhead* can be significant. Once transformed, the remaining overhead is determined by the *performance of the translated code*. Current code transformation systems can only become part of mainstream computing only if these overheads are eliminated. Nevertheless, certain application and computing environments exist in which code transformation systems can be effectively deployed. This thesis identifies two such environments, persistence and mixed execution.

Persistence leverages previous execution characteristics to address the transformation overhead. This is accomplished by capturing the translated executions at the end of their first invocation. The captured executions are cached on disk for re-use. All subsequent invocations of the run-time system using the same application cause the system to reuse the cached executions. Since applications exhibit similar behavior across varying input data sets, this execution model successfully diminishes the transformation overhead across multiple invocations. Persistence in the domain of dynamic binary instrumentation is highlighted as an example.

Mixed execution accepts that the performance of the code generated by today's code transformation systems is in no position to compete with original execution times. Therefore, this technique proposes executing a mix of the original and translated code

sequences to keep the translated code performance penalties within bounds. This execution model is a more effective alternative to pure Just-in-Time compiler-based code transformation systems, when low overheads and minimal architectural perturbation are the critical constraints required to be met. A dynamic compilation framework for controlling microprocessor energy and performance using this model is presented in light of its effectiveness and practicality.

Contents

Tables

Table

Figures

Figure

Chapter 1

Introduction

Dynamic code transformation systems have the potential to impact the design and use of modern computer systems since they can provide a number of services at run-time, such as instrumentation, optimization, translation and security. These systems have an inherent advantage over static techniques, as they can collect and exploit run-time execution characteristics. The information collected can be used to adapt the execution of the target application. However, since the execution of current run-time system is generally interleaved with the execution of the application, there is a substantial overhead penalty. In the context of performance analysis and program behavior tools, the overhead of a dynamic code transformation system may be acceptable. Nonetheless, execution penalty is a major barrier towards deploying run-time code transformation in computing systems, specifically in use in large-scale application environments.

While prior works provide considerable insight into the potential of run-time adaptation of applications, such work provides only limited insight into the elements that are vital to the practical deployment of traditionally-proposed run-time systems. Other code transformation models exist such as the use of multi-core hardware and special-purpose architecture-support for evaluating the full potential of code transformation systems; however this thesis centers on the deployment of run-time transformation in a software-based domain. In this context, program behavior and the design of the code

transformation system have the largest impact on the execution overhead incurred by an application under a run-time system. The resulting overhead is an aggregate of the (1) *transformation cost* and (2) the penalty due to the poor *translated code performance*.

Transformation overhead is determined by the complexity of run-time analysis and transformation components applied to the code (run-time compiler, register allocator, optimizer, cache manager etc.). Likewise, the overhead of each system component is not fixed, but depends on the run-time characteristics of the application. For instance, at startup, most programs have tremendous amounts of code that are executed once or only a few times. A dynamic code transformation system will not be able to amortize the run-time overhead costs for these code sequences. Run-time systems rely on the repeated execution of the translated code sequences to amortize their overhead.

The performance of the translated code is controlled by factors such as whether optimization is applied or instrumentation code is added during transformation. In addition, the code generated by a run-time system often varies from the original instruction stream to make sure that the application remains under the control of the run-time system at all times. Likewise, care can also be taken to ensure that the application is not perturbed by the presence of the run-time system. These changes to the instruction stream often contribute to a significant amount of overhead.

It is imperative for researchers to understand the resulting transformation and translated code performance overheads as they are currently beyond tolerable. Only when run-time overheads break even with original execution times can current code transformation systems become integrated into mainstream computing environments. However, certain application and computing environment characteristics exist that can still benefit from the deployment of current code transformation systems.

This thesis presents two models of code transformation that exploit unique computing scenarios: *persistence* and *mixed execution*. Both models leverage aspects of an application and run-time system dealing with execution overheads. These executions

models are evaluated in a state-of-the-art Just-In-Time (JIT)-based code transformation system to mitigate the transformation and translated code performance penalties. The conclusion made, based on their evaluations, is that existing dynamic code transformation systems can be deployed into mainstream computing environments when specific execution properties and conditions exist.

Persistence leverages an application facet; applications exhibit very similar execution paths across varying input data sets. The persistent code-transformation model mitigates transformation overhead across executions by caching translated executions. Translated code sequences, captured at the end of the first execution, are cached on disk for re-use. All subsequent invocations of the run-time system using the same application cause the system to reuse the cached executions. The Persistence execution model in the domain of dynamic binary instrumentation is highlighted as an example using the Pin [31] framework. The approach successfully mitigates the overhead of SPEC2K applications, by as much as 25%, and over 90% for everyday applications like web browsers, display rendering systems, and spreadsheet programs.

Mixed execution addresses a challenging aspect of run-time systems which is poor performance of translated code. The motivation behind this execution model is that code executed as a mix of the original and transformed code sequences can avoid many pitfalls of a fully translated code execution sequence such as the removal of costly return and indirect branch handling sequences. Therefore, mixed execution is a practical alternative to pure Just-In-Time compiler-based code transformation systems when extremely low overheads and minimal architectural perturbation are critical constraints. The mixed execution model is evaluated in the Pin [31] run-time system addressing the rising concern of microprocessor power consumption. It exhibits minimal overheads, which are as low as $\sim$4%. Consequently, the run-time power-saving optimizer successfully achieves energy savings up to 70% for applications in the SPEC2K benchmark suite, with an $\sim$0.5% performance degradation.

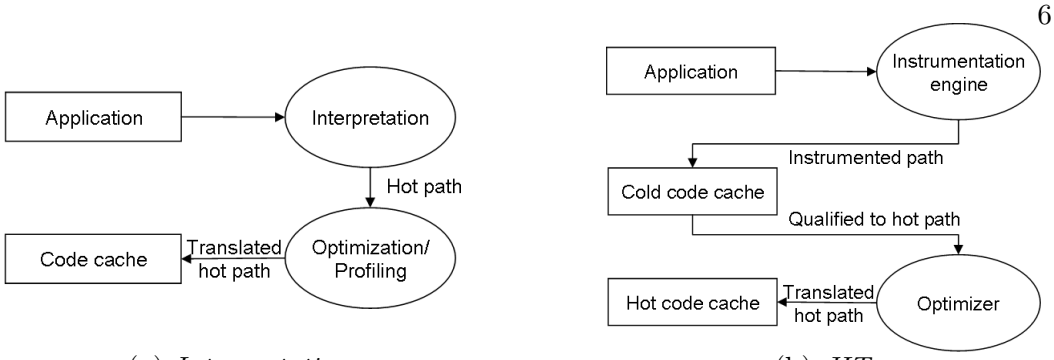
The remainder of this thesis is organized as follows: Section 2 presents background into run-time code transformation systems and describes their fundamental barriers. Section 3 characterizes the various challenges faced by these systems. Two distinct models of addressing these barriers are proposed and evaluated in Section 4 and Section 5. Finally, Section 6 draws a summary of the work accomplished in this thesis.

Chapter 2

Characteristics of dynamic code transformation systems

Existing dynamic code transformation systems rely on a Just-In-Time (JIT) compiler, or an interpretation system to follow the dynamic execution path of a program. Figure 2.1 illustrates how dynamic optimization is applied in either of the two methods of dynamic code transformation. In the Interpretation model, shown in Figure 2.1 (a), instructions are emulated while at the same time profile information is gathered inside the interpreter. Interpretation's advantage is that it is relatively simple to implement, especially in the case of software translation systems [26, 14] where the input ISA is different than the host ISA. However, interpretation generally is a slow process. In the JIT model, shown in Figure 2.1 (b), a run-time compiler is utilized due to its inherent ability to generate more efficient code sequences. Since all JIT approaches work on the granularity of at least a basic block, there is more scope for run-time optimizations. Such is not the case in the interpretation model since the scope is limited to just one instruction.

Generally, these systems disrupt the target program execution with multiple steps to initiate, monitor, and consider every code sequence for optimization/transformation. As the primary goal of these systems is to perform their task under execution time constraints, implementing practical systems based on these run-time models can be challenging in certain commercial and scientific environments that contain a substantial



(a) *Interpretation* (b) *JIT*

Figure 2.1: Overview of (a) interpretation and (b) Just-In-Time (JIT) based dynamic optimization systems.

number of unique execution paths.

2.1 Applications

This section presents background into existing run-time code transformation systems. The section first motivates the reason for performing the tasks at run-time versus at static compile time, followed by referrals to seminal works.

2.1.1 Optimization

For years, the steadily growing clock speed of new processors has been relied upon to consistently deliver increasing performance for a wide range of applications. However, processor frequency scaling has stalled, driving microprocessor companies to investigate adding value through alternative technologies. To continue making gains in system performance, the overall system must dynamically adjust resource allocations to meet system needs. Unfortunately, existing systems do not provide such support. Without such support, static compilers, due to their inherent limitations, fail to realize the interaction between the hardware and application the code. In fact, aggressive compile-time optimization may even lower performance [46] when run-time characteristics and architecture resource constraints are not accurately addressed. Overcoming such issues

involves embracing new architecture and software ideas, rather than simply attacking the problem by scaling hardware resources. As such, the need for run-time system support to adapt and optimize application behavior to hardware resources is increasing across each generation of emerging systems.

Taking advantage of the run-time behavior has been the motivation behind many recent works in the area of dynamic optimization (code transformation) [3, 6, 11, 29]. Generally, dynamic optimization refers to the process of modifying an application binary at run-time to promote desirable execution characteristics, such as high performance [6, 11, 3], low power [38, 44], or managed resource consumption [21]. Traditional dynamic optimizers first observe run-time characteristics to determine frequently executed sequences of code for performing optimizations within a given instance of execution. These sequences span procedure boundaries as well as cross into shared libraries and therefore have more potential for optimization.

There are several successful implementations of dynamic optimization systems such Dynamo [3], DynamoRIO [6], Mojo [11], DELI [13], and ADORE [29]. DynamoRIO, Mojo and DELI are examples where execution is initially done from a code cache where code is being monitored for optimization opportunities. Once an execution threshold metric (hot code detection) is exceeded, the systems go into code specialization mode of laying out a single-entry multiple-exit code sequence for the detected hot code region. The threshold identification metrics rely on software mechanisms that promote code from the initial cache to the optimized cache. DynamoRIO[6], an extension of Dynamo, supports applications in the Windows and Linux environments. These two optimizer infrastructures primarily focus on reducing the execution time of the application by identifying hot sections of the code and optimizing them to make the programs run more efficiently. The general performance improvement technique is centered on code layout optimizations. ADORE [29] is a dynamic optimizer that uses the hardware performance monitoring (HPM) system of a processor to detect hot code sequences

and memory bottlenecks. The hardware information replaces the need for dynamic instrumentation and significantly lowers the overhead of the run-time code transformation system. ADORE is best known for dynamically deploying software prefetch instructions through the run-time identification of cache bottlenecks.

Feedback-directed optimization (FDO) also falls into the domain of optimization. It describes any technique that automatically alters a program's execution based on the run-time execution tendencies observed in current or previous runs. A good overview of the techniques and challenges of FDO are presented in [40]. Feedback-directed optimization tools such as Spike [12] and Vulcan [41] are static post-link optimizers that can statically optimize application binaries in response to user input and data patterns. Relatedly, Ispike [30] utilizes performance counters available on the Intel Itanium Architecture to automate the optimization between runs. The term *continuous optimization* has also emerged and is centered on performing continuous profiling, and therefore optimizing a program across multiple invocations. DCPI [1] is an example of such a system which continually samples a hardware performance monitoring unit to collect profiles. Kistler [28] explores a similar framework using hardware to collect profiles and continuously optimize programs between different runs.

2.1.2 Translation

Related to dynamic optimization, translation systems facilitate the execution of binaries compiled for one Instruction Set Architecture (ISA) to execute on a different ISA. This is an effective means for companies to migrate from one ISA to another while maintaining backward compatibility.

With the recent announcement that Apple Computer will ship systems with Intel chips using Transitive's Rosetta translation software (to convert PowerPC applications to x86 at run-time) [33], code transformation systems have moved into the mainstream and are no longer an impractical full-system deployment strategy. IA32EL [4], HP-

Aries [48], Daisy [14], Transmeta's CMS [26] and FX!32 [19] are other such translation systems. All of these systems with the exception of FX!32 are JIT based translation systems. FX!32 translates code from the x86 architecture to the Alpha architecture. It profiles the x86 application as it is emulated; afterward, the profile is used to generate optimized native code from the translated x86 code generated during the run. Future invocations utilize the native code, eliminating the full cost of performing profiling and transformation at execution time.

2.1.3 Instrumentation

Understanding application characteristics for debugging or performance evaluation on systems is important to ensure that binaries compiled and shipped to customers function correctly and operate at peak performance. Applications can be analyzed at various stages of their creation - source code, link, or run-time. With the growing complexity of everyday software systems, the method chosen for introspection is critical to effectively and efficiently study the application behavior.

Source code instrumentation is extremely time consuming since the code has to be explicitly added/removed by editing the original program files. This form of instrumentation by definition requires the application source code. This requirement makes it an impractical approach in certain circumstances. Post-link and run-time instrumentation systems work at the binary level. They are classifiable into static or dynamic approaches.

The static approach relies on rewriting the object or executable file with the required instrumentation, thereby generating a new instrumented executable. Most static tools suffer from tool-chain dependence which requires the cooperation of the compiler and/or the linker when the binary is created. Dynamic instrumentation is a more effective and practical approach of instrumenting a binary since it imposes no special requirements.

Dynamic binary instrumentation systems are widely accepted and used as a means of characterizing program behavior for performance, code coverage and program correctness. They are also applicable in payload delivery and target function replacement [24].

Dynamic binary instrumentation is implementable either via (1) binary modification or (2) the recompilation of the application at run-time. Binary modification relies on using probes/trampolines which overwrite the original program instructions to invoke special code that handles the process of instrumenting the application code. The latter does not touch the original code. Instead, it decodes the original instructions and encodes them into a separate area of memory commonly referred to as the code cache, while inserting instrumentation at the specified points.

Dyninst [7], Gilk [37], KernInst [42] and DTrace [2] are examples of instrumentation systems that rely on code splicing to apply the instrumentation. Pin [31] and Valgrind [34] are systems that rely on the recompilation at run-time strategy to perform instrumentation.

2.1.4 Security

Run-time systems are gaining acceptance as the vehicle of delivering security against malicious code execution. All security systems, regardless of whether they are dynamic or static in nature, are bound to impose execution overhead on the application being monitored. However, due to the fundamental design of run-time systems, dynamic security systems can adapt their monitoring mechanisms to be either active or inactive to limit performance degradation. Furthermore, applying run-time system based security has the added benefit that modifications are not required to the host environment. Most successful tools [8, 15, 9, 36] require a customized environment to operate.

2.2 Fundamental barriers

The usefulness of run-time code transformation systems is apparent from the applications presented in the previous section. Yet, these dynamic code transformation systems are not prevalent in everyday systems. The lack of enthusiasm in realizing their potential is because these systems face numerous challenges.

Low overhead: All run-time systems experience overhead since their execution is interleaved with the execution of the application itself. To keep the overhead minimal run-time systems are required to collect dynamic information, as well as perform low-overhead optimizations.

Hot code detection: To keep the overhead low enough, the optimizations performed, or the information collected, must be applied to those sections of application code that are vital for performance and carry out optimizations accordingly.

Efficient optimizations: Upon having identified the hot code, these systems need to carry out optimizations and apply them to the execution run. This step involves two critical aspects: (1) time to optimize and (2) the strength of the optimizations. Both of them are closely tied together; with an increased amount of time, more efficient code can be generated. However, increased time for analysis and optimization can hurt performance if a significant amount of time is not spent executing the optimized code sequences to make up for the time lost generating the sequences. This balance is hard to approximate without information such as the average lifetime of the program.

The work in this thesis aims at addressing the current barriers to state of the art JIT-based dynamic code transformation systems by specifically targeting their *low overhead* issue. The following section explains the framework utilized to study the issue.

2.3 Framework used to delineate and address the fundamental barrier

All the work presented in this thesis has been done using Pin [31]. Pin is a JIT-based instrumentation engine that supports binary introspection on the IA32, EM64T, IPF and XScale platforms via the use of Pin Tools that export a rich user interface. Without applying instrumentation, the system can be viewed as a native-to-native binary translator. 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.

The system is illustrated in Figure 2.2. The core of the system is the Virtual Machine (VM). The VM works by injecting itself into the address space of the application being introspected, thereafter becoming a part of the application. It is not considered a separate process by the host operating system. The system does not execute any instructions out of the original application. Rather it utilizes the JIT compiler to recompile the application dynamically into the code cache. The recompilation occurs in small units of compilation called a *trace*, which is comprised of dynamic basic blocks. The motivation for doing so is explained in Section 3.2.1. Once the compilation is complete for that unit, control is transferred by the dispatcher to the code/translation cache, where the execution of the compiled unit occurs. If an execution path has not yet been previously visited and is about to be executed, control is transferred over to the virtual machine first, which generates the path. Once generated, subsequent executions of that path no longer invoke the VM. Instead, control is directly transferred to the target. An emulation module in the VM exists to assist with corner cases such as handling system calls.

Though Pin supports various platforms, this work focuses only on the IA32 platform. Data about application behavior is gathered using Pin Tools. Run-time system

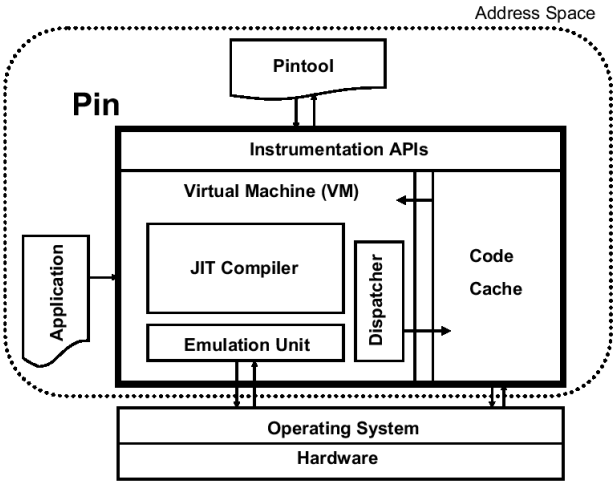


Figure 2.2: Overview of the run-time system utilized as the evaluation framework.

behavior is also evaluated using the Pin infrastructure.

Of all the available run-time systems, Pin is chosen as the subject of evaluation since it is widely used both by industry and academia. Furthermore, it is currently looked upon as the state of the art run-time code transformation system in the domain of run-time binary instrumentation. Since its fundamental design resembles a general code transformation system, any conclusions drawn based on this system are fairly accurate and representative of other run-time code transformation systems. When evaluating run-time system characteristics, no instrumentation is applied unless stated otherwise. All experiments were performed on a system that hosted an Intel 1.7GHz microprocessor with 1GB main memory running the RedHat 7.0 Linux distribution unless explicitly stated.

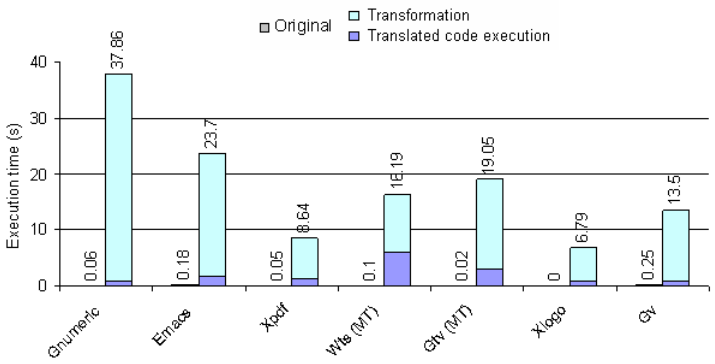
Chapter 3

Challenges faced by state of the art run-time transformation systems

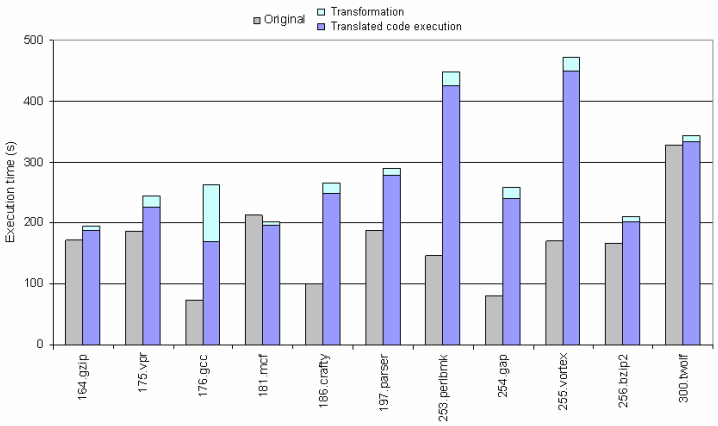
The fundamental barrier to run-time code transformation systems is the overhead incurred during execution. The slow-down incurred by applications running under a run-time system stems primarily from the transformation cost and the performance of the translated code.

Transformation cost: In a JIT-based run-time system, original code is recompiled. This process incurs a certain amount of overhead. Figure 3.1 (a) compares the distribution of the time spent generating/recompiling the code versus executing it for the startup phase of graphic applications. The transformation and translation overheads are stacked on top of each other to illustrate the ratios of where time is being spent. The original startup times are less than a second for all the applications. However, under the run-time system, the applications experience several magnitudes of slow-down. The performance penalty is between 40x to 400x. A significant amount of time is spent transforming the code. When so much time is spent transforming 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.

Translated code performance: This overhead is a result of the performance of the code generated by the run-time system. Figure 3.1 (b) compares the performance of the SPEC2K INT benchmarks against the performance of its translated counterpart. Once again, the run-time system transformation and translation overheads are stacked on top



(a) *Everyday applications*



(b) *SPEC2K INT*

Figure 3.1: Performance of the translated code relative to the original program. Execution time bars of the application running under the run-time system are stacked to show the distribution of the time spent generating the translated code versus executing the translated code.

of each other to illustrate the ratios of where time is being spent. Only SPEC2K INT benchmarks are shown since the applications contain fairly complex control flow relative to the SPEC2K FP benchmarks. The floating point benchmarks experience negligible overheads. From observing the data, it is seen that the overheads vary significantly from one application to another. In applications like *181.mcf* and *300.twolf* the overhead is tolerable. In others, like *176.gcc*, *186.crafty*, *253.perlbmk* and *255.vortex*, the slow-down experienced is over 2x.

Figure 3.1 confirms that both the transformation and translation overheads are

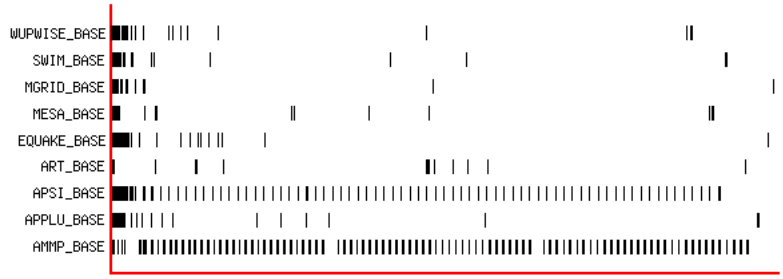
significant and need to be addressed. In the following sections, the transformation and translation costs are broken down to explain their origins. The overheads are explained in relation to the design of run-time systems and the behavior of applications. Following that, Section 3.3 takes a more holistic approach to the source of these problems. It demonstrates the impact of compilers and machine resources on run-time system performance.

3.1 Transformation cost

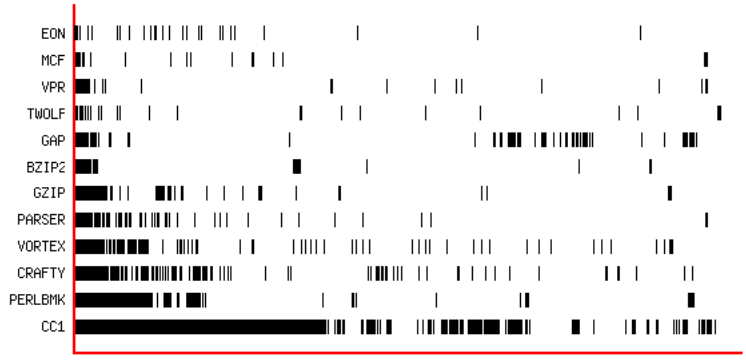
The first form of overhead is determined by the run-time compilation steps and the amount of code that requires run-time compilation. The cost varies based on the execution characteristics of the application. Programs with large footprints, such as *176.gcc*, tend to stress the infrastructure more heavily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

3.1.1 Start-up (initialization) transformation overhead

Figure 3.2 shows the usage of the Pin binary transformation infrastructure by the SPEC2K benchmark suite. They are sorted in ascending order based on how aggressively they utilized the infrastructure. The x-axis represents time while the y-axis corresponds to a unique benchmark. Therefore, a vertical line on the graph corresponds to a service request from the run-time system to either generate a new code sequence or to emulate a selective system call. At this point, the application is making no progress since the run-time system is executing and time is being spent in the Virtual Machine (VM). Space between two lines implies the application is making progress and time is being spent in the code cache. From observing the figure, specifically the SPEC2K INT benchmarks, it is evident that demands on the run-time system vary drastically from one application to another.



(a) *SPEC2K FP*



(b) *SPEC2K INT*

Figure 3.2: Time spent in the run-time code transformation system while executing the SPEC2K benchmark suite. Every black line represents the compilation of a new code sequences. Space between adjacent black lines indicates time being spent in already compiled code paths.

Benchmark *176.gcc* while running under the run-time system experiences significant slow-down of approximately 3x. This is because the benchmark has a very large footprint with minimal code reuse. Therefore, a significant amount of time is spent by the run-time system compiling new code paths. Only near the end of each application run is the majority of the time spent executing the translated application.

Repeatedly invoking the code transformation system degrades performance since the effects of invoking the system are similar to a process context switch from the viewpoint of an operating system. Architectural resources like the cache, translation look-aside buffer, and branch target buffers, which exist to help application performance, are polluted every time the virtual machine is invoked. The insight gained from this data is that a benchmark with a large footprint will consume a substantial transformation

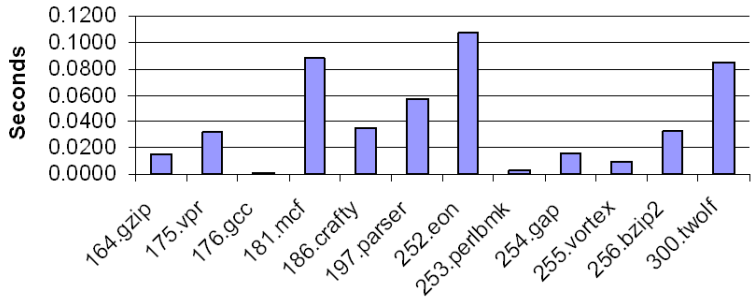


Figure 3.3: Average time between requests from the run-time system's virtual machine.

time. Also a significant amount of transformation time is spent on infrequently executed code.

Contrasting to benchmark *176.gcc* is benchmark *181.mcf*, which experiences no execution overhead while running under the run-time system as shown in Figure 3.1 (b). This benchmark performs well because it has a small working set that is quickly captured. As such, the run-time system is not invoked frequently. This is evident from Figure 3.2; the benchmark has fewer vertical lines. The same traits are also evident in benchmarks like *252.eon*, as well as the entire SPEC2K FP suite. Such characteristics are ideal for run-time systems. The initial overhead incurred due to the run-time system is easily amortized by repeated executions of the already cached translated code sequences.

Data similar to Figure 3.2 is presented in Figure 3.3, which calculates the execution time divided by the number of code cache traces created for each application. The data indicates that the applications with reduced code footprints, such as *181.mcf*, *197.parser*, and *300.twolf* all have longer time intervals between subsequent invocations of the run-time system. On the other hand, benchmark *176.gcc* has a very short time period. Such a short period implies significant demands on the run-time system.

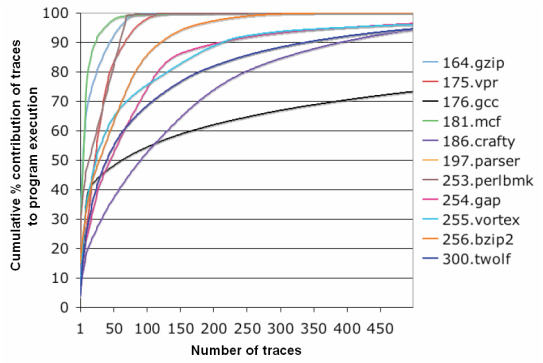


Figure 3.4: Cumulative percentage of the number of traces that dominate program execution.

3.1.2 Infrequently executed code

Cold code, code executed once or a few times, is yet another source of transformation overhead for run-time systems. The execution of such code is challenging for run-time systems because they rely on amortizing the cost of the transformation overhead by repeated executions of the translated code sequences. Figure 3.4 shows the sorted distribution of code sequences generated by the run-time system to cover the execution of each program. 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. Benchmark *176.gcc* has the largest code footprint, and therefore requires the most individual traces (greater than 500) to cover its entire execution. In fact, the run-time system used for evaluation, Pin, generates over 17,000 code sequences that contribute only about 1% to the complete execution. This is because much of the code is infrequently executed and also a result of code layout (explained in Section 3.2.1).

3.1.3 Code transformation components

Figure 3.5 presents the components of the run-time system and their associated overheads when starting up graphic applications. These components describe the com-

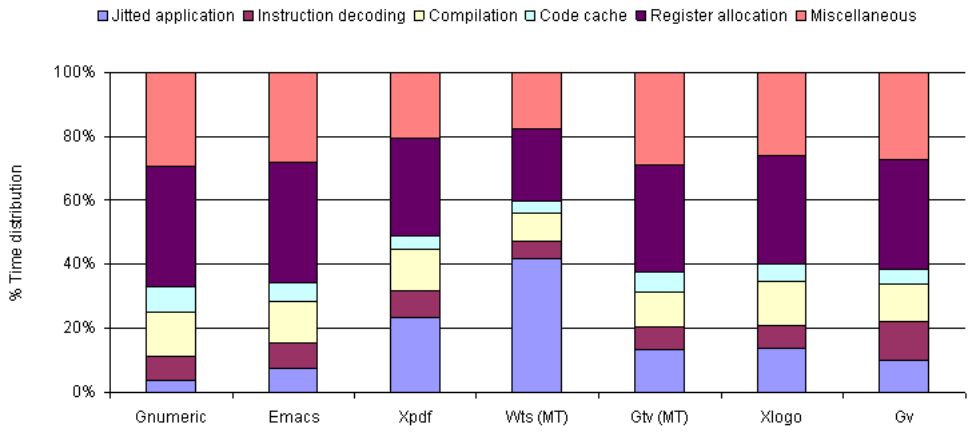


Figure 3.5: Distribution of time spent in components of the run-time system.

plexity of each of the targeted application's ability to be transformed at run-time. By describing the components (stages) in detail, interesting points can be derived about issues with code transformation systems. The description of components are as follows: *Jitted application* is the time spent executing the code generated by the run-time system. *Instruction decoding* is the time to fetch instructions from the main applications text section. *Compilation* is the time to compile the fetched instruction stream from the native form into its corresponding translated form. *Code cache* is the time to perform trace linking optimizations. It also includes other services like those detailed in [17]. *Register allocation* accounts for the time to assign registers to the fetched code sequence so that register liveness properties are identical to the original code. *Miscellaneous* accounts for a series of other events in the system. These events include the allocation of basic blocks, setting up the state required to transition in and out of the dispatcher, the interception of system calls etc.

It can be seen in Figure 3.5 that for the applications shown, on average, less than 20% of the execution time is spent in *Jitted application*. All the others can be attributed to the time spent inside the run-time system transforming the code (i.e. compiling, allocating registers etc). On average, 25% of the time inside the run-time

system is spent performing register allocation. This is a considerable amount of time and contributes a significant amount to the execution overhead.

It is likely to be more efficient if this expensive optimization is applied in two phases. During the first phase, the system opts not to perform expensive register allocation on all the code sequences it generates. Rather, it utilizes the compiler generated register assignments. When the system itself requires registers to perform its task, it spills and fills the registers as required. This approach is likely to function well in comparison to the overhead resulting from this aggressive optimization. Once certain code regions are discovered as being *hot*, the system can regenerate them using expensive register allocation (the second phase). Since these code regions are likely to be executed for a significant amount of time, the time spent optimizing during the second phase can be amortized. This two pass technique is likely to work well with speculative fetching (described in Section 3.2.1) since a significant amount of time is not wasted applying this optimization on code that is never/rarely executed.

Taking a closer look at the miscellaneous costs associated with programs, the system call interception ensures that the run-time system is aware of things such as memory mapping, system calls etc. Some of these have to be emulated and cannot be directly executed. If not, the run-time system may lose control of the application. The emulator executes an instruction manually and lets the run-time system regain control immediately. As such, the application characteristics directly impact the distribution of the run-time system time. For instance, in the SPEC2K suite, programs with a high number of calls to memory management routines, such as *181.mcf* demonstrate higher miscellaneous overhead. This is important to note since this time cannot generally be improved with better code transformation framework design. Rather, is the overhead experienced by ensuring that system calls are obeyed properly. Benchmark *176.gcc* has more complex code, in terms of both functionality and control-flow; the register allocation time for generating code traces is much higher than the average of most other

applications.

In order to build effective run-time code transformation systems, designers must evaluate every component to ensure that they work efficiently. Figures 3.4 and 3.5 indicate that transformation overhead can be large. It can result from various things such as the infrequent execution of translated code sequences or the severe demands of an application on the miscellaneous services. Therefore, addressing the transformation overhead is critical to improving the performance of these systems.

3.2 Translated code performance

While certain applications suffer from transformation overhead, others suffer from poor performance of the translated code. Translated code overhead is determined by a number of factors that include the characteristics of the original code, modifications applied to the code (optimization or 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 may have significant overhead as detailed in the following sections.

3.2.1 Code layout

Overhead can be added based on the layout of the translated code in memory since this determines the I-cache, translation look-aside buffer (TLB), and memory system performance. Most JIT-based run-time systems compile the application code in units, each referred to as a *trace*. A trace consists of a pre-defined number of conditional control flow instructions, a pre-defined number of fetched instructions or it is code selected due to its execution characteristics. If a trace is built without applying prior knowledge of the dynamic execution path of program, it can be said that the trace consists of a speculative instruction stream. This is so because the system cannot determine whether the instructions in the trace will all be executed or not.

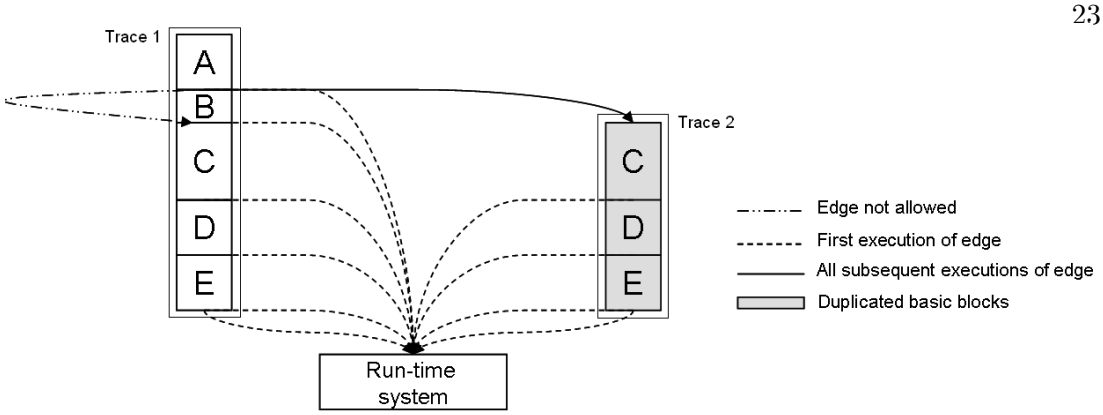


Figure 3.6: Code duplication that results as a result of speculative fetching and trace formation.

Most dynamic code transformation systems speculatively fetch instructions into the compilation unit in order to minimize the overhead of being frequently invoked. Through speculative fetching the run-time systems hope to maximize the time spent executing the translated code. A trace is a single-entry, multiple-exit unit. Therefore, even if the source and target of a control flow instruction are in the same trace, the source cannot branch to the target basic block. A new trace must be generated beginning at the target. These units of compilation are utilized by most systems such as [3, 6, 31, 4].

Speculative fetching and the constraint of a single-entry, multiple-exit code sequence has a negative side effect, which is code duplication. This is illustrated in Figure 3.6. *Trace 1* has basic blocks: *A, B, C, D,* and *E*. The conditional taken path of basic block *A* is basic block *C*. Since *A* and *C* are in the same trace, and *C* cannot be reached from *A*, a new trace beginning at C must be created. Because the principle of speculative fetching is applied, *Trace 2*, that begins with basic block *C*, speculatively fetches basic blocks *D* and *E* prior to reaching its trace termination condition. As a result of the speculative fetching, basic block *C*, *D* and *E* are duplicated in the code cache.

The concept of trace formation is akin to the concept of *superblocks* in the domain of static compiler optimization. However, the consequences of such code duplication

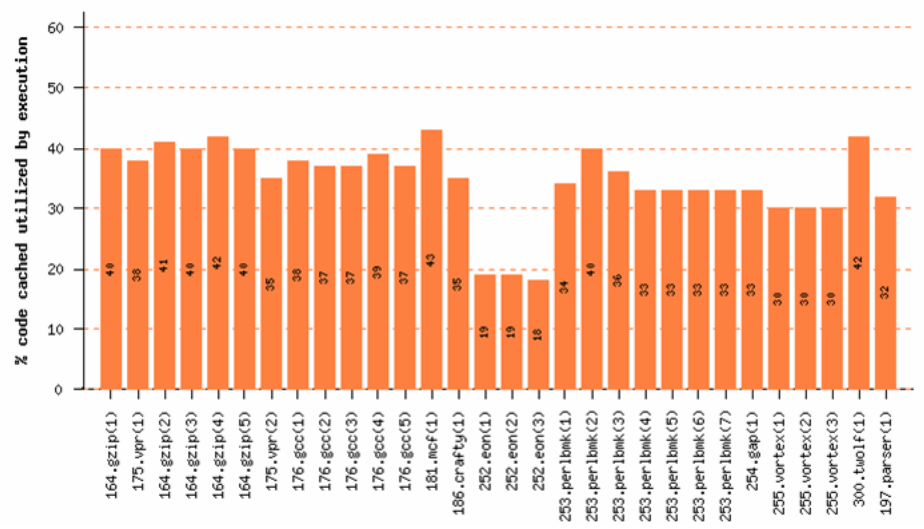


Figure 3.7: Percentage of all the translated application code utilized by the run-time system.

are quite different from its static counterpart. Code duplication in a run-time system implies more time to compile the code, as well as to execute it. The application makes no progress during the compilation process. Furthermore, duplication may also result in code sequences that never executed. The run-time system under evaluation, due to its speculative fetching, utilizes only 40% of the code it generates at run-time, as shown in Figure 3.7. Usage is shown for every reference input set; *164.gzip(1)* refers to the first SPEC2K INT reference input data set followed by *164.gzip(2)*, which corresponds to the second input data set, etc. Another observation about the utilization of the code generated is the consistent behavior of applications regardless of the input data set. Consider benchmark *253.perlbmk*, its cache utilization does not vary significantly over 33% despite the input data set. All benchmarks with multiple input data sets, *164.gzip*, *176.gcc*, *253.perlbmk*, and *255.vortex*, exhibit no variation in code utilization despite the input data set. This observation illustrates the effects of trace creation are independent of the input data set.

The result of speculative fetching is an increase in compilation time. Also, if

the branch out of basic block *A* is frequently taken, the result is worse architectural performance. No I-Cache locality will be exhibited since all instructions brought in following basic block *A* will never/rarely be utilized. If *Trace 2* is located elsewhere in memory, perhaps on a different physical page, the result may be an increase in the execution time due to potential increase in instruction TLB misses.

Figure 3.8 shows measurements made using the PAPI performance counters interface to count the increased number of instruction TLB misses experienced during the execution of an application under the control of a run-time system. Multiple inputs were used to evaluate the impact of the code transformation system on the performance of the TLB. Measurements indicate a severe number of additional instruction TLB misses for programs that exhibit a large number of service requests from virtual machine. At each virtual machine invocation, the application is deterred for long periods of time, waiting for the system to finish its task. By observing multiple inputs for several of the applications, it can be deemed that for some applications like *164.gzip*, *255.vortex* and *256.bzip2*, run-time transformation disruption can be very consistent regardless of the input. Other benchmarks like *176.gcc* and *253.perlbmk* vary dramatically based on the input. As such, the design of modern run-time systems must be careful not to base design techniques on an aggregate value of the characteristics of all applications (e.g. average TLB miss rate). Instead, design decisions need to be evaluated in response to the application behavior based on varying the input data sets, and by comprehending the behavior of an application under a run-time system.

3.2.2 Varying branch target instructions

Code that includes branches with varying targets requires special handling under a run-time system. Figure 3.9 illustrates how varying branch target instructions are handled in Pin. The system utilizes a virtual register, *indirectReg*, to resolve indirect branch targets at run-time. The original instruction, `0x805494b RET` is translated into two in-

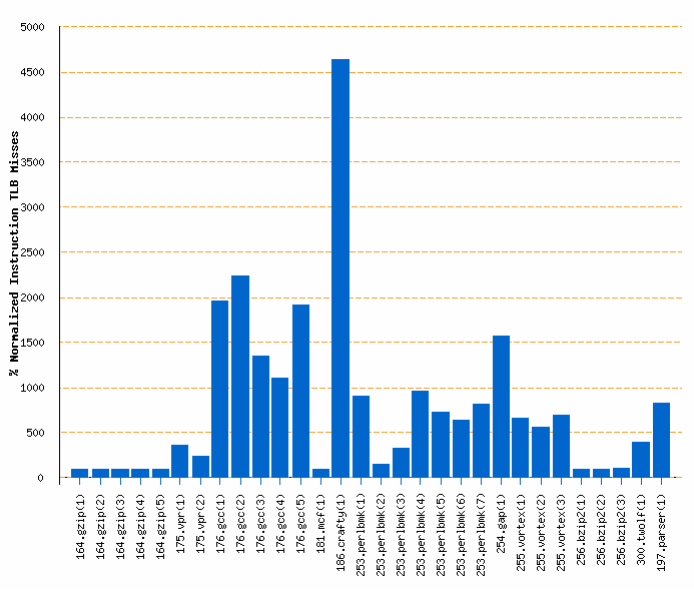


Figure 3.8: Percentage increase in instruction TLB misses when running the applications under the control of the run-time system.

structions: (1) a `POP %indirectReg` instruction to move the return target from the stack into the virtual register followed by (2) an unconditional branch, `JMP 0x78000000`, to the first translated return target trace. `RET` instructions can have multiple return sites, as such, the run-time system must make sure that control is transferred to the correct translated return target. To verify this a check is performed at the entry point of all return target traces. For example, the first three of a return target trace instructions starting at address `0x78000000` verify if the return site is address `0x8054900`. If not, control is transferred to another return target trace beginning at address `0x78002000`. If the current return target does not match any of the previously generated traces, the run-time system is invoked to generate a new return trace. The new trace is added to the end of the list of possible translated return sites for the `RET` instruction at address `0x805494b`. This approach is different from other systems; for instance, the virtual machine is invoked if the first prediction fails (e.g. DynamoRIO [6]).

In the presented approach, if the varying branch target instructions have numer-

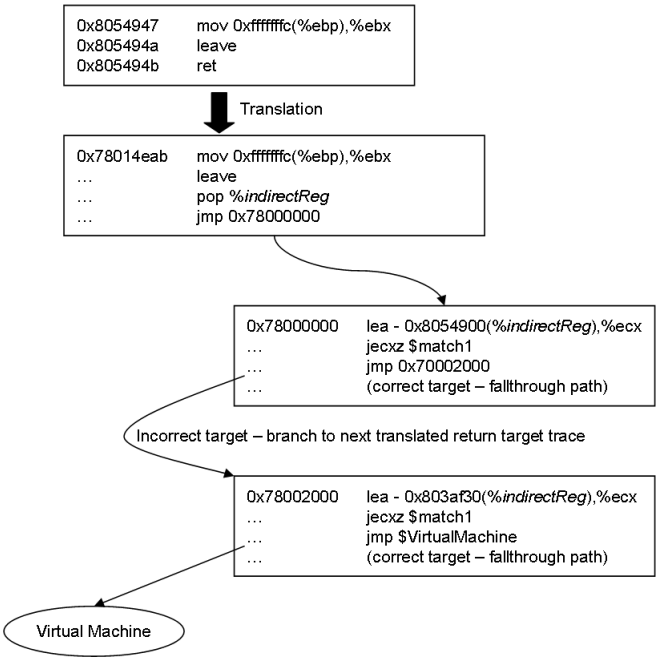


Figure 3.9: Indirect branch predictions.

ous sites they can return to, the list sizes are likely to get large. Long lists are bad for performance; they will consume significant amounts of time to locate the trace that corresponds to the target. If the traces are located across different physical pages, performance may be degraded due to the poor performance of the I-Cache and the I-TLB.

3.2.3 Maintaining application transparency

Most run-time systems implement complex optimizations to ensure that the application being monitored does not alter execution because of the presence of the system. These optimizations come at a price, an increase in the execution overhead.

The run-time system used for evaluation, applies an optimization called *stack-switching*. In this optimization, the system ensures that it does not use the application stack for its own use. This is important to ensure application correctness. For example, certain applications access data beyond the top of the stack pointer. And if the run-time system places its own data on the top of the stack, it is possible to break the program.

While putting data on top of the stack is a matter of program correctness, the system itself must not make the bug visible. Therefore, optimizations like stackswitching are critical despite the overhead (stackswitching results in ∼33% execution overhead). Other optimizations exist that run-time systems implement to enforce transparency (e.g. system call emulation).

3.3 Execution environment effects on run-time systems

Transformation and translation overheads explained thus far are from the perspective of application characteristics and how they relate to run-time system design. Another perspective on the challenges faced by run-time systems is the execution environment. Understanding the host environment of a run-time system can help us understand how to benefit from the potential of these systems. Code transformation and translated code performance can vary significantly based on the environment. Elements of the environment include the compiler used to generate the binaries, the optimizations applied or even architectural features of the machine. The current model of binary generation and delivery needs to be adapted in the future to facilitate the widespread use of these systems. The following sections illustrate that there exists a dire need for us to re-think the traditional model. The sections also make a brief point or two on how the current models can be adapted to cater for these systems.

3.3.1 Compiler vendors

Compiler makers like *Gnu* and *Intel* try to out perform each other by generating efficient binaries. To achieve good performance their binaries are put through various levels of optimization. And once shipped to customers, the binaries are commonly referred to by their parent compiler and the optimization level applied during their generation (e.g. *Gcc 3.2.2* using -O3 flags). The actual set of optimizations applied at a given optimization level may vary from one compiler to another. As such, it is

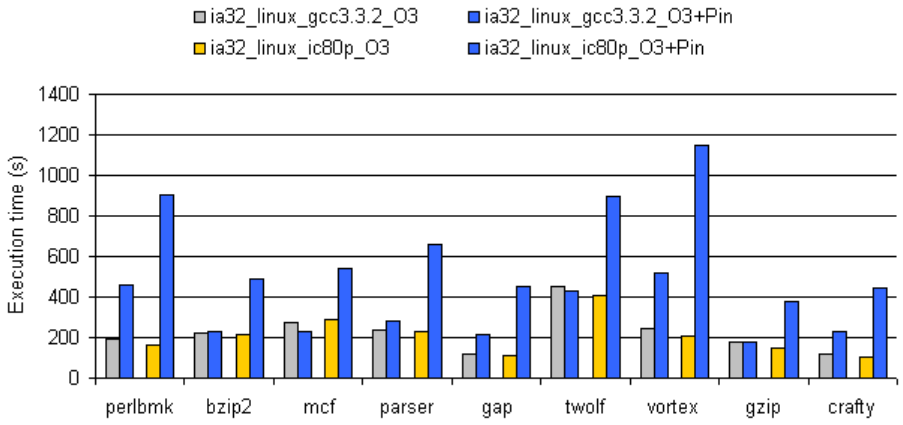


Figure 3.10: Performance impact based on the compiler used to generate the binaries.

important to understand the performance variation of an application under a run-time system when it is generated by different compilers at the same level of optimization.

Figure 3.10 compares the execution time of binaries generated by different compilers with and without a run-time system. The binaries were compiled using the *Gcc 3.3.2* and *Icc 8.0* compilers using level 3 optimizations. Each application has four bars that are grouped into two sets. The first set corresponds to *Gcc* compiled binaries. Of the two bars in the first set, the first corresponds to the execution time of the application running by itself. The second corresponds to the execution time under the run-time system. Correspondingly, the next set of two bars reflect the execution time of binaries generated using the *Icc* compiler running with and without the run-time system.

Running without the run-time system, the original *Icc* binaries perform slightly better than those generated by the *Gcc* compiler. Contrasting to this are the execution times when the applications are running under the control of the run-time system. The times vary significantly between the two compilers. The run-time system performs significantly better on the *Gcc* compiled binaries. Also, when benchmark *181.mcf*, generated by the *Gcc* compiler, is run under the run-time system, it consistently yields better performance than the original run. The application benefits because of good code

layout.

All the *Icc* generated binaries require more compilation and register allocation time. These binaries contain more control flow compared to the *Gcc* compiled binaries. The *Icc* binaries also have more indirect branch instructions. Such instructions require special handling as explained in Section 3.2.2. Figure 3.10 shows that product compilers compete and evaluate their system performance against one another. As such, with run-time systems moving into mainstream computing environments, it is critical that compiler vendors evaluate the performance of their binaries under these systems.

3.3.2 Compiler optimizations

Binary optimizations applied by compilers can affect the performance of a code transformation system. Two sets of binaries were generated using the *Icc 8.0* compiler under varying levels of optimization. Each application contains four bars that are grouped into two sets as shown in Figure 3.11. The first set corresponds to the execution time of the application compiled using the standard O3 flags running with and without the code transformation system. The second set corresponds to the binaries compiled using more aggressive, as well as chipset specific optimizations.

There is not much gain in performance between the two sets of binaries when they are run by themselves without the transformation system. However, the aggressive optimizations have a dramatic performance improvement on the run-time system as shown in Figure 3.11. The results indicate that the system overhead is reduced on average by nearly 50%, and dramatically reduced in cases of *255.vortex*, *253.perlbmk*, and *197.parser*.

To understand this behavior, vortex, compiled using the standard O3 flags is analyzed. It has the worst performance of all benchmarks compiled using the standard O3 flags. *255.vortex* has subroutines that are heavily invoked from numerous callsites. A subroutine `CALL` instruction almost always terminates with the execution of a `RET`

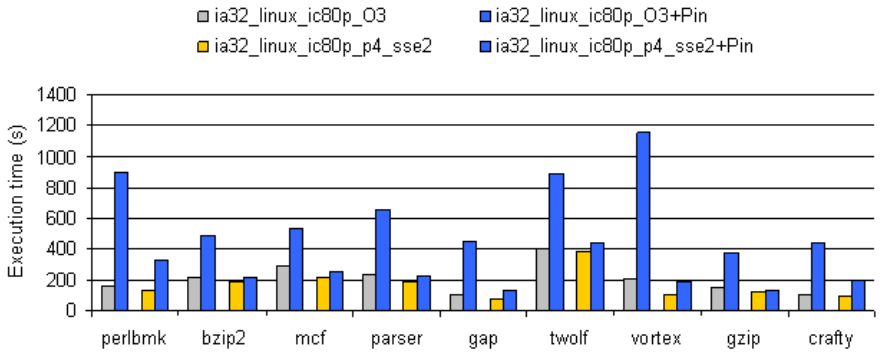


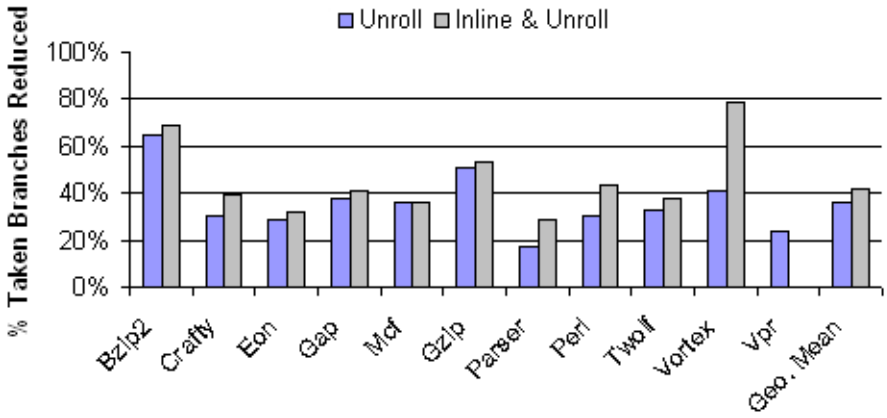
Figure 3.11: Effects of compiler optimizations on the performance of code transformation and translated code execution time.

instruction. Of all the indirect branches executed, the benchmark executes ~94% `RET` instructions. Furthermore, the number of unique callsites to a single function is as high as 233. And as explained in Section 3.2.2, varying branch target instruction requires special handling in a run-time system and are the source of significant overheads.

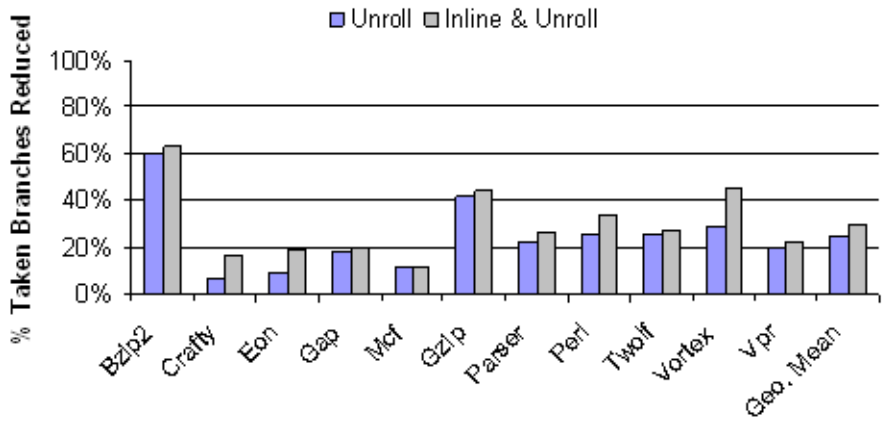
A solution to mitigate the performance overhead resulting from the numerous `RET` instructions is the aggressive inlining of subroutine calls. Aggressive inlining eliminates the `CALL` and `RET` instructions along with other prologue and epilogue instructions. This reduces the number of indirect `RET` instructions to handle. As a result, performance of benchmark *255.vortex* improves by nearly 50%. Therefore, it is extremely critical that the static compiler is aware of the effects of its optimizations on a dynamic code transformation system. Such an awareness can help keep the overheads minimal.

3.3.3 Compiler consequences on dynamic optimization

Run-time systems are being explored to boost the application performance by exploiting execution characteristics [3, 6, 11, 29]. However, it is critical to first understand the scope of dynamic optimization as it stands in todays computing environment (i.e. potential for run-time specific optimizations). Figure 3.12 shows the ability of a Pin-based code optimization system (known as the O-Pin framework) to perform dy-



(a) *Binaries compiled using Gcc*



(b) *Binaries compiled using Icc*

Figure 3.12: Comparison of dynamic optimization (loop unrolling and inlining) on binaries generated using different compilers: (a) Gcc and (b) Intel (Icc).

namic loop unrolling and function inlining on translated code sequences. As illustrated in earlier sections, the compiler can directly impact the application behavior as observed by the code transformation system. For instance, the compiler may have already performed a large amount of optimization. If so, it is possible that the scope for run-time transformations is limited.

Figure 3.12 (a) shows the percent reduction in taken branches using dynamic loop unrolling and inlining on code originally generated using the *Gcc 3.2.2* compiler. While Figure 3.12 (b) show the same percent reduction for code compiled with the *Icc 8.0*

compiler. The *Gcc* compiler generated binaries achieve a 40% reduction in the number of taken branches, while the *Icc* compiler generated binaries achieve only 30% reduction in taken branches. The latter set of binaries exhibit relatively fewer opportunities for the run-time system because they have already been optimized using loop unrolling and inlining.

This leads to an interesting question: is it possible to inhibit potential run-time optimizations due to static code structuring techniques? The static compiler may be able to aid a run-time transformation system without generating code that assumes certain run-time behavior. Overall, the future of proposing any profile-based compilation is at question since the run-time system may be able to eliminate its need.

3.3.4 Machine resources

Microarchitectural features exist to help boost application performance. The L1/L2 cache, Branch Target Buffer (BTB), Trace Cache, Return Address Stack etc. are all such resources. The effectiveness of these resources is determined by the dynamic instruction stream of the application. For instance, the BTB's usefulness is mitigated if the dynamic instruction stream consists of a significant number of branch instructions. An excess of branch instructions (static count) can cause BTB thrashing. Consequently, it hinders the system from capturing the history pattern of certain branch instructions. As a result, more branch predictions are likely to be incorrect. This directly impacts the performance of the pipeline. Incorrectly predicted branch instructions flush the pipeline resulting in an increase of the execution time. Certain static compiler optimizations are aware of this and perform their task accordingly. Along those lines, the following sections present the overall impact of architecture resources on the performance of run-time systems.

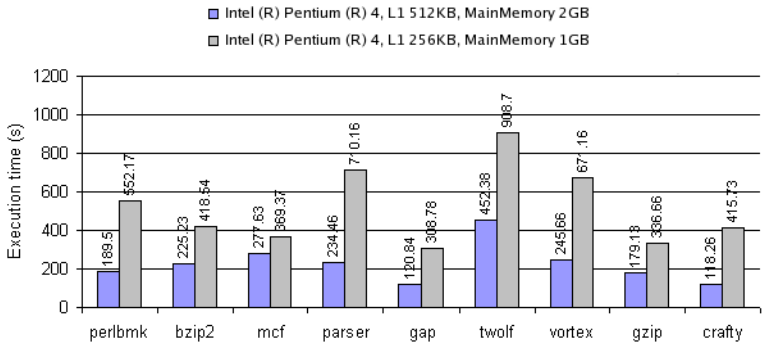


Figure 3.13: Run-time system performance observed under different processor/system resources.

3.3.4.1 Resource impacts

In order to evaluate the impact of machine resources on a run-time system, two systems were selected to evaluate SPEC2K INT benchmarks. By averaging the execution time across inputs, and comparing the difference between two systems, insight is gained about the operating characteristics of the code transformation system. The results are shown in Figure 3.13.

The presented results do not show the percentage relative to the improvement in the base. Rather, they highlight the performance improvement obtained to the run-time system given more resources. The two systems differ in processor speeds and on-chip cache capacity by a factor of two. The observed performance improvement is on average 60%. One of the largest improvements is seen in benchmark *255.vortex*. The benchmark is a memory intensive application [10] that suffers from cache limitations. This factor is directly attributable to the increase in processor resources. It experiences less interference with the code transformation system as a result of a larger cache (increase from 512KB to 1M level-2 cache).

Several run-time analysis and optimization techniques can be activated in a host system with larger architectural resources since the corresponding overheads can be mitigated. An interesting study is required to determine the exact limits to which

dynamic code transformation systems can be stretched between processor generations.

3.3.4.2 Utilizing architectural features

Most modern processors (e.g. Pentium and Athlon) maintain an internal stack called the Return Address Stack (RAS) to help boost application performance. This technique is capable of improving performance by as much as 15% in programs that execute subroutine calls frequently [27]. This stack is updated every time a `CALL` or `RET` instruction is executed. When a `CALL` instruction is executed, the original application return target address is put on top of the return address prediction stack. The return target is also placed on top of the regular application stack; on the x86 architecture this is referred to via the `%esp` register. The execution of a `RET` instruction *pops* the top address off of the return address stack, as well as the regular stack. An instruction stream, beginning at the address on top of the return prediction stack, is fetched to keep the processor pipeline busy.

The RAS functionality, as explained above, is tied to the `CALL` and `RET` instructions. Most run-time systems modify these instructions into their run-time idiom form to ensure proper execution of the program in the presence of the run-time system. There are two fundamental issues associated with these modifications/translations. First, it imposes overhead due to the special handling required for `RET` instructions (explained in Section 3.2.2). Second, it inhibits the use of the RAS.

A `CALL` instruction is translated into the following: (1) an instruction that puts the address following the `CALL` instruction on top of the application stack. And (2) an unconditional branch, `JMP`, to the first instruction of the target subroutine. Likewise, a `RET` instruction is translated into the following two instructions: (1) a `POP` instruction that removes an address from the top of the stack. Followed by (2) a `JMP` to the address just removed. Since the original `CALL` and `RET` instructions have been substituted, the RAS is not activated for translated subroutine `CALL` and `RET` instructions.

To facilitate the use of the RAS by the translated `CALL` and `RET` instructions, they must not be translated into their run-time idiomatic form. Rather, they must be executed natively. There are two benefits of this approach. First, `RET` instructions contribute to the varying branch target overhead. As such, this overhead can be reduced by the proposed approach because it resembles the original program model; most `RET` instructions transfer control to the instruction following the `CALL` instruction that invoked the subroutine. Therefore, the penalty incurred as a result of return trace lists such as the one illustrated in Figure 3.9 can be reduced. Second, native execution of the `CALL` and `RET` instructions results in the hardware RAS working in conjunction with this approach. This facilitates improved I-Cache performance as the hardware RAS can keep the processor pipeline busy using the translated address on top of the stack just prior to removing it. However, there are certain challenges that must be addressed to facilitate native execution of these instructions:

Cannot modify application stack: JIT-based run-time systems copy instructions and execute them from the translation/code cache. Placing any translated addresses on the application stack can break C++ exception handling, garbage collection or the execution of setjmp()/longjmp() instructions. Furthermore, certain applications unintentionally look beyond the top of the stack for data. These may fail if any translated addresses are put on the application stack. Both the `CALL` and `RET` instructions touch the stack; `CALL` instructions put the instruction address following it on top the stack. Therefore, their native execution from the code cache may cause a program to fail.

To avoid touching the application stack, a separate stack must be maintained to allow the native execution of the `CALL` and `RET` instructions. This stack address must be swapped in and out of the stack pointer register just before and after executing the `CALL` and `RET` instructions natively in the code cache. This separate stack is referred to as the software return address stack (SRAS). It is named as such because it imitates the hardware prediction stack.

Empty prediction stack: When running an application under a run-time system, it is possible the number of `CALL` instructions executed are less than the number of `RET` instructions executed. This can cause the run-time system to loose control of the application or cause program failure. The SRAS is populated only by the execution of `CALL` instructions. Therefore, it is possible that the return address prediction stack is empty even though a valid `RET` is about to be executed.

Just before executing a `RET` instruction on the SRAS, the stack must be checked to ensure it contains a valid translated return target. Else the system needs to make sure there is a fall-back mechanism to handle the execution of the instruction. One such mechanism is illustrated in Figure 3.9.

Position independent code (PIC): This form of code is heavily utilized in shared libraries to facilitate code relocation at load time. On the x86 architecture there is no instruction to get the instruction pointer. A `CALL` instruction targeting the next instruction is executed to materialize the program counter on the application stack. Such instructions are called PC-materialization instructions. These instructions have no corresponding `RET` instruction.

As a result of PIC code, PC-materialization instructions can upset the accuracy of the proposed optimization. This can result in more instances of an empty return prediction stack. For example, if the inner most leaf function of a nested function call executes a PC-materialization instruction, all the return predictions back up the nested call stack will be incorrect. Upon reaching the top of the stack, the prediction stack will be empty. Therefore, precaution must be taken to ensure that SRAS predictions of the return target are correct. Also, there must be a mechanism to handle incorrect predictions.

No interprocedural optimizations are allowed: As explained in previous sections, run-time systems apply optimizations to reduce their overheads. The optimizations applied by a run-time system must not violate the calling conventions for the proposed

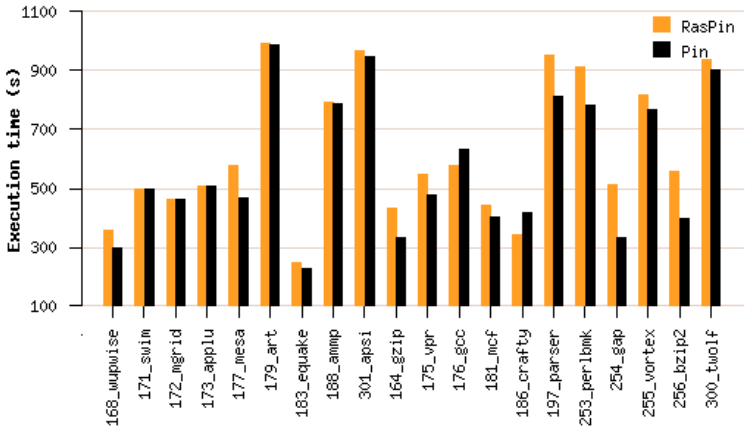


Figure 3.14: Performance of the run-time system utilizing the software based return address prediction stack.

mechanism to function properly. This is akin to the rules followed by compiler optimizations that are non-interprocedural. If optimizations are applied, state must be restored prior to the execution of the `RET` instruction.

Ensuring that all of the above challenges are properly addressed imposes significant amounts of overhead on the run-time system. The performance results are shown in Figure 3.14. The results are relative to the approach illustrated in Figure 3.9. The performance of the run-time system using the technique detailed in Section 3.2.2 is labeled Pin. The version of the run-time system that relies on the suggested approach is marked as RasPin.

The large overhead shown in Figure 3.14 indicates that run-time systems cannot drastically benefit from existing architectural features as do normal applications. In fact, any attempt to do so as results in performance degradation. Only benchmarks *176.gcc*, *186.crafty* and *252.eon* (not shown because it makes the rest of the execution time numbers look insignificant due to its large y-axis data-point) benefit from this optimization. Their average improvement is around 22%. In all others benchmarks, significant performance loss is seen.

The execution time degradation is a result of code explosion. Two instructions (`CALL` and `RET`) are expanded into nearly fifteen instructions. This expansion is because of the safeguards required to address the stated challenges. For instance, eight of the instructions result from having to change the register holding the stack pointer from the application stack to the prediction stack and vice versa.

The results presented in Figure 3.14 confirm that current run-time systems cannot directly hope to exploit all existing architectural resources. Certain resources designed for regular application domains are ill-suited for run-time code transformation systems, these systems need additional architectural support to be effective. For instance, having another register to hold the prediction stack pointer with variations of the `CALL` and `RET` instructions that work on the prediction stack can significantly alter the performance numbers. This is not too far fetched as architectural support for virtual machines is on the rise [25].

Due to the lack of support for run-time systems, several techniques such as *function cloning* and *indirect branch resolution* optimizations are attempted by current state of the art systems like [3, 6, 13, 31] to mitigate the overheads. However, current techniques are far from being successful in reducing the overheads to negligible. As such, it is imperative for computing environments to adapt for run-time systems.

3.3.5 Everyday application environment

In order to understand the impact of code behavior on the code transformation system, several non-standard applications were evaluated. These applications use several shared libraries, where as the SPEC2K codes have only limited use. The applications include user interface programs such as *emacs*, *xpdf*, and *xlogo*. In general, the specific details of how a code transformation system behaves with graphic based user-interactive programs has not yet been evaluated. These applications are evaluated only for their start-up costs (generation of initial screens, etc.).

Figure 3.1 (a) presented the overheads for these applications under the run-time system. The original execution time of each application is very small compared to the excessive amount of execution time under the transformation system. The applications slow-down by about 40x-400x relative to their original execution time. The slow-down is because of their large footprint due to code across multiple shared libraries. Although these results indicate the poor state of operation of a modern code transformation system to handle normal (non-benchmark) applications, the experiment opens the opportunity for creative directions in run-time code transformation system design. For instance, several of these application use the same library behavior to display graphics. As such, it may be possible to design dynamic code transformation systems that leverage such common traits across executions and applications.

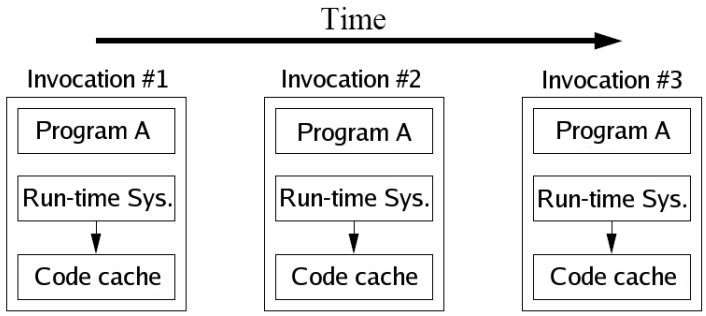
Furthermore, this highlights that code transformation design should focus on workloads other than SPEC. SPEC's benchmark spectrum is often limited because of portability issues [18]. However, run-time systems are capable of handling generic applications and so designers ought to evaluate performance on a more diverse set of benchmarks.

Chapter 4

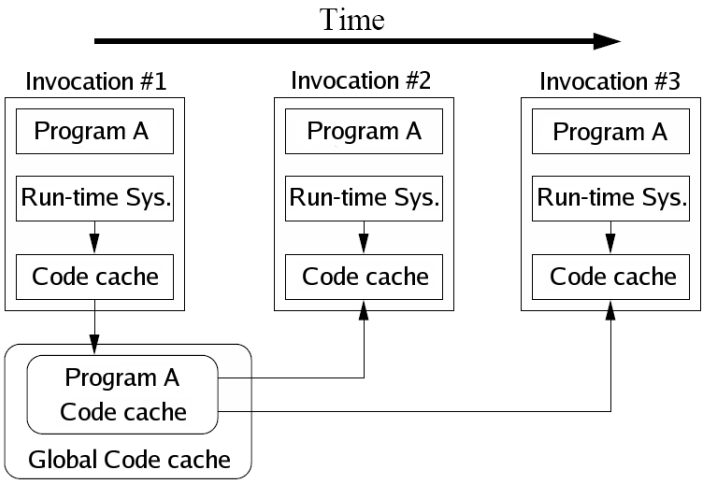
Exploiting persistent characteristics to address transformation overhead

Thus far, the work has elaborated on the barriers to run-time code transformation systems and makes a case centering on finding new design strategies of these systems. Exploring the design space for these systems is critical for their development and in realizing their true potential. This chapter makes a case that the dynamic instruction stream of an application does not vary significantly across different input sets. As such, a model of code transformation, *persistence*, is proposed that leverages this invariance trait to address the transformation overhead explained in Section 3.1.

Traditionally, dynamic code transformation system designs have primarily focused on achieving a given transformation task on a single execution instance of a program as shown in Figure 4.1 (a). Repeated invocations do not exploit identical characteristics such as frequent/identical code paths or other information from previous executions as suggested in Figure 4.1 (b). This is a significant drawback since applications tend to exhibit identical characteristics across executions as shown in Section 4.1. Exploiting shared behavior can dramatically reduce the run-time overhead since the time spent performing identical tasks such as analysis and optimizations in a repeated invocation can be avoided.



(a) *Traditional run-time systems*



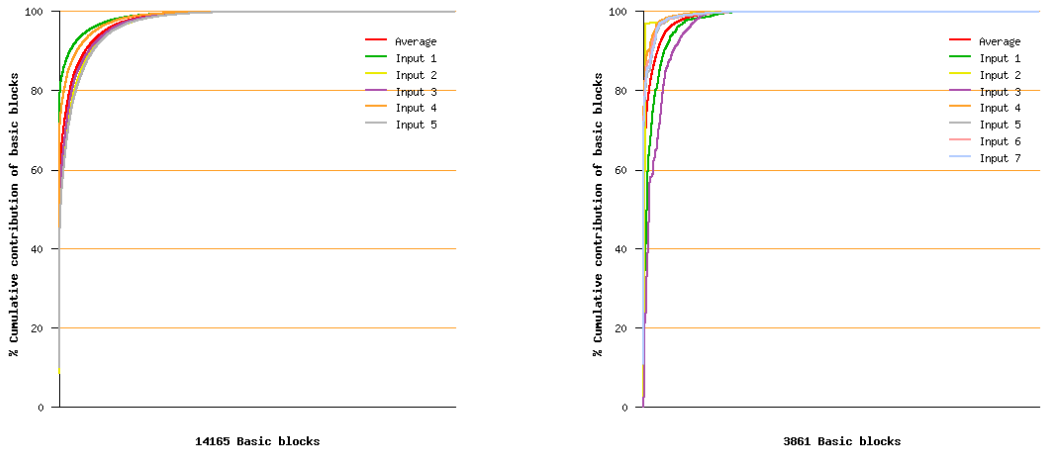
(b) *Proposed design for future run-time systems*

Figure 4.1: (a) Current model applied by dynamic code transformation systems where they target accomplishing their task only within the current instance of execution (b) Proposed design for run-time systems in the future that exploit execution characteristics across multiple invocations of the same program.

4.1 Execution path variance across different inputs

To effectively exploit the concept of reusing information from prior runs, the requirement is that applications exhibit identical behavior across invocations regardless of the input data set. To verify if this is indeed the case with applications, dynamic execution counts of all basic blocks were collected across all the SPEC reference input data sets. A cumulative graph is presented in Figure 4.2 for all the input data sets along with the cumulative average of all basic blocks across all inputs.

Only *176.gcc* and *253.perlbmk* are presented here since they have the most number



(a) *176.gcc* (b) *253.perlbmk*

Figure 4.2: Cumulative contribution to program execution by basic blocks for *176.gcc* and *253.perlbmk* across all their SPEC2K reference input data sets.

of reference input data sets. Also, of all the benchmarks that have multiple reference input sets (*197.art*, *164.gzip*, *175.vpr*, *176.gcc*, *252.eon*, *253.perlbmk*, *255.vortex* and *256.bzip2* in the SPEC2K suite), they were the only two benchmarks that yielded maximum variation in response to the input data set variation. Figure 4.2 is interpreted as follows, the x-axis consists of basic blocks across all the inputs regardless of whether that block was executed in an input or not. The y-axis is the corresponding dynamic execution contribution of that basic block to the execution. A horizontal shift on the x-axis with zero or tiny gradient implies that the basic block contributed nothing or very little to the execution when the program was run using that particular input data set was utilized.

The data clearly indicates that nearly 60% of the application is executed by the same basic blocks across all the inputs. Past that point, there is a slight deviation in the contribution of basic blocks across the different input sets. However, since most of the basic blocks still contribute to execution regardless of their dynamic execution, the case stands that programs do exhibit persistent characteristics across varying input sets. The variance in the dynamic execution counts of the basic blocks in relation to

the work presented here is beyond the scope of this thesis.

This data confirms that despite the input used on a binary, code variation in the SPEC2K suite is minimal and thus the suggested strategy of exploiting previous executions is an effective strategy to tackling the transformation overhead.

4.2 Applications of persistence

The practicality of exploiting persistent execution characteristics in a real environment is described in this section to strengthen the argument in support of it.

Instrumentation: Persistent caching of executables instrumented via a run-time binary instrumentation system 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 tool using the Pin infrastructure experiences a 1.4x slow-down simply due to transformation overhead. Separate invocations of the program with different input sets would all therefore result in an identical transformation slow-down. 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.

Optimization: Dynamic optimizers have thus far focused on optimizing just a single run of an application. While this has been a step forward from static compiler optimizations in being able to leverage execution characteristics, it has its limitations. For example, it is inefficient to spawn dedicated processes of a dynamic optimizer on every application running on a system simply due to the vast number of processes.

Rather it is more practical to envision a dynamic optimization system that is deployed system wide to allow applications to share code from different runs of the same program or even perhaps across programs (shared library code).

In a server domain, a daemon may spawn multiple threads which execute identical code sections. A dynamic optimizer could pay the cost of identifying the hot code once and reuse the optimized code paths for other threads that are spawned there after. The idea is that the high cost of performing optimizations is only paid once and subsequent runs would benefit from the prior work. Another strategy would be that the optimization analysis is spread across multiple threads, the main dynamic optimization thread could spend minimal amounts of time in the optimization and analysis phase for every thread in order to ensure it does not drop the response time for the thread. Eventually, the system would be able to gain enough information to efficiently deploy the optimizations.

To evaluate persistence and its benefits, a working solution has been implemented in the Pin binary instrumentation framework described in Section 2.3. In the following section, the persistence model is demonstrated to be an effective solution to the transformation cost in Pin. This model of Pin is referred to as Persistent Pin (PPin). PPin is capable of the following:

- Caching pure and instrumented translations of program executions.
- Reusing cached instrumented executions across separate invocations.
- Reusing a persistent cache for a previously unseen input.
- Supports multi-threaded applications.
- Fully inheriting Pin's run-time optimizations across executions.

4.3 A persistent run-time system

Figure 4.3 illustrates the control flow of a persistent run-time system. The system is started with some basic initialization steps. Past the initialization phase, the system checks if a prior execution exists in the persistence cache database. If a prior execution

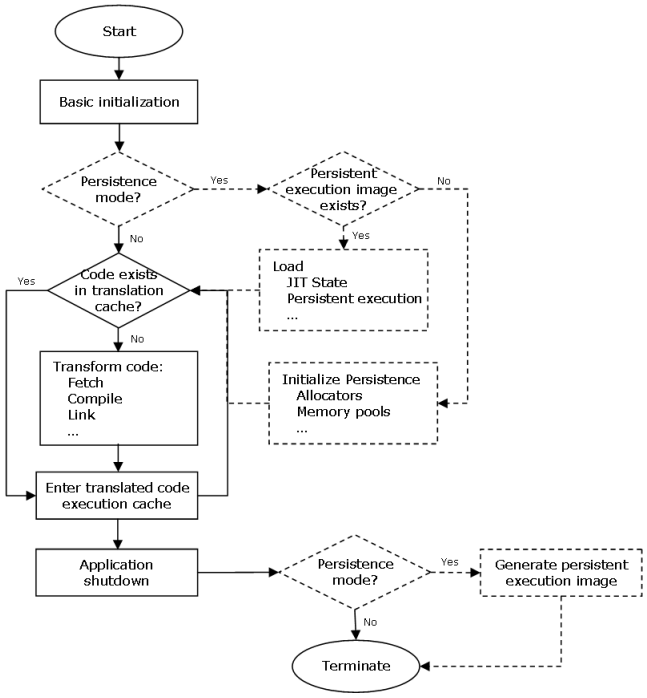


Figure 4.3: Overview of a persistent code transformation system. Dotted lines indicate Persistence specific states.

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.

Past the initialization phase, the system defaults to steps normally taken by any dynamic binary code transformer such as those explained in Section 2.3. At the end of the execution, the translated code sequences and their relevant data structures 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.

4.3.1 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 dynamic code transformation system 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) [36]*:* 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.

Absolute 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 code base.

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

4.3.2 Persistence in a binary instrumentation system

Persistent Pin (PPin) is specifically designed to reduce the overhead of dynamic instrumentation. The overhead 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 previously visited will not require Pin's compilation.

PPin was evaluated on the SPEC2K suite 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 SPEC2K programs, which are poor indicators of everyday application characteristics on run-time systems.

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.

4.3.3 Overhead reduction over the lifetime of programs

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.

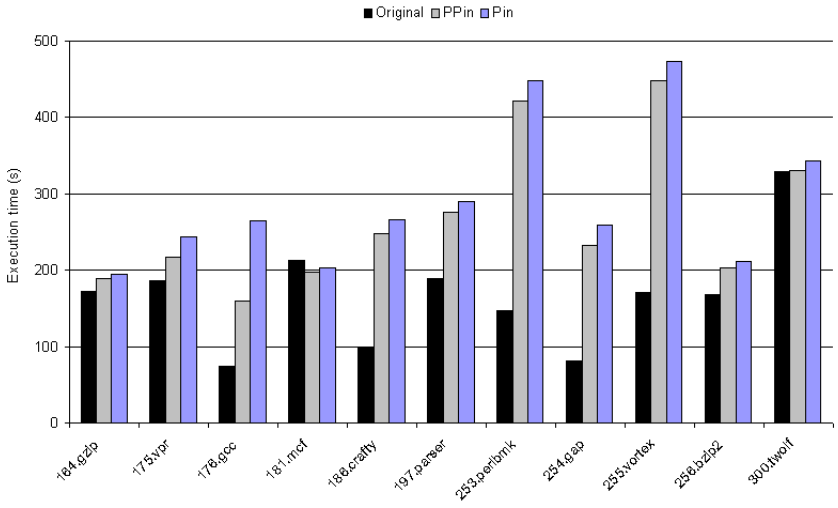


Figure 4.4: Persistent Pin's performance in comparison to native and Pin's performance.

Without instrumentation, Pin is highly representative of a generic run-time system. 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 4.4 shows the execution times of the SPEC benchmarks running the original binary, under Pin, and using an already cached execution (PPin). Only SPEC2K INT benchmark performance is reported since it has been shown in [31] that Pin does affect SPEC2K FP 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 4.5. 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. Benchmark *176.gcc* benefits most from persistence with an improvement of 30%

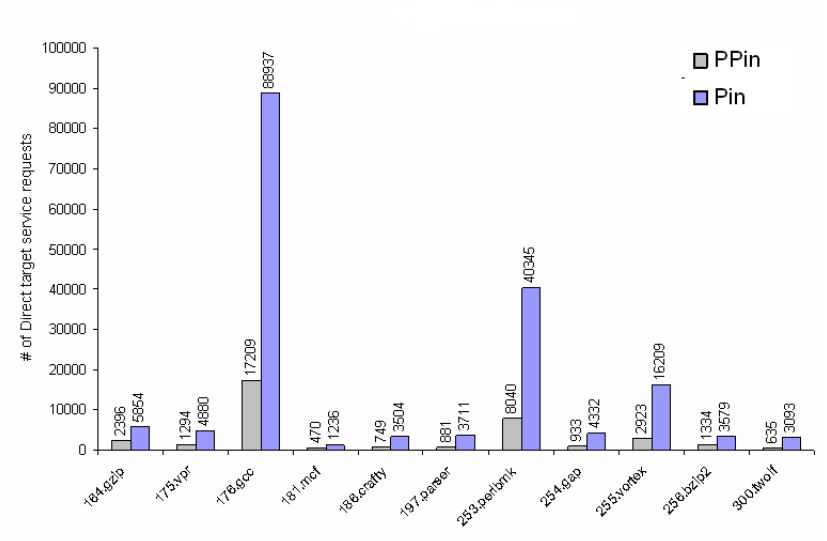


Figure 4.5: Pin service requests from the translated code.

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. 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, these reductions in service requests do not improve the respective execution time since these applications are affected by the performance of the translated code. Both applications have a very high dynamic execution of indirect branches. The challenges in handling such instructions has been elaborated in detail in Section 3.2 and in works such as [3, 31]. This is not the task of the proposed solution and thus is not discussed further.

Proceeding to the evaluation of persistence in instrumentation, basic block (BBL)

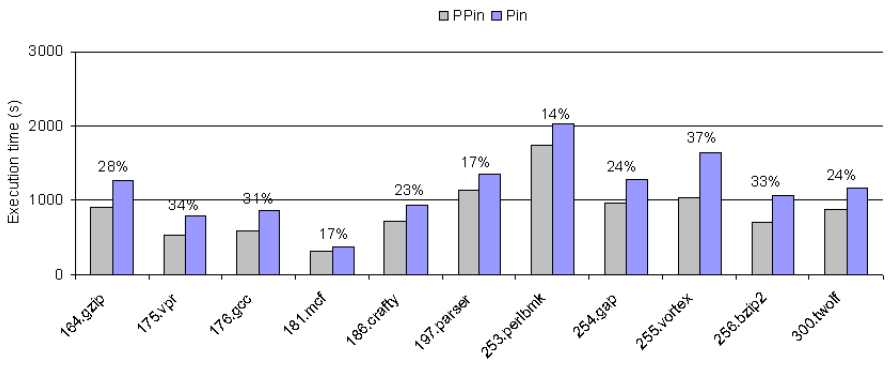


Figure 4.6: Execution time for BBL instrumentation using Persistent Pin vs. Pin. Percentage improvements of PPin over Pin are shown.

profile instrumentation was applied to the SPEC2K INT benchmarks. Figure 4.6 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. The benefits of caching instrumented code will vary based on the granularity of the instrumentation. Instrumenting every instruction will result in more benefits than caching instrumented code that instruments every load or store instruction since less code is generated in the latter case.

4.3.4 Start-up cost reduction

The start-up phase is the most difficult part for any code transformation system since it is constantly invoked to generate new code paths that have not been seen yet. Applying these systems to interactive programs is difficult because users do not tolerate especially long pauses at start-up. This slow-down is a result of a lot of shared library initialization. Cold code that is bad for run-time systems because there is no way to amortize the overhead incurred during the start-up phase.

Figure 4.7 shows the execution time to start the graphic programs and to shut

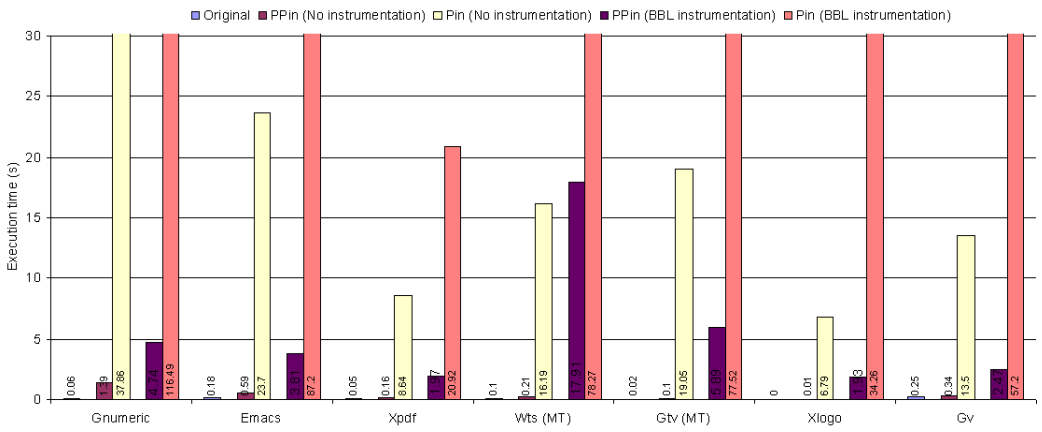


Figure 4.7: Execution time of Persistent Pin vs. Pin for BBL instrumentation of everyday applications. Applications were started and immediately shutdown, this is the worst case scenario and illustrates the benefits of caching cold code.

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 pure and instrumented 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 run-time system that does not use persistence-like services.

4.3.5 Persistent cache analysis

With the growing size of the executables and the number of shared libraries they interact with, code caches tend to be fairly large in size as detailed in [17]. The size of the persistent executable image is critical. The run-time system shares the address space

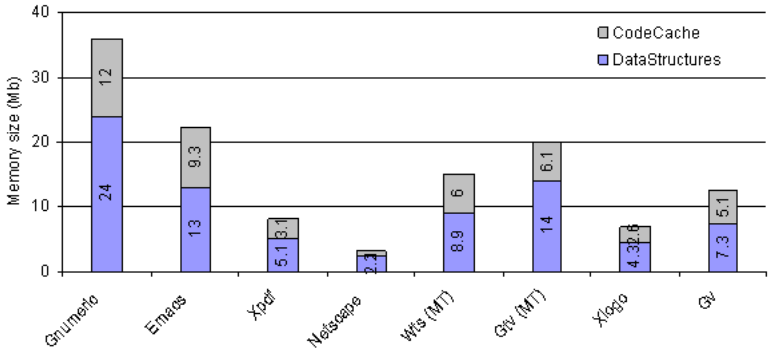


Figure 4.8: Persistent cache sizes.

of the main application being monitored, so if the cache size is too large the address space may be exhausted. However, with the advent of 64-bit address space systems, persistent cache sizes are highly unlikely to be problematic.

The persistent cache sizes for SPEC2K applications are relatively small with a maximum cache of approximately 3MB with the exception of benchmark *176.gcc* which reached a size of 24MB. Figure 4.8 shows the persistent cache sizes of everyday applications, they are significantly larger than most of the SPEC2K applications.

It is interesting to note the size of the translated execution bits relative to their corresponding data structures. The latter is nearly twice the size of the execution bits, this data is representative of even the SPEC2K benchmark suite. Further investigation revealed that majority of the cached data structures were those that corresponded to the cache directory ($\sim$70%) as shown in Figure 4.9.

The cache directory is a structure that maintains mappings of original program addresses to their translated code addresses and other relevant metadata. The directories are so large because they are polluted with entries that are not utilized. This piece of data can be related to the code cache utilization presented in Figure 3.7 which shows that many of the translated code sequences are not utilized due to speculative code generation (Section 3.2.1). The cache directory has to keep track of every compilation unit regardless of its utilization. It is a critical component of any run-time system and is

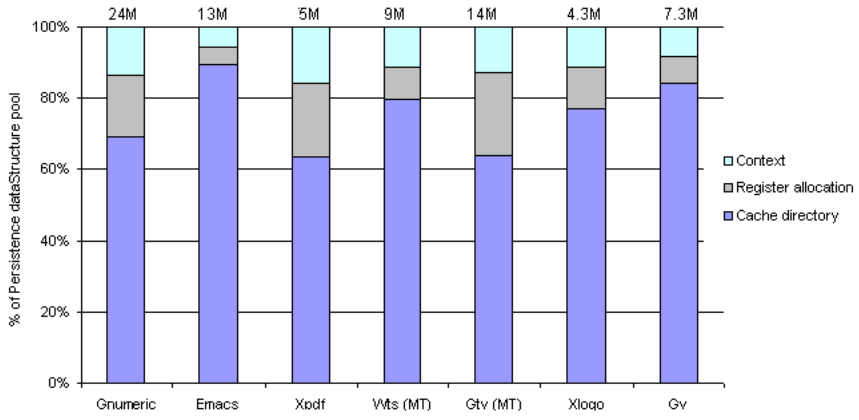


Figure 4.9: Breakdown of the data structures persistent cache.

heavily utilized during code transformation. It facilitates trace-linking, an optimization that is extremely critical for keeping the overheads small in run-time systems. Therefore it is important that it has good run-time performance. A large directory could impact performance since traversal time during code transformation is likely to be high, thereby adding to the execution overhead.

The other two components of the data structures are the re-register allocation optimization information and the run-time system's context information. Re-register allocation information is cached across executions since it is required to facilitate adding new code paths to the cache. All subsequent executions have to utilize the same register binding as the cached instance. Context is the metadata of the run-time system itself and is state required for the system to be able to support persistence. These too are utilized during code transformation and their performance is critical as well.

Chapter 5

Mitigating the translation overhead via mixed execution

The previous chapter addressed the dynamic code transformation problem experienced when applications have fairly large footprints. However, Figure 3.1 shows that certain applications experience significant overhead as a result of the translated code performance. All run-time systems apply optimizations specifically aimed at reducing the translation overhead. Despite all past work, no one system has shown that the translation overhead can be successfully mitigated to negligible numbers. Overcoming the translation barrier is difficult due to the challenges presented in Section 3.2. This raises the question as to whether pure JIT-based run-time systems are practical to implement in everyday systems.

In lieu of the execution overhead faced by JIT-based run-time systems, a different execution model is proposed as a technique capable of keeping translation overheads minimal. It is called *mixed execution*. The code executed is a mix of translated code and instructions from the original binary. In this model, the run-time system's JIT compiler is involved only at points of interest rather than the entire execution. The premise of this model is that executing original code intermittently results in minimal translation performance penalties. Therefore, by carefully selecting the regions of interest for code transformation, the performance degradation is kept within limits. Mixed execution is an effective alternative to its JIT-based counterpart, when constraints like extremely low overheads and minimal architectural perturbation are critical to execution.

In the following section, the underlying technique facilitating mixed execution, *code splicing*, is introduced, and its limitations on the x86 architecture are discussed. Additionally, the methodology is discussed further to illustrate how mixed execution generates effective code sequences. The section also introduces *function migration*, which is a novel, and previously unexplored, technique surpassing certain limitations of code splicing.

5.1 Code splicing

Code splicing is the modification of a binary's static instruction stream. The technique is useful to redirect control elsewhere during execution. The target of the redirection can be one of two: (1) transformed code in the translation cache or (2) the entry point of the run-time system. The first has the disadvantage that the code needs to be generated prior to execution. This is achievable by scanning the binary and splicing instructions of interest just before executing the first instruction. As such, it is possible to generate code sequences that will never be executed. Code is guaranteed to execute only if instructions in the dynamic instruction stream are spliced. In the second, code is generated just prior to execution. It closely resembles JIT compiler systems.

Thus far, code splicing has been prominent in the domain of dynamic instrumentation [1, 7, 37]; especially for kernel level instrumentation. Here, the JIT and interpretation methods are impractical because it is a timing sensitive environment. In theory, the process of code splicing is relatively simple. However, there are complications when implementing code splicing on the popular x86 architecture. These challenges are discussed later in Section 5.1.1.

The mixed execution approach is practical in a domain where overheads beyond certain thresholds are not acceptable. For example, such domains can be run-time code optimization and light weight instrumentation. For instance, if the only interest is to profile the execution count of all functions executed, it is better to apply the mixed

execution approach versus the pure JIT approach. Figure 5.1 compares the performance of the original program with the execution time of the mixed approach and JIT-based instrumentation in order to profile functions at two different granulates.

The first experiment, *Functions Visited*, counts the number of times a function is invoked until a threshold of four is reached. Once the execution count of a function reaches four, the profiling code is removed. The overheads between the mixed execution and JIT approach vary significantly for benchmarks in the SPEC2K INT suite. Benchmarks like *186.crafty*, *253.perlbmk* and *255.vortex* experience a slow-down of ∼2x. Contrasting the performance of the JIT-based approach is the mixed execution approach. It matches the performance of the original program across the entire benchmark suite. Under the mixed execution approach, benchmarks `301.apsi` and *164.gzip* serve as interesting observations. Both these benchmarks consistently outperform their original performance. SPEC2K performance does not vary much under either of the approaches. Changes in the I-Cache and I-TLB behaviors, due to the addition of profiling code, are suspected for the performance change. Unlike benchmarks in the SPEC2K INT suite, the SPEC2K FP applications do not exhibit performance variation. The floating point benchmarks are not as aggressive as the SPEC2K INT benchmarks in subroutine calls.

The second experiment, *Function execution profile*, collects the execution profile of how many times a function is invoked during execution. Both approaches show no performance degradation on the SPEC2K FP suite. Once again, the floating point benchmarks do not make too many subroutine calls. Also, they have relatively smaller execution foot-print than their counterpart SPEC2K INT programs. Figure 3.2 (a) shows that the run-time system quickly captures the small foot-print. Thereafter, execution time is spent in the translated code sequences. This allows the application to make progress. The SPEC2K INT applications perform differently under the two approaches. Mixed execution once again significantly outperforms its JIT-based run-time

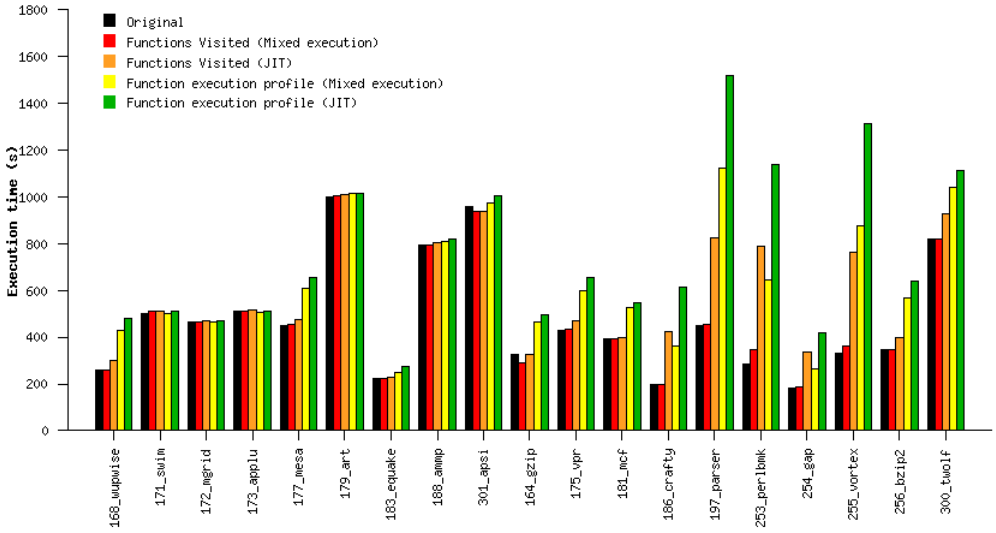


Figure 5.1: Time to collect the dynamic execution count of all functions executed.

counterpart.

In both experiments, the JIT approach is slower because of the transformation cost, as well as the poor performance of the translated code. The JIT-based system incurs more transformation overhead because the system must compile the entire dynamic execution path of a program. On the other hand, the mixed execution approach needs to compile just enough code to facilitate profiling.

The performance of the translated code varies because the JIT approach incurs overhead as a result of making sure that it maintains control at all times. The performance of benchmarks *253.perlbmk* and *255.vortex* under the JIT approach, in relation to the mixed execution, is degraded by nearly 60% due to translation penalties. Mixed execution does not incur such penalties to perform function profiling (technique is elaborated in Section 5.1.2). As such, it experiences significantly lower translated code performance overheads.

5.1.1 Limitations

The challenges of code splicing depends on the instruction set architecture (ISA). For fixed length ISA's, splicing does not prove problematic since every instruction is fixed size. Overwriting an instruction with another in order to redirect control elsewhere is safer than doing so in the x86 architecture. In the x86 architecture, this is more challenging due to its variable length instruction set. There are three potential hazards when splicing (1) over instruction and basic block boundaries, (2) threaded applications and (3) position independent code.

Splicing over instruction and basic block boundaries: The x86 architecture requires a splice to be the same size as the instruction it overwrites. Only then does code-splicing guarantee program correctness. However, this is challenging in the x86 architecture. The code cache offset is often too big to fit into eight and sixteen bit branch instruction offset fields. One way around this limitation is to overwrite the smaller instructions with the splice, migrate the overwritten instructions to the target of the splice, and then execute the instructions there. However, this is risky. One of the overwritten instructions may be the target of a branch elsewhere in the application. In this case, a branch may be taken during program execution that would jump into the middle of the newly-written spliced instruction and cause the program to fault. Consequently, not all instructions can be safely spliced unless additional information is present. One piece of critical information that can distinctly aid the splicing process is complete control flow graph information of a subroutine. With precise control flow graph information, basic block boundaries can be determined and other splicing options then become available.

Another alternative is to write a special x86 trap instruction (`0xCC`, one byte) to splice arbitrary points in the original program. Executing this instruction causes an `INT 0x3` exception, which invokes the kernel. Its is a standard practice used in debugging

systems. In the same sense, a splicing system can register a signal handler to catch the execution of that special trap (splice). And then redirect the control flow elsewhere. Nonetheless, this is an impractical technique because a kernel trap costs many thousands of cycles due to context switch overhead.

Splicing a threaded application: In a threaded environment, splicing cannot cross instruction boundaries, even if the control flow graph indicates a point is safe to splice. Consider this scenario, a splice that spans multiple instructions is being written. One of the instructions being overwritten is the resume address of a sleeping thread. This sleeping thread will fault the very instant it is scheduled for execution because its resume address has been overwritten. As such, it is resuming execution in the middle of an instruction. Therefore, it is risky to splice a threaded application across instruction boundaries.

Position Independent Code (PIC): PIC relies on computations that require the instruction pointer of an instruction to find targets, such as the the global offset table (GOT). The x86 has no instruction to get the current instruction pointer address. The only way is to execute a PC-materialization call instruction. It is a `CALL` instruction to the following instruction address. Executing such an instruction puts the current instruction pointer on top of the stack. This value is moved into a register using a `POP` instruction. The value in the register is then used to compute PIC target addresses. An example is shown below in which the PIC is generating the GOT offset.

```
(gdb) disassemble _fini
dissemble _fini
Dump of assembler code for function _fini:
0x08048440: push  %ebp
0x08048441: mov   %esp,%ebp
0x08048443: push  %ebx
0x08048444: call  0x8048449 <_fini+9>
0x08048449: pop   %ebx
0x0804844a: add   $0x111f,%ebx
0x08048450: push  %edx
0x08048451: call  0x8048310 <__do_global_dtors_aux>
0x08048456: mov   0xfffffffc(%ebp),%ebx
```

```
0x08048459: leave
0x0804845a: ret
End of assembler dump.
(gdb)
```

Instructions at address `0x08048444`, `0x08048449` and `0x0804844a` are computing a PIC target address. Instruction `CALL 0x8048449` cannot be spliced. If this PC-materialization instruction is migrated and executed elsewhere, the instruction pointer will be different. As a result, the PIC target computation will be incorrect causing program failure. Such special cases of five byte instructions must not be considered for splicing.

From the limitations, it becomes evident that code splicing may severely limit the transformations applied to the original code. Table 5.1 shows where mixed execution and JIT-based approaches are applicable. The applications presented are in reference to the systems presented in Chapter 2. Optimization, instrumentation and security are listed as being task-dependent for mixed execution because they are limited by whether or not a given point in the program is safe to splice.

For an optimization system, the JIT approach is marked as being dependent on the task because it is unlikely to function properly when optimization decision making algorithms rely on architectural performance data. JIT-based systems affect the architectural behavior since their execution is interleaved with the execution of the application itself. This means that any field of study relying solely on observing the architectural behavior will most likely not be able to utilize a JIT-based approach. A case study is presented in Section 5.2 to illustrate this claim.

5.1.2 Methodology

The fundamental method of splicing an instruction is illustrated in Figure 5.2. Control flow is redirected by *splicing* the original code to a point in the translation/code

Type of system	*Mixed execution*	*JIT*
Optimization	Task dependent	Task dependent
Instrumentation	Task dependent	Yes
Translation	No	Yes
Security	Task dependent	Yes

Table 5.1: Systems where the mixed execution mode and JIT-based run-time systems apply.

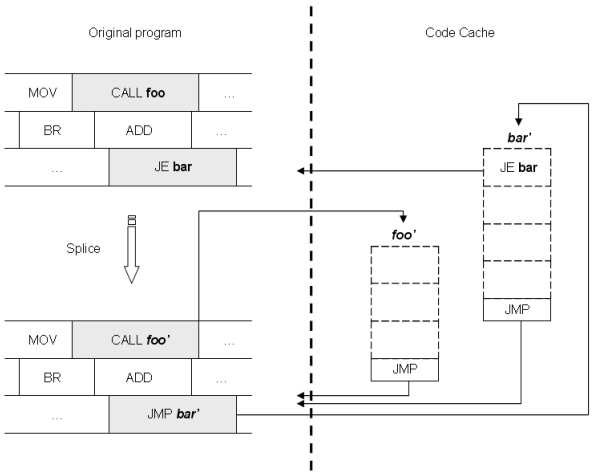


Figure 5.2: Using code splicing as a means of transferring program control from the application to the code cache sequences. The gray shaded boxes are 5 byte x86 instructions, so they are patchable.

cache. The control flow redirection is forced by replacing the instruction `JE bar` with an unconditional branch instruction `JMP bar'`. The replaced instruction is executed at the splice target. It can be redirected by overwriting the immediate field of the `CALL foo` instruction as `CALL foo'`. At the end of a code sequence in the code cache, there is an unconditional control transfer instruction (`JMP`) that transfers control to the appropriate place in the original application code.

Example uses of code splicing are presented in the following paragraphs. The example applications collect profiles of various program elements: function callsites, functions, and loops. Any issues regarding the usage of this technique are also discussed.

Callsite profiling: This can be accomplished via the following sequence: (1) copying the `CALL` instruction over to the code cache, (2) instrumenting the copied instruction with the required profiling code, (3) adding an unconditional branch after the copied instruction to transfer control back to the original program, and (4) overwriting the original `CALL` instruction with a splice (an unconditional `JMP`) to the copied instruction. Every time the program reaches the callsite, control is transferred to the instrumented `CALL` in the cache. The instrumented instruction executes the profiling code either before or after it invokes the instruction target in the original program depending on the instrumentation. It then transfers control back to the original program via an unconditional branch. From experience, all `CALL` instructions generated by the *Gcc* compiler have 32-bit offsets. Therefore, the issue of overwriting a smaller instruction with a larger 5-byte splice (`JMP` instruction) has thus far not surfaced as an issue. However, callsite profiling is not guaranteed to be safe due to PIC code which is explained in Section 5.1.1.

Function invocation profiling: This is achieved by overwriting the first instruction of the function with an unconditional `JMP` instruction. It transfers control to an instrumented copy of the instructions it overwrote. This is a common technique. It is illustrated in works such as [24, 37, 35, 43].

Loop profiling: This is challenging to achieve via the mixed execution approach because of the variable length x86 ISA. To profile this program element, instrumentation needs to be added to the loopback edge, as well as the entry and exit points of a loop. However, compilers generate at most 16-bit loopback branch instructions. These branches do not have a large enough offset to reach the code cache. Therefore, the instruction cannot be overwritten with a splice; splice instruction offsets normally need to be 32-bits wide. This is detailed in Section 5.1. A solution capable of addressing this issue is discussed below. *Function migration* is a technique that can facilitate loop profiling via the mixed execution approach. It is believed this technique is a novel

contribution to prior work in code splicing. There are other proposed techniques such as those discussed in [24, 35].

5.1.2.1 Function migration

Function migration generates a copy of the original function and places it in the code cache. The only difference between the copied and the original function is the eight and sixteen bit offset changes applied to all the PC-relative branch instructions. They are modified into their corresponding 32-bit versions. There are two fundamental reasons for this modification: First, if the branch target is outside the bounds of the copied function, the branch offset must be large enough to reach the original code from the code cache. Second, applying transformations to the function changes the distance between instructions within the migrated function. So, all PC-relative branch instructions need to be updated. This migrated (copied and transformed) function is made available to the main application by the function invocation technique described in the previous section.

The primary advantage of this technique is that it facilitates the fine-grained transformation of x86 variable sized instructions. However, it comes at the cost of having to compile an entire function at run-time. Nevertheless, this approach is better than the pure JIT approach in certain scenarios. For example, this approach can successfully perform loop profiling. It is also applicable if just a few functions are of interest. Executing a mix of the original and transformed functions results in small transformation and translation code performance penalties.

5.2 Addressing a rising concern with run-time code transformation: power management

To illustrate when mixed execution is a more suitable approach than the JIT approach, a case study is presented. This case study addresses the growing concern

of microprocessor power consumption using a run-time dynamic voltage and frequency scaling system (RDO). The system utilizes the mixed execution to perform its task.

Scaling microprocessor energy and power control, Dynamic voltage and frequency scaling - DVFS, based on an application's execution behavior, is an effective means of conserving power in a system. It is implemented in many modern processors [5, 16]. Significant efforts are dedicated to DVFS control. Many of them are hardware based [23, 32, 39]. The rest are dependent on the OS time-interrupt or are static compiler techniques [20, 47]. All of these techniques rely on a fundamental concept of scaling of the processor frequency. The frequency is scaled down when the ratio of memory to arithmetic logic unit operations is high for a prolonged period and vice versa.

Existing hardware or OS time-interrupt-based DVFS techniques typically monitor some system statistics, such as issue queue occupancy [39], in fixed time intervals, and also decide DVFS settings for future time intervals [32, 39, 45]. Since the time intervals are re-determined and are independent of program structure, the DVFS control by these methods is not efficient in adapting to program phase changes. One reason is that program phase changes are generally caused by the invocation of different code regions, as observed in [22]. Thus, the hardware, or OS techniques, are not in a position to infer precise application code characteristics and the most effective adaptation points. Also, program phase changes are often recurrent (i.e. loops). In this case, the hardware, or OS, schemes would need to detect and adapt to the recurring phase changes repeatedly.

Static compilers apply DVFS optimizations based on the use of some profiler. A major limitation to static compiler DVFS is that the setting obtained at static compile time is possibly not appropriate for the program at execution. This is due to different execution environments for the profiler and the actual program. Program behavior in terms of memory boundedness is dependent on run-time system settings such as machine/architecture configuration, program input size and patterns. For example, machine/architecture settings such as cache configuration or memory bus speed affect

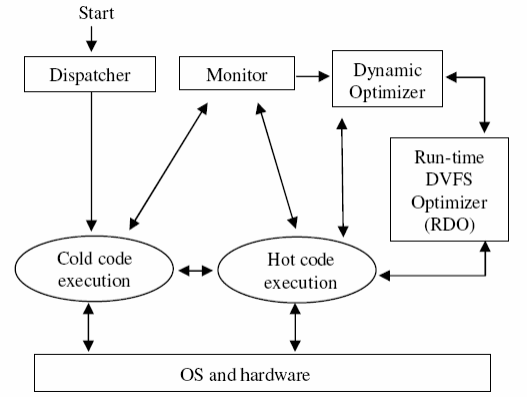


Figure 5.3: Overall system structure showing the operation and interactions among different components of a dynamic compiler DVFS optimization system.

how much CPU idle time exists. Different program input sizes or patterns affect how much memory is used. Additionally, as well as how it is going to be used. Therefore, it is difficult for a static compiler to make DVFS adaptive decisions based on the above factors. However, a dynamic compilation framework utilizes run-time information and can therefore make input-adaptive and architecture-adaptive decisions yielding better results.

5.2.1 A run-time dynamic voltage and frequency scaling optimizer

The run-time power optimizer structure is illustrated in Figure 5.3. Its requirements are follows: First, dispatch executable code for execution by the hardware. Second, monitor the code execution to identify hot code regions. These code regions are memory bound/intensive with regards to dynamic voltage and frequency scaling. Third, periodically apply, as well as remove, DVFS optimization to the code regions based on the execution history.

5.2.2 Challenges

The most significant RDO challenge is every cycle spent optimizing might be a cycle lost to execution. Therefore, the primary challenge becomes applying simple and

inexpensive analysis and decision algorithms in order to minimize the run-time system cost. For the analysis to be successful, the overhead resulting from code transformation and the translated code performance must be within bounds. As mentioned previously, a key task of the run-time power optimizer is to identify hot code regions (functions and loops) by monitoring whether their execution frequency reaches a certain threshold. It has been shown in Figure 5.1 that this overhead is significant if a JIT-based approach is used. From this data, it can be drawn that a JIT-based approach is not the means by which to tackle the profiling aspect of this problem.

Yet another critical challenge to address is the process of identifying run-time information. Once the RDO identifies hot code regions by execution thresholds, it needs to segregate them into CPU or memory bound regions, the latter being the region of interest. This data is aggregated using performance counters. Because data is collected via the run-time system, perturbation as a result of running the system itself must be minimal. Figure 3.8 shows the architectural perturbation as a result of running the run-time system. It is far from negligible. Therefore, data collected by the system is likely to be polluted. And any analysis drawn based on this data is also likely to be incorrect.

5.2.3 Experimental framework

RDO was implemented in a real system using an Intel development board, running a Pentium-M (855GME, FW82801DB) processor. The processor is capable of six DVFS settings (1.6GHz, 1.4GHz, 1.2GHz, 1.0GHz, 800MHz, and 600MHz) with two 32K L1 caches and one unified 1M L2 cache. The board has a 400MHz FSB bus and a 512M DDR RAM. The system was setup to run a Linux 2.4.18 kernel with loadable module support. The modules provide user level hardware performance counter interface and DVFS control settings in the form of system calls.

The optimization system uses the Pin run-time system as the fundamental soft-

ware platform. It supports programming interfaces for dynamic optimizations. This version of Pin is referred to as Optimization Pin (O-Pin). O-Pin has added features to support dynamic optimizations, such as adaptive code replacement and customized trace selection. It selectively generates code and does more light-weight profiling and optimization of the dynamically compiled code (i.e. loops inside a function and function entry points). It relies on mixed execution and employs techniques described in Sections 5.1.2 and 5.1.2.1 to ensure its overheads are small.

Functions and loops are transformed to use instrumentation to monitor execution thresholds and/or extract data from the hardware performance counters. Specifically, the system utilizes callsite and function invocation profiling to identify hot, as well as cold code regions. Loops are instrumented on their backedges. This determines their execution weights. If a certain threshold is met the system applies its optimizations to the loop header and exit paths. By translating a few regions of the original application, RDO achieves lesser transformation and translation overhead compared to the regular JIT-based system.

5.2.4 Methodology

Of the three key steps described in Section 5.2.1, the one relating to the work in this thesis is step two, monitoring the code execution to identify code regions that are memory bound/intensive.

Figure 5.4 illustrates the operation flow graph of the system. The RDO instruments all function calls and loops in the program to monitor and identify frequently executed code regions. If a candidate code region is found hot (i.e. the execution count is greater than a hot threshold), DVFS testing and decision code is applied to collect run-time information and decide how memory-bound the region is. This information is collected using two hardware performance counters. They keep track of the number of memory bus transactions and the total number of micro-ops retired. The ratio is used

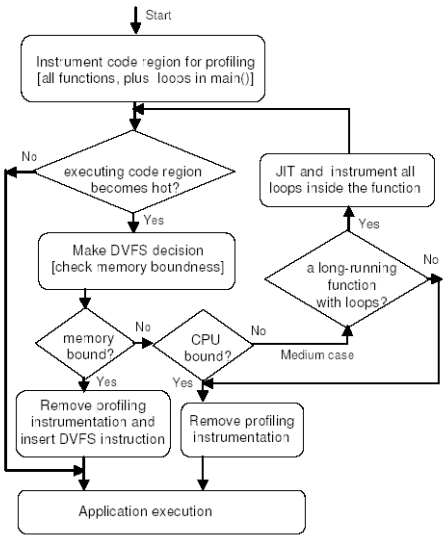


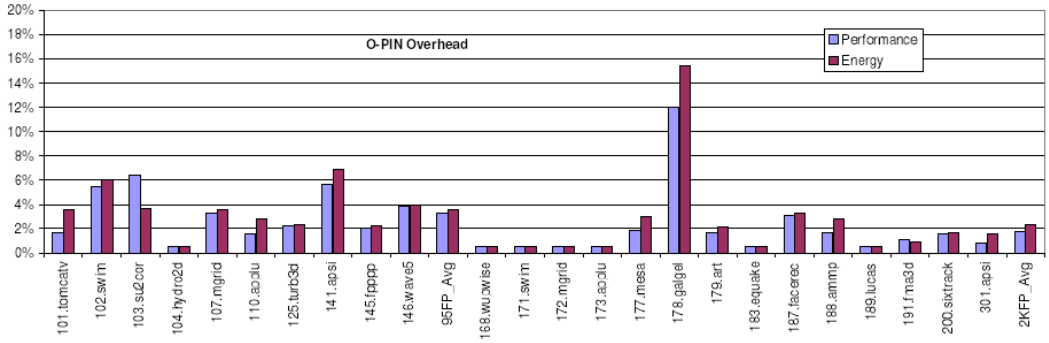
Figure 5.4: Operation flow graph of the run-time dynamic voltage and frequency scaling system.

to approximate how memory/CPU bound an application is.

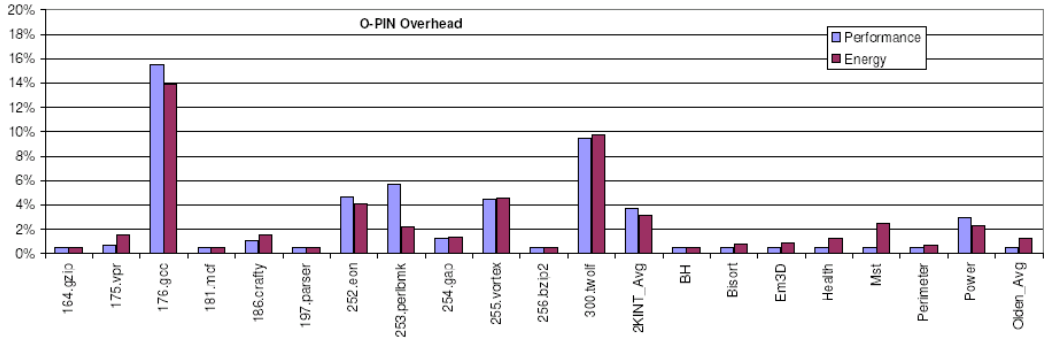
RDO removes the profiling code if a code region is found to be memory bound. It proceeds to insert DVFS mode set instructions to scale the frequency of the processor down, and resumes execution. If a region is found to be CPU bound, no DVFS instructions are inserted. However, there is still a medium case that the candidate code region exhibits mixed memory behavior. This is likely because the code region contains both memory-bound and CPU-bound sub-regions. For such cases, the system determines if it is a long-running function containing loops. So, a copy of the function is dynamically generated and all loops inside the candidate function are identified and instrumented, and the process repeats.

5.2.5 RDO baseline overhead

Since the optimization of interest is power, it is critical to identify the overhead incurred from running the RDO system along side the application, while no DVFS optimizations are deployed. The overhead is a sum of the time to select, apply, and execute



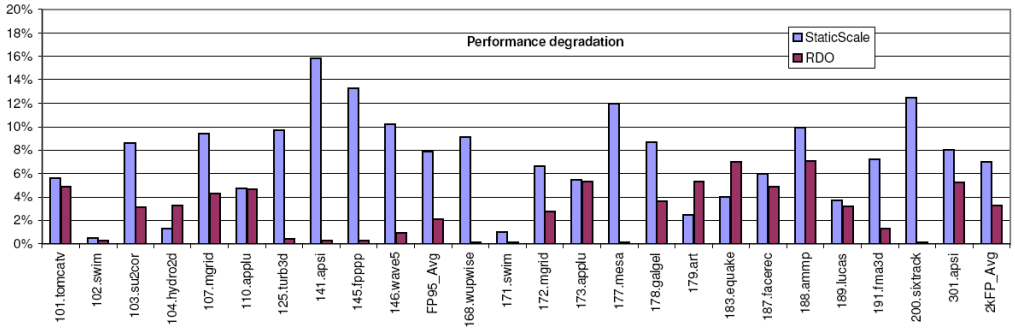
(a) *Spec95 and SPEC2K FP benchmarks*



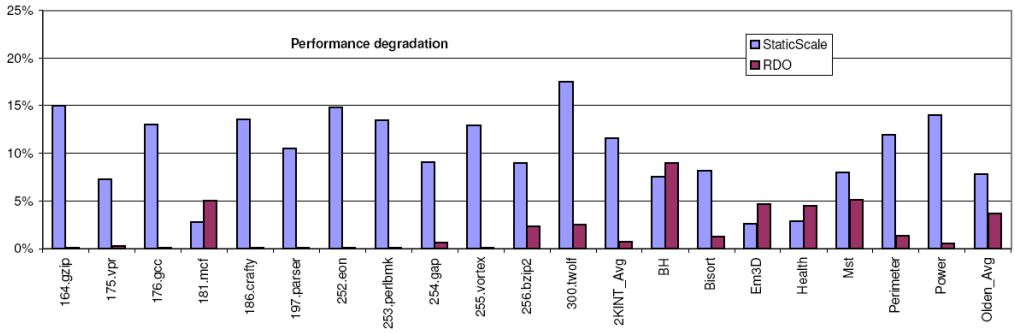
(b) *SPEC2K INT and Olden benchmarks*

Figure 5.5: Performance and energy overhead for (a) Spec95 and SPEC2K FP and (b) SPEC2K INT and Olden benchmarks resulting from the RDO system without applying DVFS optimizations.

profiling instrumentation on selected functions and loops. Section 5.2.1 details the selection and execution methodology. This overhead is overcome by the system breaking even with the native application performance. Figures 5.5 shows the performance and energy overhead for the basic O-Pin infrastructure (computed relative to the original program). For individual benchmarks, the performance overhead is as low as 0.5% for benchmarks like *164.gzip* and *171.swim*, and is as high as 15% for benchmarks like *176.gcc*. On average, the performance overhead for O-Pin is about 3.3% for SPEC95 FP, 1.8% for SPEC2K FP, 3.7% for SPEC2K INT, and 0.5% for Olden benchmarks. The energy overhead values are similar. These values are significantly lower than the basic overhead for a pure JIT-based system, because of the low-overhead mixed execution approach.



(a) *Spec95 and SPEC2K FP benchmarks*

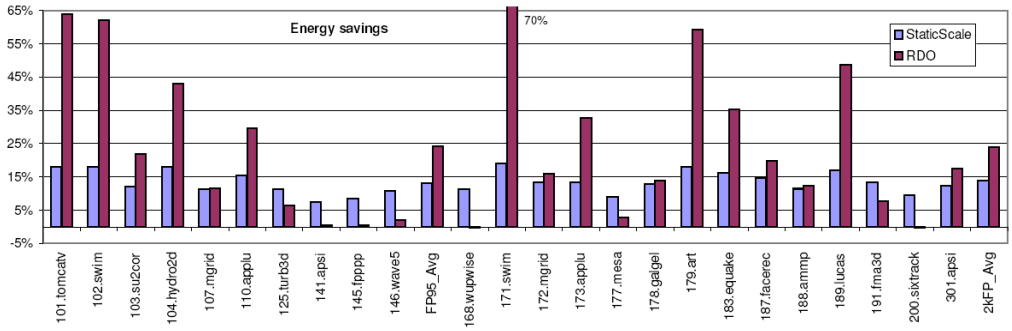


(b) *SPEC2K INT and Olden benchmarks*

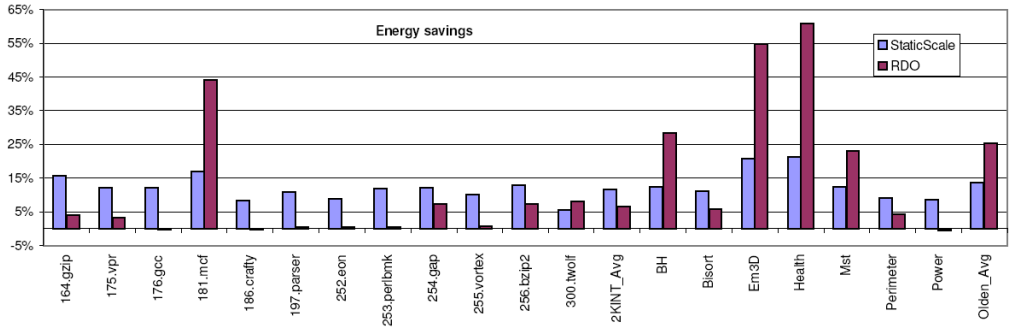
Figure 5.6: Performance degradation because of running RDO along side the application.

5.2.6 RDO performance

An application's performance is degraded when the RDO is run next to it. Monitoring application behavior and applying DVFS optimizations results in overhead that degrades performance. This is because time is consumed to run the RDO, the application makes no progress while the RDO is running. Performance degradation is shown in Figure 5.6. However, the slight performance loss is sustained by the significant power savings achieved by the RDO, shown in Figure 5.7. Both these results are compared against *Static Scale*. Static Scale refers to an established compiler DVFS optimization. The results vary significantly between the benchmark suites. On the high end, RDO achieves a maximum of 64% energy savings (4.9% performance loss) for SPEC95 FP (*101.tomcatv*), a maximum of 70% energy savings (0.5% performance loss) for SPEC2K



(a) *Spec95 and SPEC2K FP benchmarks*



(b) *SPEC2K INT and Olden benchmarks*

Figure 5.7: Energy savings achieved by the run-time dynamic voltage and frequency scaling optimizer.

FP (*171.swim*), a maximum of 44% energy savings (with 5% performance loss) for SPEC2K INT (*181.mcf*), and a maximum of 61% energy savings (4.5% performance loss) for Olden benchmarks. On the low end, the efficiency of a DVFS control is largely constrained by the memory boundedness of an application. The more memory bound an application is, the more opportunities and energy saving potentials there are for DVFS.

These results confirm that the mixed execution approach is a practical alternative to run-time system challenges. Specifically, the technique is successful when low overhead and architectural perturbation must be minimal. Additionally, despite the challenges in 5.1, the method is still capable of serving certain domains where the JIT-based approach is impractical. Complete details on RDO are available in [38].

Chapter 6

Summary and conclusion

Dynamic code transformation systems contribute significantly to computing environments by providing services such as program optimization, security, ISA translation etc. Despite their potential, these systems are yet to become prevalent. Run-time systems are faced with the fundamental challenge of keeping the execution overhead minimal. This overhead is an aggregate of the *transformation cost* and the *translated code performance*. This thesis presents these overheads in light of application's behavior, as well as the design of a run-time system.

Existing run-time system overheads must be understood and mitigated for these systems to become part of mainstream computing. However, applications and run-time systems exhibit certain traits that are leveraged to the benefit of code transformation systems. This thesis presents, *persistence* and *mixed execution*, two execution models that gain from distinct aspects of an application and a run-time system to address the execution overheads. These aspects are exploited in the proposed models to mitigate the transformation and translated code performance penalties. These executions models are evaluated in a state-of-the-art Just-In-Time based code transformation system. Their evaluation leads to the conclusion that, in certain computing environments, existing dynamic code transformation systems can be successfully deployed.

Persistence exploits the fact that applications exhibit minimal execution path variance across input data sets. As such, it addresses the substantial overhead due to

code transformation by performing execution caching. Subsequent invocations of the same program experience lower run-time transformation overhead due to the re-use of already translated code sequences. The concept is evaluated in a run-time binary instrumentation engine, Pin, creating a system known as Persistent Pin (PPin). PPin is capable of caching instrumented program executions and reusing the cached instrumented executions in separate invocations. Additionally, it is capable of caching new code paths in subsequent executions, if the application changes behavior from prior executions and supports multi-threaded applications. The model is shown to be very effective for Pin. Experimental data shows code transformation overhead reductions up to 90% for interative graphic program start-up costs and up to 25% for SPEC benchmarks for instrumented executions.

Mixed execution addresses a challenging aspect of run-time systems which is poor performance of the translated code. Thus far, solutions addressing the translated code performance problem have been unsuccessful. As such, mixed execution deals with the problem by executing a mix of the original and translated code, achieving the required task. As a result, minimal translation overheads are incurred relative to its pure JIT-based run-time system approach. It is a practical alternative to pure when extremely low overheads and minimal architectural perturbation are the critical constraints.

This thesis contributes a new concept called *function migration*, that facilitates mixed execution using the fundamental technique, code splicing. The concept addresses an existing code splicing limitation of possible breakage of execution if smaller instructions are overwritten with a larger instruction. An implementation of the concept enables safe and fine-grained transformation of x86 instructions. This is accomplished by generating a transformed copy of the original function in the code cache.

Mixed execution's practicality, in contrast to its JIT counterpart, is evaluated by addressing the issue of microprocessor power consumption within a run-time system. The goal of the system is to achieve power savings without considerable performance

loss. To successfully accomplish its task, the system itself needs to exhibit minimal overheads. Using the mixed execution approach, the system successfully achieves energy savings up to 70% for applications in the SPEC benchmark suite with an approximate 4% performance degradation in execution time.

The above execution models successfully address the transformation and translation overheads discussed in this work. However, domains exist in which either one of the methods are ill-suited. It is possible to surpass this obstacle with more support from the host environment, or software and hardware layers.

In summarizing, this work concludes that while general-purpose computing environments have much to gain from run-time code transformation, significant amounts of future research is required in order to further understand their practical deployment.

Bibliography

[1] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In Proceedings of the 16th ACM Symposium of Operating Systems Principles, pages 1–14, October 1997.

[2] M. W. Shapiro B. M. Cantrill and A. H. Leventhal. Dynamic instrumentation of production systems. In Proceedings of the 2004 USENIX Conference, pages 15–28.

[3] 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.

[4] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigal Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In Proceedings of the 36th International Symposium on Microarchitecture, December 2003.

[5] B. Brock and K. Rajamani. Dynamic power management for embedded systems. In Proceedings of the IEEE SOC Conference, Sep 2003.

[6] Derek Bruening, Timothy Garnett, and Saman 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.

[7] Bryan R. Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. In Journal of High Performance Computing Applications 14 (4), Winter 2000.

[8] D. Maier J. Walpole-P. Bakke S. Beattie A. Grier P. Wagle Q. Zhang H. Hinton C. Cowan, C. Pu. In Proceedings of the 1998 USENIX Security Symposium, pages 63–78, January 1998.

[9] S. Beattie Greg Kroah-Hartman C. Cowan, M. Barringer. Formatguard: Automatic protection from printf format string vulnerabilities. In Proceedings of the 2001 USENIX Security Symposium, August 2001.

[10] Cache Performance for SPEC CPU2000 Benchmarks. http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/.

[11] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3), December 2000.

[12] Robert S. Cohn, David W. Goodwin, and P. Geoffrey Lowney. Optimizing alpha executables on windows nt with spike. Digital Technical Journal, 9(4):3–20, 1998.

[13] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. DELI: A new run-time control point. In 35th Annual International Symposium on Microarchitecture, December 2003.

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

[15] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In Proceedings of the 10th USENIX Security Symposium, r001.

[16] S. Gochman, R. Ronen, and Others. The Intel Pentium M processor: Microarchitecture and performance. Intel Technology Journal, 07(2), 2003.

[17] Kim Hazelwood. Code cache management in dynamic optimization systems. Master's thesis, Department of Computer Science, Harvard University, Cambridge, Massachusetts, May 2005.

[18] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. Computer, 33(7):28–35, July 2000.

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

[20] C-H Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In Proc. of PLDI-2003, pages 38–48, June 2003.

[21] Shiwen Hu, Madhavi Valluri, and Lizy Kurian John. Effective adaptive computing environment management via dynamic optimization. In CGO '05: Proceedings of the international symposium on Code generation and optimization, pages 63–73, Washington, DC, USA, 2005. IEEE Computer Society.

[22] M.C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In Proceedings of ISCA, June 2003.

[23] C. J. Hughes and S. V. Adve. A formal approach to frequent energy adaptations for multimedia applications. In Proc. of 31st ISCA, June 2004.

[24] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In Proceedings of the 3rd USENIX Windows NT Symposium, pages 135–143, July 1999.

[25] Intel Virtualization Technology. http://www.intel.com/technology/computing/vptech/.

[26] John P. Banning-Richard Johnson Thomas Kistler Alexander Klaiber Jim Mattson James C. Dehnert, Brian Grant.

[27] M. Martonosi K. Skadron, P. S. Ahuja and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In Proceedings of the 31st International Symposium on Microarchitecture, December 1998.

[28] T. Kistler and M. Franz. Continuous program optimization. In IEEE Transactions on Computers vol. 50 n. 6, June 2001.

[29] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. In Journal of Instruction-Level Parallelism 6(2004), pages 1–24, April 2004.

[30] C-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Ispike: A post-link optimizer for the intel itanium architecture by chi-keung luk, robert muth, harish patil, robert cohn, geoff lowney (intel). In Proceedings of the 2nd International Symposium on Code Generation and Optimization, March 2004.

[31] 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.

[32] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. In In Workshop on Complexity Effective Design, Vancouver, Canada, June 2000., June 2000.

[33] John Markoff and Laurie J. Flynn. Apple's next test: Get developers to write programs for intel chips. June 2005.

[34] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In Proceedings of the 3rd Workshop on Runtime Verification, July 2003.

[35] Paradyn Parallel Performance Tools. http://www.paradyn.org/.

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

[37] David J. Pearce, Paul H. J. Kelly, Tony Field, and Uli Harder. GILK: A dynamic instrumentation tool for the linux kernel. In Proceedings of the 12th International Conference on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation (TOOLS '02), pages 220–226, 2002.

[38] Youfeng Wu Jin Lee-Dan Connors David Brooks Margaret Martonosi Doughlas W. Clark Qiang Wu, V.J. Reddi. A dynamic compilation framework for controlloing microprocessor energy and performance. In Proceedings of the 38th Annual International Symposium on Microarchitecture (MICRO-38), November 2005.

[39] G. Semeraro, D.H. Albonesi, S.G. Dropsho, G. Magklis, S. Dwarkadas, and M.L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In Proc. of the 35th Micro, pages 356–367, November 2002.

[40] M. Smith. Overcoming the challenges to feedback-directed optimization. In ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, 2000.

[41] A. Srivastava. Vulcan. Technical Report TR-99-76, Microsoft Research, September 1999.

[42] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In Operating Systems Design and Implementation, pages 117–130, 1999.

[43] The HP Digital Continuous Profiling Infrastructure (DCPI). http://h30097.www3.hp.com/dcpi/documentation.htm.

[44] V. Venkatachalam M. Franz V. Haldar, C. Probst. Virtual machine driven dynamic volrate scaling. Technical Report CS-03-21, Department of Computer Science, University of California, Irvine, October 2003.

[45] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In Proceedings of CASE'02, Oct 2002.

[46] Wei-Chung Hsu Pen-Chung Yew Xiaoru Dai, Antonia Zhai. A general compiler framework for speculative optimizations using data speculative code motion. In 3rd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA, pages 89–96, March 2005.

[47] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pages 49–62, New York, NY, USA, 2003. ACM Press.

[48] Cindy Zheng and Carol Thompson. Pa-risc to ia-64: Transparent execution, no recompilation. Computer, 33(3):47–52, 2000.