A Brief Survey of Profile-Guided Optimization Techniques and Implementation Strategies

CS243: High Performance Architectures and Their Compilers

Christopher A. Wood

December 13th 2013

Abstract. This report contains a summary of a subset of work related to profile-guided optimizations found in the literature. We provide summaries of each studied work and conclude with overarching thoughts about the state of the field, challenges that need to be addressed, and opportunities for future work. This literature survey was conducted in parallel with our own PGO optimization pass implementations for the LLVM compiler infrastructure in order to better interpret the results from our experimental analysis, improve our understanding of the difficulty in implementing useful PGO passes in a compiler, and ultimately gain a more general perspective of this area of compiler development as it has evolved over the past two decades.

Table of Contents

1. Introduction

The idea of feeding runtime profile information back into the compilation step of a program in order to improve the overall performance is by no means a novel idea. In fact, profile-guided optimization (PGO), which is a special case of feedback-driven optimization (FDO), has been around for over two decades. The motivation for using such runtime information is quite natural: statically analyzing the behavior of a program based solely on its source places an intrinsic limit on the number of modifications and optimizations that can be made by conservative compilers. Therefore, by leveraging runtime profile information during program compilation, or recompilation, it is possible to apply more aggressive optimization passes after identifying new opportunities for improvement based on the program's runtime behavior.

PGO has a lengthy but sporadic history in the field of compiler development, having been applied to simple optimization passes such as function inlining, loop unrolling, and basic block placement as well as to more advanced passes such as partial dead code elimination, partial redundancy elimination, strength reduction, and load-store elimination from loops. In this report we seek to capture some important techniques and applications of PGO that have been developed and deployed in the past two decades. We emphasize that this survey is incomplete as it was our goal to become more acquainted with PGO techniques in the literature while also continuing the development of our own PGO techniques in the LLVM compiler infrastructure.

The remainder of this report is organized as follows. In Section 2 we survey published work that reveals PGO techniques for improving program performance after recompilation, which is then followed up by non-performance-related PGO techniques in Section 3. The literature survey then ends with a presentation of working PGO systems and implementation strategies in Section 4. All survey information is partitioned on a paper-by-paper basis for a clear separation between different works. Our overall thoughts and suggestions for future research ideas, which are ultimately inspired by each reading and experience implementing PGO techniques for LLVM, are offloaded to Section 5.

2. PGO Techniques for High Performance

In this section we detail several PGO techniques formally studied in the literature that are geared towards high performance applications. We present our survey of relevant works in chronological order to emphasize the trends that have been followed in the field of compiler development over the past two decades.

[1] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Profile-Guided Automatic Inline Expansion for C Programs. *Softw. Pract. Exper.* 22, 5 (May 1992), 349-369.

This paper describes some very early work experimenting with profile-guided optimizations for automatic function inlining, and as such is more of an engineering technical report than scholarly research. The majority of their discussion is dedicated to various design issues with integrating an automatic profile-guided optimization into a compiler, such as how to partition compiler tasks throughout the lifecycle of a program (from source code to binary to a recompiled binary) and program representation. The authors correctly identify several other issues of importance to consider with automatic inlining, perhaps the most critical of such are the extent to which candidate functions should be inlined so as to minimize activation stack explosion (e.g., with recursive functions that continually place and subsequently remove parameters from each stack frame upon entrance and exit) and code expansion (e.g., if a function call is inside the body of a loop to be unrolled). They also give thorough treatment to how the program source is actually changed to ensure correctness after inlining, which requires variable names to be uniquely named so as to avoid conflicts among multiple function invocations, among other issues.

They conclude with a discussion of their experimental results with the profile-guided automatic inliner in a prototype C compiler. As one would expect, all production-quality C programs tested with this compiler experienced an increase in the resulting executable (up to 32% more in one case - for the one xlisp program considered), with an average code expansion factor of approximately 16%. However, this size expansion was compensated by the fact that inlining showed speedup improvements for all programs considered (up to 46% improvement, again for the same xlisp program), where the average speedup was approximately 11%.

While not incredibly significant, the fact that this inlining is done only once after profile data is collected to achieve speedups beyond what traditional static compilers can accomplish supports the continued investigation of PGO techniques for compilers to this day. However, it is important to note that analogous speedups are very difficult to achieve with today's compilers, which are significantly more sophisticated than those used at the time of this work. As such, profile-guided inlining cannot achieve the same speedups with the same amount of effort in today's compilers.

[2] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path Profile Guided Partial Dead Code Elimination Using Predication. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques* (PACT '97). IEEE Computer Society, Washington, DC, USA, 102-.

Path profile-guided dead code elimination (PDE) is a technique that leverages the execution frequency of paths in a program's control-flow graph (CFG) representation in a cost-benefit analysis to determine whether or not moving dead code instructions from hot program paths (i.e., critical paths) to cold paths will improve locality and reduce the critical path length. This general code motion technique is referred to as sinking, since expressions are taken off hot paths and sunk to lower, less executed paths. Traditional code sinking algorithms statically attempt to sink instructions while maintaining the semantics of the program. However, there are some situations where such static analysis will skip a valuable sinking opportunity because dead code (e.g., an assignment that is later overwritten along a critical path) cannot be directly moved to a later point in the CFG without guaranteeing that it will not violate the program's correctness.

The key insight made by the authors is that predicating the execution of this dead code on cold paths after sinking would solve this problem. Accordingly, they propose a complete cost-benefit analysis for PDE that uses CFG path traversal frequency to determine points in the program to which instructions can be sunk and then guarded with a predicate (see Figure 1). Their cost-benefit analysis can handle acyclic and cyclic CFGs (with some modifications to how accept sink points are identified) to perform PDE, and also supports the ability to inject temporary predicates into the program CFG in the event that a branch-controlling predicate is not readily available. For example, if a program does not compute a predicate that is used to determine the direction of a branch to a hot or cold path, their predicate sinking framework will introduce a new "virtual" predicate variable at the root node dominating the branch, which is initially set to true and then set to false on all paths during which the instruction to be sunk should not be executed. This has the effect of only executing the sunken instruction on the cold path(s) at the cost of extra assignment instructions. Unfortunately, the authors do not provide experimental results for their procedure to compare against traditional PDE algorithms.

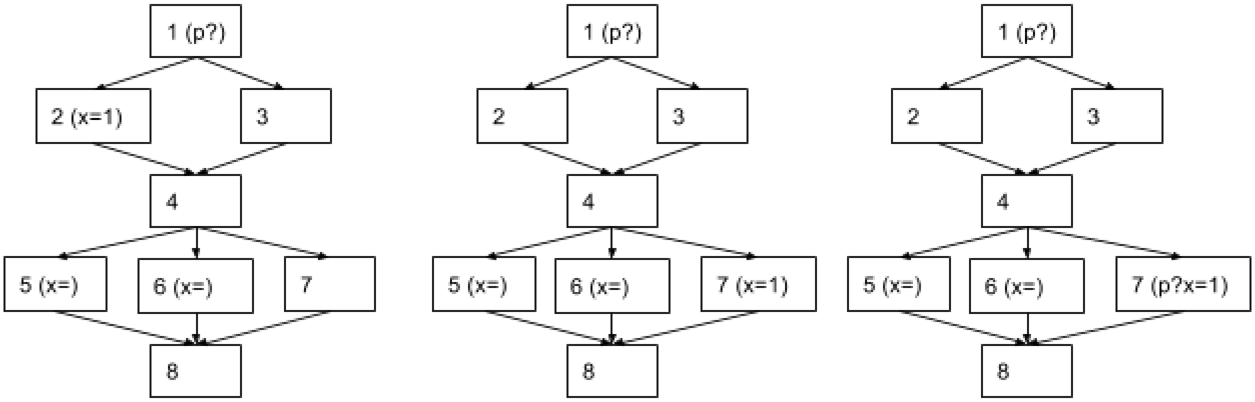

Figure 1. Visual depiction of predicate-based PDE using profile information. In this program, the paths 1-2-4-(5/6)-8 are executed more frequently than 1-3-4-7-8. The original version of the program is on the left, where the instruction in Node 2 is dead (since it is not used in Node 7 or 8 and is reassigned in nodes 5 and 6). Directly sinking this instruction off the critical path to Node 7 may not be acceptable since there is no guarantee that it will be needed later on in the program if the 1-3-4-7-8 path is taken. Thus, the predicate introduced in the version on the right ensures that the instruction is executed only if the predicate p is true (as in the original version) while also

getting the instruction off the less-frequent code paths.

[3] Thomas Kistler. Dynamic Runtime Optimization. Springer Berlin Heidelberg, 1997.

In this early work, Kistler argues in favor of a modularized program architecture that uses profile-guided optimizations at runtime to continually improve the performance of program modules based on their dynamic behavior. In the proposed system, profile data such as function call frequency and duration is collected from executables (represented in a fixed intermediate representation language) at runtime using an adaptive profiler, fed to a dynamic compiler that optimizes and recompiles the modules that would most benefit from optimizations, and replaces the old module executables with the new ones in an online fashion.

The natural advantage of this approach is that optimizations are applied only to those sections of code that are executed most frequently, which is in contrast to the static optimizations traditionally performed on an entire program. The dynamic profile-guided optimizations recommended for this system include intermodular inlining, which is function inlining across module boundaries (note that online optimization enables inlining to be performed across boundaries, which is not the case when modules are statically compiled separately), intermodular register allocation (i.e., giving CPU register priority to nodes across the CFG that spans all program modules based on their execution frequency), and intermodular code elimination and code motion. The predicted benefits of intermodular inlining are illustrated by examining the procedure size distribution of the Oberon operating system (see Figure 2). The large number of “small” procedures is a clear indication that inlining these functions could improve locality, and without online profile-guided information it would not be easy, if at all feasible, to determine which of these functions should be inlined and which should not.

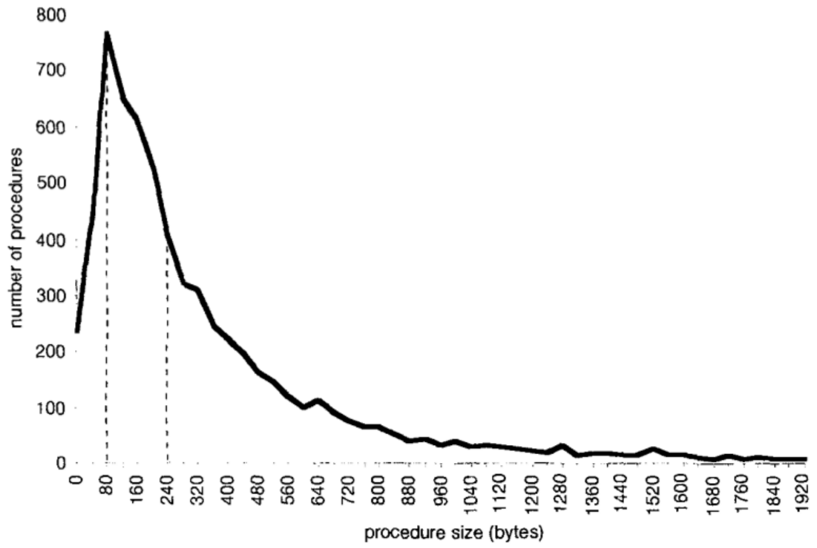

Figure 2. Distribution of the procedure size in the Oberon operating system. The large number

of small procedures is an indication that intermodule inlining may be of significant benefit to this system if deployed with the modular, dynamic recompilation engine. Image acquired from [3].

Novel improvements to both instruction and data cache optimizations are also discussed, with particular emphasis on instruction cache optimizations that are possible online since cache hits can be detected using the physical distance between instructions for different procedure calls that span multiple modules. For example, if one procedure invokes another procedure in a different module frequently, it may be worthwhile to reorganize the instruction sequence in either module to exploit this relationship for improved locality. A detailed performance analysis of this particular optimization is not included in the work.

[4] Ronald Barnes, Ronnie Chaiken, and David M. Gillies. Feedback-Directed Data Cache Optimizations for the x86. In *2nd ACM Workshop on Feedback-Directed Optimization (FDO)*. 1999.

This paper explores a very early attempt at integrating profile information relating to memory access patterns to guide recompilation code generation. In particular, the authors explore the concept of using memory access patterns to determine when and where to insert cache prefetch instructions (in the x86 ISA) to improve data locality and minimize cache misses. Such access patterns are characterized based on (1) the cache miss frequency of all address references, (2) the instruction sequence gaps between memory accesses, (3) cache-data reuse frequency analysis, and (4) evicted entries that are quickly brought back into cache after being written to main memory. This information is generated by instrumenting all memory references within an executable at the level of machine code. Naturally, this incurs additional overhead during the profiling stage, but such instrumentation is removed upon recompilation.

After analyzing these memory access and usage patterns, the compiler is able to make more informed decisions about how to leverage cache-controlling instructions (e.g., cache prefetch and streamed store instructions). Empirical evidence suggests that integrating such memory traces into the recompilation stage can yield percent improvements of approximately 6.8%, ranging from 3% to 27%. The lack of adoption of this technique is likely due to the significant overhead needed to capture such fine-grained memory access patterns. In automatic and dynamic compilation engines, such instrumentation overhead often negates and sometimes even degrades the overall program performance.

[5] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization* (DYNAMO '00). ACM, New York, NY, USA, 52-64.

This paper reports on the feasibility and utility of profile-guided function inlining over static function inlining in the context of the Java programming language. This work is particularly interesting due to the many obstacles inherent in the implementation and execution of such object-oriented programming languages, including dynamic method dispatch, reflection, and dynamic class loading. The authors of this work approached function inlining as a combinatorial optimization problem similar to the well-known NP-hard knapsack problem. More specifically, function (method) inlining was performed according to an inlining plan, which is analogous to a set of function call sites that are flagged for inlining or not depending on whether or not they can “fit” into an inlining “bucket” without exceeding some predefined threshold for code expansion. Framing the inlining this way enabled them to employ a greedy approximation algorithm to solve for the near-optimal inlining plan given a set of function call sites and other relevant information. Accordingly, the amount of information made available during the construction of such plans depends on whether or not profile-guided information is acquired, and furthermore, how such information is mapped onto the program representation. The authors considered three different program representations with three corresponding algorithm that are used by the greedy knapsack heuristic to determine whether or not a function should be inlined. These representations include a static CFG, dynamic CFG with node weights, and dynamic CFG with edge weights.

After implementing this inlining approach into a Java compiler for the Jalapeño VM (a Java VM that does not do any bytecode interpretation), the authors then conducted experiments on a set of five large-scale programs. The most important empirical result was that the dynamic CFG with edge weights provided the most information when used with the corresponding inlining heuristic and thus yielded the best speedups averaging from 11% to 57% across all programs. One of the suspected causes for this result is that more thorough and complete profile information naturally means that better inlining decisions can be made, thus reducing code explosion and reducing the difficulty of register allocation while improving cache locality. However, their results also indicated that the number of functions inlined does not directly correlate with the final speedups, supporting the community-wide view that function (method) inlining is a difficult task to do well by any compiler.

[6] Mongkol Ekpanyapong, Michael Healy, and Sung Kyu Lim. Profile-Driven Instruction Mapping for Dataflow Architectures. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 25, 12 (December 2006), 3017-3025.

This paper is unique in that it does not deal directly with software compilers. Rather, it discusses the use of profile information to guide the place-and-route algorithm for dataflow-oriented architectures used in very-large-scale integration (VLSI) designs. First, the authors show how to use profile information to determine the execution time for each path in the program’s data-flow graph (DFG). Specifically, given the DFG of an algorithm or application in assembly code that is to be synthesized and mapped to some available hardware resources, the total execution time for a single path in the DFG is estimated as a weighted sum of the delay induced by traversing each edge in the corresponding path. In this case, the frequency with which each edge in the path is traversed is the weight used in this summation. The total execution time for the DFG is then the maximum execution time across all paths (i.e., the time for the critical path).

Second, the authors present a DFG mapping algorithm that uses the DFG path frequency to (1) cluster DFG nodes based on execution frequency, (2) place and route clusters to architectural clusters (i.e., sets of ALUs, on-chip memories, etc), and (3) introduce routes between architectural clusters. Omitting many of the technical details, DFG clustering uses profile information to iteratively build clusters by adding one node N at a time based on the relative frequency with which N is traversed to or from the nodes in a particular cluster. That is, a node N is added to the cluster to which its cumulative connections have the highest execution frequency. Profile information is then also used in the architectural place and route step. The authors formulate the place and route procedure as an integer programming problem (ILP) that seeks to minimize the total cost (physical distance) resulting from placing DFG nodes onto clusters. The cost of placing two nodes onto clusters is defined as the resulting distance between them, and the distance between each pair of nodes is weighted by the statistical frequency between these two nodes. Finally, this same statistical frequency information is later used in another ILP formulation of the inter-cluster routing establishment phase in order to scale the cost of a particular route between two nodes.

Widespread usage of this profile information enabled their DFG mapping algorithm to outperform (in terms of execution time of the application after layout) the results from traditional wirelength-driven algorithms by 14% and 21% in the average case before and after routing was done, respectively. Thus, while leveraging this profile information did increase the time and complexity of the place-and-route procedure, the resulting application was superior to its static counterpart.

[7] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (ASPLOS XIII), 2008. ACM, New York, NY, USA, 277-286.

This paper proposes a new online technique for thread partitioning and scheduling using runtime profile information. A natural impediment to multithreaded code is the existence of data synchronization and sequential bodies of code. Further obstacles may even lie outside of the context of the program. For example, off-chip bandwidth (e.g., shared memory bandwidth, disk drive access bandwidth, etc.) may serve as points of contention for multiple threads, and such issues are difficult to detect at compile time. Due to these observations, when program threads are distributed across multiple cores it is necessary to identify and reduce the impact of these issues in order to improve performance while also reducing on-chip power consumption. Unfortunately, determining such impact at compile time is often times intractable, regardless of the implementation of the program.

Accordingly, feedback-driven threading (FDT), as is coined in this paper, is a framework for using profile information to perform thread partitioning and scheduling in order to optimize performance and power consumption. The FDT framework, which takes small samples of program behavior periodically at runtime, was used to implement both (data) synchronization-aware threading (SAT) and bandwidth-aware threading (BAT), and then a hybrid SAT+BAT policy. Their x86 simulation-based experimental results showed that SAT could achieve a performance speedup and power reduction of 66% and 78%, respectively, BAT could achieve a 17% reduction in power, and SAT+BAT could achieve an average performance improvement and power reduction of 17% and 59%, respectively. Even though the results were based on a simulation, the results show great promise with the growing presence of multicore architectures in low-power embedded devices such as smart phones.

[8] Arun Kejariwal, Alexandru Nicolau, Veidenbaum, Alexander V., Utpal Banerjee, Constantine Polychronopoulos. Efficient Scheduling of Nested Parallel Loops on Multi-Core Systems. In *Parallel Processing, 2009. ICPP '09. International Conference on* , vol., no., pp.74,83, 22-25 Sept. 2009.

In this paper the authors present a technique for mapping the iteration space of nested parallel for loops to different cores in multicore systems in a way that is aware of the computational cost and amount of cache thrashing during each iteration. Such information is not available during static compilation and optimization passes and therefore optimal nested loop scheduling cannot be guaranteed without cache thrashing and execution profile information collected at runtime. Generally speaking, the technique for mapping the iteration space of a nested parallel for loop to multiple cores is to couple the cache miss profile information and so-called density of an iteration space i (for each iterator space $i = 1, ..., N$ for N iterations of the outermost loop) to compute the *volume* of cache misses along the space. Then, this total volume is appropriately partitioned and mapped to the available P cores by determining partial volumes for each point $i = 1, ..., N$ along the space and then evenly distributing the iteration points across each processor P such that the partial volume “slices” for each processor are minimized. The net effect is that each processor is expected to encounter less cache misses as a result of executing its designated iteration space.

Their approach was tested on kernels extracted from the industry-standard CFP2000 and CFP2006 benchmarks, among others. The experimental results on seven distinct nested parallel for loops indicated that speedups ranging from 2.7% to 13.4% could be obtained. As such, despite the seemingly large computational cost of computing the cache miss profile and volume functions, which is performed offline, this profile-guided optimization technique to schedule loops is quite promising in the presence of high performance programs targeting multicore systems.

3. Non-Performance-Related PGO Techniques

In this section we survey PGO technique that have been formally studied in the literature that are not directly related to application performance. While such work may be outside the scope of this project, we felt that it was important to study PGO techniques for all applications to gain an understanding about the difference in utility and difficulty of static versus dynamic (i.e., feedback-driven or profile-based) program modifications.

[9] Saumya Debray and William Evans. Profile-Guided Code Compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (PLDI '02). ACM, New York, NY, USA, 95-105.

Motivated by the empirical 80-20 rule which states that approximately 80% of a program's execution time is spent running 20% of the code, the authors of this paper present a technique to use profile data collected at runtime to identify cold blocks of code to compress on memory-constrained devices. The intuition is that if compression thresholds (i.e., code path execution frequency limits that determine whether to compress or not) are tailored based on program behavior collected at runtime, then an optimal best balance between the executable's memory footprint and average running time can be achieved. At a high level, executables are compressed at the level of the functions, and the body of each function whose execution account exceeds this threshold will be replaced with a “lookup-stub” to enable compilation and linking to succeed. The purpose of the lookup-stub is to serve as an index into a table, deemed the "function-offset table”, containing compressed versions of the original functions. This offset is then used by the decompressor to identify the address of the function to invoke from the table, decompress the code and insert the corresponding instructions into a runtime buffer, and then jump to the buffer to execute the instructions of the original function.

The experimental results from this work were quite promising, showing that, on average, a compression of approximately 20% was achieved with negligible impact on the executable's running time for all programs considered. This empirical evidence indeed supports the 80-20 law, and as such can be a very useful optimization for code-compressing compilers.

[10] Shukang Zhou, Bruce R. Childers, Naveen Kumar. Profile Guided Management of Code Partitions for Embedded Systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings* , vol.2, no., pp.1396,1397 (Feb. 2004), 16-20.

In this short paper the authors study the benefits of using runtime profile information for a program running on an extremely resource constrained embedded device to drive a software-based partial reconfiguration strategy. It is assumed that such systems only store a portion of executable code in a “code partition buffer” on the device and continually swap out code with a remote code server, to which they are connected via a wireless link, when needed. In order to reduce the amount of thrashing in the code partition buffer and minimize the amount of wireless transfers, runtime profile information is used to determine which pieces of code are executed more frequently and should thus be given priority during the swapping phase. In essence, their proposed architecture uses profile information in a weighted LRU replacement policy for fetching new code from the code server to execute, where hot pieces of code are less likely to be replaced than cold pieces of code. As a preliminary research report, the authors only present the results of simple experiments estimating the frequency with which a small sample of smart-card applications from the telecom, security, and control domains execute instructions, indicating that on average about 27.5% of the instructions for each application are executed only once. Thus, these code portions should be immediately swapped out after executed since they consume unnecessary space on the device.

4. PGO-Enabled Compilers and Implementation Strategies

In this section we survey working systems and implementation strategies for integrating PGO techniques into modern compilers. The scope of this topic is broad, and we only seek to capture a subset of the work in the literature to gain some familiarity with progress that has been made in the field and also acquire some perspective about the difficulties associated with this type of compiler development. Also, the work presented below does not appear in chronological order.

[11] Georgios Tournavitis, Zheng Wang, Bjorn Franke, and Michael F.P. O'Boyle. Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping. In *SIGPLAN Not.* 44, 6 (June 2009), 177-187.

Given the difficulty of hand-crafting parallel code that is semantically equivalent to its sequential counterpart due to data dependencies between shared variables and across iteration spaces, program parallelization is often left to experienced programmers who are capable of identifying and exploiting such dependencies. However, the ubiquitous presence of multicore systems calls for compilers that can perform such transformations automatically. Unfortunately, static compiler optimizations are overly conservative in parallelizing code because such dependencies cannot always be inferred at compile time.

The authors of this work present an approach to solve this problem which relies on dynamic program information collected through profiling and an IR-level instrumented version of a program to gather data dependency and control flow information. The reliance on instrumentation at the level of the intermediate representation language is needed in order for data dependencies obtained in the profile information to be used by the compiler in the later optimization pass that introduces the appropriate parallelism before machine-level code generation. Indeed, the compiler's job of parallelizing machine-level code is much more difficult than parallelizing IR code prior to code generation. Using this data and control flow information, a control data flow graph (CDFG) is constructed to characterize for loops as parallel or sequential. Generally speaking, the data dependency information embedded in the CDFG based on the profile data is used to identify non-private variables within a loop, which indicate a sequential dependency, determine opportunities for applying parallel reduction operations to some scalar variable or shared collection, and minimize the number of barriers needed if a loop is deemed parallel. The latter technique is of critical importance if barrier setup and teardown is performed frequently to support thread synchronization, as this is an inherently sequential task that can negate the performance gains of parallelization. In cases where a loop cannot be characterized as a parallel candidate with absolute certainty, the system prompts the user to manually determine if the loop is parallel or sequential. This is done to ensure program correctness without being overly conservative.

To evaluate their technique of identifying parallel loop candidates, the authors examined the false positive and negative rate for a variety of applications. In the small subset of applications they considered, they encountered zero false positives (i.e., sequential loops incorrectly marked as parallel), and the number of false negatives was also quite small, ranging from 0 to 3 across all applications. In all programs considered, roughly half of the loops were covered, which is not as high as the coverage compared to the 90% coverage rate hand-crafted parallel programs. Still, given that the loop coverage was sufficiently high across all programs, these results are a good indication that profile-guided information can help streamline the preliminary parallelization of high-performance and time-sensitive code and decrease the overall time-to-market.

[12] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. *SIGPLAN Not.* 46, 4 (May 2011), 65-83.

In this paper the authors present and discuss the internal design of online recompilation support in the Jalapeño JVM. While the emphasis is on the design of the JVM, the most significant profile-guided element is their online sampler that dynamically recomputes program CFG edge weights that are used to determine which methods to inline during future recompilations. The recompilation engine will periodically inject newly optimized code into a currently running executable, where such optimizations are made based on perceived behavior of the program. Since the behavior of a program may change over time, statistically sampling and modifying the CFG enables the program to be recompiled in such a way that the resulting machine code accurately reflects and supports the application's expected behavior. This is a novel change from previous work in which profile-guided optimizations, like static compiler optimizations, were used in a "once and for all" fashion to generate an optimized executable. Their experimental evaluation on a set of ten distinct programs showed that using online profile data to perform optimizations and recompilation yielded noticeable speedups in all but one case when compared to the performance of each application at startup, i.e., before any profile-guided inlining was applied (see Figure 3 for a visual depiction of these results).

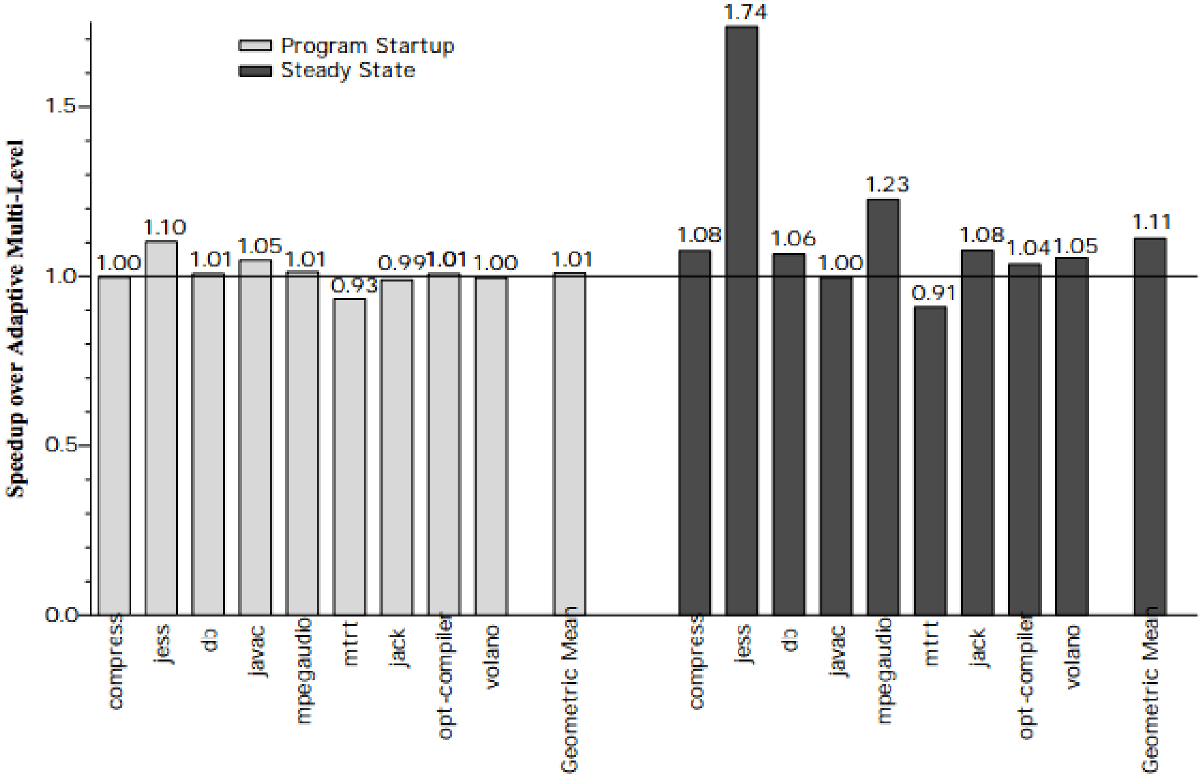

Figure 3. Visual depiction of the performance improvement of applications after several recompilations (i.e., in their steady state) over their preliminary startup performance after a single recompilation. The numbers above each bar indicate the relative speedups with respect to the unoptimized versions [12].

[13] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online Feedback-Directed Optimization of Java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (OOPSLA '02). ACM, New York, NY, USA, 111-129.

This paper reviews the implementation strategy and experimental results from an automatic, feedback-guided optimizing compiler for Java. We omit an overview of the implementation details as we are mostly concerned with the results. We note, however, that this system builds upon the traditional JIT (just-in-time) compilers found in Java VMs by adding a continual instrument-optimize feedback loop after JIT optimizations are applied (analogous to the work presented in [12] above). This general workflow is depicted below in Figure 4.

The profile data gathered via low-overhead edge profile instrumentation is used to apply loop unrolling, function and method inlining, loop splitting, and code reordering (i.e., basic block placement), and was demonstrated to show an average peak performance improvement of 6.2%. Furthermore, their system is capable of converging to a better performing steady state than is possible with traditional JIT optimizations since the continual instrument-optimize feedback loop that is automatically applied online throughout the program’s lifetime enables more frequent and targeted recompilations. For example, with respect to the *mtrt* benchmark program used in this work, their system was able to converge to a steady state performance of approximately 7s per run, whereas the traditional JIT optimizations converged to a steady state performance of 8.8s per run. Given the non-negligible performance improvements that can be achieved without performance degradation due to instrumentation, this technique should have been integrated into other high-level language VMs. Unfortunately, to the best of our knowledge, this is not the case.

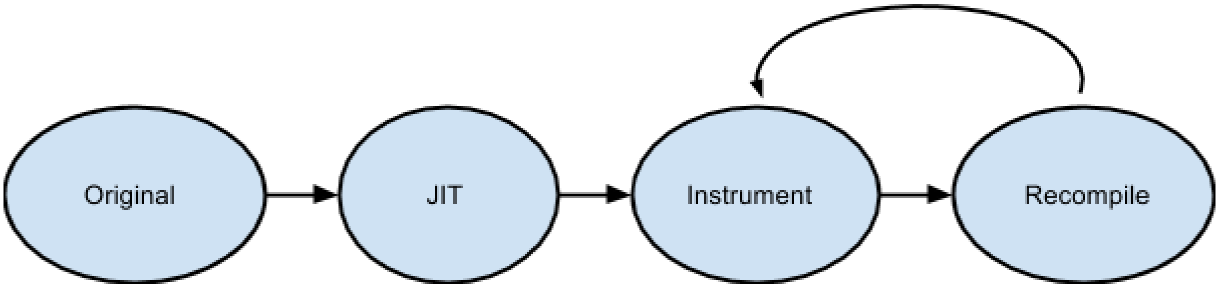

Figure 4. Feedback-driven recompilation in the JVM.

[14] K. Vaswani, and Y. N. Srikant. Dynamic Recompilation and Profile-Guided Optimisations for a .NET JIT Compiler. In *IEEE Proceedings of Software*, vol.150, no.5, pp.296,302, 27 Oct. 2003.

This paper presents a comprehensive overview and experimental analysis of the profile-guided optimizations implemented in the .NET Rotor JIT architecture, the design of which is shown below in Figure 5. The Rotor profilers consist of runtime stack-based samplers, which can be used to infer program hotspots since the call-stack of a program is an indication of where most of its time is spent, and more fine-grained profilers that collect basic block and CFG execution frequency information. These profilers are executed in two levels by the recompilation controller to converge towards optimally performing executables at runtime. Interestingly, the only profile-guided optimizations beyond identifying which blocks of code to optimize based on runtime stack usage are function and method inlining, loop unrolling, and basic block ordering, which are exactly the same optimizations explored in this project.

Their experimental results indicate that stack inspection only introduces at most 1.5% overhead to program execution at runtime, which is caused by the inherent context switching that is needed to pause execution, inspect the stack from a different internal VM process, and then resume execution. Level two profile-guided optimizations only showed significant improvements for programs that were run for a long period of time (e.g., up to 42.3% performance improvement), and conversely, programs with a short runtime did not show any significant benefit (e.g., on the order of only 0.5% performance improvement). This result appeals to intuition because the effects of runtime data used to guide the program optimization and native code generation processes in the .NET JIT are intensified as the program runtime is elongated. Another obvious implication of their result, which is especially true for dynamic recompilation engines, is the need for minimally intrusive profile data collection techniques, such as lightweight instrumentation techniques. Of course, significant work has been done on profile data collection and PGO techniques since this paper was published in 2003, but the results are still significant.

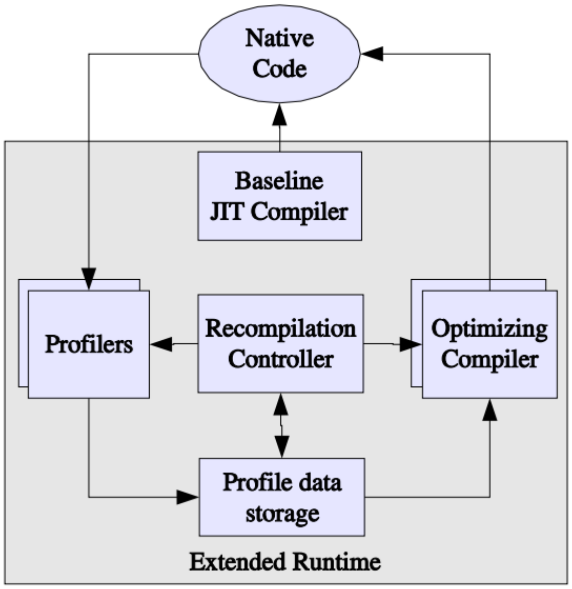

Figure 5. Overview of the extended .NET Rotor architecture that incorporates dynamic profile information into the runtime recompilation process [14].

[15] Michael D. Bond and Kathryn S. McKinley. Practical Path Profiling for Dynamic Optimizers. In *Proceedings of the international symposium on Code generation and optimization* (CGO '05). IEEE Computer Society, Washington, DC, USA, 205-216.

This paper focuses on improving program instrumentation techniques that are used to collect path profile information (i.e. CFG path execution frequencies) for dynamic, just-in-time compilers. Prior to this work, path profiling techniques developed by [19] and [20] have been shown to contribute anywhere from 31-97% and 15-53%, respectively, thus making them difficult to justify for use with high-performance applications. The new path profiling technique, dubbed practical path profiling (PPP), typically requires fewer paths in the program CFG to be instrumented at runtime, thus contributing to less instrumentation overhead, and also reduces the number of additional instructions needed for such instrumentation.

These improvements are due in large part to a flow metric used to measure the amount of execution of a path. In contrast to previous path profiling techniques, the flow of a path in PPP is computed as the product of the execution count of a path and the number of branches in the path. Using the branch number as a weight effectively means that the length of each path, which may vary from among branch paths, influences the total flow. Then, to reduce the number of paths that are instrumented, the PPP technique computes the coverage of a path, defined as the ratio of the minimum guaranteed flow and actual measured flow, and omits instrumentation if this ratio is above a certain threshold. The rationale for this modification is that the minimum guaranteed flow, which can be cheaply computed edge weights in the CFG, is a good enough predictor of the flow and can therefore be used to determine if instrumentation should be added without degrading the application's performance.

There are a variety of other new heuristics incorporated into the PPP technique, such as path candidate removal based on local bias (i.e., the relatively frequency compared to local edges in the CFG) and global execution frequency, as well as selective edge instrumentation based on execution count (specifically, frequently traversed CFG edges are less likely to be instrumented). Edge inlining and unrolling, based on profile information, are also performed prior to reinstrumentation and recompilation in order to decrease the total number of paths, increase path lengths, and exploit the new flow metric that is used to determine which paths are ultimately instrumented.

Collectively, these path profiling improvements perform slightly better than the prior profiling techniques by approximately 5% in terms of added program runtime due to instrumentation, and is capable of predicting hot paths correctly an average 96% of the time and never less than 90% (in the test cases considered). While these may not seem like much, when considered in the context of very large programs such as *gcc*, these prediction improvements and overhead decreases can be quite noticeable.

[16] Robert Cohn and Geoffrey Lowney. Design and Analysis of Profile-Based Optimization in Compaq's Compilation Tools for Alpha. *Journal of Instruction-Level Parallelism* 3, 2000, 1-25.

This is a technical report detailing various profile-guided optimizations implemented in Compaq's compiler toolchain for the Alpha architecture that were capable of showing great speedups when applied to very small and specialized portions of a program. The PGO techniques employed by this compiler include: (1) a function inliner, (2) "commando" loop optimizer which pulls less-frequently executed loops (inside nested loops) out to be subject to static optimizations, (3) profile-enhanced register allocation pass that tries to enforce less-frequently functions to use as few registers as possible to reduce main memory accesses and high-frequency functions to use as many registers as necessary, (4) tracer (i.e., a component that converts complicated spaghetti-like paths in a program CFG to larger basic blocks using block join and loop peeling techniques [15]), and (5) a basic block placement pass. Their experimental evaluation with these PGO techniques showed minimal speedup over baseline programs, ranging from approximately 2% to 20% with several positive (e.g., 47% and 59%) and negative (e.g., -1%) outliers. As expected, the most significant speedup improvements were attributed to the PGO function inlining technique. The authors also compared the resulting code size and found that in cases where there was non-negligible code growth, it was usually positive in the sense that code size reduced while simultaneously achieving performance improvements.

[17] Weifeng Zhang, Brad Calder, and Dean Tullsen. An Event-Driven Multithreaded Dynamic Optimization Framework. In *Parallel Architectures and Compilation Techniques, 2005. PACT*

2005. 14th International Conference on , vol., no., pp.87-98 (Sept. 2005), 17-21.

This paper presents a framework that facilitates online profile-guided recompilation in multicore systems. The underlying design for this framework is to utilize hardware-based instrumentation mechanisms to dynamically identify program hot spots and then invoke recompilation events on separate control threads running on under-utilized CPU cores. The hardware-assisted techniques to guide recompilation events include a hot path profiler to identify frequently executed branches in the program CFG, hot value profiler to identify values frequently loaded from main memory, (instruction) cache access counters to determine which instructions are executed most frequently, and monitoring of a rather new data structure coined the “optimized trace watch table.” This data structure keeps a record of all program traces and their expected completion time, in terms of CPU cycles estimated as instruction counts, and is periodically updated based on the program’s behavior. The table is also asynchronously monitored to determine if a particular trace has deviated too far from its expected completion time, which then triggers an event in the Trident system. Such events are classified as hot path, hot value, or code cache invalidation events, and the response to each of these events is to spawn a helper thread to examine the contextual information surrounding each event (e.g., for a hot path event, a helper thread will construct a “hot trace” through the program) and determine if recompilation is needed. Since the hardware-assisted event generation techniques and helper threads run concurrently with the program at execution, both of which with seemingly negligible overhead based on empirical evidence, this framework may be well suited to support online PGO passes in systems with multicore CPUs. We omit the experimental results from this work as it is focused on software-based value specialization, an optimization technique outside the scope of this project.

[18] Paul Berube, Adam Preuss, and Jose Nelson Amaral. Combined Profiling: Practical Collection of Feedback Information for Code Optimization. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering* (ICPE '11). ACM, New York, NY, USA, 493-498.

This in-progress research report motivates the need for profile-guided compilers to support the usage of profile data collected from multiple sources. In the standard PGO workflow, a program is run under a specific use case to collect a set of profile training data that is then used to recompile the program for that particular use case. In cases where use cases change over time, this profile and recompilation procedure can be automatically repeated so that the the executable is tailored to the most recent set of program characteristics. Unfortunately, neither solution is adequate when such use cases change with high frequency. In this scenario, it would be ideal if the program was compiled for the common case to avoid such dramatic changes in the executable throughout its lifetime. This can be achieved by combining profile data from multiple samples collected under different use cases to construct distributions of program characteristics, or behaviors, rather than scalar values. For example, it might be more beneficial to assess the probability distribution associated with the execution count of edges in a program CFG rather than scalar values. Doing so enables outliers in program behavior, which arise when its usage changes dramatically over time, to be smoothed into the distribution, thus minimizing the degree to which they impact the resulting executable after every recompilation event.

Driven by this intuition, the authors propose a technique for representing program behavior distributions using empirical and normalized histogram models. They made a conscious decision to use histogram models instead of parametric probability models because the former is both efficient and does not make any assumption about the shape of the distribution being modelled, which is necessary for conservative compilers. Finally, since this was merely a work-in-progress report, there is not any significant experimental evaluation. The authors do note, however, that they are developing support for combined profile data in LLVM.

5. Challenges and Opportunities for Future Work

Even with our brief coverage of PGO techniques in the field, there are a large set of common problems that are encountered and discussed at length in many independent publications. Perhaps the more significant of such problems is the difficulty of implementing low-overhead instrumentation techniques for software executables. PGO techniques require some means of acquiring this program runtime information about a target program, and generally speaking there are only two ways to acquire such information: *internal* code instrumentation and *external* program sampling. The former technique tends to be the favored option in both scholarly work and industry-standard compilers because code instrumentation enables more fine-grained and accurate metadata about the program to be collected at runtime. Unfortunately, such instrumentation also necessarily introduces overhead into the original program executable, and extreme care must be taken so that such overhead does not significantly alter the program's perceived performance. Accordingly, low-overhead instrumentation techniques are needed.

Based on the results in the literature above, potential ways to address this challenge are to first identify the minimal set of profile data that is needed to adequately perform PGO techniques. Given that the majority of PGO techniques rely on the weighted program CFG representation of the program, such data should be sufficient for most useful PGO passes. Therefore, low-overhead code to capture this data should be injected into a program executable. We do not have sufficient compiler development experience to recommend optimal techniques for performing this instrumentation. However, a first approach might be to associate each CFG node with a globally unique identifier. The instrumentation code would then allocate an appropriately sized array to maintain execution frequencies for each CFG node, and short instruction sequences would be injected at the beginning of every CFG node to increment the appropriate index in this array. If function-level execution frequencies are needed, an analogous data structure could be maintained for each (non-inlined) function that is modified at the beginning of every function body. One foreseeable problem with this technique is that write instructions to update the execution frequency arrays might induce otherwise absent cache misses, thus degrading the overall performance.

If the target program is inherently sequential and single-threaded, it may be possible to leverage independent and unused cores to maintain these execution frequency tables. In particular, rather than injecting instruction sequences that write directly to the table from the same thread, these instructions could signal a software interrupt to another update thread that would asynchronously update the table instead. If low-overhead inter-thread communication mechanisms are supported by the target operating system, then this technique may yield significant improvements over the former approach described above.

In the event that code instrumentation simply induces too much overhead to warrant its application for gathering profile information, the second approach of externally monitoring the program at runtime from a separate thread can be employed, as is done in [17]. Unfortunately, while effective, their hardware-assisted period sampling techniques are limited to a single platform. This limitation brings the fundamental gap between hardware platforms and software domains to light as something that must be addressed in the future. In particular, if their framework was generalized to arbitrary platforms then such hardware-assisted techniques could be leveraged in many more mainstream compilers or in cross-platform VMs for high-level languages like Java and C#.

This issue sheds light on another avenue for potential work in the field. In particular, there needs to be a shift in research focus from traditional imperative (object-oriented or otherwise) languages such as C and C++ to modern imperative and functional languages such as Java, C#, Scala, and Haskell. Based on the empirical evidence presented in the surveyed works, PGO passes for scientific computations and legacy programs can no longer compete with the performance improvements that come from increased spatial and temporal parallelism. Such programs are typically compiled once with a specified set of optimization goals and then run many times with similar data sets. In contrast, modern enterprise-scale applications written in new languages such as Java and Scala run in significantly more complicated environments and have use cases that are orders of magnitude more complicated. Spatial and temporal parallelism are significantly more difficult to correctly extract due to the wide variety of use cases. For this reason, applying or integrating PGO passes into the compilers or virtual machines of these languages may be yield noticeable performance gains.

In fact, as the work in [12] and [14] shows, this is exactly the direction towards which PGO-based compiler development is moving. However, if such techniques are to be applied to larger, enterprise-scale applications, the work in [12] and [14] should be augmented to make use of the combined profile characterization methods of [18]. Application stress, load, and verification tests can be used to collect profile information that is then normalized and merged together to form profile behavior distributions to guided VM optimizations and JIT recompilation. It is interesting to note that web applications may be particularly well-suited to this type of online profiling. In particular, it would be possible to augment the server-side business layer and backend services to capture information about common use cases among all clients that are served (in an anonymous way) while also leveraging the low-overhead instrumentation techniques previously discussed to capture fine-grained performance information about the server's execution. Such information could be leveraged by the server to modify load-balancing strategies and database prefetching and caching decisions, among other key performance factors. While this is not a strict example of PGO, since profile information isn't used to actually recompile the server application, it can certainly be characterized as a form of profile or feedback-driven execution that targets high performance.

The last challenge that we feel needs to be addressed is the gap between scholarly work on PGO techniques and those that are present in mainstream compilers. Even in the small subset of work considered in this survey, there were many promising ideas presented whose working implementations only serve to benefit the paper. Compiler developers, especially those involved with GCC and LLVM, should carefully monitor such academic publications and make an honest effort to integrate prospective techniques into their respective compiler. Although, bridging the gap between academia and industry is a challenge faced by every field of computer science, it is especially important for compiler development given the potential utility and benefit that could be made to software developers worldwide, and as a result, everyone who uses the software they build.

References

[1] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Profile-Guided Automatic Inline Expansion for C Programs. *Softw. Pract. Exper.* 22, 5 (May 1992), 349-369.

[2] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path Profile Guided Partial Dead Code Elimination Using Predication. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques* (PACT '97). IEEE Computer Society, Washington, DC, USA, 102-.

[3] Thomas Kistler. Dynamic Runtime Optimization. Springer Berlin Heidelberg, 1997.

[4] Ronald Barnes, Ronnie Chaiken, and David M. Gillies. Feedback-Directed Data Cache Optimizations for the x86. In *2nd ACM Workshop on Feedback-Directed Optimization (FDO)*. 1999.

[5] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization* (DYNAMO '00). ACM, New York, NY, USA, 52-64.

[6] Mongkol Ekpanyapong, Michael Healy, and Sung Kyu Lim. Profile-Driven Instruction Mapping for Dataflow Architectures. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 25, 12 (December 2006), 3017-3025.

[7] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs. In

Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII), 2008. ACM, New York, NY, USA, 277-286.

[8] Arun Kejariwal, Alexandru Nicolau, Veidenbaum, Alexander V., Utpal Banerjee, Constantine Polychronopoulos. Efficient Scheduling of Nested Parallel Loops on Multi-Core Systems. In *Parallel Processing, 2009. ICPP '09. International Conference on* , vol., no., pp.74,83, 22-25 Sept. 2009.

[9] Saumya Debray and William Evans. Profile-Guided Code Compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (PLDI '02). ACM, New York, NY, USA, 95-105.

[10] Shukang Zhou, Bruce R. Childers, Naveen Kumar. Profile Guided Management of Code Partitions for Embedded Systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings* , vol.2, no., pp.1396,1397 (Feb. 2004), 16-20.

[11] Georgios Tournavitis, Zheng Wang, Bjorn Franke, and Michael F.P. O'Boyle. Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping. In *SIGPLAN Not.* 44, 6 (June 2009), 177-187.

[12] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. *SIGPLAN Not.* 46, 4 (May 2011), 65-83.

[13] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online Feedback-Directed Optimization of Java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (OOPSLA '02). ACM, New York, NY, USA, 111-129.

[14] K. Vaswani, and Y. N. Srikant. Dynamic Recompilation and Profile-Guided Optimisations for a .NET JIT Compiler. In *IEEE Proceedings of Software*, vol.150, no.5, pp.296,302, 27 Oct. 2003.

[15] Michael D. Bond and Kathryn S. McKinley. Practical Path Profiling for Dynamic Optimizers. In *Proceedings of the international symposium on Code generation and optimization* (CGO '05). IEEE Computer Society, Washington, DC, USA, 205-216.

[16] Robert Cohn and Geoffrey Lowney. Design and Analysis of Profile-Based Optimization in Compaq’s Compilation Tools for Alpha. In *Journal of Instruction-Level Parallelism* 3, 2000, 1-25.

[17] Weifeng Zhang, Brad Calder, and Dean Tullsen. An Event-Driven Multithreaded Dynamic Optimization Framework. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on* , vol., no., pp.87-98 (Sept. 2005), 17-21.

[18] Paul Berube, Adam Preuss, and Jose Nelson Amaral. Combined Profiling: Practical Collection of Feedback Information for Code Optimization. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering* (ICPE '11). ACM, New York, NY, USA, 493-498.

[19] Thomas Ball and James. R. Larus. Efficient Path Profiling. In *International Symposium on Microarchitecture*, pp 46–57, 1996.

[20] Rahul Joshi, Michael D. Bond, and Craig Zilles. Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems. In *International Symposium on Code Generation and Optimization*, pp 239–250, 2004.