Overcoming the Challenges to Feedback-Directed Optimization

Michael D. Smith
smith@eecs.harvard.edu
Division of Engineering and Applied Sciences
Harvard University

Abstract

Feedback-directed optimization (FDO) is a general term used to describe any technique that alters a program's execution based on tendencies observed in its present or past runs. This paper reviews the current state of affairs in FDO and discusses the challenges inhibiting further acceptance of these techniques. It also argues that current trends in hardware and software technology have resulted in an execution environment where immutable executables and traditional static optimizations are no longer sufficient. It explains how we can improve the effectiveness of our optimizers by increasing our understanding of program behavior, and it provides examples of temporal behavior that we can (or could in the future) exploit during optimization.

1 Introduction

We often optimize applications to eliminate unnecessary generality and to streamline their execution on modern computing platforms. *Feedback-directed optimization* (FDO) is a general term used to describe any technique that alters a program based on information gathered at run time. This paper focuses on FDO techniques for improving performance. It presents a view of the field today and a vision for its future. It points out that current trends in application development, software engineering, hardware technology, and the Internet are increasing the need for FDO. And it argues that, to meet this need adequately, we should alter some commonly-held perceptions about executables and the execution engine.

Traditionally, FDO has been viewed as an *off-line* technique: The programmer runs an application one or more times to gather statistics that summarize the program's prevalent behavior and describe its execution environment. These statistics are then used to create a new program binary. Off-line refers to the fact that the optimization takes place *after* (as opposed to *during*) program execution. This type of FDO has a long history of use by programmers interested in tuning their code to increase performance. In addition, researchers in programming languages and compilation have proposed and built a wide range of systems that are able to perform FDO with little or no programmer intervention.

Though the application of FDO is most often associated with off-line techniques, perhaps the greatest commercial success of FDO has come from the area of computer architecture. In the design of modern microprocessors, computer architects dedicate a large amount of their silicon budget to both capturing and exploiting program tendencies at run time. Caches and out-of-order execution are just two examples of hardware techniques that use feedback to affect a program's execution. Caches reduce the cost of some memory accesses at the expense of others based on observations of previous program activity. Out-of-order execution dynamically adjusts the order of the instructions in the instruction stream based on observations of instruction latency. Caches and out-of-order execution are *on-line* versions of FDO. As I will discuss, there is a growing interest in software-based on-line techniques for FDO.

My claim that programmers, compilers, and hardware routinely perform many kinds of FDO shows that I have a very broad view of the definition of FDO. Specifically, I view any technique that alters the realization of a program based on tendencies observed in the present run or in past runs as a FDO technique. Furthermore, these techniques may be implemented in software, hardware, or some combination of the two. Tasks that we have traditionally viewed as compiler tasks, such as instruction scheduling and register allocation, are now routinely done in hardware. Many of the recently-announced on-line techniques for FDO rely on a combination of hardware and software mechanisms.

In addition to my broad view of what FDO is, I also have a broad view of when we should be able to apply FDO. My view is a result of the following answers to two simple questions: When is the first time that I would like my program to run quickly? When will I no longer need my program to run quickly? The obvious answers are, respectively, immediately after it is compiled for the first time and once I have run it for the last time. If we define the *lifetime of a program* as the time between its first compilation and its last execution (inclusive), then I am saying that we should be able to perform FDO at any time during a program's lifetime.

Thinking about optimization as *recurring when needed* is convenient for several reasons:

- First, this view directly addresses the well-known issue that we cannot build a fully optimizing compiler, one capable of transforming any program into an equivalent program with identical input/output behavior and a provably-minimal execution time [7]. If optimization can occur at any point in a program's lifetime, we can imagine scenarios in which it is possible to apply new optimization techniques to existing binaries even after they have been shipped.
- Second, this view gives us the freedom to perform an optimization when the data for that optimization is available (Adve et al. [1] refer to this idea of an *optimization continuum*) and to vary the persistence of the optimizations that we perform. Optimizations that are produce undesirable side effects, such as binary bloat and machine dependency, would be moved toward the shorter end of the persistency scale. Today, software vendors hesitate to perform traditional profile-driven and machine-specific optimizations because these optimizations will persist for the entire lifetime of a program. This persistence implies that the usage patterns and machine environment identified at compile time remain unchanged over the program's lifetime. For many commercial applications, this is a bad assumption. By no longer requiring the effect of an optimization to persist indefinitely, we can allow executables adapt to changes in their usage and environment.
- Third, this view helps us to regain the original promise of software—that it is flexible and easy to change. Much of the rapid acceptance of run-time binding techniques has come from the software industry's realization that one can use these techniques to simplify the task of patching shipped binaries and integrating third-party software extensions.

The rest of this paper is organized as follows: Section 2 describes how recent advances in hardware and software technology are increasing the need for FDO (in general) and on-line optimization (in particular). Having established the importance of the area, Section 3 proceeds with a brief review of the current state of affairs in FDO, and it presents a categorization of existing approaches. From this review, we extract two definite trends: a movement toward units of optimization based on run-time program behavior; and a movement toward executables as mutable objects. Sections 4 and 5 discuss each of these trends in turn. Section 6 enumerates several challenges to the vision presented in this paper, and it highlights on-going research projects that may address these challenges. Finally, Section 7 concludes.

2 The Increasing Need for FDO

This section describes the promise of FDO and motivates the need for the broader approach to program optimization discussed above. Section 2.1 begins with a brief discussion of profile-guided compilation (PGC), its growing commercial acceptance, and its future potential. Though PGC is a powerful technique for FDO, it has many shortcomings. Section 2.2 describes the limitations of the traditional static model of optimization, and it lists several trends in hardware and software technology that make this traditional static model an ineffective and incomplete solution. Dynamic FDO systems have been built that address one or more of these trends, and in Section 3, we look at a representative subset of those efforts.

2.1 Profile-guided compilation

As Brian Kernighan and Rob Pike state in the preface of their book entitled *The Practice of Programming*, simplicity, clarity, and generality form the three basic principles of good software [31]. PGC addresses the performance impact of generality, the third of these principles. By generality, Kernighan and Pike mean that a program should work well in a wide array of situations [31]. However, a program written to function in many situations is typically slower than one written to handle one or a few specific situations. In PGC, the compiler attempts to mitigate the cost of a program's generality by using information, such as a summary of how often each basic block in the compiled program executed in one or more previous program runs (i.e. block profiles), to focus its optimization efforts on the frequently executed portions of the program and to understand the run-time tendencies within these portions.

Though PGC has been around for many years, it has only recently begun to be commercially accepted and widely used. Today, almost every popular production compiler, with the notable exception of the GNU C compiler, has a mode in which it performs PGC. PGC exists commercially not because it is easy to support or use (as discussed below), but because it achieves noticeable performance improvements. The rest of this section will focus on the effectiveness of PGC in the Compaq GEM compiler, as reported by Cohn and Lowney [14]; this work is representative of the results reported for other commercial compilers [8,42].

Cohn and Lowney report that the SPECint95 benchmarks run 17% faster (on average) when compiled with FDO than when compiled with GEM's most aggressive level of classical optimization [14]. This sizable speed-up is quite impressive given the maturity of the baseline against which it is compared. Even so, even more impressive results are possible given the trends in application development and the recent research results in PGC.

Though the benchmarks in SPECint95 are real applications, several researchers have noted the qualitative differences between these applications and the graphical and interactive desktop applications used by a large segment of the population. Lee et al. [33] suggest that the dominant execution paths in these desktop programs may be less predictable than the paths in SPEC benchmarks. They note that the desktop benchmarks have a larger number of features and that more of their execution is determined by interactions with a user. Wang, Pierce, and McFarling [51] state that small changes in the position of the mouse or windows on a screen can cause large changes in the execution paths of today's popular interactive applications. Wang and Rubin [50] go farther and show how differences in the usage of these desktop applications impact profile-based program translation. In addition, Cohn and Lowney [14] mention briefly that in their experience the benefit from PGC often grows as the size and complexity of the application grows.

The GEM compiler, like most other commercial compilers, uses *point* profiles to direct its optimization efforts. Block and edge profiles are examples of point profiles—profiles that provide an aggregate execution count for individual program points. In research, there has been a resurgence in the area of PGC because of the development of more detailed profiles. A path profile, which records execution counts for sequences of program blocks (i.e. program paths), is an example of one of these more detailed profiles [11,54]. Path profiles provide a greater level of insight into a program's run-time tendencies. Several researchers [22,25,26,53] have shown that the use of more detailed profiles during optimization can yield greater improvements in program performance.

Overall, work in the field of PGC demonstrates that the potential benefits of program specialization will increase as

- applications become more complex, include more features, and depend more on user input, and
- compilers are configured to use more detailed kinds of profile information.

The question is: Can we regularly achieve the full potential of this approach? Unfortunately, I believe that the answer is no. Even if we ignore the demands involved in profiling an application, which are well documented elsewhere (e.g., see Conte et al. [17]), it is still quite difficult to achieve sizable and consistent benefits using only PGC.

2.2 Ineffectiveness of traditional model

PGC uses a fairly static model of program specialization: In order to optimize the program for a new set of program tendencies, the program must be re-compiled. As such, PGC works well only when the actual run-time program tendencies match those used during specialization. We can make a similar statement about machine-specific optimization: It works well only when the characteristics of the actual execution engine match those assumed during optimization. The obvious answer to these problems is to perform FDO at a point closer in time to the program execution. In this way, the optimizer can use accurate and timely information concerning the execution environment and program usage. This has led researchers to view optimization as a continuum [1] and to explore a wide range of more dynamic techniques for FDO. (I use the generic term *dynamic optimization* when referring to the range of techniques beyond PGC.) In a moment, I will reflect on this range, but first I focus on the reasons for the recent explosion[1] in the level of interest and activity in the area of dynamic optimization.

Much of the recent interest in dynamic optimization stems from a widely-held belief that these techniques can address the performance needs of the recent trends in computer architecture, software engineering, and the Internet. Stated another way, it is the ineffec-

1. There have been three successful workshops on binary translation, feedback directed optimization, and dynamic optimization all during the first five months of the 1999-2000 academic year.

tiveness of the traditional static model of optimization in handling these trends that has fueled the recent explosion in dynamic techniques.

To better understand the impact of the recent trends on the traditional static model, let us quickly review the history behind compile-time optimization. Assemblers and then compilers were originally developed to raise the level of programming abstraction. By removing the need for the programmer to deal with the intricacies of the target machine, the programmer was freed (in theory) to focus on algorithmic issues and to produce code that was easy to understand, debug, and maintain. Compile-time optimization was developed to eliminate the performance penalties of this abstraction. As Aho et al. [2] state, "if a compiler can be relied upon to generate efficient code, then the user can concentrate on writing clear code." In fact, much of the early success of FORTRAN is attributed to the fact that the IBM FORTRAN compiler was able to produce optimized code that rivaled the performance of hand-coded assembly [9].

Trends in computer architecture. As mentioned in the introduction, hardware techniques for FDO began to appear as soon as computer architects noticed the growing gap between processor and memory speeds. The problem was that the traditional static model of optimization froze in the executable not only what was to be done at run time but also what was thought to be the best way to do it. As the pace of hardware evolution and the importance of support for legacy binaries have grown over time, the burden of re-optimizing a binary for the current usage and machine environment has fallen on the hardware. Re-compilation to achieve re-optimization is viewed as commercially impractical. As Smith and Sohi [44] state in their survey paper on the design of modern superscalar microprocessors, hardware should simply view executables as a specification of what has to be done and not how it should be done.

Unfortunately, simply augmenting the traditional static model of optimization with run-time optimization in hardware is not a panacea. In fact, it does not even adequately address all of the optimization issues introduced by computer architects. For example, architects extend existing ISAs with new instructions (e.g., prefetch or multimedia instructions) because they want the compiler to generate and the executable to use these new instructions directly. There would be no need for these instructions to appear in the ISA if the hardware could dynamically transform existing executables and achieve the same level of performance. Furthermore, next-generation architectures like the IA-64 [18] have been designed with the expectation that the compiler will apply sophisticated profile-guided and interprocedural optimizations—many of which are still being developed. Without these optimizations, much of the potential performance benefits of these new architectures will be lost.

Trends in software engineering. One of the claims made by proponents of object-oriented programming is that this paradigm leads to code that is easy to understand, reuse, and maintain. However, object-oriented mechanisms like dynamic dispatch and programming styles like code factoring yield programs that are difficult to optimize using traditional static approaches. There exists a large body of work addressing the performance penalty of object-oriented languages and programming styles; specific solutions and approaches can be found in the proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) and the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). With respect to the point of this paper, we note a rising use of object-oriented techniques in the commercial world and a rising interest in on-line FDO techniques for overcoming the aforementioned performance penalties. In addition, we note the fact that applications written in these languages have smaller programmer-specified code units than applications written in imperative languages like C and FORTRAN.

The commercial acceptance of run-time binding techniques also severely limits the effectiveness of traditional static optimization. A growing number of software manufacturers choose to ship their applications as collections of dynamically linked libraries (DLLs). DLLs are easier to create and update than statically linked libraries [34]. They simplify for a software vendor the task of patching shipped binaries and integrating third-party extensions. However, static optimization across module boundaries becomes impossible if the only time that the entire executable exists is at run time.

Trends in the Internet. The tremendous growth and interest in the Internet has brought with it a call for the development of *mobile code*. The idea behind mobile code is that applications written using this paradigm can be distributed across computer networks and automatically run upon arrival at the network end point. Implicit in this idea is the expectation that these applications would be able to run across a wide range of hardware platforms and computing environments. Traditional PGC assumes that the target hardware platform and computing environment are relatively stable. With projects like HotSpot [47] and Jalapeño [12], computer companies have already come to the realization that, if they want mobile code written in Java to not only run but run efficiently, they must adapt traditional profile-guided optimization techniques to the run-time environment.

3 FDO Today

As I mentioned in the introduction, FDO is a well-accepted technique used by almost every hardware manufacturer to improve the performance of the applications running on their processors. Though this is the most successful use of FDO, many other approaches exist. Table 1 presents a categorization of these existing approaches based primarily on how dynamic each is (i.e., how quickly each reacts to changes in program tendencies or the execution environment). In general, PGC is least dynamic of all of the approaches, while run-time optimization in hardware is the most. For each category, I provide several examples.

I will also use this categorization to make general comments about the transparency, scope, and run-time overhead of each approach. *Transparency* refers to the amount of programmer/user effort involved in performing FDO. I consider a system that is able to perform FDO automatically (i.e., without any programmer assistance and without the knowledge of the user) to be *fully* transparent. *Scope* refers to the size of the code segment analyzed during optimization; I refer to this segment as the *unit of optimization*. Typically, the larger the scope is the more aggressive, and thus effective, the optimization. Section 4 discusses this issue in much greater detail.

I discussed PGC in Section 2.1, and there I mentioned that PGC is often characterized by large optimization scopes and zero run-time overheads (all of the analysis and transformations for optimization are done at compile time). However, as I noted in Section 2.2, it is becoming increasingly difficult to maintain a large optimization scope due to recent trends in software engineering. Furthermore, this approach suffers compared to the others due to its lack of transparency and inability to react quickly (or at all) to changes in program behavior or the execution environment. Approaches for off-line optimization systems based on continuous profiling were developed to address these problems.

Systems like Morph [55] had the goal of making profile collection and executable re-optimization automatic. This goal was accomplished by running a profile collection agent and a re-optimization agent directly on the end user's machine. The profile collection agent would continuously gather and maintain a database of

Category	Examples	
Profile-guided compilation	GEM [14], IMPACT [29], Machine SUIF [45]	*less dynamic*
Off-line optimization using continuous profiling	DCPI [5], FX!32 [28], Morph [55]	
Run-time code generation	'C [20], DyC [23], Tempo [15]	↕
On-line optimization in software	Crusoe [32], DAISY [19], Dynamo [10], Dynamic Rescheduling [16], Jalapeño [12], HotSpot [47]	
On-line optimization in hardware	Trace caches [40], dynamic trace optimizations [21]	*more dynamic*

Table 1: Categorization of existing approaches for FDO. The lists of example systems are not meant to be exhaustive. In particular, these lists focus mainly on the recent efforts.

profiles. Since these profiles were gathered on the end user's machine, they obviously reflected the prevalent program usage. In the background, after the user had stopped running an application, the re-optimization agent would run, analyze the profile data recently collected, and re-optimize the application if it determined that re-optimization would be worthwhile. The run-time overhead of this kind of approach is kept low by using low-overhead profiling systems (e.g., DCPI [5]) and because re-optimization occurs off-line. The scope of this approach is actually better than PGC since optimization can occur on the linked executable. Notice that re-optimization may include translation from one instruction set to another.

Instead of completing the optimization process at compile time and then attempting to re-optimize the application after one or more program runs, techniques for run-time code generation stage the compilation process so that optimization can occur during the program run. In general, systems like DyC [23,24] perform the majority of the optimization process at compile time and leave only selected pieces for completion at run time. In particular, these systems use concepts and techniques from the partial evaluation community to determine what program segments could benefit from optimization based on information available only at run time. The compiler then creates an executable capable of capturing that run-time information, performing the associated optimization, and completing the code generation process. Because the bulk of the work is done at compile time, these systems can exhibit low run-time overheads. As structured today, the systems for run-time code generation rely on the programmer to indicate what program segments to optimize dynamically. There is however on-going work investigating ways to improve the transparency of this kind of an approach (e.g., see Mock et al. [37]).

I separate the on-line optimization systems in the last two categories in Table 1 from those listed under dynamic code generation because all of the remaining approaches share the goal of being as transparent as possible. Still, these last two categories encompass a wide range of approaches and techniques. I divide them into only two broad categories: those that use only software or a combination of hardware and software techniques (*on-line optimization in software*), and those that use only hardware techniques (*on-line optimization in hardware*).

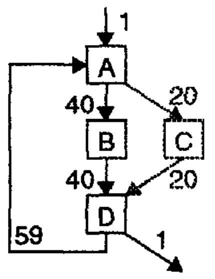
The hardware-only approaches are simply the next steps in the logical progression of the computer industry's work on caching and out-of-order execution. Computer architects are working hard to increase the optimization scope from something akin to peephole optimizations to something closer to trace-based optimizations (e.g., see Friendly, Patel, and Patt [21]). In the future, the hardware will not just renumber registers, rearrange the instruction stream, and remove unconditional branches, but it will change and even eliminate large sequences of instructions.

The approaches that include some element of software support vary significantly in the scope of their on-line transformations, since some of these approaches also perform binary translation. However, in terms of their application of sophisticated optimization, all of the recent systems have been *selective*: optimization occurs only on those program segments that account for the majority of the execution time. This emphasis is a direct consequence of the need to minimize the run-time overhead of the optimizer. To get the most benefit from this time spent optimizing, we want to identify not only those program segments that are dynamically important, but also those that are most amenable to optimization. How to best accomplish this is on-going research and the topic of Section 4. Overall, I feel that we have just begun to explore the potential of the systems in this category.

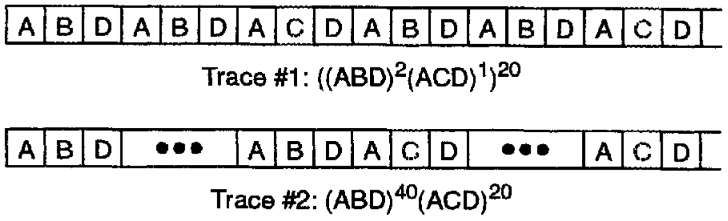
In summary, Table 1 illustrates that there exists a wide range of approaches that use FDO to alter a program and improve its run-time performance. If we step back from the details however, we can see two definite trends emerging from this set of technologies. The first is a movement toward units of optimization based on a program's run-time behavior and not on a set of programmer-specified boundaries. The reasons for this movement are further discussed in Section 4. The second is a movement away from executables as immutable objects. As we have discussed, the importance of FDO has grown as hardware and software technology has advanced and as the lifetimes of applications have increased. Since FDOs depend upon more than just information gleamed from the static code base (i.e., they depend upon information that may change in the future or may not even be available at compile time), it is unreasonable to expect that the compiler can produce an executable that is appropriate and effective for the entire lifetime of any long-lived application. In Section 5, we present further arguments for the concept of *mutable executables*.

4 The Unit of Optimization

Traditionally, static optimizers have used code boundaries defined by the programmer as the boundaries for their units of optimization. For example, compilers often optimize each procedure in an application in isolation. As programmers concern themselves more with issues of understandability and maintainability and as more programmers adopt the object-oriented programming paradigm, these programmer-defined boundaries will make less and less sense as delimiters for units of optimization. The brute-force approach of analyzing and transforming the application as a single, monolithic piece of code fails because many of our existing global



(a) *Edge profile.*


A B D A B D A C D A B D A B D A C D
Trace #1: ((ABD)²(ACD)¹)²⁰
A B D ••• A B D A C D ••• A C D
Trace #2: (ABD)⁴⁰(ACD)²⁰

(b) *Two traces that produce the edge profile in (a).*

Figure 1. Example of how point profiles limit our understanding of program behavior. From the edge profile, we cannot determine if the program ran trace #1 or #2.

optimizations do not scale well as the optimization scope grows. This approach also runs counter to the trend toward shipping applications as a collection of independent DLLs. A different approach is needed.

4.1 Setting boundaries based on run-time behavior

This section argues that the optimization of a program by independently optimizing pieces of that program is not the problem. By continuing this approach, we can continue to use and benefit from the large pool of existing global optimization algorithms. Instead of changing how we determine the size of the units of optimization, this section argues for changing how we determine the contents of them. In particular, optimizers should select optimization units that reflect the current program behavior (as determined by analysis of feedback information) and are reasonably sized (given the computational complexity of the optimization techniques).

As noted by Cohn and Lowney [13], static optimizers appear to miss a large number of optimization opportunities when one reviews the dynamic instruction stream. Computer architects have also noticed this fact and proposed techniques such as value prediction [35] and instruction reuse [46], which exploit the redundancies in the dynamic instruction stream. From the perspective of this paper, these observations argue that we group together in the units of optimization those segments of the application that execute frequently together.

This idea is not entirely new. Hank, Hwu, and Rau [27] proposed this kind of an approach for traditional PGC systems, and the mechanism for instruction scheduling and register allocation in the Multiflow compiler [36] is a limited example of this kind of an approach. All of the prior work however has used *point* profiles—profiles such as edge or block counts that aggregate how often individual program points appeared in an execution trace—to identify the segments of the application that execute frequently together. Though these profiles provide us with more information about a program's behavior than we can determine through static analysis alone [49], the level of understanding that we can gain from point profiles is limited, especially if the program's execution involves non-trivial control flow. Figure 1 provides an example where two different executions of the same code segment produce the same edge profile.

Recent work in the area of *temporal* profiles—profiles such as path counts that aggregate how often sequences of basic blocks appeared in the execution trace [11,54]—provide more insight into the program's temporal behavior. Using temporal profiles, it is possible to build units of optimization that more accurately reflect the run-time behavior of an application. For instance, Young and Smith [53] describe how to use path profiles to construct and schedule superblocks that match the most frequently occurring program traces. Using the edge profile in Figure 1, a traditional superblock unroller would produce an unrolled loop body with the block sequence ABDABDABD (assuming an unrolling factor of three), since all it can determine is that block B occurs more frequently than block C. Using path profiles of the traces in Figure 1, Young and Smith's algorithm would produce the following: a single unrolled loop body with the block sequence ABDABDACD for the path profile of trace #1; and two unrolled loop bodies with the block sequences ABDABDABD and ACDACDACD for the path profile of trace #2. Clearly, we can achieve a higher completion rate by using the path-profile-based unrollings than the edge-profile-based one. Gloy and Smith [22] collect a different kind of temporal profile and show how to use it to achieve better procedure layouts.

4.2 Understanding phased behavior

Though the optimization community has gained greater insights into program behavior based on existing temporal profiling techniques, I believe that we have only begun to find ways to identify interesting program behavior and use that information to direct program optimization. This section describes one as yet largely untapped aspect of program behavior, called *phased behavior*, that software-based on-line optimization systems can use to make their optimization efforts cost effective. In this section, I define what I mean by the term phased behavior, illustrate some examples of phased behavior, and argue why such behavior might be interesting to an on-line optimization system. Section 4.3 uses some of the insights in this section to describe how we can build on-line monitoring systems that are able to identify the important run-time behaviors of an application in time to exploit them.

I define phased behavior as the tendency for a piece of code to exhibit a sequence of behaviors, each for an extended period of time. For the purposes of this definition, the time period may correspond to a single program run, or it may span multiple runs. In an application that displays phased behavior, there exists at least one code region where optimizing for aggregate behavior is sub-optimal. We can obtain better performance if we apply one optimization strategy to that region for one phase of the program's execution and another strategy during another phase. Phased behavior is problematic when the persistence of a FDO is longer than the phase. For example, phased behavior across program runs is problematic for the off-line optimization schemes that assume the aggregate behavior of a code segment is dominant.

Branch-based phased behavior. I begin with examples of phased behavior associated with the execution of conditional branches. The run-time handling of conditional branches has a tremendous effect

```
if (flat_object(isect.object)) {     // b35
  if (n_dot_v < 0.0) {               // b36
    // ...
  }
```

Figure 2. Code snippet from shade() *in* ray.c.

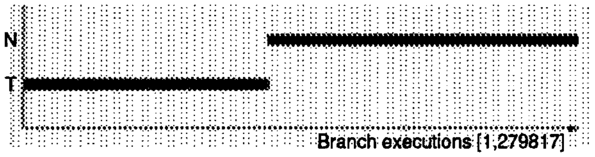


Figure 3. Execution trace of b36 on the input scene1. *The horizontal axis plots time from left to right as measured in executions of this conditional branch. This branch executed a total of 279,817 times. For each branch execution, we mark in the vertical dimension whether the branch was taken (T) or not taken (N).*

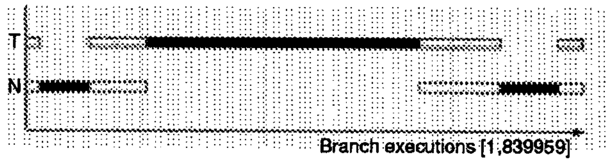


Figure 4. Execution trace of b35 on the input scene1. *We use shading to distinguish contiguous segments of taken (or not-taken) executions from those segments that have an interleaving of taken and not-taken executions. For example, between executions 183272 and 595513, inclusive, b35 is only taken.*

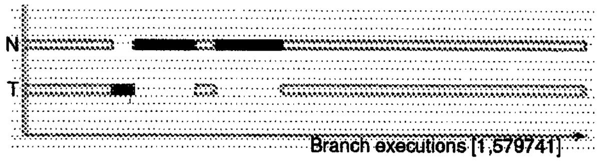


Figure 5. Execution trace of b36 on the input scene24.

on the performance of modern systems, and thus any new insights that we can gain in this area will likely have an important impact on the industry. I start with obvious examples of phased behavior in the execution of individual branches and then move to less obvious (but probably more common) examples. I conclude with some thoughts on the effect of these observations on program optimization. The interesting issue here is that all of the examples involve branches that are weakly biased; current optimization strategies concerned with conditional branches are tailored toward the identification and exploitation of highly-biased branches. Except for code transformations like static correlated branch prediction [54], weakly biased branches are largely ignored by modern optimization techniques.

My first few examples focus on two conditional branches that occur in the `shade` procedure from `ray.c` of *rayshade*, the C ray-tracer developed by Peter Holst Andersen [6]. Figure 2 lists the code associated with these two branches, which I label b35 and b36. This code looks much like the code associated with the other conditional branches in `ray.c`, and thus there is not any obvious syntactic clue alerting us to the following interesting program behavior.

Figure 3 plots the execution trace of b36 when *rayshade* is run on the input *scene1*. In the aggregate, b36 is a weakly-biased branch, a branch taken 44.0% of the time. As Figure 3 illustrates however, b36 is taken exclusively in its first 123,090 executions, and it is never taken after that. This is a clear example of phased behavior; b36 exhibits one behavior and then another, each for a long period of time.

Figure 4 plots the execution trace of b35 during the same run. This figure displays an interleaving of taken and not-taken executions (non-shaded segments) that does not appear in Figure 3. Even so, the execution of b35 contains a long period of time where it is always taken and two smaller periods where it is always not taken. Overall, nearly 50% (49.1%) of this branch's execution is spent in the large taken segment—fairly amazing for a branch that is taken only 66.9% of the time in the aggregate.

Figure 5 shows that the phasing behavior in Figure 3 is not something that we could exploit using traditional profile-driven optimization. Figure 5 plots the execution trace of b36, but this time when *rayshade* is run on the input *scene24*. Comparing Figures 3 and 5, we see that b36's execution is now more "random" than it was under *scene1*. The phased behavior of this branch is influenced by the input data set. However, the aggregate behavior of this branch has not changed noticeably; b36 is taken 39.8% of the time under *scene24* and 44.0% of the time under *scene1*.

As one might expect, the execution trace of a weakly-biased branch does not always contain long periods of time where the branch executes in only one direction. As illustrated in Figure 6 however, some of these weakly-biased branches still exhibit phased behavior that is interesting to an optimization system. In particular, Figure 6 looks at phased behavior in the bias of a conditional branch (labeled b99) in the SPECint92 benchmark *compress*.

Figure 6 shows that the bias of b99 oscillates between periods of low and high bias over time. In the aggregate, this branch would be classified as a weakly-taken branch, based on its average taken percentage of 78.0%. Clearly, this is the wrong classification for periods, like the beginning of the program, where the branch is nearly always not taken.

The actual shape of the curve in the graph plotting bias against time depends upon the characteristics of the input data set. To understand this, I first must explain a bit more about the functioning of the code in this section of *compress*. Branch b99 is part of the hashing code for the code table in *compress*. Specifically, b99 checks to see if the probed slot in the code table is empty after we found that we did not have a hash hit. This explains the initial increase in the bias of this branch; the table is initially empty and slowly fills, changing the branch's bias. The later abrupt changes from a high to a low bias are due to the fact that *compress* implements block compression with an adaptive reset, as described in the comments in the source code of *compress*. If the compression ratio begins to decrease after the code table reaches a predefined fill threshold, the program will clear the code table, output a special CLEAR code, and begin a new encoding. Since this criterion depends upon the characteristics of the input, the shape of the plotted curve also depends upon the characteristics of the input.

The vast majority of optimizations concerned with branching behavior focus their attention on the aggregate behavior of each branch. For the branches discussed above, this would result in lost optimization opportunities. The clear phased behavior in Figure 3 leads us to consider one optimization strategy for the first 123,090 executions of b36 and a different strategy for the last 156,727. Furthermore, given the dependence of b36's behavior on the input data set, we would probably want to implement this flexibility in an on-

```
// starting at line 790
if (hit in hash table)
  // ...
else if (empty slot)  // b99
  // ...
else
  // probe hash table
```

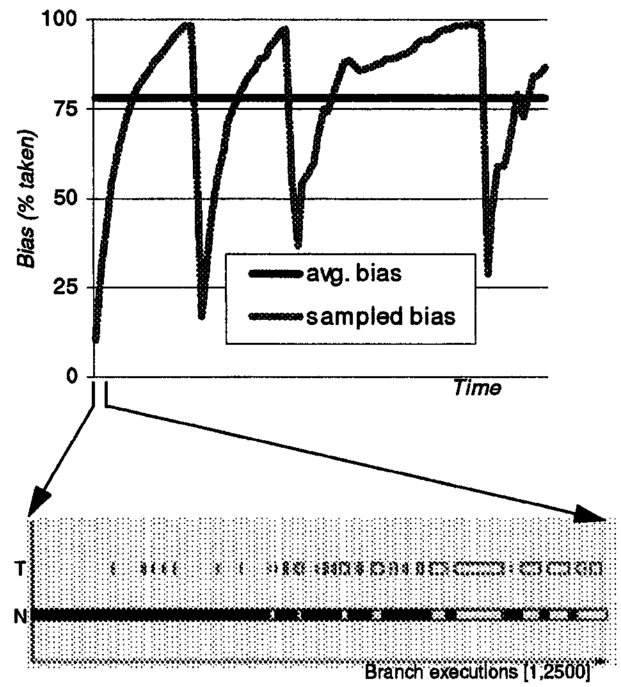


Figure 6. Execution trace of b99 in compress.c. *The top graph plots the bias of this branch at a granularity of 5000 executions. This branch executed a total of 433,223 times. The bottom graph illustrates the pattern of taken and not-taken outcomes in the first 2500 executions.*

line optimizer. Considering b35, adaptation between an optimization strategy involving speculation (to take advantage of long execution runs in one direction) and predication (for those execution runs with no obvious predominant direction) is a potentially fruitful avenue of attack for the branch execution pattern in Figure 4.

Existence of sizable periods of high bias in the execution history of weakly-biased branches implies a change in the importance of individual program paths over time. This observation may affect how we build path-based optimizers. For example, if a branch within a program path transitions from mostly taken to mostly not-taken (or vice versa), it is likely that the common paths before and after this transition share code blocks. To get the most benefit, a path-based optimizer would duplicate the common blocks. Code duplication impacts performance by stressing the memory hierarchy. If we can identify the phase transitions, we could dynamically generate and cache the appropriate path-based optimization—potentially improving performance by never having both versions in memory at the same time and by linking each in as if the other had never existed.

Value-based phased behavior. Phasing occurs not only in a program's control flow, but also in its value stream. The following are

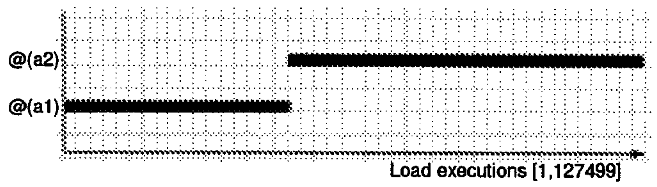


Figure 7. Value trace of load d2542 in vortex. *The horizontal axis plots the executions of this load from left to right. For each load execution, we record in the vertical dimension the value returned. This load returns only two unique values: the base addresses of arrays named (for convenience) a1 and a2.*

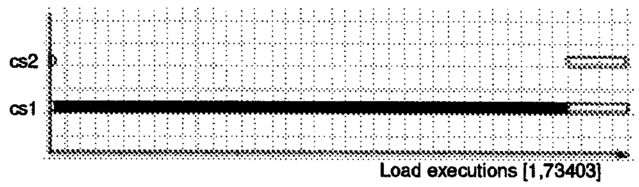


Figure 8. Value trace of load d4531 in m88ksim. *This load restores the return address register. Since the load returns only two unique values,* `addunsigned` *is called from only two call sites during this run.*

two clear examples of phased behavior in the values returned by load instructions. Techniques like value prediction [35] have increased the research community's interest in the values loaded by programs. Like conditional branches, the handling of memory loads has a large impact on the overall performance of applications.

Figure 7 plots the value trace of a load (called d2542) from the procedure `BMT_CommitParts` in `bmt10.c` of the SPECint95 benchmark *vortex*. This instruction loads the base address of the array `Vlists` from the global pointer table. As Figure 7 shows, the program initially accesses one array in memory through this pointer variable and then switches to another array for the remainder of the run. Even though this load returns more than one value, a run-time optimizer could take advantage of the fact that the values are not interleaved in the value trace.

As we saw earlier, interesting phased behavior may occur in traces where there is an interleaving of different results. Figure 8 plots the value trace of a load (called d4531) at the end of procedure `addunsigned` in `dpath.c` of the SPECint95 benchmark *m88ksim*. This load restores the return address register just before it is used in the procedure return. The value trace shows that `addunsigned` is called from just two call sites (named cs1 and cs2 for convenience) during the program run. The interesting aspect of this trace is that, except for two small pieces at the beginning and end of the trace, `addunsigned` is called exclusively from cs1. During this extensive time period, an on-line optimizer could elect to inline `addunsigned` at cs1 and do so without increasing the instruction memory footprint.

Related Work. Though this section has presented examples of phased behavior, we do not as yet understand how common this behavior is or how much benefit we could gain from exploiting it. Even if we cannot use an improved understanding of phased behavior to make applications run faster, there are other ways in which we can use this information. For example, Sherwood and Calder [43] are investigating this kind of program behavior to help reduce the amount of time it takes to perform architectural simulations. By understanding the location and duration of a program's phases, computer architects could state with confidence that a simulation of only small pieces of a program's execution would match the results obtained from a simulation of the entire program execution. Albonesi [3] has proposed an adaptive processor architecture based on observations of variations in a program's instruction level parallelism (ILP). (Jouppi [30] was the first to examine the distribution of ILP in programs.) Albonesi describes a processor that automatically reduces the complexity of its pipeline when ILP falls so that the clock frequency (and thus performance) can improve. Finally, we may be able to use our understanding of program phases to design better techniques for dynamically managing processor temperature and power [39].

4.3 Monitoring to find the important run-time units

To this point, we have discussed the need to build optimization units whose boundaries reflect a program's current behavior, and the potential benefits of collecting many different and extensive kinds of temporal information. To take advantage of this information in an optimizer, there are two more questions that we need to discuss: how we can monitor a program's execution to collect the desired data about the program's behavior; and how to use this information to identify the actual unit of optimization. This section concentrates on the issue of program monitoring. Young [52] provides an answer to the second question for optimizers based on path profiles; more general answers to this question will have to wait until we have gained more experience with temporal profiles and their use in selecting optimization units.

In this section, I focus on the category of systems that perform optimization on-line and in software. Many of the existing systems in this category already contain on-line monitoring systems that allow them to focus their optimization efforts and better amortize their optimization costs. For example, the IBM DAISY system [19] and the IBM Jalapeño dynamic optimizer [12] both employ a two-tiered approach to optimization: each code region is "lightly" optimized at first and then later aggressively optimized when it has been shown to be frequently executed. The HP Dynamo system [10] implements a similar policy: it distinguishes between infrequently and frequently executed code for the purposes of determining which code to emulate and which code to optimize and place in its code cache.

Though these existing monitoring systems work quite well for their intended uses, they are also quite simple. They each use something akin to the execution counts in point profiles as a threshold for determining how to classify a program point (i.e., as frequently or infrequently executed). Dynamo [10], for example, sets an execution threshold on blocks that are the target of backward branches, and it optimizes the first trace of blocks extending from such a block when the threshold is exceeded. Perhaps this is all that we will ever be able to afford in a monitor for an on-line optimizer; however if our monitoring system is to collect information about program behavior that will help select the dynamically-important units of optimization, we need something more.

To see how we can afford more, consider the following observations about a system that monitors for phased behavior. Due to the inherent characteristics of phased behavior, we are going to have to monitor a program's behavior during its entire execution. This may sound like an expensive proposition, but realize that we do not have to monitor all of the details all of the time. Programs are predictable. Once we have identified and optimized a program for some common behavior, we do not have to keep recognizing that behavior. We simply want to know when the common behavior changes. Furthermore, an on-line optimizer cannot afford to optimize every segment of a program's execution. It must be selective and focus on the common behaviors, behaviors where we can amortize the overhead of run-time optimization. In other words, the monitoring system should ignore the program's exceptional (uncommon) behaviors.

The mechanism in Dynamo [10] for flushing its code cache in reaction to changes in an application's working set is an example of a mechanism employing many of the above observations. At Harvard, we are building a software-based on-line optimizer, called Deco, with a multi-level approach to program monitoring that exhibits all of the characteristics mentioned above. We first use a low-overhead sampling system, such as those found in DCPI [5] or Morph [55], to identify the frequently executed program regions. We then instrument these dynamically important regions to record details of the program behavior in that those regions. We refer to the instrumentation as *ephemeral* because it exists only temporarily—only long enough to identify the unit of optimization and its tendencies accurately.

5 Mutable Executables

In this paper, I have expounded the virtues of being able to change the specification of a program's execution while running it. To change the specification requires the ability to write into the instruction stream, and the organization of modern, high-performance microprocessors does not make this easy to do. The cost of flushing a set of instructions from the instruction cache, for example, is relatively high on most machines. Furthermore, there exists a common perception that we should not need to write into the instruction stream of a running executable. Most modern operating systems initialize the text segments of an application as read-only. In effect, we are saying that there is something special about the executable. Software, once compiled and shipped, is no longer "soft" or mutable.

Figure 9 compares the process of creating an executable with that of creating a microprocessor. Hardware architects figured out long ago that the rapid changes in hardware technology made it inappropriate to consider any implementation of an instruction set architecture (ISA) as something "special" and immutable. Today, computer architects are wary of including technology-dependent "solutions" to performance problems (e.g., delayed branches) in their ISA; these solutions inevitably become "bad" ideas due to technology advances. As such, the hardware community is in a better state of mind for thinking about on-line optimizers than the software community.

Even though executables are generally thought of as immutable, the software community has taken a few small steps toward the idea of executables as mutable objects with its acceptance of DLLs and dynamically-loadable classes. Furthermore, existing systems that employ run-time code generation also write executables while they run. Yet, the most aggressive on-line optimizers (the lower two categories of Table 1 on page 4) often take extraordinary steps to make sure that they do not touch the original executable or cause any visible side effects that deviate from the sequence of side effects that would have occurred during execution of the original executable. Again, this is an odd view given that the same executable would incur a different sequence of, say, TLB misses on two different implementations of the same ISA.

My point is that we as a community are not consistent in our views and expectations of the execution environment. If you believe my earlier arguments about the increasing need for FDO, we should invest in discussions and experiments that help to identify the level of mutability that we could profitably support in our executables. In addition, we should consider what hardware and operating system primitives would aid in the development and support of mutable executables. Perhaps, in the future, mutable executables will be as accepted as virtual memory is today.

$$\begin{tikzcd}
\text{problem} \arrow[r, "\textit{programmer}"] & \text{algorithm in HLL} \arrow[r, "\textit{compiler}"] & \textbf{executable} \\
\text{class of problems} \arrow[r, "\textit{architect}"] & \textbf{ISA} \arrow[r, "\textit{designer}"] & \text{implementation}
\end{tikzcd}$$

Figure 9. A comparison of the process of compiling an executable with the process of designing a microprocessor. In the upper flow, the executable is considered (mostly) immutable. In the lower flow, the ISA is considered (mostly) immutable.

6 Challenges

There are many challenges to overcome if we are to make the vision described in the prior sections into reality. This section presents my list of the most important of these. You should not view this as an exhaustive list; its purpose is as a starting point for further discussion. I truly believe that we can overcome each of these challenges. As proof of my belief, I provide references to some of the promising current research efforts that may provide full or partial solutions. I also include a set of lofty objectives that I would like to see the community attain while overcoming each barrier.

Mindset. The biggest challenge to the vision in this paper is our existing mindset. I have found it difficult for people to think about executables as mutable objects. Some of this aversion comes from the concern of programmers that the process of optimization can silently turn a correct program into a buggy one. Kernighan and Pike [31, page 176] state that "the more aggressively the compiler optimizes, the more likely it is to introduce bugs into the compiled program." This seems to be a common perception, though I could not find any published studies supporting this view.

We may achieve some leverage on this problem by adapting some of the work on safety and correctness in the area of mobile code. For example, Rinard [38] proposes that we build compilers that validate the correctness of each transformation when it is applied. He believes that this is a more fruitful approach than one in which we prove that the compiler works correctly on all possible legal programs.

Debugging. Even if we can guarantee that the optimizer performs transformations correctly, many software vendors refuse to compile with high levels of optimization because optimized code is difficult to debug. As optimizations have become more sophisticated, it has become increasingly difficult to have the debugger mask the effects of the optimization process from the programmer. If automatic and on-line optimization systems are to be successful, we need to build tools that help the programmer debug optimized executables. Tice [48] proposes one such tool. Her system helps programmers to understand the effects of optimization on their applications without having to know how or exactly what optimizations were performed.

If the current trends continue, hardware manufacturers will soon consider the implementation of on-line code transformations that software vendors hesitate to invoke at compile time. Since it is already difficult for programmers to debug their own optimized code, it is unreasonable for the industry to assume that a user could diagnose the cause of a fault in an on-line optimized executable and create a bug report. A bug report should be generated by someone intimate with the details of the program execution at the time of the fault. On-line optimization systems provide us with such an agent—the monitor. Perhaps we could adapt existing monitoring techniques to gather information helpful in diagnosing a program's run-time faults and useful in transforming that program so that it avoids those faults in future runs.

Infrastructure. Compilers are very large pieces of software that take many person-years of work to develop. One of the goals of the Harvard Deco project is to find ways to share code and technology between our traditional compile-time optimizers and our experimental on-line optimizers. Though we may wish to adapt our current optimizations so that they execute more efficiently in a runtime environment, simply moving them between the compile-time and run-time environments should not be a reason to have to recode. In Machine SUIF [45], we have developed a programming interface for specifying optimization analyses and transformations that will allow us to test our on-line optimizations in a traditional compile-time environment and do so without having to re-code them.

Multi-disciplinary approach. To make significant changes to today's execution environment requires that the members of many research communities cooperate. I have seen the beginnings of such cooperative efforts in the development of DCPI [5] and Morph [55], two profiling systems based on statistical sampling. These two projects drew together researchers from the operating system and compiler communities. In my discussions with computer architects from large microprocessor vendors, I have heard them express their willingness to implement new performance monitors in hardware, but they first need some justification that what is requested will be useful.

Commercial realities. A large body of research exists in support of the performance potential of FDO. Only recently, however, has the research community begun to investigate the hurdles that must be crossed in order for an approach like PGC to be used daily within the development group of a large commercial software vendor. To achieve such acceptance, we must look scientifically at the problems of profile collection and maintenance. The recent works of Albert [4], Wang et al. [51], and Savari and Young [41] are noteworthy examples of this kind of research. Finally, there are other commercial realities like testing that must be carefully considered.

7 Conclusions

In this paper, I proposed a broad view of what FDO is and when it should be used. By considering any technique that alters a program based on information gathered at run time to be FDO, it is easy to see that this approach is already successful and ubiquitous—nearly every hardware manufacturer implements some set of FDO techniques in its microprocessors. I also presented a classification that encompasses much of the prior work in FDO. Even with this rich body of prior work, I believe that we are just beginning to tap the potential of and explore the design space in FDO.

Furthermore, I discussed several important trends in application development, software engineering, hardware technology, and the Internet that are increasing the need for and interest in FDO. To meet these needs adequately, I argued for a movement away from two static models: units of program optimization based on programmer-defined code boundaries, and immutable executables. Instead, we should build optimization units with boundaries defined by the program's run-time behavior. This frees the programmer to set code boundaries that make the program easy to understand and maintain, and it enables the optimizer to uncover optimization opportunities unavailable when it considers all program paths to be equally important. We should also consider our executables to be mutable objects. By making it easy to change executables after they have been shipped and the instruction stream while programs run, we can build optimizers that are able to adapt applications so that they always run well, no matter how things change.

8 Acknowledgments

The ideas presented in this paper benefited greatly from discussions with Vas Bala, Glenn Holloway, and the members of my research group at Harvard.

Michael D. Smith is funded in part by a NSF Young Investigator award (grant no. CCR-9457779), NSF grant no. CDA-94-01024, DARPA grant no. NDA904-97-C-0225, and research grants from AMD, Compaq, Digital Equipment, HP, IBM, Intel, and Microsoft.

9 References

[1] S. Adve, et al. "Changing Interaction of Compiler and Architecture," *Computer*, 30(12):51–58, December 1997.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1988, p. 589.

[3] D. Albonesi. "Dynamic IPC/Clock Rate Optimization," *Proc. 25th Annual International Symposium on Computer Architecture*, pp. 282–292, June 1998.

[4] G. Alpert. "A Transparent Method for Correlating Profiles with Source Programs," *Proc. Second Workshop on Feedback-Directed Optimization*, held in conjunction with MICRO-33, pp. 33–39, November 1999

[5] J. Anderson, et al., "Continuous Profiling: Where Have All the Cycles Gone?," *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[6] P. Andersen. "Partial Evaluation Applied to Ray Tracing," unpublished technical report, January 1995.

[7] A. Appel. *Modern Compiler Implementation in C*, Cambridge University Press, Cambridge, UK, 1998.

[8] A. Ayers, S. de Jong, J. Peyton, and R. Schooler. "Scalable Cross-Module Optimization," *Proc. ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pp. 301–312, June 1998.

[9] J. Backus. "The History of Fortran I, II, and III," *IEEE Annuals of the History of Computing*, 20(4), Oct.-Dec. 1990.

[10] V. Bala, E. Duesterwald, and S. Banerjia. "Dynamo: A Transparent Runtime Optimization System," to appear in the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, June 2000.

[11] T. Ball and J. Larus, "Efficient Path Profiling," *Proc. 29th Annual International Symposium on Microarchitecture*, pp. 46-57, 1996.

[12] M. Burke, et al. "The Jalapeño Dynamic Optimizing Compiler for Java," *Proc. ACM Java Grande Conference*, June 1999.

[13] R. Cohn and G. Lowney. "Hot Cold Optimization of Large Windows/NT Applications," *Proc. 29th Annual International Symposium on Microarchitecture*, pp. 80–89, December 1996.

[14] R. Cohn and G. Lowney. "Feedback Directed Optimization in Compaq's Compilation Tools for Alpha," *Proc. Second Workshop on Feedback-Directed Optimization*, held in conjunction with MICRO-33, pp. 3–12, November 1999.

[15] C. Consel, et al. "Tempo: Specializing systems applications and beyond," *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.

[16] T. Conte and S. Sathaye, "Dynamic rescheduling: A technique for object code compatibility in VLIW architectures," *Proc. 28th Annual International Symposium on Microarchitecture*, November 1995.

[17] T. M. Conte, K. N. Menezes and M. A. Hirsch, "Accurate and practical profile-driven compilation using the profile buffer," *Proc. 29th Annual International Symposium on Microarchitecture*, pp. 36–45, December 1996.

[18] C. Dulong, et al. "An Overview of the Intel IA-64 Compiler," *Intel Technology Journal*, November 22, 1999 (Q4).

[19] K. Ebcioglu and E. Altman. "Dynamic Compilation for 100% Architectural Compatibility," *Proc. 24th Annual International Symposium on Computer Architecture*, pp. 26–37, June 1997.

[20] D. Engler, W. Hsieh, and M. Kaashoek. "'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation," *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 131–144, January 1996.

[21] D. Friendly, S. Patel, and Y. Patt. "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," *Proc. 31st Annual International Symposium on Microarchitecture*, pp. 173–181, November 1998.

[22] N. Gloy and M. Smith. "Procedure Placement using Temporal-Ordering Information," *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, September 1999.

[23] B. Grant, et al. "Annotation-Directed Run-Time Specialization in C," *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 163–178, June 1997.

[24] B. Grant, et al. "An Evaluation of Staged Run-Time Optimizations in DyC," *Proc. ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pp. 293–304, May 1999.

[25] R. Gupta, D. Berson, and J. Fang. "Path Profile Guided Partial Dead Code Elimination Using Predication," *Proc. of Parallel Architectures and Compilation Techniques*, pp. 102–115, November 1995.

[26] R. Gupta, D. Berson, and J. Fang. "Path Profile Guided Partial Redundancy Elimination Using Speculation," *Proc. of IEEE International Conference on Computer Languages*, pp. 230–239, May, 1995.

[27] R. Hank, W. Hwu, and B. Rau. "Region-Based Compilation: An Introduction and Motivation," *Proc. 28th Annual International Symposium on Microarchitecture*, pp. 158–168, November 1995.

[28] R. Hookway and M. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal*, 9(1):3-12, 1997.

[29] W. Hwu et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing* 7(1/2):229-248, Kluwer Academic Publishers, May 1993. Also see http://www.crhc.uiuc.edu/Impact.

[30] N. Jouppi. "The Non-Uniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance," *IEEE Transactions on Computers*, 38(12):1645–1658, December 1989.

[31] B. Kernighan and R. Pike. *The Practice of Programming*, Addison Wesley, Reading, MA, 1999.

[32] A. Klaiber. "The Technology Behind the Crusoe Processors," White paper (http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf), Transmeta Corporation, January 2000.

[33] D. Lee, P. Crowley, J.L. Baer, T. Anderson, and B. Bershad. "Execution Characteristics of Desktop Applications on Windows NT," *Proc. 25th Annual International Symposium on Computer Architecture*, pp. 27–38, June 1998.

[34] J. Levine. *Linkers and Loaders*, Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[35] M. Lipasti and J. Shen. "Exceeding the Dataflow Limit Via Value Prediction," *Proc. 29th International Symposium on Microarchitecture*, pp. 226–237, December 1996.

[36] P. Lowney, et al. "The Multiflow Trace Scheduling Compiler," *The Journal of Supercomputing*, 7:51–142, January 1993.

[37] M. Mock, M. Berryman, C. Chambers, and S. Eggers. "Calpa: A Tool for Automating Dynamic Compilation," *Proc. Second Workshop on Feedback-Directed Optimization*, held in conjunction with MICRO-33, pp. 100–109, November 1999.

[38] M. Rinard. "Credible Compilation," MIT Laboratory for Computer Science technical report MIT-LCS-TR-776, Cambridge, MA, March 1999.

[39] E. Rohou and M. Smith. "Dynamically Managing Processor Temperature and Power," *Proc. Second Workshop on Feedback-Directed Optimization*, held in conjunction with the 32nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 73–82, November 1999.

[40] E. Rotenberg, et al. "Trace Processors," *Proc. 30th Annual International Symposium on Microarchitecture*, pp. 138–148, December 1997.

[41] S. Savari and C. Young. "Comparing and Combining Profiles," *Proc. Second Workshop on Feedback-Directed Optimization*, held in conjunction with the 32nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 50–62, November 1999.

[42] W. Schmidt, et al. "Profile-Directed Restructuring of Operating System Code," *IBM Systems Journal*, 37(2):270–297, 1998.

[43] T. Sherwood and B. Calder. "Time Varying Behavior of Programs," UC San Diego Technical Report CS99-630, August 1999.

[44] J. Smith and G. Sohi. "The Microarchitecture of Superscalar Processors," *Proceedings of IEEE*, 83(12):1609–1624, December 1995.

[45] M. Smith. "Extending SUIF for Machine-dependent Optimizations," *Proc. First SUIF Compiler Workshop*, Stanford, CA, pp. 14–25, January 1996. Also see http://www.eecs.harvard.edu/machsuif.

[46] A. Sodani and G. Sohi. "Dynamic Instruction Reuse," *Proc. 24th International Symposium on Computer Architecture*, pp. 194–205, July 1997.

[47] Sun Microsystems. "The Java Hotspot Performance Engine Architecture," White paper (http://java.sun.com/products/hotspot/whitepaper.html), Sun Microsystems, April 1999.

[48] C. Tice. "Non-Transparent Debugging of Optimized Code," Ph.D. Dissertation, Computer Science Division Tech Report Number UCB/CSD-99-1077, University of California at Berkeley, October 1999.

[49] D. Wall. "Predicting Program Behavior Using Real or Estimated Profiles," *Proc. SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 59–70, June 1991.

[50] Z. Wang and N. Rubin. "Evaluating the Importance of User-Specific Profiling," *Proc. 2nd USENIX Windows NT Symposium*, August 1998.

[51] Z. Wang, K. Pierce, and S. McFarling. "BMAT — A Binary Matching Tool," *Proc. Second Workshop on Feedback-Directed Optimization*, held in conjunction with MICRO-33, pp. 40–49, November 1999.

[52] C. Young, "Path-based Compilation," Ph.D. Dissertation, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA, January 1998.

[53] C. Young and M. Smith, "Better Global Scheduling Using Path Profiles," *Proc. 31st Annual International Symposium on Microarchitecture*, pp. 115-123, December 1998.

[54] C. Young and M. Smith. "Static Correlated Branch Prediction," *ACM Transactions on Programming Languages and Systems*, 21(5): 1028–1075, September 1999.

[55] X. Zhang et al., "System Support for Automatic Profiling and Optimization," *Proc. 16th ACM Symposium on Operating Systems Principles*, 1997.