Characterizing Inter-Execution and Inter-Application Optimization Persistence

Kim Hazelwood and Michael D. Smith
Division of Engineering and Applied Sciences
Harvard University
{hazelwood, smith}@eecs.harvard.edu

Abstract

Dynamic translation and optimization systems employ code caches to improve performance and to support reuse of dynamically generated code sequences within a single run of an application. However, these intra-application caching techniques are ineffective at amortizing runtime costs on short-running applications or scripts that repeatedly invoke the same application. For these situations, dynamic translation systems have successfully used persistent code caching—saving the code cache during one run to prime later executions—as a means for amortizing runtime translation costs. The success of persistent code caching in dynamic optimizers will depend heavily on the amount of inter-execution and inter-application *optimization persistence* that can be found in software applications. Our experiments with the DynamoRIO dynamic optimization system demonstrate that many of the most heavily executed code traces in SPEC-CPU2000 are identically optimized during successive executions. Our results indicate that there is a significant opportunity for leveraging inter-execution persistence, and there is even a small opportunity for inter-application persistence for small programs.

1 Introduction

Dynamic optimization is a means for leveraging information that is available at runtime to improve program performance. This information may include runtime constants, user preferences and access patterns, and traits of the particular processor architecture or memory system. Runtime data is used to apply late and/or temporal optimizations [17] to program binaries, typically without the luxury of source code. Dynamic optimization is an invaluable technique for recovering the performance opportunities missed by shipping standardized, ISA-compatible executables.

Dynamic translation has been used to produce fast simulators [7] and efficient emulators [11]. Runtime information is used to avoid the need for source code and the problems of code discovery inherent in static translation techniques. Dynamic translation is an invaluable technique for code compatibility and design space exploration.

The major cost of these dynamic techniques is that, unlike static techniques, the overhead of dynamic translation or optimization directly impacts the runtime performance of the executing program. Therefore, designers often rely on caching of the dynamically generated code to amortize the runtime costs of the dynamic system. This paper explores the question of when these cached code sequences should persist to drive the overhead of dynamic techniques as low as possible. In particular, we are primarily interested in dynamic optimization systems, for reasons stated below, though all of our findings are generally applicable to dynamic translation systems as well.

Most modern dynamic translation and optimization systems cache the code sequences generated during a program's execution for that execution. We refer to this approach as one that exploits *intra-execution persistence*. This approach works well when applications exhibit a sizable amount of code reuse and a large amount of reuse for individual sequences. Long-running programs often exhibit such characteristics, but this is not typically the case for short-running programs. Yet, the performance of short-running programs can be important if most of the application workload comprises such applications or if long-running scripts repeatedly invoke short-running programs.

To address the problem of short-running applications, a dynamic translator or optimizer will

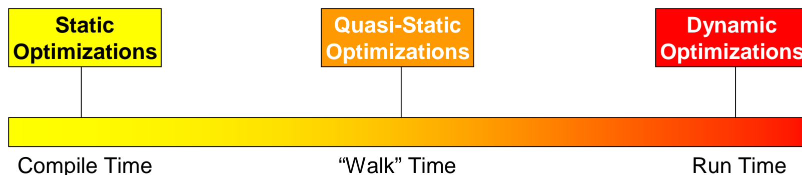


Figure 1: The optimization opportunity scale. On the far left, at compile time, we may apply a standard set of universally safe optimizations. On the far right, at runtime, we may apply dynamic input-dependent and otherwise unsafe optimizations. In this paper, we explore the middle domain, known as walk time [13]. Here we may apply machine or user-dependent optimizations.

need to amortize its costs over multiple executions of the short-running application, an approach we refer to as one that exploits *inter-execution persistence*. Though some dynamic translation systems [16] reuse translations from earlier program runs, current dynamic optimization infrastructures do not support inter-execution persistence of optimized code sequences. We refer to persistence of optimized code sequences as *optimization persistence*. Understanding optimization persistence is an important step in building such functionality into dynamic optimization systems.

In this paper, we find that there is a sizable amount of inter-execution optimization persistence in SPECCPU2000 under the DynamoRIO dynamic optimization system. By understanding the type of inter-execution optimization persistence that exists, we can in the future consider techniques that reduce the overhead of dynamic optimization by replacing the cost of start-up profiling and initial optimization generation with the cost of priming the dynamic optimizer's code cache with persistent optimizations.

Once we are focused on code reuse beyond the boundaries of a program's execution, we can consider using optimizations and translations from past application executions for new programs that have never previously executed. Interestingly, we show that a non-trivial amount of optimized code persists across program boundaries. That is, identical code traces appear in the DynamoRIO code cache despite the fact that different programs are executing. This result points out the potential benefit of *inter-application persistence*.

We can explain inter-application optimization persistence by realizing that the DynamoRIO optimizes across module boundaries. Therefore, by priming the code cache with optimized code traces from commonly executed shared libraries, we may achieve overhead reductions even during the first execution of a new application.

In this paper, we focus on systems for dynamic optimization, because the least is known about the beneficial effects of inter-execution and inter-application optimization persistence on these types of systems. The paper analyzes the potential overhead reduction possible in the DynamoRIO dynamic optimization system when dynamically optimized portions of code are maintained after program termination for use in successive executions. This is accomplished by analyzing the amount of potential persistence that occurs in a dynamic optimizer, first for successive executions of the same program (with changing inputs), then for different programs. Section 2 discusses current dynamic optimization efforts, the mechanisms they use to reduce overhead, and the benefits of persistence known today. Section 3 describes the DynamoRIO dynamic optimization system we used for gathering our results. Section 4 then discusses our findings in inter-execution and inter-application persistence. Finally, Section 5 concludes.

2 Dynamic Optimization and Caching Techniques

Several groups are currently developing dynamic optimization systems. Dynamo [3] is a system developed at Hewlett-Packard Laboratories that provides a software-based mechanism for selecting and optimizing program traces. Several successors to Dynamo have recently surfaced, such as DELI and DynamoRIO. DELI [12] is a VLIW version of Dynamo geared toward embedded processor applications, whereas DynamoRIO [4] is an IA-32 (Windows and Linux) version that is intended for academic research. Wiggins/Redstone [10] is a dynamic optimization and specialization system developed at Compaq, and Jikes RVM is a Java VM that provides adaptive optimization [1]. Finally, Microsoft's

Mojo [6] is a dynamic optimizer that focuses on optimizing IA-32/WinNT binaries.

Each of these dynamic optimization systems performs the following tasks:

- runtime profiling to determine frequently-executed (hot) portions of code;
- copying the hot code into a software-based cache (possibly performing optimizations on the code en route);
- executing future instances of the hot code directly from the software-based cache.

Speedups result not only from the optimizations performed on frequently-executed code, but due to improved instruction cache locality from code reordering and layout.

One of the main challenges for the dynamic optimization infrastructure designer is maintaining reasonable system overhead. The major contributor to the runtime overhead of dynamic optimization systems is the cost of online profiling and optimization. Online profiling includes the time that the dynamic optimizer spends determining program hot spots and collecting profile information. Low-overhead profiling techniques, such as *instrumentation sampling* [2], have been proposed and demonstrated to reduce this cost. Similarly, work exists on lightweight, low-overhead optimizations (e.g., see [14]). Yet little research has focused on techniques for reducing or eliminating the underlying need for start-up profiling and re-optimization in general. In particular, none of the systems mentioned previously employs a persistent caching scheme for storing dynamically optimized code between successive program executions.

Fisher [13], in his paper on *walk-time* techniques, describes an optimization spectrum, depicted in Figure 1, and discusses the types of optimization opportunities in each major category of this spectrum. Optimization persistence as we describe it clearly falls into Fisher's walk-time opportunity category. Rather than rediscover such optimization opportunities during every execution, a primed dynamic optimizer will be able to benefit from the work it performed during a prior execution.

Persistent state has been maintained for several other applications. For example, Hazelwood et al. [15] explored maintaining a trace of program memory accesses for reducing data cache misses during future executions. In addition, Conte et al. [9] explored hardware techniques for maintaining rescheduled pages of VLIW code for use in future program executions. FX!32 [16] exploits persistence when storing program translations between executions, and Spike [8] performs persistent optimizations, such as code placement, offline.

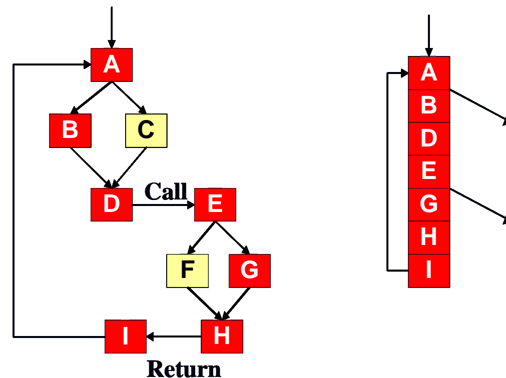


Figure 2: An example of a code trace. On the left is the dynamic instruction execution, with the actual executed blocks shaded. On the right is the corresponding code trace produced by DynamoRIO.

3 Experimental Procedure

The dynamic optimization infrastructure selected for our experiments was the x86/Linux version of DynamoRIO. DynamoRIO is a black-box research infrastructure. It is equipped with a set of low-level application programming interfaces (APIs) for dynamically modifying an application's text image as it runs, and relaying pertinent information back to the user regarding the progress of dynamic optimization. All results were generated on a Pentium-4 2.5 GHz processor system with 2GB RAM running Red Hat Linux 7.2.

DynamoRIO works by injecting itself into a running executable at load time and taking control of execution. Frequently executed portions of code are optimized and cached at the granularity of a code trace. A code trace is a superblock region that contains a single entry point and multiple exit points, as shown in Figure 2. In order to simplify optimization processing, internal loops and side entries are not allowed. However, a trace may span procedure boundaries, exposing optimization opportunities that are available only once the dynamic libraries have been loaded. Not only does this increase the optimization instruction window, but the code layout also results in improved spatial locality in the instruction stream.

The official SPECCPU2000 test, training, and

	test	training	reference
gzip	352	389	440
vpr	1,040	1,086	1,184
gcc	11,304	10,878	18,450
mcf	208	253	240
crafty	1,758	1,901	2,027
parser	2,526	2,542	2,883
eon	755	780	861
perlbmk	1,216	2,805	4,345
gap	2,454	2,805	2,692
vortex	2,073	2,074	2,198
bzip2	249	317	442
twolf	888	1,575	1,573
ammp	321	380	414
art	116	250	269
equake	283	394	400
mesa	111	232	311

Figure 3: The number of *hot traces* recognized by the dynamic optimizer for our benchmarks and input sets.

reference inputs were run to completion under the control of DynamoRIO using the `runspec` script. Flags were set indicating that DynamoRIO should output a log of all actions performed, including hot trace detection, optimization, and caching. Figure 3 shows the number of code traces that DynamoRIO recognized as *hot* and optimized.

4 Optimization Persistence

Just as poor profile data can adversely affect the performance of an executable produced by a static compiler, low correlation between successive runs of an executable could result in poor dynamic optimization performance if persistent optimization state is maintained.

Therefore, before it will be possible for researchers to propose or design an optimization caching technique that will store optimizations between program executions, it is first worthwhile to clearly understand how much opportunity exists. We did so by measuring exactly how much work was duplicated during successive program executions. In this case, *program executions* can be defined as either successive executions of the *same* program or successive executions of *different* programs. We refer to the former case as *inter-execution persistence*, and the latter case as *inter-application persistence*. We investigated both opportunities. Our particular experiments and results are discussed in the following subsections.

4.1 Inter-Execution Persistence

Our first study looked into the correlation of frequently executed instruction sequences from one execution of a single program to the next with changing program inputs. We performed this study by running the SPECCPU2000 benchmarks to completion under the control of the DynamoRIO dynamic optimization system.

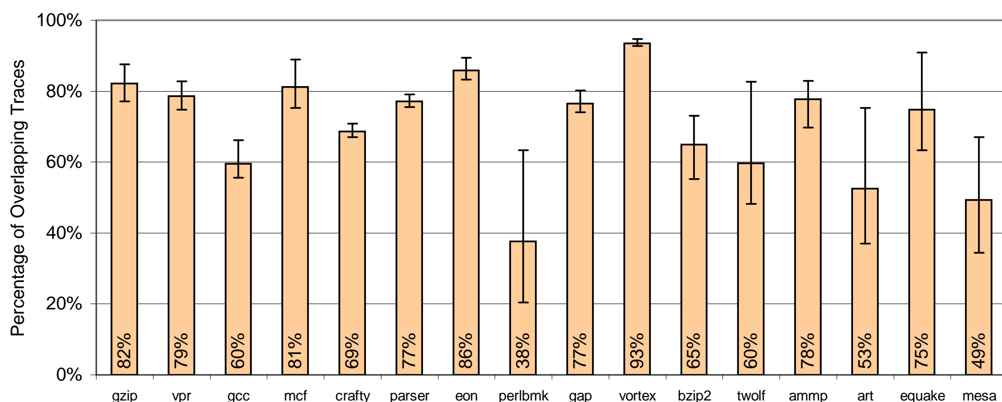
By setting an environment variable, we were able to request that DynamoRIO log all of the hot instruction traces after program completion. Instruction traces chosen for optimization were compared with one another as we varied the input set. This comparison was performed offline after the benchmarks had completed execution.

The SPECCPU2000 benchmarks have three official input sets per benchmark – test, train, and reference. We used these three input sets to study the effect of changing input sets on dynamic optimization decisions. For our experiments, we performed comparisons of all permutations of the input sets, then calculated the overlapping percentage using Equation 1.

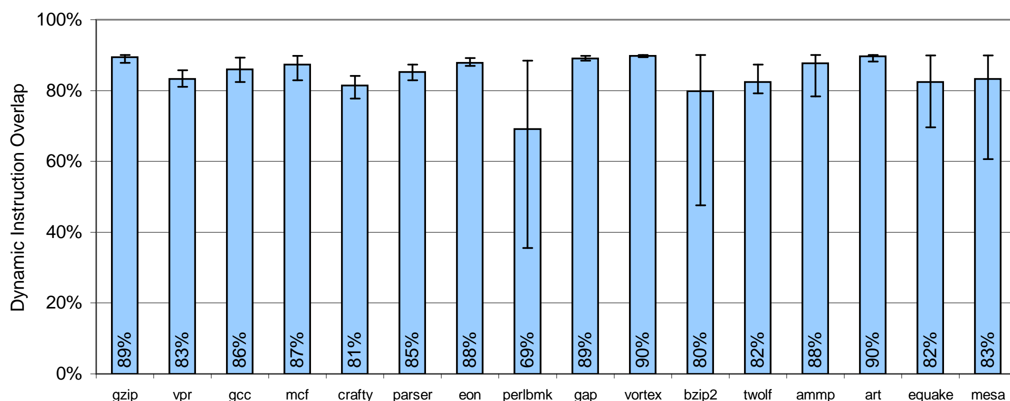
$$\frac{|hotTraces_A \cap hotTraces_B|}{|hotTraces_A|} \tag{1}$$

We then averaged the results of the six permutations to form the results shown in Figure 4. The average values are shown in the wide graph bars, while the ranges are depicted in the error bars. From Figure 4a, we can make the overall observation that on average 70% of the optimized code produced by DynamoRIO is regenerated every execution, regardless of program inputs. In the best case, `vortex` resulted in 93% persistence, while in the worst case, `perlbmk` showed only 38% persistence. As `perlbmk` is an interpreter program, its dynamic behavior is highly dependent on the program input, therefore the low persistence and high variance is not surprising. Additionaly, `art` and `mesa` resulted in less-than-average persistence; however, in their case, this can be attributed to the fact that they are small programs (producing only 269 and 311 traces, respectively) and therefore small changes in their code traces resulted in a higher percentage of trace changes.

Figure 4b shows the percentage of dynamic instructions that can be attributed to the persistent code. Interestingly, a consistently high percentage of time is spent in the code that persists from one run to the next. Even in the case of `perlbmk`, where only 38% of the instruction traces were regenerated

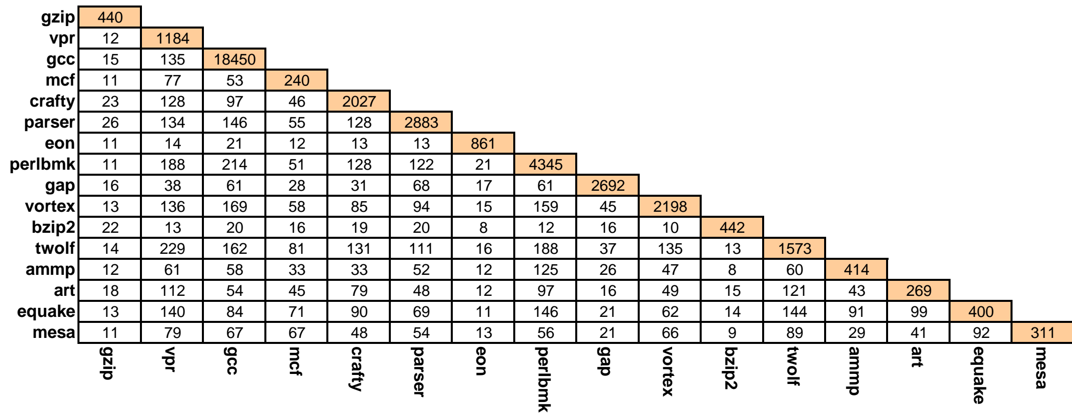


(a) Percentage of Common Traces



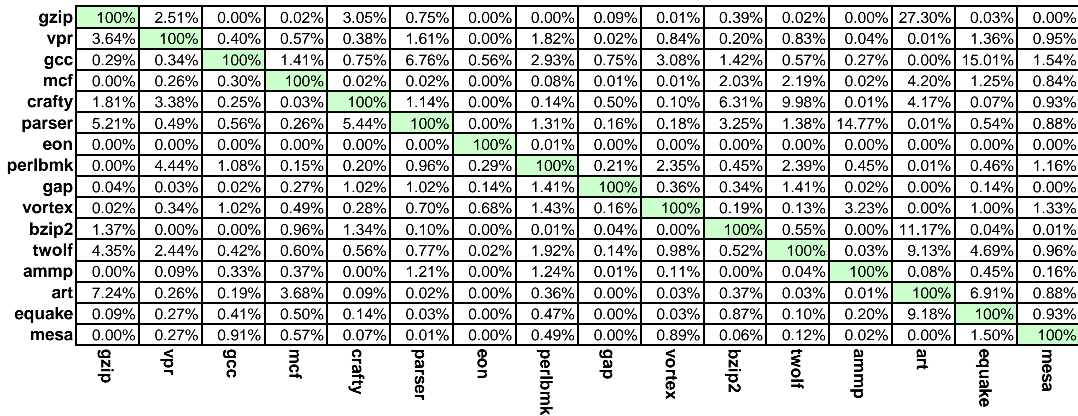
(b) Percentage of Overlapping Dynamic Instructions

Figure 4: This figure depicts the inter-execution persistence occurring between the different input sets of the SPECCPU2000 benchmarks. On the top, we show the percentage of traces that reappear in the code cache during successive executions, regardless of the changing input set. On the bottom, we show the percentage of dynamic instruction count spent in the code traces that reappear during successive executions.



	gzip	vpr	gcc	mcf	crafty	parser	eon	perlbmk	gap	vortex	bzip2	twolf	ammp	art	equake	mesa
gzip	440															
vpr	12	1184														
gcc	15	135	18450													
mcf	11	77	53	240												
crafty	23	128	97	46	2027											
parser	26	134	146	55	128	2883										
eon	11	14	21	12	13	13	861									
perlbmk	11	188	214	51	128	122	21	4345								
gap	16	38	61	28	31	68	17	61	2692							
vortex	13	136	169	58	85	94	15	159	45	2198						
bzip2	22	13	20	16	19	20	8	12	16	10	442					
twolf	14	229	162	81	131	111	16	188	37	135	13	1573				
ammp	12	61	58	33	33	52	12	125	26	47	8	60	414			
art	18	112	54	45	79	48	12	97	16	49	15	121	43	269		
equake	13	140	84	71	90	69	11	146	21	62	14	144	91	99	400	
mesa	11	79	67	67	48	54	13	56	21	66	9	89	29	41	92	311

(a) Count of Common Traces



	gzip	vpr	gcc	mcf	crafty	parser	eon	perlbmk	gap	vortex	bzip2	twolf	ammp	art	equake	mesa
gzip	100%	2.51%	0.00%	0.02%	3.05%	0.75%	0.00%	0.00%	0.09%	0.01%	0.39%	0.02%	0.00%	27.30%	0.03%	0.00%
vpr	3.64%	100%	0.40%	0.57%	0.38%	1.61%	0.00%	1.82%	0.02%	0.84%	0.20%	0.83%	0.04%	0.01%	1.36%	0.95%
gcc	0.29%	0.34%	100%	1.41%	0.75%	6.76%	0.56%	2.93%	0.75%	3.08%	1.42%	0.57%	0.27%	0.00%	15.01%	1.54%
mcf	0.00%	0.26%	0.30%	100%	0.02%	0.02%	0.00%	0.08%	0.01%	0.01%	2.03%	2.19%	0.02%	4.20%	1.25%	0.84%
crafty	1.81%	3.38%	0.25%	0.03%	100%	1.14%	0.00%	0.14%	0.50%	0.10%	6.31%	9.98%	0.01%	4.17%	0.07%	0.93%
parser	5.21%	0.49%	0.56%	0.26%	5.44%	100%	0.00%	1.31%	0.16%	0.18%	3.25%	1.38%	14.77%	0.01%	0.54%	0.88%
eon	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100%	0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
perlbmk	0.00%	4.44%	1.08%	0.15%	0.20%	0.96%	0.29%	100%	0.21%	2.35%	0.45%	2.39%	0.45%	0.01%	0.46%	1.16%
gap	0.04%	0.03%	0.02%	0.27%	1.02%	1.02%	0.14%	1.41%	100%	0.36%	0.34%	1.41%	0.02%	0.00%	0.14%	0.00%
vortex	0.02%	0.34%	1.02%	0.49%	0.28%	0.70%	0.68%	1.43%	0.16%	100%	0.19%	0.13%	3.23%	0.00%	1.00%	1.33%
bzip2	1.37%	0.00%	0.00%	0.96%	1.34%	0.10%	0.00%	0.01%	0.04%	0.00%	100%	0.55%	0.00%	11.17%	0.04%	0.01%
twolf	4.35%	2.44%	0.42%	0.60%	0.56%	0.77%	0.02%	1.92%	0.14%	0.98%	0.52%	100%	0.03%	9.13%	4.69%	0.96%
ammp	0.00%	0.09%	0.33%	0.37%	0.00%	1.21%	0.00%	1.24%	0.01%	0.11%	0.00%	0.04%	100%	0.08%	0.45%	0.16%
art	7.24%	0.26%	0.19%	3.68%	0.09%	0.02%	0.00%	0.36%	0.00%	0.03%	0.37%	0.03%	0.01%	100%	6.91%	0.88%
equake	0.09%	0.27%	0.41%	0.50%	0.14%	0.03%	0.00%	0.47%	0.00%	0.03%	0.87%	0.10%	0.20%	9.18%	100%	0.93%
mesa	0.00%	0.27%	0.91%	0.57%	0.07%	0.01%	0.00%	0.49%	0.00%	0.89%	0.06%	0.12%	0.02%	0.00%	1.50%	100%

(b) Percentage of Overlapping Dynamic Instructions

Figure 5: This figure depicts the inter-application persistence occurring between the different benchmarks in SPECCPU2000. On the top, we show the number of traces that appear in code caches of both benchmark programs. On the bottom, we show the percentage of dynamic instructions corresponding to the overlapping code traces. Each column lists the percentage of dynamic instructions spent executing code that appeared in each row.

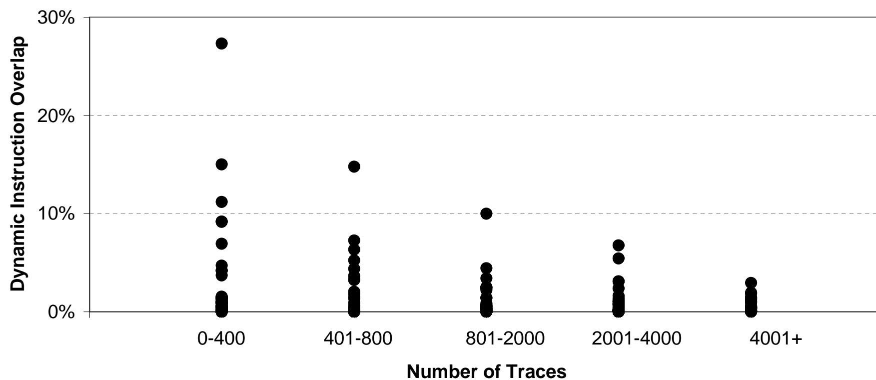


Figure 6: In this scatter plot, benchmarks are bucketed based on the number of traces produced during their execution, then their overlapping dynamic instructions are shown as graph points.

during successive executions, those 38% of traces accounted for 69% of the dynamic instruction count. Similarly, for the small benchmarks `art` and `mesa`, their less-than-average trace persistence ended up accounting for 90% and 83% of the dynamic instruction count, respectively.

While our original hope was to see a high inter-execution optimization persistence to reduce the overhead of short-running programs, we actually found high persistence regardless of program size or execution duration.

4.2 Inter-Application Persistence

Our next set of experiments attempted to quantify the amount of persistence that occurs between different programs. Using a technique similar to that of Section 4.1, we ran each of the SPECCPU2000 benchmarks using the reference input set under the control of DynamoRIO. Again, we instructed DynamoRIO to output a log of all optimized code traces. We compared the log files of every permutation of the benchmarks to determine the overlap that occurred between different programs.

Intuitively, one might assume that there is no common code between any of the SPECCPU2000 benchmarks. Encouraged by the work of Calder et al. [5] who showed that shared libraries exhibit very predictable branch behavior from one execution to the next, we were interested in determining if similar conclusions could be made in the optimization domain. Our results show that up to 32% of optimized traces generated by DynamoRIO are also produced while executing a different benchmark. Since the traces generated by DynamoRIO can span procedure boundaries, the dynamic optimizer has the ability to optimize code contained inside system libraries. Therefore, optimized code from the standard C libraries (such as the `printf` routine) will often be duplicated in many different applications.

Figure 5 shows the number of common traces between each of the SPECCPU2000 benchmarks using the reference inputs. The top half of the figure shows the number of overlapping traces, and the bottom half shows the resulting dynamic instruction percentage that is spent executing the overlapping code. From the tables, we see that there is up to 32% trace commonality between different programs. We see that while the traces generated for `gzip` account for 27.3% of the instruction execution of `art`, only 0.0008% of `gcc`'s instruction can be attributed to instructions also appearing in `gzip`.

If we categorize each benchmark according to the number of traces generated during execution, we begin to see an interesting trend. Figure 6 groups the benchmarks into five categories: very small (0-400 traces), small (401-800 traces), medium (801-2000 traces), large (2001-4000 traces) and very large (4001+ traces.) It then shows the overlapping dynamic instruction count experienced by each of the benchmarks falling into the respective buckets. As program size increases (as measured by the number of traces produced during execution), the opportunity for inter-application persistence diminishes. While programs in the *very small* category experienced up to 27% persistence, those in the *very large* category never experience more than 3% persistence. Therefore, we can conclude that while opportunities do exist for leveraging inter-application persistence, the benefits are skewed toward smaller programs.

5 Conclusions

Analysis of the amount of optimization persistence that occurs can improve dynamic optimization systems in several ways. Whether the end goal is one of optimization or compatibility, it is certainly useful to recognize whether or not earlier efforts can be reused to reduce future overhead. In this paper, we analyzed the number of optimizations that were repeated during successive program executions, both for the same and for different programs. We found that regardless of changing inputs, on average 70% of the optimizations performed during one execution are repeated during successive executions, and those optimized code traces account for roughly 85% of the dynamic instruction count. Furthermore, we found that up to 27% of a short-running program's dynamic instructions were even being spent in identical code to that of a different program.

6 Future Work

The future work that this research enables falls into two categories. First, there are ways that we could extend our persistence analysis to gain a deeper understanding of the problem at hand. For example, we could investigate the effect of various individual optimizations on persistence. This would allow us to categorize each optimization in terms of inter-execution robustness, and fill the pool of retained code with machine- and environment-specific optimizations, rather than input-specific optimizations. We could also study the amount of optimization per-

sistence that occurs in larger benchmarks, such as the Microsoft Office suite, to determine if our conclusions also hold for interactive applications.

The second category of future work includes technologies that are enabled by the understanding of optimization persistence. A logical follow-up to this research is to modify a modern dynamic optimizer such that persistent optimization state is maintained between program executions. This would allow us to quantify the engineering costs of loading persistent data, and also explore whether we should always load persistent data, or first compare an initial profile with summary data from the previous execution.

There are several adjacent areas of future work. One area involves novel storage formats that are specialized for storing dynamic optimization data. There exists a body of prior research describing techniques for storing profiling data between program executions, such as the work surrounding Digital's FX!32 dynamic translator [16] and the persistent rescheduled-page cache [9]. It would be interesting to determine whether conclusions made in these related domains can be directly applied to dynamic optimization systems.

Acknowledgments

We wish to acknowledge Hewlett-Packard Company and MIT for the use of DynamoRIO. We would also like to thank Glenn Holloway, Omri Traub, and the anonymous reviewers for their helpful review of this paper.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proc. of the 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[2] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

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

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

[5] B. Calder, D. Grunwald, and A. Srivastava. The predictability of branches in libraries. In *Proc. of 28th International Symposium on Microarchitecture*, pages 24–34, 1995.

[6] W.-K. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In *Proc. of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 81–90, 2000.

[7] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[8] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: An optimizer for alpha/nt executables. In *USENIX Windows NT Workshop*, pages 17–24, 1997.

[9] T. Conte, S. Sathaye, and S. Banerjia. A persistent rescheduled-page cache for low-overhead object code compatibility. In *Proc. of the 29th Symposium on Microarchitecture*, pages 4–13, 1996.

[10] D. Deaver, R. Gorton, and N. Rubin. Wiggins/redstone: An on-line program specializer. In *Proc. of IEEE Hot Chips XI*, 1999.

[11] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. of Symposium on Code Generation and Optimization*, pages 15–24, 2003.

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

[13] J. A. Fisher. Walk-time techniques: Catalyst for architectural change. *IEEE Computer*, 30(9):40–42, September 1997.

[14] K. Hazelwood and T. M. Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization. In *Proc. of the 2000 Conference on Parallel Architectures and Compilation Techniques*, pages 71–80, 2000.

[15] K. Hazelwood, M. C. Toburen, and T. M. Conte. A case for exploiting memory-access persistence. In *Proc. of the 2001 Workshop on Memory Performance Issues*, 2001.

[16] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, pages 3–12, February 1997.

[17] M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, pages 1–11, 2000.