RC23143 (W0312-097) December 17, 2003 UPDATED May 18, 2004
Computer Science

IBM Research Report

A Survey of Adaptive Optimization in Virtual Machines

Matthew Arnold, Stephen J. Fink, David Grove,
Michael Hind, Peter F. Sweeney
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A Survey of Adaptive Optimization in Virtual Machines

Matthew Arnold, Stephen J. Fink, David Grove,
Michael Hind, and Peter F. Sweeney, *Member, IEEE*

Abstract

Virtual machines face significant performance challenges beyond those confronted by traditional static optimizers. First, portable program representations and dynamic language features, such as dynamic class loading, force the deferral of most optimizations until runtime, inducing runtime optimization overhead. Second, modular program representations preclude many forms of whole-program interprocedural optimization. Third, virtual machines incur additional costs for runtime services such as security guarantees and automatic memory management.

To address these challenges, vendors have invested considerable resources into adaptive optimization systems in production virtual machines. Today, mainstream virtual machine implementations include substantial infrastructure for online monitoring and profiling, runtime compilation, and feedback-directed optimization. As a result, adaptive optimization has begun to mature as a widespread production-level technology.

This paper surveys the evolution and current state-of-the-art of adaptive optimization technology in virtual machines.

Index Terms

adaptive optimization, dynamic optimization, feedback-directed optimization, virtual machines

I. Introduction

The past decade has witnessed the widespread adoption of programming languages designed to execute on virtual machines. Most notably, the Java programming language [1] and more recently the Common Language Runtime [2] have driven virtual machine technology into the mass marketplace.

Virtual machine architectures provide several software engineering advantages over statically compiled binaries, including portable program representations, some safety guarantees, built-in automatic memory and thread management, and dynamic program composition through dynamic class loading. These powerful features enhance the end-user programming model and have driven the success of new languages. However, in many cases, these dynamic features frustrate traditional static program optimization technologies, introducing new challenges for achieving high performance.

In response, the industry has invested heavily in adaptive optimization technology. This technology aims to improve performance by monitoring a program's behavior and using this information to drive optimization decisions. This paper surveys the major developments and central themes in the development of adaptive optimization technology in virtual machines over the last thirty years. We categorize such techniques into three categories: 1) *selective optimization*, techniques for determining when, and on what parts of the program, to apply a runtime optimizing compiler, 2) *feedback-directed code generation*, techniques for using profiling information to improve the quality of the code generated by an optimizing compiler, and 3) *other feedback-directed optimizations*, other uses of profiling information to improve performance.

After defining some terminology (Section II), we present a brief history of adaptive optimization in virtual machines (Section III). The core of the paper then addresses the following topics: selective optimization (Section IV), profiling techniques (Section V), feedback-directed code generation (Section VI), and other feedback-directed optimizations (Section VII). The paper concludes with a discussion of future research topics (Section VIII) and conclusions (Section IX).

II. TERMINOLOGY

Some software programs serve only to provide an execution engine for other programs. This class of software execution engines spans a wide range of domains, ranging from microcode [3] to binary translators [4] to interpreters for high-level languages such as APL [5]. Across these domains, the technological drivers and implementation trade-offs vary significantly depending on the executed program's representation and level of abstraction.

Rau [6] classified program representations into three categories:

- *high level representation (HLR)*: a high-level language such as ALGOL, APL, C++, or the Java programming language,
- *directly interpretable representation (DIR)*: an intermediate representation with a simple syntax and a relatively small set of simple operators, such as JVM bytecode or CLI, and
- *directly executable representation (DER)*: an executable binary representation bound to a machine architecture, such as PowerPC, IA32, or SPARC.

HLRs carry semantic information at a high level of abstraction, with program representations designed to help human understanding rather than machine execution. HLR optimizers focus on providing an effective translation from the high-level abstractions and program representation to a DIR or a DER. Because this translation must cross a wide gap, even simple translation is generally considered too expensive to apply at runtime; so, most HLR compilers operate offline. Optimizing HLR compilers apply aggressive program transformations, often relying on high-level semantic information expressed in the HLR.

Most HLR compilers emit DER code, which a machine can execute directly with no runtime translation overhead. The main disadvantage of DER is that it lacks portability across machine architectures. Furthermore, a DER carries relatively little semantic information, obscuring opportunities for further program transformations. As a result, a DER translator has comparatively little opportunity to optimize a program based on its runtime environment.

DIRs provide a middle ground, which has gained favor in recent years. A DIR provides portability across machine architectures, yet most machines can execute DIR with relatively little runtime translation overhead. A DIR carries semantic information that falls between the other two categories of representations, which facilitates more aggressive program transformations than are easily attainable on DER.

We define a *virtual machine* (VM) to be a software execution engine for programs in Rau's DIR category. This includes implementations of the Java Virtual Machine, Common Language Runtime, and Smalltalk v-code.

In addition to providing a direct execution engine, modern VM architectures provide a managed execution environment with additional runtime services. Cierniak et al. [7] provide a discussion of typical runtime services in a managed execution environment, including automatic memory management, type management, threads and synchronization, dynamic loading, reflection, and exceptions.[1] As Cierniak et al. discuss, each of these services introduces some runtime overhead. This paper reviews adaptive optimization technologies designed to mitigate the overhead of these services.

III. A Brief History of Adaptive Optimization in Virtual Machines

The fundamental concepts of adaptive optimization date back at least to the advent of software. Due to space constraints, this paper focuses on the work that has most directly transferred into today's VM

[1] Cierniak et al. [7] and others use the term *managed runtime* to refer to what we call a virtual machine. Rattner [8] proposes terminology whereby a "virtual machine is but one element of a modern runtime environment", as distinct from other subsystems such as compilers and the garbage collector. We use the single term *virtual machine* to indicate the sum of all these parts.

implementations. We refer the reader to Aycock [9] for an in-depth review of the genesis of these techniques, as well as a survey of related dynamic compilation techniques in domains other than virtual machines.

In the domain of virtual machines, the following five developments stand out as milestones in the evolution of adaptive optimization: Lisp interpreters [10], Adaptive FORTRAN [11], ParcPlace Smalltalk [12], SELF [13], [14], and Java [1].

Lisp interpreters [10] probably represent the first widely used virtual machines. Although Lisp is best-classified as a HLR, its syntax allows simple parsing and nearly direct interpretation. Lisp implementations pioneered many of the characteristic virtual machine services; notably, Lisp drove the development of automatic memory management technology for several decades. Lisp even provided an early precursor to dynamic loading; the Lisp `eval` function would evaluate an expression in the current environment, in effect dynamically adding code to the running program.

Hansen's Adaptive FORTRAN work [11] provided the first in-depth exploration of issues in online adaptive optimization. Although FORTRAN certainly is a HLR, Hansen's system executed a DIR compiler-centric intermediate program representation. Hansen's excellent 1974 thesis describes many of the challenges facing adaptive optimizers with regard to selective recompilation, including models and heuristics to drive recompilation, dealing with multiple optimization levels, and online profiling and control systems.

In their influential 1984 paper [12], Deutsch and Schiffman described the ParcPlace Smalltalk virtual machine, the first "modern" virtual machine. Their Smalltalk implementation introduced many of the core concepts used today, including a full-fledged Just-In-Time compiler, inline caches to optimize polymorphic dispatch, and native code caches. Furthermore, this work demonstrated convincingly that software-only virtual machines were viable on conventional architectures, and could address the key performance challenges without language-specific hardware support.

The SELF project [13], [14] carried on where Deutsch and Schiffman left off, and developed many of the more advanced techniques that appear in virtual machines today. Some technical highlights include polymorphic inline caches, on-stack replacement, dynamic deoptimization, selective compilation with multiple compilers, type prediction and splitting, and profile-directed inlining integrated with adaptive recompilation.

Sun Microsystems introduced the Java programming language [1] in 1995, which stands as a milestone since the JVM became the first virtual machine with major penetration into mainstream markets. Competing for customers, Java vendors poured resources into virtual machine implementations on an unprecedented scale. In the quest for Java performance, vendors embraced the adaptive optimization technologies pioneered in Smalltalk and SELF, and have spurred a renaissance of research and development in the area. The Microsoft Corporation followed the Java programming language with the Common Language Runtime [2], and it appears that virtual machines will continue to proliferate for the foreseeable future.

IV. Selective optimization

We use the term *selective optimization* to denote the policy by which a virtual machine chooses to apply its runtime compiler. In this section, we review the development of the dominant concepts that have influenced selective optimization technology in today's virtual machines. See [9] for a more in-depth review of the early development of these ideas and applications in arenas beyond virtual machines.

A. Interpreters

As discussed in the previous section, early interpreters, such as those for Lisp, may be considered the first virtual machines. Following Lisp, other interpreter-based language implementations gained popularity, including implementations for high-level languages such as APL [5], SNOBOL [15], BCPL [16], and Pascal-P [17]. To this day, interpreters enjoy widespread use in implementations for languages such as Perl [18], Python [19], and MATLAB [20].

Interpreters have employed various techniques to improve performance over the simplest, `switch`-based implementations. Some researchers investigated specialized hardware (e.g. [21]–[23]) to accelerate interpreters. However, at the present, economics of general-purpose hardware have driven this approach out of favor.

Perhaps the most important software optimization for interpreters is *threading* [24]. With basic threading techniques, the interpreter jumps with indirect branches directly from the implementation of one bytecode to the next, simplifying dispatch logic.

Recent work has applied dynamic techniques to improve on basic threading, using runtime translation to customize a threaded interpreter implementation for the input program. Piumarta and Riccardi [25] describe techniques to dynamically generate the threaded code in order to eliminate a central dispatch site and to "inline" common bytecode sequences. Ertl and Gregg [26] extended this work by replicating code sequences and investigating various interpreter generation heuristics, focusing on improving branch prediction accuracy. Gagnon and Hendren [27] adapted Piumarta and Riccardi's techniques to work in the context of dynamic class loading and multi-threading. Sullivan et al. [28] describe cooperation between an interpreter implementation and an underlying dynamic binary optimizer, which improves the efficacy of the underlying optimizer on the interpreter execution.

Despite these advances, an interpreter cannot match the performance of an optimizing compiler. A few early papers discussed techniques to cache interpreter actions, in effect producing a simple runtime compiler (see [9] for an in-depth review). This work progressively evolved into the full-fledged runtime compilers common in today's VMs. Nevertheless, interpreters remain an attractive option for some domains, such as space-constrained devices [29], [30].

B. *Just-In-Time Compilers*

To improve interpreter performance, virtual machines began to incorporate runtime compilers that could perform an optimizing translation from DIR to DER on-the-fly. The simplest scheme is commonly known as a Just-In-Time (JIT) compiler, which compiles each code sequence directly to native code immediately before it executes.[2]

This approach was typified by the seminal work on ParcPlace Smalltalk-80 [12]. This software-only Smalltalk implementation executed interpretable virtual code (v-code) by translating it to native code (n-code) when needed. This system architecture supported three mutually exclusive execution strategies: an interpreter, a simple non-optimizing translator, and an optimizing translator. The execution strategy was chosen prior to execution.

Code space was a scarce resource for the system, and optimized n-code was measured as five times larger than the corresponding v-code. To avoid paging (in fact, the experimental system reported had no virtual memory), Deutsch and Schiffman [12] introduced a code cache for n-code, and would discard and regenerate n-code as needed. Subsequently, code caches have proved effective in other contexts for platforms where code space is scarce (e.g. [31]).

Several later projects adopted the "JIT-only" strategy, including the initial SELF implementation [32] and the initial production Java virtual machines.

C. *Selective Optimization Concepts*

The JIT-only strategy introduces compilation overhead before any code sequence can execute, imposing a heavy burden when compilation resources, either compiled code space or compile time, are scarce. For

[2] The term *JIT* is also commonly used to describe any runtime compiler in a virtual machine, even if it is used selectively to compile some methods while others are interpreted.

these situations, a virtual machine can exploit the well-known fact that most programs spend the majority of time in a small fraction of the code [33], and focus compilation resources only on frequently-executed code sequences, or "hot spots". To this end, the next wave of virtual machines used two implementations, a "cheap" default implementation (either interpreter or a fast non-optimizing compiler) and a more expensive optimizer applied only to program hot spots.

A system employing selective optimization requires three basic components:

1) a profiling mechanism to identify candidates for further optimization,
2) a decision-making component to choose the optimizations to apply for each candidate, and
3) a dynamic optimizing compiler to perform the selected optimizations.

Since these components operate online during program execution, the system must minimize their overhead in order to maximize performance. Concerning profiling, two mechanisms have emerged for obtaining a low-overhead, coarse-grained data to drive selective optimization: *counters* and *sampling*.

The counter mechanism associates a method-specific counter with each method. The counter counts method invocations and perhaps loop iterations.

The sampling mechanism periodically interrupts the system and records the method (or methods) on the top of the call stack. Often, an external clock triggers sampling, which allows significantly lower overheard than incrementing a counter on each method invocation. However, using an external clock as the trigger introduces non-determinism, which can complicate system debugging.

Based on the profile data, the decision-making component selects methods or code sequences for optimization. Many systems employ a simple predetermined threshold strategy, where a value (method counter or sample counter) exceeding a fixed threshold triggers the compilation of one or more methods. Other systems use more sophisticated strategies to select methods and levels for optimization. The next section reviews the implementations and strategies used in selective optimization systems.

D. Selective Optimization Systems

Early work (e.g. [33]) suggested that the user optimize code selectively based on profile information. To our knowledge, Hansen implemented the first system to perform adaptive optimization automatically, in Adaptive FORTRAN [11]. The Adaptive FORTRAN (AF) compiler produced an intermediate form that could either be directly interpreted, or further optimized. The system would selectively apply optimizations to basic blocks or loop-like "segments" of code. After empirical tuning, Hansen settled on a system with three levels of optimization in addition to direct interpretation. The experimental results showed that

AF's adaptive optimization provided better overall performance than JIT-only strategies with any single optimization level.

The SELF-93 implementation [34] applied many of Hansen's techniques. Because SELF was targeted as an interactive programming system, the system could not afford the pauses if it compiled each method with the highest level of optimization. So, SELF-93 initially compiled each method with a fast non-optimizing compiler, and invoked the optimizing compiler on a subset of the frequently executed (or hot) methods. They also considered the observed pause times for an interactive user, rather than just the total time spent by the runtime compiler. SELF-93 identified hot methods using method invocation counts, which decayed over time. They considered using sampling to identify hot methods, but rejected the idea because of the coarse granularity of sampling. In [34], the SELF authors discuss many of the open problems in choosing a counter heuristic, but the project did not report any in-depth studies of the issues.

Detlefs and Agesen [35] studied mixed-mode execution in more detail by exploring the trade-off between an interpreter, a fast JIT compiler, and a slow "traditional" compiler adapted for use as a JIT. They found that a combination of the fast JIT and judicious use of the slow JIT on the longest running methods provided the best results on a collection of benchmarks. They used an oracle to determine the longest running methods. Some recent Java VMs have adopted a compilation-only strategy [36], [37], while others use an interpreter in a mixed-mode strategy [38]–[41].

SELF-93 also integrated recompilation decisions with inlining decisions. When an invocation count passed a threshold, the SELF system would traverse the call stack starting from the top method, using a heuristic to find a "base" method suitable for recompilation. The compiler would then inline the traversed call stack into the base method. The SELF team reported exploring a wide range of tuning values for the various recompilation policy heuristics. The HotSpot Server VM recompilation policy reported in [38] closely resembles the SELF-93 technique, including the inlining heuristic. The initial IBM mixed-mode interpreter system [39] relied on invocation counts to drive recompilation decisions, but used different inlining policies.

In addition to a counter-based selective optimization heuristic, the Intel Microprocessor Research Labs VM [36] implemented continuous compilation that uses a spare processor to periodically scan the profile data to determine which methods are hot and need to be recompiled.

All these counter-based policies rely on myriad heuristic tuning knobs. Hansen [11] reports on a deeply iterative ad hoc tuning process, to find reasonable settings for the variety of counter thresholds that drive recompilation. In current industry practice, VM vendors perform a similar process, laboriously tuning values to meet performance goals. Hansen lamented the situation in 1974:

> Determining optimization counts heuristically has its limitations, for we found it hard to change an optimization count so only a portion of the performance curve is affected. Therefore, if any appreciable progress is to be made, a more theoretical basis for determining them must be developed. [11], p.112.

In recent years, a few academic projects have begun exploring less ad hoc counter-based strategies, and more theoretically-grounded policies for selective optimization. Plezbert and Cytron [42] considered several online strategies for selective optimization. Although they did not draw a connection explicitly, their “Crossover” strategy is exactly the ski-rental problem [43], an online two-competitive algorithm that guarantees that the online cost will be at most twice the offline (or optimal) cost. Plezbert and Cytron presented a simulated study based on C compilation with a file-based granularity. They compared “JIT-only” and selective optimization approaches, as well as considering a background compilation thread that uses a spare processor to continuously compile. The study simulated a number of scenarios and calculated break-even points that indicate how long a program must run to make a particular recompilation heuristic profitable.

Kistler [44], [45] performed a similar analysis in a more realistic study using a virtual machine for Oberon. Kistler considered a more sophisticated online decision procedure for driving compilation, in which each compiler phase estimates its own speedup based on a rich set of profile data. Kistler performed an extensive study of break-even points based on this model, but did not implement the model-driven online algorithm in the virtual machine.

Jikes RVM [37] used call stack sampling to feed a model-driven optimization policy, relying on a cost-benefit model to select between multiple levels of optimization. Jikes RVM recompiles a method at a particular optimization level when it estimates that the benefit of additional optimization outweighs the cost of recompilation. To estimate these quantities, the system relies on models to predict the speedup in generated code due to optimization, the cost of recompilation, and the period of time that a method will execute in the future. The system estimates these quantities by relying on aggregates of offline profile data, and by estimating that at any point in time a method will execute for twice as long as it has to that point.

E. Deferred and partial compilation

Most current production VMs apply selective recompilation policies at method-level granularity. However, it has been recognized that methods may not provide the best delimiters for compilation units.

Hansen's Adaptive FORTRAN [11] selectively recompiled basic blocks and single-entry regions that contain loops. Adaptive FORTRAN associated a counter with each basic block, which triggered selective compilation of each code sequence according to empirically derived heuristics. Because Hansen's compiler did not perform inlining, the compiler applied selective optimization only to portions of an individual method, but could not selectively optimize across procedure boundaries.

SELF-91 [13] could optimize across method boundaries via inlining, and also could optimize partial methods with *deferred compilation*. SELF-91 would defer compilation of code that was predicted to execute infrequently, called *uncommon code*. This policy saved compilation time, as the compiler did not spend resources on uncommon code. Should the program branch to uncommon code, the system performed "uncommon branch extension", jumping to trampoline code that would generate the necessary code and execute it.

Uncommon branch extension presents a nontrivial engineering challenge, as a method's executable code must be rewritten while the method executes. The transition between compiled versions is called *on-stack replacement (OSR)*.

The HotSpot Server VM [38] adopted the SELF-91 techniques, inserting "uncommon traps" at program points predicted to be infrequent, such as those that require dynamic class loading. In addition, the HotSpot Server VM transfers from the interpreter to an optimized form for long-running loops, performing an OSR at a convenient point in the loop. This technique has also been adopted by other JVMs [46], [47], which is especially important for some microbenchmarks.

Others [46]–[48] have investigated the interplay between deferred compilation and inlining, and found some modest performance benefits.

F. Dynamic deoptimization

Some virtual machine services, such as some forms of introspection and debugging, require the virtual machine to interpret stack frames and program state generated by optimized code. When an adaptive optimization system produces code at various optimization levels, according to various conventions, it can complicate introspective virtual machine services.

For example, Smalltalk-80 exposed method activation state to the programmer as data objects. To deal with this, Deutsch and Schiffman [12] implemented a mechanism to recover the required v-code activation state from an n-code closure record. The optimizing translator identified distinguished points in the code sequence where the user may interrupt normal execution to examine activation state. At these points (and only at these points), the compiler recorded enough mapping information to recover the desired v-code state.

SELF pioneered the use of deoptimization to facilitate debugging [49]. The SELF compiler would mark distinguished safe points in optimized code, where the system would maintain enough state to recover the original, unoptimized state. At a debugging breakpoint, the system would dynamically deoptimize any optimized code, use OSR to transfer to the unoptimized version, and provide debugging functionality on the unoptimized representation. This technique allows nearly “full-speed” debugging, and has been adopted by today’s leading production Java virtual machines [50].

The same deoptimization technology can be used to implement speculative optimizations, which can be invalidated via OSR if needed. More discussion of this technique appears in Section VII-B.

V. Profiling for Feedback-directed optimization

Having a compiler as part of the runtime system allows the VM to apply fully automatic online feedback-directed optimization (FDO). Smith [51] provides an excellent discussion of the motivation and history of FDO. In particular, Smith highlights the following three factors as compelling motivation for FDO:

1) FDO can overcome limitations of static optimizer technology by exploiting dynamic information that cannot be inferred statically,
2) FDO enables the system to change and revert decisions when and if conditions change, and
3) Runtime binding allows more flexible and easy-to-change software systems.

A number of studies have reported program transformations that effectively use offline profile information to improve performance over static optimization technology (e.g. [52]–[56]). However, to implement fully automatic online FDO effectively, a virtual machine must also address the following challenges [57]:

1) Compensate for the overhead in collecting and processing profile information and performing associated runtime transformations, and
2) Account for only partial profile availability and changing conditions that affect profile data stability.

This section describes the most significant profiling technology for driving FDO, and its use in today’s virtual machines today.

A. Profiling techniques

A key technical challenge for effective online FDO is to collect accurate profile data with low overhead.

Although selective optimization systems monitor the running program to identify candidates for runtime optimization (as previously discussed in Section IV-C), feedback-directed optimizations often require more fine-grained profiling information. For example, many FDO techniques require profile data regarding individual program statements, individual objects, or individual control flow paths. Collecting fine-grained profile data with low overhead presents a major challenge, making many forms of FDO difficult to perform effectively online.

To address this challenge, VM implementors have investigated a variety of low-overhead fine-grained profiling techniques. In recent years, several sophisticated and effective techniques have emerged.

We organize the most common mechanisms used by virtual machines to gather profiling information for FDO into four categories: *runtime service monitoring*, *hardware performance monitors*, *sampling*, and *program instrumentation*. We next review examples of each category, as well as approaches that combine several techniques.

1) Runtime service monitoring: With this technique, the system monitors state associated with various virtual machine runtime services. In some cases, the program's use of a runtime service exhibits temporal locality that the system can exploit for optimization. Section VII-A discusses a variety of optimizations for dynamic dispatch, which monitor runtime data structures that record past dispatch behavior. Section VII-B discusses optimizations that monitor other runtime state, relating to hashcodes and synchronization.

The memory management system provides a particularly rich body of information that can drive FDO. The memory manager can observe trends in allocation, garbage collection, and heap utilization, in great detail. Section VII-C describes optimizations to exploit this information.

2) Hardware performance monitors: Many microprocessors provide specialized hardware that can provide online profile information regarding processor-level events. Despite the fact that mainstream processors provide a rich variety of hardware performance monitors, few VMs have exploited this approach for driving FDO. We are aware of only one published report [58], which describes the ORP VM's use of hardware performance monitors to guide memory prefetch injection. Their work takes advantages of the hardware support for sampling cache misses on the Itanium(R) 2 platform. DCPI [59] provides a sampling system that uses interrupts generated by the hardware performance counters on the ALPHA processor to identify the frequently executed portions of a program. However, to our knowledge no virtual machine has employed this technique.

We speculate that VMs have not generally exploited hardware performance monitors due to the complexity of architecture-specific counter infrastructures, and the difficulty in mapping low-level counter data to high-level program constructs. It remains to be seen if VMs will develop techniques to overcome these difficulties and more effectively exploit hardware performance monitors.

3) Sampling: With sampling, the system collects a representative subset of a class of events. By observing only a limited percentage of the events, sampling allows the system to limit profiling overhead. Sampling alone can provide sufficient profile data to drive some VM services. As discussed in Section IV-C, some virtual machines sample executing methods to derive an execution time profile to drive selective optimization. For many feedback-directed optimizations, VMs additionally sample the program's call-stack. Call-stack sampling can provide context-sensitive call graph profiles for guiding feedback-directed inlining [34], [60]. SELF-93 [34] used a count-down scheme to determine when to sample the call-stack, while Jikes RVM [37] used a timer-based approach.

Many feedback-directed optimizations rely on fine-grained profile information, such as basic block frequencies or value profiles, which can be difficult to collect efficiently with a purely sample-based approach. A few systems have collected such profiles with the help of hardware support [59], [61], [62].

4) Program instrumentation: By inserting intrusive instrumentation in a running program, a virtual machine can collect a wide range of profile data at a fine granularity.

Many studies report using *offline* profiles collected via instrumentation to guide feedback-directed optimization [52]–[56]. However, many types of instrumentation can impose intolerable runtime overhead; slowdowns ranging from 30%–1,000% above non-instrumented code are not uncommon [63]–[65], and overheads in the range of 10,000% (100 times slower) have been reported [66]. Therefore VMs must apply techniques to reduce these overheads to be able to apply instrumentation online with acceptable performance.

The primary mechanism to reduce instrumentation overhead is to limit the time during which instrumented code executes. For example, the VM can instrument unoptimized code (or interpreted code) only, allowing the instrumentation to terminate automatically when the VM's selective optimization mechanism recompiles a hot method. Several contemporary VMs apply this technique [38], [39]. This approach has a number of advantages: 1) the instrumentation likely imposes minimal overhead over and above the already poor performance of unoptimized code, 2) the optimizer has profile data available when it first recompiles a method, enabling early application of FDO, and 3) implementing this approach requires relatively low engineering effort.

However, despite these advantages, instrumenting unoptimized code has two significant limitations. First, because the system profiles methods only during their early stages of execution, the profile may not reflect the dominant behavior if the behavior changes after the early stages. Second, certain profiles for guiding FDO are more difficult to collect in unoptimized code. For example, optimizations such as aggressive inlining drastically change the structure of a method. Determining hot paths through inlined code can be nontrivial when using a profile obtained prior to inlining.

To avoid profiling start-up behavior, Whaley [48] proposed a three-stage model in which fine-grained profiling is inserted in the second stage. A more general solution adopted by several systems [39], [44], [57] inserts instrumentation into fully optimized code. This solution avoids the aforementioned drawbacks of instrumenting unoptimized code, but introduces challenges of its own. The fact that a method is selected for optimization suggests that the method may be executed frequently; thus, naive instrumentation could result in severe performance degradation.

Kistler [44] inserted instrumentation in optimized code without any special mechanism to enforce a short profiling interval. Although this approach can be effective for some applications, it could degrade the performance of others to unacceptable levels, making this approach too risky for use in a production-level VM. The IBM DK 1.3.0 [39] instrumented optimized code, but enforced a short profiling period by using code patching to dynamically remove instrumentation after it has executed a fixed number of times. Recent improvements [67] use a hierarchical structure for enabling and disabling the counters to collect more accurate profiles with fewer samples. However, profiling in shorter bursts increases the probability that the observed behavior does not accurately reflect the overall behavior.

5) Combining instrumentation and sampling: Some work has combined instrumentation with periodic sampling to observe program behavior over a longer window of execution. The code-patching technique described above can be repeated, enabling and disabling instrumentation to collect data in sets of short bursts. However, the overhead of maintaining cache consistency when patching can limit the sample frequency achievable with this approach. Arnold and Ryder [68] describe a technique for sampling instrumentation that allows the system to enable and disable instrumentation at a finer granularity. Their technique introduces a second version of code within each instrumented method, and lightweight checks determine when the instrumented version should be executed. This technique has been used online in a virtual machine to collect profiling information for FDO [57]. Chilimbi and Hirzel [69], [70] modified this technique in an online-binary optimizer to guide prefetching optimizations.

B. Stability and phases

As an online system, a VM has the ability to consider the stability of a profile across phases of a single execution, and may attempt to react appropriately when the program enters a new phase. Kistler [44] pioneered this approach in an interactive Oberon VM by periodically capturing a vector composed of the number of occurrences of an event, such as a basic block counter. The system computed a similarity metric between two recent vectors. When this value exceeded a threshold, the system initiated a new profile/optimization stage.

Aside from Kistler, little published work describes online phase detection in a VM. One contributing factor is that certain types of adaptive optimization do not require explicit phase shift detection, as the profiling scheme will automatically discover changes in behavior. For example, most implementations of selective optimization do not require explicit phase shift detection. When the application's behavior changes so that new methods become "hot", the selective optimization systems detect the new "hot" methods and guide optimization appropriately. Most current VMs do not take any action when a previously optimized method becomes "cold", although discarding generated code would be important for a system with a limited code cache.

In theory, many optimizations, such as inlining, code layout, and specialization, might benefit from reconsideration when the running program enters a new phase. Additionally, an optimization decision that is profitable in one phase might degrade performance when a subsequent phase exhibits different characteristic behavior. It remains open whether adaptive optimization systems will be able to exploit phase detection to tailor optimizations effectively and efficiently online.

A number of offline studies have examined issues relating to profile stability across different inputs for basic blocks [71], receiver types [65], and procedure calls, indirect control transfers, and nonaligned memory references [72]. Wall [71] found the level of stability of basic block profiles depended heavily on the event being profiled. Grove et al. [65] found that receiver type profiles produced stable results. Wang and Rubin [72] found that different users of interactive programs have different usage patterns, and observed up to 9% performance degradation when profiles from a different user are used to drive profile-directed optimizations.

The Dynamo [31] binary translator addressed similar issues, by monitoring the creation rate of optimized code fragments. When the system detected an increase in optimization activity, such as is likely to occur when a different collection of instructions begins executing, it would flush the code cache. Chilimbi and Hirzel's online optimization system [70] accounted for potential phase shifts by periodically re-gathering profile data based on a fixed duration.

The computer architecture community has also found the problem of profile stability of interest in both offline (after program execution) [73], [74] and online [75]–[78] contexts. Some studies have employed phase detection to identify a phase of a profile that is representative of the complete program's behavior to reduce simulation time [73], [74]. Online phase detection has been used to dynamically adapt multi-configuration hardware to program behavior [75], to tailor code sequences to frequently executed phases [76], and to track and predict phases [77].

VI. Feedback-directed code generation

In this section, we review the use of feedback-directed code generation in virtual machines. This section concentrates on feedback-directed techniques to improve the quality of the code generated by an optimizing compiler. Section VII discusses other forms of FDO used by virtual machines. The remainder of this section describes feedback-directed variants of inlining, code layout, instruction scheduling, multiversioning, and miscellaneous other forms of feedback-directed code generation.

A. Inlining

Inlining, the replacement of a call site with the contents of the method it calls, has proved to be one of the most important compiler optimizations, especially for object-oriented programs. However, overly aggressive inlining can impose steep costs in both compile-time and code size.

Many studies have examined the cost-benefit trade-offs of profile-directed inlining using offline profile data. Much of this work has considered inlining as a KNAPSACK [79] problem, measuring the inlining benefit that can be obtained with various inlining budgets [80]–[84] using a greedy algorithm to choose inlining candidates. All these studies have concluded that profile-driven greedy algorithms can judiciously guide inlining decisions that cover the vast majority of dynamic calls, with reasonable compile-time overhead.

To our knowledge, no virtual machines have adopted greedy inlining algorithms based on the KNAPSACK formulation. A central obstacle is that virtual machines tend to consider recompilation decisions on a local, method-by-method basis, whereas the KNAPSACK inlining studies rely on a comprehensive, global view of the entire program. Instead, virtual machines have relied on a variety of ad hoc heuristics to drive profile-directed inlining.

The SELF implementations introduced a number of techniques for more effective inlining. Compared to most mainstream languages, SELF placed an even greater premium on effective inlining, to deal with extremely frequent method calls. As reviewed in Section VI-D, SELF-89, SELF-90, and SELF-91 introduced progressively sophisticated optimizations to predict types, and inline and split code based on static type estimates. SELF-93 [34] augmented these techniques with *type feedback*, where the VM would provide the runtime compiler with a profile of receiver types collected from the current run. The SELF compiler used this information to choose inlining candidates and to guide the transformations to deal with inlined dynamic dispatch. The reported results show significant speedup (1.7x improvement) from using type feedback, and show that the profile-directed approach results in significantly better code than a more sophisticated optimizer (SELF-91) that relied on static type estimates.

As reviewed in Section IV, the SELF-93 adaptive optimization system incorporated inlining decisions into the recompilation policy, walking the call stack to find a suitable root method to recompile with inlining. The HotSpot JVM [38] adopted the SELF-93 technique of driving recompilation policies based on inlining decisions. It performs guarded inlining when class hierarchy analysis or the profile indicated a single receiver type.

Dean and Chambers [85] presented Inlining Trials, an approach to more systematically drive inlining decisions. In this work, the SELF compiler would tentatively inline a call site, and monitor compiler transformations to quantify the resultant effect on optimizations. The virtual machine maintained a history of inlining decisions and resultant effects, and would drive future inlining decisions based on the history. This approach could help guide more intelligent inlining decisions because the effect of inlining on optimizations is difficult to predict. Waddell and Dybvig [86] report a similar approach for a dynamic Scheme compiler. We are not aware of any production virtual machines that have adopted this methodology.

Several studies [36], [37], [57], [87], [88] report on fully automatic online profile-directed inlining for Java that improves performance by factors of approximately 10-17%, as compared to comparable strategies that ignore profile data.

Jikes RVM [37] incorporated inlining into the cost-benefit model for selective recompilation by increasing the expected benefit of recompiling a method that contains a hot call site. Arnold et al. [57] augment this scheme by profiling hot methods to determine hot basic blocks within those methods. Inlining budgets for call sites in such hot blocks are increased.

Suganuma et al. [87] explored online profile-directed inlining heuristics, relying on an approximation of the dynamic call graph collected by instrumenting hot target methods for short time periods. They concluded that for non-tiny methods, heuristics based solely on profile data outperformed strategies that also rely on static heuristics.

The StarJIT compiler [88] uses call site frequency to augment inlining heuristics and improve guarded devirtualization decisions. Hazelwood and Grove [60] explored more advanced inlining heuristics that consider both static and dynamic characteristics of the call stack.

B. Code layout

Code layout, or code positioning, is one of the most frequently implemented forms of feedback-directed optimization. For this transformation, the compiler rearranges code to maximize instruction locality and improve branch prediction; it attempts to lay out frequent code paths contiguously in the address space.

Pettis and Hansen [52] detailed the most popular approach to profile-directed code positioning, and used an offline study to show that significant performance improvements are possible.

Arnold et al. [57] employ a variant of the top-down code positioning in an online manner to obtain modest improvements. A similar online variant has been reported for the IBM DK for Java [89]. Adl-Tabatabai et al. [88] also employ a variant of the top-down code positioning algorithm, extended to perform tail duplication of the block being laid out. They also use profiles to drive method splitting (partitioning compiled code into a hot and cold section) and alignment of branch targets.

C. Instruction Scheduling

In a related vein, researchers have documented significant improvements with feedback-directed instruction scheduling. Instruction scheduling seeks to order instructions to maximize flow of instructions through a microprocessor pipeline. The IMPACT compiler project conducted a number of influential studies of offline profile-directed instruction scheduling [90], [91].

Naturally, the efficacy of instruction scheduling depends highly on the underlying instruction architecture and implementation. Out-of-order superscalar processors with large reorder buffers mitigate the need for instruction scheduling. However, statically scheduled processors such as Itanium [92], [93] place a premium on effective instruction scheduling.

Adl-Tabatabai et al. [88] perform online profile-directed instruction scheduling in the StarJIT, using edge information obtained from either instrumentation or derived from sampling Itanium’s performance monitors. The StarJIT combines instruction scheduling with trace formation and tail duplication to address phase ordering issues among these techniques.

D. Multiversioning

Multiversioning is an adaptive technique in which the compiler generates multiple implementations of a code sequence, and emits code to choose the best implementation at runtime.

In “static multiversioning”, the compiler generates the various versions based purely on information known at compile time. For example, Byler et al. [94] describe a compiler that generated several versions of Fortran loops, and at runtime, chooses the best implementation for runtime values of loop bounds, increment values, and access patterns.

Similar techniques have been applied for Java programs [95], [96]. For example, Artigas et al. [95] describe static multiversioning to create “safe regions” in Java code that are free of exception dependencies and certain aliases. In these safe regions, the compiler can apply more aggressive optimizations.

Many virtual machines use a form of multiversioning to speculatively inline targets of dynamic dispatch. The simplest mechanism is to introduce a “diamond” by testing a runtime condition before the inlined method body, and branch to an out-of-line call should the runtime condition fail. Arnold and Ryder proposed a mechanism called *thin guards*, whereby a potentially expensive set of runtime checks can be replaced by a small set of boolean checks [97].

To avoid the entire runtime cost of a conditional check, a VM may instead speculatively insert a no-op instruction, or *patch point*, where it would otherwise insert a runtime check [36], [98]. On invalidation, the VM can replace the patch point with a code sequence equivalent to the aforementioned runtime check.

A more aggressive implementation can “break the diamond” using deferred compilation as discussed in Section IV. This technique was pioneered in SELF-91 [13] and later adopted by HotSpot [38] and Jikes RVM [47]. A failed inline guard jumps to an OSR point, which contains no code and models a method exit. If control flow reaches this point, the system dynamically generates code to handle the failed guard. Because the infrequent path models a method exit, forward dataflow facts from this path do not merge back into the frequent path, resulting in improved dataflow properties in the code following the guard. Fink and Qian [47] evaluated the impact of breaking diamonds in Jikes RVM and did not find significant value from improved dataflow from guarded inlining.

The SELF implementations pioneered more sophisticated static multiversioning in techniques they call *message-splitting*. The SELF compiler would replicate control flow downstream of a merge point to preserve unmerged type information. With this technique, the compiler can optimize downstream by exploiting more specific type information. The SELF project documented progressively sophisticated forms of downstream splitting; starting with local splitting immediately after a merge [99], then extended splitting to handle any type information lost by merges [32], and then splitting for loops and path-based splitting [13]. SELF-93 [34] enhanced the efficacy of splitting by incorporating *type feedback*, an online profile of observed receiver types.

One problem with static multiversioning is that due to space overhead and compile-time costs, it is not always possible to generate every possible variant of a code sequence. However, a runtime compiler can use profile information to select a few of the many possible versions. SELF-89 [99] and later implementations relied heavily on this form of dynamic multiversioning, with a technique called *customization*. The SELF system would generate a new version of each method customized to the type of the receiver. With this technique, the compiler can resolve each call to the `self` object at compile-time, bypassing an expensive dynamic dispatch and allowing more effective inlining.

Some work has focused on *specialization*, or dynamic multiversioning based on speculative runtime constants or properties. Most of the research has relied on programmer annotations or directives to guide multiversioning policies(e.g. [100]–[102]). Mock et al. [56] extended this work with automated techniques to derive the appropriate directives. To our knowledge, only one product VM has documented fully automatic profile-directed specialization. Suganuma et al. [39] describe a sophisticated automatic approach to exploit runtime constant primitive values, types, array lengths, type relationships, aliasing relationships, and thread-local properties.

A few other systems have performed simpler forms of dynamic multiversioning based on runtime profiles. Arnold et al. [57] used edge profiling to split control flow graph merge nodes. Similarly, Adl-Tabatabai et al. [88] perform hot path splitting via tail duplication for IA32 and Itanium architectures.

In most multiversioning, the generated code uses some absolute criteria, based on runtime values, to determine which implementation to execute. An alternative approach, *empirical optimization*, has the system measure performance of the various implementations during a “training” period, and then choose the best implementation for a “production” period. Diniz and Rinard [103] describe an instantiation of this technique called *dynamic feedback*, a fully automatic compiler-supported system that selected among several possible synchronization optimizations at runtime.

Voss and Eigenmann describe a more advanced empirical adaptive multiversioning system called ADAPT [104]. In ADAPT, the user describes possible optimizations in a domain-specific language. The ADAPT compiler generates an application-specific runtime system that searches for the best optimization parameters at runtime. At any given time, the runtime system maintains the best known version, and performs experiments to evaluate an experimental version generated with different optimization parameters. The ADAPT system has a runtime hot-spot detector, and focuses its effort on important loops identified by profile data. The ADAPT system as implemented is a loosely coupled coarse-grain system that invokes a full-fledged static Fortran compiler as its runtime optimizer. It is targeted for compiler writers as a tool for prototyping variants of adaptive optimizations. It would be interesting to evaluate this approach in a full-fledged virtual machine, as a technique to deal with the potential non-intuitive effects of adaptive optimization.

E. Others

Today’s production VMs collect a potpourri of profile data during the course of execution, and use of the profile data has tended to seep into many aspects of the runtime compiler. Many times, these FDO are fairly straightforward applications, and each individual optimization may have limited impact. As a result, there have been few comprehensive studies of miscellaneous profile-directed optimizations in

VMs.

In addition to the optimizations discussed in other sections (inlining, code reordering, splitting) Arnold et al. [57] also evaluate the impact of using edge profiles to improve loop unrolling heuristics in Jikes RVM. Additionally, the Jikes RVM optimizing compiler uses profile information to drive register allocation spill heuristics and live range splitting. The runtime improvements from the latter two optimizations were small, and have not been formally documented.

The HotSpot Server compiler [38] reports similar techniques. The HotSpot interpreter collects counts of method entries and backward branches, type profile at call sites, is-null information, and branch frequencies. The optimizer exploits this information for various optimizations including inlining heuristics and global code motion.

In recent years, researchers from IBM's Tokyo Research Lab have documented a number of FDO applications in the IBM DK for Java. Suganuma et al. [89] report that the JIT uses runtime trace information to guide code generation and register allocation, in addition to the code layout reported earlier. Ogasawara et al. [105] describe a feedback-directed optimization to accelerate exception handling, which required profiling to determine call tree paths frequently traversed by exceptional control flow. Suganuma et al. [46] describe a speculative stack allocation optimization, relying on escape analysis and multiversioning with on-stack replacement to invalidate and lazily instantiate objects.

Researchers at Intel's Microprocessor Research Labs have also explored dynamic optimization techniques to accelerate exception handling and handle speculative optimizations in the presence of exceptions [36].

Several researchers have examined software prefetching based on offline profile data to discover access patterns (e.g. [106], [107]). Recently, a few papers have reported results on automatic online profile-directed prefetching. Inagaki et al. [108] developed a sophisticated lightweight profiling mechanism to discover stride access patterns both within and across loop iterations. Adl-Tabatabai et al. [58] use hardware performance monitors to help guide the placement of prefetch instructures to improve access to linked data structures. Chilimbi and Hirzel [70] describe an automatic online DER rewrite tool that profiles the application to find frequently recurring data reference sequences, called "hot data streams". Having identified a hot data stream, the system automatically rewrites the binary to detect prefixes of hot data streams, and insert software prefetch instructions to exploit the stream's reference pattern.

VII. Other feedback-directed optimizations

In this section, we review feedback-directed techniques other than code generation that have been used in virtual machines. The first subsections explore two broad themes in adaptive runtime systems: exploiting temporal locality and the speculative tailoring of runtime services to other aspects of the application's dynamic behavior. The final subsections discuss two facets of memory management: optimizations to improve garbage collection and to improve the program's spatial locality.

A. Temporal Locality and Caching

A number of runtime services can be quite expensive in the worst case, but can be significantly accelerated in the average case by applying some form of caching, assuming that the application exhibits some exploitable form of temporal locality.

For example, the key characteristic of object-oriented programming is that an object's behavior depends on its *runtime* type, as opposed to a declared static type. Since program behavior depends so strongly on runtime types, object-oriented systems pioneered adaptive techniques to improve the performance of dynamic dispatch and type testing. Many of these techniques rely on temporal locality to be effective.

Early Smalltalk-80 systems used dynamic caching [109] to avoid performing a full method lookup on every message send. The runtime system began method lookup by first consulting a global hash table that cached the results of recent method lookups. Although consulting the hash table was significantly cheaper than a full method lookup, it was still relatively expensive.

Therefore, later Smalltalk systems added inline caches [12] as a mechanism to mostly avoid consulting the global cache. In an inline cache, the system overwrites the call to the method lookup routine with a direct call to the method most recently called from the call site. The system modifies the callee method's prologue to check that the receiver's type matches, and calls the method lookup routine when the check fails. Inline caches perform extremely well if the call site is monomorphic (has one target), or at least exhibits good temporal locality, but perform poorly if a call site dispatches to multiple target methods in quick succession.

Polymorphic inline caches (PICs) [110] were developed to overcome this weakness of inline caches. In a polymorphic inline cache, the call site invokes a dynamically generated PIC stub that executes a sequence of tests to see if the receiver object matches previously seen cases. If a match is found, then the stub invokes the correct target method; if a match is not found, the PIC terminates with a call to the method lookup routine (which may in turn choose to generate a new PIC stub for the call site, extended to handle the new receiver object). Some implementations of PICs use a move-to-front [111] heuristic to further exploit temporal locality.

Similar issues arise in dispatching interface method calls in the Java programming language. Some JVMs use PICs to dispatch interface methods. Another commonly used technique is the *itable*, which is a virtual method table for a class, restricted to those methods that match a particular interface of the class implementation [112], [113]. To dispatch an interface method, the system must locate the appropriate itable for a class/interface pair and then load the target method from a known offset in this itable. In general, the runtime system must search for the relevant itable at dispatch-time. In a straightforward implementation, search time increases with the number of interfaces implemented by the class. However, as with PICs, most systems augment the basic mechanism with an itable cache or move-to-front heuristic to exploit temporal locality.

Many object-oriented languages have other language constructs, such as Java's `instanceof`, that require the runtime system to test the runtime type of an object. A number of non-adaptive schemes for answering these questions efficiently have been explored; a discussion of many of these techniques can be found in Krall et al. [114]. More recent work has specialized these schemes for the particular semantics of Java [115], [116]. Some virtual machines have also used caching to exploit temporal locality in type testing. For example, the IBM DK for Java caches the result of the most recent type inclusion test in the class object [89].

Finally, temporal locality has also been exploited to reduce the cost of Java synchronization. Kawachiya et al. [117] observe that it is common for an object to be locked repeatedly by the same Java thread. They exploit this *thread locality* of locking by allowing the lock on an object to be reserved for a single thread. This thread can obtain the lock cheaply, while all other threads that attempt to acquire the lock must first cause the reservation to be canceled (an expensive operation).

B. Speculative Optimizations for Runtime Services

In Section VI-D, we reviewed techniques whereby a compiler can speculatively emit code, and recover should a speculative invariant fail. Similarly, the runtime system can apply speculative techniques to optimize runtime data structures.

The most pervasive data structures in object-oriented languages, objects, depend on the VM's *object model*. An object model dictates how the VM represents objects in storage; the best object model will maximize efficiency of frequent language operations while minimizing storage overhead. In addition to the programmer-specified data elements in the object, the virtual machine adds additional state in an *object header* to support operations such as virtual dispatch, garbage collection, synchronization, and hashing.

It has been observed that the usage of the different portions of the object header varies from one object instance to another. Therefore, some virtual machines use adaptive object models that elide some portions of the object header until it has been determined that a particular object instance will actually need them.

For example, it has been observed that most Java objects never need their default hash code. Thus, rather than allocating space for the hash code in all the headers of all object instances, many virtual machines use the tri-state encoding technique from Bacon et al. [118] (also developed independently by Agesen and used in the Sun EVM), where the states of an object are *unhashed*, *hashed*, and *hashed-and-moved*. For the first two states, the hash code of the object is its address. When the garbage collector moves an object whose state is *hashed*, it changes its state to *hashed-and-moved* and copies the old address to the end of the new version of the object. In this scheme most objects never have space allocated in their header for a hash code, but if necessary the virtual machine will adapt the object model on a per-object basis to accommodate the hash code.

Similarly, most Java objects are never synchronized and therefore do not need the *lockword* portion of their object header. One heuristic for deciding which object instances are likely to use their lockword is to predict that an object of a class C is likely to be locked if and only if C has at least one synchronized method, or if any of its methods contain synchronized(this) statements. This heuristic was used by Bacon et al. [119] to define a family of object models with one-word headers; they also proposed adaptively adding the lockword back into object instances that were mispredicted by this heuristic during a copying garbage collection.

Speculative optimizations require runtime support to invalidate the optimization when conditions change. For example, despite the possibility of dynamic class loading, most JVMs apply some simple whole program analyses by speculating that the set of currently loaded classes *are* the entire program (i.e. that dynamic class loading will not occur in the future) and optimizing accordingly. Virtually all production virtual machines speculatively apply class hierarchy-based inlining [120]. This optimization assumes that the class hierarchy is complete; a number of techniques including preexistence [121], code patching [98] and on-stack replacement [49] are used to recover when dynamic class loading changes the class hierarchy and invalidates a speculative optimization.

C. Heap Management and Garbage Collection

Automatic memory management, commonly referred to as garbage collection (GC), is one of the more complex services provided by a VM. Most garbage collectors are inherently somewhat adaptive in that the rate of garbage collection and the amount of work done in each collection cycle depends heavily on the application's runtime behavior. Jones and Lins [122] describe the rich diversity of GC algorithms that have been developed. Virtually all of these algorithms have been deployed in some VM; there is no generally accepted "best" GC algorithm. In this section we discuss three areas in which VMs apply more interesting forms of adaptation: online choice of GC algorithms, heap size management, and the scheduling of garbage collection.

One interesting approach is to adaptively switch garbage collection algorithms to adjust to the application's dynamic behavior. Printezis [123] reports on dynamically switching between Mark&Sweep and Mark&Compact algorithms to manage the mature space of a generational collector. Soman et al. [124] describe a more radical approach in which an extended version of Jikes RVM can dynamically swap between radically different garbage collection algorithms to adapt to the application's behavior. To our knowledge, this level of adaptive GC has not been deployed in production-level VMs.

Most production VMs dynamically adjust the size of their heap, the portion of the VM's address space that is used to support the application's dynamic memory allocation requests. The heuristics used are rarely fully described in the literature. Dimpsey et al. [125] contains a detailed description of a policy used in some versions of the IBM DK that adjusts the VM's heap size based on heap utilization and the fraction of time spent in GC.

Generational GC algorithms divide the heap into multiple regions. Objects are initially allocated into a *nursery* space and are promoted (or tenured) into a *mature* space after they survive a certain number of collections. Some systems use a fixed size nursery. Appel [126] describes a system in which the fraction of the total heap utilized for the nursery is dynamically adjusted based on the application's behavior.

VMs can also schedule GC with adaptive policies. Normally, a VM triggers GC when some memory resource is exhausted (or almost exhausted); the exact details vary from algorithm to algorithm. Some systems have explored heuristics for triggering a GC before memory is actually exhausted in the hopes of increasing the efficiency of GC (reclaiming more free memory for each unit of GC work). Hayes [127] observed that there are often key objects whose unreachability indicates with high probability that large data structures have also just become unreachable, and thus indicates an attractive time to schedule GC. Hirzel et al. [128] build on this idea by suggesting the scheduling of GC based on a connectivity analysis. Less sophisticated heuristics have been proposed that schedule garbage collections based on stack height [129] or the number of pops from the stack [130].

D. Heap Optimizations for Spatial Locality

Although automatic memory management imposes overheads as discussed in the previous section, it also offers opportunities for the VM to improve performance. When a safe language prevents user code from directly reading or writing pointers, it gives the VM freedom to rearrange the heap layout to improve spatial locality.

Some researchers have explored *field reordering*, a technique to rearrange layout of fields in an object to improve spatial locality [45], [131]–[133]. This technique attempts to lay out fields in an object such that concurrently accessed hot fields fall on the same cache line. Additionally, Kistler and Franz [45] show that the order of fields within a cache line can impact performance because of the order in which the hardware fills individual bytes in a cache line. They compute field layout with a model that takes this factor into account.

Chilimbi et al. [131] proposed *object splitting*, which divides an object's hot and cold fields into separate sub-objects using an offline profile. The system accesses the cold sub-object indirectly from a pointer in the hot sub-object. By bringing only hot fields into cache, object splitting improves spatial locality.

While Chilimbi's work addressed objects, Rabbah and Palem [134] proposed a form of object splitting for arrays that also uses offline profile information. This work splits an array of objects of a particular type into multiple arrays, each containing a subset of the type's fields. Rabbah and Palem analyzed an object reference trace along an application's hot spots to determine a splitting policy that matched the data access patterns.

A third technique, *object co-location*, places concurrently accessed objects on the same cache line. Two online approaches have been reported. Chilimbi et al. [135] used an object affinity graph for dynamic object co-location of Cecil programs. A generational garbage collector interpreted the graph to determine how to co-locate objects when copying objects into a semispace. Kistler and Franz [45], [132] used a temporal relationship graph for dynamic object co-location in Oberon. Their technique detects when it is advantageous to change the layout of a particular data structure, recompiles all affected code in the background, and then atomically updates the code and the data structure's layout. Although these papers reported significant speedups from object co-location, the high runtime cost of maintaining an object affinity graph remains a problem that has impeded the adoption of these techniques in production VMs.

Shuf et al. [136] proposed a simpler, type-affinity based co-allocation alternative. Using offline profile data, Shuf et al. identified a small set of *prolific* types, types that are allocated frequently, but are short lived. Given the set of prolific types, the system was enhanced with a method to allocate together a cluster of prolific type objects that reference each other.

A few other papers have explored other profile-directed techniques to improve locality, based on profile data collected offline (e.g. [64], [137]). We are not aware of any online object co-location, object splitting, or field reordering results in production VMs at this time.

VIII. Discussion

Having reviewed the evolution of adaptive optimization technology to the present, we now discuss a few areas that appear ripe for further research.

A. *Optimization Control Policies*

As reviewed, many of the current techniques for controlling optimization rely on ad hoc policies demanding extensive tuning. Although many would agree that a more theoretically grounded approach would be valuable, there are significant technical challenges on this path. Most dauntingly, we have no satisfactory methods to automatically predict the future impact of a particular transformation.

One promising approach to this problem centers on “empirical optimization policies”, as described in Section VI-D. A few papers reviewed here have begun applying this approach to adaptive optimization; notably the ADAPT [104] system and the Inlining Trials [85] work.

In the domain of high-performance numerical libraries, empirical optimization has succeeded in a number of high-profile projects [138]–[140]. On the other hand, at least one paper has reported that model-driven optimization can compete with the empirical approach for simple numerical kernels [141]. The trade-offs of empirical vs. model-driven optimization for the applications described in this paper are not well understood.

B. *Deep Analysis*

None of the optimizations reviewed in this paper involve substantial interprocedural analysis (IPA). Since IPA would be much more expensive than method-granularity analysis, it remains an open question on how to design policies so that runtime IPA justifies the effort. A few recent papers have examined runtime IPA optimizations [142]–[146]; however, to our knowledge these techniques have not yet appeared in production VMs.

Another approach is to stage analysis effort, performing the bulk of the work offline and some cheap analysis at runtime. The QuickSilver project [147] investigated a *quasi-static* compiler that would precompile some code offline, and could integrate the code into a full-fledged VM at runtime. Philipose et al. [148] describe a scheme to automatically generate compilers with staged analysis, which could be applied in “quasi-static” scenarios. Although the potential for dynamic class loading hampers many potential transformations that rely on interprocedural analysis; language features such as Java’s sealed packages can mitigate the impact of dynamic class loading [149]. Venugopal et al. [29] propose an even more drastic approach: specializing the entire VM with respect to a fixed application. Although this approach sacrifices dynamic class loading, it may suit many embedded devices.

In general, deep analysis such as IPA can attempt to prove invariants, but with profile information, the system can instead speculate that invariants hold without generating a proof. A VM can apply many transformations speculatively, relying on invalidation mechanisms such as OSR and heap rewriting mechanisms. It is not clear whether the dominant trend will be deep analysis to prove invariants, aggressive speculation with advanced invalidation, or some combination of the two.

C. RAS

Runtime compilation, especially driven by sampling, adds another level of non-determinism to the running program. So, adaptive optimization makes testing more difficult, as it provides more opportunities for non-deterministic bugs to sneak past testing. Additionally, a sophisticated runtime optimizer adds significant complexity to the VM.

Most work has ignored the RAS (reliability, availability, serviceability) implications of adaptive optimization. A few academic papers have started to address the subject. The DejaVu project [150] investigated enhancing a VM with deterministic replay, to aid debugging. Bruening [151] describes a similar deterministic replay facility in the Rivet virtual machine.

The RAS implications of adaptive optimization technology, if left unaddressed, threaten to slow the adoption of new technologies.

D. Compiler Engineering

Some research work on dynamic code generation has focused on extremely cheap code generation. For example, Lee and Leone [101] report on a system that consumes 6 cycles per instruction generated, and Engler [152] reports 6-10 cycles per instruction generated for a VCODE optimizer. On the other hand, current VMs have evolved to the other side of the compiler cost spectrum, by providing full-fledged traditional runtime compilers that spend many thousands of cycles per instruction generated. This fact has driven the work on selective optimization discussed in Section IV. However, it remains to be seen if the pendulum will swing back toward extremely cheap techniques. For example, VMs could evolve toward ubiquitous speculative specialization with frequent invalidation.

Alternatively, we have seen incremental improvements to compile-time. For example, some VMs have adopted cheap, near-linear register allocation techniques in place of graph coloring [34], [88], [89], [153]–[155]. Chen and Olukotun [156] describe a JIT engineered with only four major passes, designed to execute significantly faster and with smaller footprint than current production JITs. Ishizaki et al. [157] evaluate the optimizations in the JIT for the IBM DK for Java, and report that the cheapest compiler optimizations provide most of the speedup benefits.

E. Architectural Impact

It remains open whether adaptive optimization will drive requirements for microprocessor architecture and design. An obvious issue is that adaptive optimization entails self-modifying code. Deutsch and Schiffman [12] noted that self-modifying code was “generally condemned in modern practice”. In current VMs, dynamic code generation is a relatively rare event compared to normal computation, and the cost of memory barriers for self-modifying code does not appear to be a significant problem.

Eeckhout et al. [158] describe an interesting study of the architectural impact of virtual machines, and a good review of related work on Java workload characterization.

A related issue concerns binary translation. It is not clear whether an underlying binary optimizer, such as Dynamo [31], can further improve performance for a VM that already performs adaptive optimization. Nevertheless, dynamic code generation can certainly help with other problems, such as providing binary portability [4] and solving the legacy problem for VLIW implementations [88]. Additionally, some recent processors such as ones from Transmeta [159] include low-level binary translators that are closely tied to the hardware and target instruction set architecture(ISA). One open question concerns whether cooperation between software virtual machines and ISA-level binary translators would deliver additional benefits.

F. New VM Domains

This paper has focused on mainstream VMs targeting fairly low-level program representations. However, current industry trends point to a proliferation of higher-level runtime systems that provide much higher level services, such as J2EE, ASP.NET, Web Services, and BPEL. These runtime systems currently provide some forms of adaptive optimization in management of runtime structures such as thread pools and communication queues. It is not yet clear whether these higher-level VMs could benefit from new or specialized forms of dynamic code generation and optimization, beyond that provided by the underlying VMs. Anecdotal evidence suggest that more abstract VMs suffer from even more severe performance problems, perhaps in areas where adaptive optimization technology could help.

Another current trend is the appearance of VMs on small, extremely space-constrained devices for embedded systems. This domain features a plethora of instruction set architectures, so portable program representations add significant value. This domain presents different optimization challenges, including requirements to minimize memory footprint and power consumption [29], [30], [155], [156], [160], and to reduce network transfer delay [161], [162].

IX. Conclusion

We have reviewed the progression of adaptive optimization technology from a niche academic interest to a widespread, competitive production technology. We have seen a direct transfer of research ideas from the technical literature into today's production systems, and an explosion of new research activity in the area since 1995. Selective optimization has clearly had a major impact on production systems, serving as a core technology in many production VMs. Although many feedback-directed optimizations have not yet progressed from research into production systems, mainstream VM vendors already support some FDO, and we expect product VMs to further incorporate FDO technology over the next few years.

The ideas reviewed in this paper do not rely on any profound theoretical concepts. To date, progress in adaptive optimization technology has centered on overcoming the formidable engineering challenges. In the past, only a few research groups possessed the substantial resources necessary to build and maintain a credible VM research infrastructure. However, the situation has changed. Today, researchers can build on a number of high-quality open-source VMs such as Jikes RVM [163], Mono [164], and ORP [165]. The availability of this technology should lower the barriers to entry in this field, and spur even faster innovation in coming years.

References

[1] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison Wesley, 1996.

[2] E. Meijer and J. Gough, "Technical overview of the Common Language Runtime."

[3] J. L. Wilkes, "Application of microprogramming to medium scale computer design," in *Conference record of the 7th annual workshop on Microprogramming*. ACM Press, 1974, pp. 135–140.

[4] R. J. Hookway and M. A. Herdeg, "DIGITAL FX!32: Combining emulation and binary translation," *Digital Technical Journal*, vol. 9, no. 1, pp. 3–12, 1997.

[5] P. Penfield, Jr., “An APL interpreter written in APL,” in *Seventh International Conference on APL*. ACM Press, 1975, pp. 265–269.

[6] B. R. Rau, “Levels of representation of programs and the architecture of universal host machines,” in *11th Annual Workshop on Microprogramming*. IEEE Press, 1978, pp. 67–79.

[7] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth, “The open runtime platform: A flexible high-performance managed runtime environment,” *Intel Technology Journal*, vol. 7, no. 1, pp. 5–18, 2003.

[8] J. Rattner, “Forward: Managed runtime technologies,” *Intel Technology Journal*, vol. 7, no. 1, 2003.

[9] J. Aycock, “A brief history of just-in-time,” *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, 2003.

[10] J. McCarthy, “History of LISP,” *ACM SIGPLAN Notices*, vol. 13, no. 8, pp. 215–223 (or 217–223??), Aug. 1978.

[11] G. J. Hansen, “Adaptive systems for the dynamic run-time optimization of programs,” Ph.D. dissertation, Carnegie-Mellon University, 1974.

[12] L. P. Deutsch and A. M. Schiffman, “Efficient implementation of the Smalltalk-80 system,” in *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, Jan. 1984, pp. 297–302.

[13] C. Chambers and D. Ungar, “Making pure object-oriented languages practical,” in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1991, pp. 1–15.

[14] U. Hölzle and D. Ungar, “A third generation SELF implementation: Reconciling responsiveness with performance,” *ACM SIGPLAN Notices*, vol. 29, no. 10, pp. 229–243, Oct. 1994.

[15] R. E. Griswold, “A history of the SNOBOL programming languages,” *ACM SIGPLAN Notices*, vol. 13, no. 8, pp. 275–275, Aug. 1978.

[16] M. Richards, “The implementation of BCPL,” in *Software Portability*, P. J. Brown, Ed. Cambridge University Press, 1977, pp. 192–202.

[17] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and C. Jacobi, “The Pascal P-compiler: Implementation notes (rev. ed.),” Institut fur Informatik, ETH Zurich, Tech. Rep. 10, 1975.

[18] “The Perl directory - perl.org,” http://www.perl.org.

[19] “Python programming language,” http://www.python.org.

[20] “The MathWorks: Main Product Page,” http://www.mathworks.com/products.

[21] G. L. Steele, Jr. and G. J. Sussman, “Design of a lisp-based microprocessor,” *Communications of the ACM*, vol. 23, no. 11, pp. 628–645, 1980.

[22] A. Berlin and H. Wu, “Scheme86: a system for interpreting scheme,” in *1988 ACM Conference on LISP and Functional Programming*. ACM Press, 1988, pp. 116–123.

[23] H. J. Burkle, A. Frick, and C. Schlier, “High level language oriented hardware and the post-von Neumann era,” in *5th Annual Symposium on Computer Architecture*. ACM Press, 1978, pp. 60–65.

[24] J. R. Bell, “Threaded code,” *Communications of the ACM*, vol. 16, no. 6, pp. 370–372, 1973.

[25] I. Piumarta and F. Riccardi, “Optimizing direct-threaded code by selective inlining,” *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 291–300, May 1998, published as part of the proceedings of PLDI’98.

[26] M. A. Ertl and D. Gregg, “Optimizing indirect branch prediction accuracy in virtual machine interpreters,” *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 278–288, May 2003, published as part of the proceedings of PLDI’03.

[27] E. Gagnon and L. Hendren, “Effective inline-threaded interpretation of Java bytecode using preparation sequences,” in *Compiler Construction, 12th International Conference*, ser. LNCS, G. Hedin, Ed., vol. 2622. Warsaw, Poland: Springer, April 2003, pp. 170–184.

[28] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe, “Dynamic native optimization of interpreters,” in *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*. ACM Press, 2003, pp. 50–57.

[29] K. S. Venugopal, G. Manjunath, and V. Krishnan, “sEc: A portable interpreter optimizing technique for embedded Java virtual machine,” in *Usenix Java Virtual Machine Research and Technology Symposium (JVM’02)*, Aug. 2002, pp. 127–138.

[30] P. Drews, D. Sommer, R. Chandler, and T. Smith, “Managed runtime environments for next-generation mobile devices,” *Intel Technology Journal*, vol. 7, no. 1, 2003.

[31] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: a transparent dynamic optimization system,” *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 1–12, May 2000, published as part of the proceedings of PLDI’00.

[32] C. Chambers and D. Ungar, “Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs,” *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 150–164, June 1990, published as part of the proceedings of PLDI’90.

[33] D. E. Knuth, “An empirical study of FORTRAN programs,” *Software – Practice and Experience*, vol. 1, pp. 105–133, 1971.

[34] U. Hölzle and D. Ungar, “Reconciling responsiveness with performance in pure object-oriented languages,” *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 4, pp. 355–400, July 1996.

[35] D. Detlefs and O. Agesen, “The case for multiple compilers,” in *OOPSLA’99 VM Workshop: Simplicity, Performance and Portability in Virtual Machine Design*, Nov. 1999, pp. 180–194.

[36] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth, “Practicing JUDO: Java under dynamic optimizations,” *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 13–26, May 2000, published as part of the proceedings of PLDI’00.

[37] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, “Adaptive optimization in the Jalapeño JVM,” *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 47–65, Oct. 2000, proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA’00).

[38] M. Paleczny, C. Vick, and C. Click, “The Java Hotspot server compiler,” in *USENIX Java Virtual Machine Research and Technology Symposium*, Apr. 2001, pp. 1–12.

[39] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, “A dynamic optimization framework for a Java just-in-time compiler,” *ACM SIGPLAN Notices*, vol. 36, no. 11, pp. 180–195, Nov. 2001, proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA’01).

[40] J. Whaley, “Joeq: A virtual machine and compiler infrastructure,” in *Workshop on Interpreters, Virtual Machines, and Emulators*, June 2003, pp. 58–66.

[41] N. Grcevski, A. Kilstra, K. Stoodley, M. Stoodley, and V. Sundaresan, “Java just-in-time compiler and virtual machine improvements for server and middleware applications,” in *Usenix 3rd Virtual Machine Research and Technology Symposium (VM’04)*, May 2004.

[42] M. P. Plezbert and R. K. Cytron, “Does “just in time” = “better late than never”?” in *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1997, pp. 120–131.

[43] R. M. Karp, “On-line algorithms versus off-line algorithms: How much is it worth to know the future?” in *Proc. IFIP 12th World Computer Congress*, vol. 1, Aug. 1992, pp. 416–429.

[44] T. P. Kistler, “Continuous program optimization,” Ph.D. dissertation, University of California, Irvine, 1999.

[45] T. Kistler and M. Franz, “Continuous program optimization: A case study,” *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 4, pp. 500–548, 2003.

[46] T. Suganuma, T. Yasue, and T. Nakatani, “A region-based compilation technique for a Java just-in-time compiler,” *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 312–323, May 2003, published as part of the proceedings of PLDI’03.

[47] S. J. Fink and F. Qian, “Design, implementation and evaluation of adaptive recompilation with on-stack replacement,” in *International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2003, pp. 241–252.

[48] J. Whaley, “Partial method compilation using dynamic profile information,” *ACM SIGPLAN Notices*, vol. 36, no. 11, pp. 166–179, Nov. 2001, proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA’01).

[49] U. Hölzle, C. Chambers, and D. Ungar, “Debugging optimized code with dynamic deoptimization,” *ACM SIGPLAN Notices*, vol. 27, no. 7, pp. 32–43, July 1992, published as part of the proceedings of PLDI’92.

[50] “Java platform debugger architecture,” http://java.sun.com/products/jpda.

[51] M. D. Smith, “Overcoming the challenges to feedback-directed optimization,” *ACM SIGPLAN Notices*, vol. 35, no. 7, pp. 1–11, July 2000.

[52] K. Pettis and R. C. Hansen, “Profile guided code positioning,” *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 16–27, June 1990, published as part of the proceedings of PLDI’90.

[53] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu, “Using profile information to assist classic code optimizations,” *Software —Practice and Experience*, vol. 21, no. 12, pp. 1301–1321, Dec. 1991.

[54] W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, “The superblock: An effective technique for VLIW and superscalar compilation,” in *The Journal of Supercomputing*. Kluwer Academic Publishers, 1993, pp. 229–248.

[55] R. Cohn and P. G. Lowney, “Design and analysis of profile-based optimization in Compaq’s compilation tools for Alpha,” *Journal of Instruction-Level Parallelism*, vol. 3, pp. 1–25, Apr. 2000.

[56] M. Mock, C. Chambers, and S. Eggers, “Calpa: A tool for automating selective dynamic compilation,” in *the 33th International Symposium on Microarchitecture*, Dec. 2000, pp. 291–302.

[57] M. Arnold, M. Hind, and B. G. Ryder, “Online feedback-directed optimization of Java,” *ACM SIGPLAN Notices*, vol. 37, no. 11, pp. 111–129, Nov. 2002, proceedings of the 2002 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA’02).

[58] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney, “Prefetch injection based on hardware monitoring and object metadata,” in *ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2004.

[59] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. tak A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, “Continuous profiling: Where have all the cycles gone?” *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 357–390, Nov. 1997.

[60] K. Hazelwood and D. Grove, “Adaptive online context-sensitive inlining,” in *International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2003, pp. 253–264.

[61] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos, “*ProfileMe*: Hardware support for instruction-level profiling on out-of-order processors,” in *the 30th International Symposium on Microarchitecture*, Dec. 1997, pp. 292–302.

[62] S. S. Sastry, R. Bodik, and J. Smith, “Rapid profiling via stratified sampling,” in *28th Annual International Symposium on Computer Architecture*, July 2001, pp. 278–289.

[63] T. Ball and J. R. Larus, “Optimally profiling and tracing programs,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 4, pp. 1319–1360, July 1994.

[64] B. Calder, C. Krintz, S. John, and T. Austin, “Cache-conscious data placement,” *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 139–149, Nov. 1998, published as part of the proceedings of ASPLOS-VIII.

[65] D. Grove, J. Dean, C. Garrett, and C. Chambers, “Profile-guided receiver class prediction,” *ACM SIGPLAN Notices*, vol. 30, no. 10, pp. 108–123, Oct. 1995, published as part of the proceedings of OOPSLA’95.

[66] B. Calder, P. Feller, and A. Eustace, “Value profiling and optimization,” *Journal of Instruction Level Parallelism*, vol. 1, Mar. 1999.

[67] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani, “An efficient online path profiling framework for Java Just-In-Time compilers,” in *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003, pp. 148–158.

[68] M. Arnold and B. Ryder, “A framework for reducing the cost of instrumented code,” *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 168–179, May 2001, published as part of the proceedings of PLDI’01.

[69] M. Hirzel and T. Chilimbi, “Bursty tracing: A framework for low-overhead temporal profiling,” in *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001, pp. 117–126.

[70] T. M. Chilimbi and M. Hirzel, “Dynamic hot data stream prefetching for general-purpose programs,” *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 199–209, May 2002, published as part of the proceedings of PLDI’02.

[71] D. W. Wall, “Predicting program behavior using real or estimated profiles,” *ACM SIGPLAN Notices*, vol. 26, no. 6, pp. 59–70, June 1991, published as part of the proceedings of PLDI’91.

[72] Z. Wang and N. Rubin, “Evaluating the importance of user-specified profiling,” in *2nd USENIX Windows NT Symposium*, Aug. 1998.

[73] T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution analysis to find periodic behavior and simulation points in applications,” in *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[74] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” in *Tenth International Conference on Architectural Support for Prog. Lang. and Oper. Sys. (ASPLOS 2002)*, Oct. 2002.

[75] A. S. Dhodapkar and J. E. Smith, “Managing multi-configuration hardware via dynamic working set analysis,” in *29th Annual International Symposium on Computer Architecture*, May 2002, pp. 233–244.

[76] R. D. Barnes, E. M. Nystrom, M. C. Merton, and W. mei W. Hwu, “Vacuum packing: Extracting hardware-detected program phases for post-link optimization,” in *the 35th International Symposium on Microarchitecture*, Nov. 2002, pp. 233–244.

[77] T. Sherwood, S. Sair, and B. Calder, “Phase tracking and prediction,” in *30th Annual International Symposium on Computer Architecture*, June 2003, pp. 336–349.

[78] A. S. Dhodapkar and J. E. Smith, “Comparing program phase detection techniques,” in *the 36th International Symposium on Microarchitecture*, Dec. 2003.

[79] M. R. Garey and D. S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[80] R. W. Scheifler, “An analysis of inline substitution for a structured programming language,” *Communications of the ACM*, vol. 20, no. 9, pp. 647–654, Sept. 1977.

[81] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu, “Profile-guided automatic inline expansion for C programs,” *Software—Practice and Experience*, vol. 22, no. 5, pp. 349–369, May 1992.

[82] A. Ayers, R. Schooler, and R. Gottlieb, “Aggressive inlining,” *ACM SIGPLAN Notices*, vol. 32, no. 5, pp. 134–145, May 1997, published as part of the proceedings of PLDI’97.

[83] O. Kaser and C. Ramakrishnan, “Evaluating inlining techniques,” *Computer Languages*, vol. 24, pp. 55–72, 1998.

[84] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney, “A comparative study of static and profile-based heuristics for inlining,” *ACM SIGPLAN Notices*, vol. 35, no. 7, pp. 52–64, July 2000, published as part of the proceedings of Dynamo’00.

[85] J. Dean and C. Chambers, “Towards better inlining decisions using inlining trials,” in *LISP and Functional Programming*, 1994, pp. 273–282.

[86] O. Waddell and R. K. Dybvig, “Fast and effective procedure inlining,” in *4th International Symposium on Static Analysis*, Sept. 1997.

[87] T. Suganuma, T. Yasue, and T. Nakatani, “An empirical study of method in-lining for a Java just-in-time compiler,” in *Usenix Java Virtual Machine Research and Technology Symposium (JVM’02)*, Aug. 2002, pp. 91–104.

[88] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman, “The StarJIT compiler: A dynamic compiler for managed runtime environments,” *Intel Technology Journal*, vol. 7, no. 1, pp. 19–31, Feb. 2003.

[89] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, “Overview of the IBM Java Just-in-Time Compiler,” *IBM Systems Journal*, vol. 39, no. 1, pp. 175–193, Feb. 2000.

[90] P. Chang, S. Mahlke, W. Chen, N. Warter, and W.-M. Hwu, “IMPACT: An architectural framework for multiple-instruction-issue processors,” in *Proc. 18th International Symposium on Computer Architecture*, 1991, pp. 266–275.

[91] W. Chen, S. Mahlke, N. Warter, S. Anik, and W. Hwu, “Profile-assisted instruction scheduling,” *International Journal of Parallel Programming*, vol. 22, no. 2, pp. 151–181, Apr. 1994.

[92] C. Dulong, “The IA-64 architecture at work,” *IEEE Computer*, vol. 31, no. 7, pp. 24–32, July 1998.

[93] I. Corporation, *IA-64 Application Developer’s Architecture Guide*, May 1999, order Number: 245188-001.

[94] M. Byler, J. R. B. Davies, C. Huson, B. Leasure, and M. Wolfe, “Multiple version loops,” in *1987 International Conference on Parallel Processing*, 1987, pp. 312–318.

[95] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira, “Automatic loop transformations and parallelization for Java,” in *2000 International Conference on Supercomputing*, May 2000, pp. 1–10.

[96] M. Gupta, J.-D. Choi, and M. Hind, “Optimizing Java programs in the presence of exceptions,” in *14th European Conference on Object-Oriented Programming*, June 2000, pp. 422–446.

[97] M. Arnold and B. G. Ryder, “Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading,” in *16th European Conference on Object-Oriented Programming*, June 2002.

[98] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, “A study of devirtualization techniques for a Java Just-In-Time compiler,” *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 294–310, Oct. 2000, proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA’00).

[99] C. Chambers and D. Ungar, “Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language,” *ACM SIGPLAN Notices*, vol. 24, no. 7, pp. 146–160, July 1989, published as part of the proceedings of PLDI’89.

[100] B. Grant, M. Philipose, M. Mock, S. J. Eggers, and C. Chambers, “An evaluation of run-time optimizations,” *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 293–304, May 1999, published as part of the proceedings of PLDI’99.

[101] P. Lee and M. Leone, “Optimizing ML with run-time code generation,” *ACM SIGPLAN Notices*, vol. 31, no. 5, pp. 137–148, May 1996, published as part of the proceedings of PLDI’96.

[102] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek, “’C and tcc: a language and compiler for dynamic code generation,” *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 2, pp. 324–369, Mar. 1999.

[103] P. C. Diniz and M. C. Rinard, “Dynamic feedback: An effective technique for adaptive computing,” *ACM SIGPLAN Notices*, vol. 32, no. 5, pp. 71–84, May 1997, published as part of the proceedings of PLDI’97.

[104] M. J. Voss and R. Eigemann, “High-level adaptive program optimization with ADAPT,” *ACM SIGPLAN Notices*, vol. 36, no. 7, pp. 93–102, July 2001, published as part of the proceedings of PPOPP’01.

[105] T. Ogasawara, H. Komatsu, and T. Nakatani, “A study of exception handling and its dynamic optimization in Java,” *ACM SIGPLAN Notices*, vol. 36, no. 11, pp. 83–95, Nov. 2001, proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA’01).

[106] C.-K. Luk and T. C. Mowry, “Compiler-based prefetching for recursive data structures,” *ACM SIGPLAN Notices*, vol. 31, no. 9, pp. 222–233, Sept. 1996, co-published as SIGOPS Operating Systems Review **30**(5), December 1996, and as SIGARCH Computer Architecture News, **24**(special issue), October 1996.

[107] Y. Wu, “Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching,” *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 210–221, May 2002, published as part of the proceedings of PLDI’02.

[108] T. Inagaki, T. Onodera, H. Komatsu, and T. Nakatani, “Stride prefetching by dynamically inspecting objects,” *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 269–277, May 2003, published as part of the proceedings of PLDI’03.

[109] G. Krasner, *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[110] U. Hölzle, C. Chambers, and D. Ungar, “Optimizing dynamically-typed object-oriented languages with polymorphic inline caches,” in *5th European Conference on Object-Oriented Programming*, July 1991, pp. 21–38.

[111] P. F. Dietz and D. D. Sleator, “Two algorithms for maintaining order in a list,” in *Nineteenth Annual ACM Symposium on Theory of Computing*, May 1987, pp. 365–372.

[112] G. Ramalingam and H. Srinivasan, “Object model for Java,” IBM Research Division, Tech. Rep. 20642, Dec. 1996.

[113] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi, “Marmot: An optimizing compiler for Java,” Microsoft Research, Tech. Rep. MSR-TR-99-33, June 1999.

[114] A. Krall, J. Vitek, and R. N. Horspool, “Near optimal hierarchical encoding of types,” in *11th European Conference on Object-Oriented Programming*, Jyvaskyla, Finland, June 1997.

[115] B. Alpern, A. Cocchi, and D. Grove, “Dynamic type checking in Jalapeño,” in *USENIX Java Virtual Machine Research and Technology Symposium*, Apr. 2001, pp. 41–52.

[116] C. Click and J. Rose, “Fast subtype checking in the HotSpot JVM,” in *The Joint ACM Java Grande - ISCOPE 2001 Conference*, 2002, pp. 96–107.

[117] K. Kawachiya, A. Koseki, and T. Onodera, “Lock reservation: Java locks can mostly do without atomic operations,” *ACM SIGPLAN Notices*, vol. 37, no. 11, pp. 130–141, Nov. 2002, proceedings of the 2002 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA’02).

[118] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, “Thin locks: Featherweight synchronization for Java,” *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 258–268, May 1998, published as part of the proceedings of PLDI’98.

[119] D. F. Bacon, S. J. Fink, and D. Grove, “Space- and time-efficient implementations of the Java object model,” in *16th European Conference on Object-Oriented Programming*, June 2002.

[120] J. Dean, D. Grove, and C. Chambers, “Optimization of object-oriented programs using static class hierarchy analysis,” in *9th European Conference on Object-Oriented Programming*, Aug. 1995, pp. 77–101.

[121] D. Detlefs and O. Agesen, “Inlining of virtual methods,” in *13th European Conference on Object-Oriented Programming*, June 1999, pp. 258–278.

[122] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Chichester, England: John Wiley and Sons, 1996.

[123] T. Printezis, “Hot-swapping between a Mark&Sweep and a Mark&Compact garbage collector in a generational environment,” in *USENIX Java Virtual Machine Research and Technology Symposium*, Apr. 2001, pp. 171–183.

[124] S. Soman, C. Krintz, and D. Bacon, “Adaptive, application-specific garbage collection,” University of California, Santa Barbara, Tech. Rep. 2003-07, Mar. 2003.

[125] R. Dimpsey, R. Arora, and K. Kuiper, “Java server performance: A case study of building efficient, scalable Jvms,” *IBM Systems Journal*, vol. 39, no. 1, pp. 151–174, Feb. 2000.

[126] A. W. Appel, “Simple generational garbage collection and fast allocation,” *Software—Practice and Experience*, vol. 19, no. 2, pp. 171–183, Feb. 1989.

[127] B. Hayes, “Using key object opportunism to collect old objects,” in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 1991, pp. 33–46.

[128] M. Hirzel, A. Diwan, and M. Hertz, “Connectivity-based garbage collection,” in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2003, pp. 359–373.

[129] P. R. Wilson, “Opportunistic garbage collection,” *ACM SIGPLAN Notices*, vol. 23, no. 12, pp. 98–102, Dec. 1988.

[130] H. Lieberman and C. Hewitt, “A real-time garbage collector based on the lifetimes of objects,” *Communications of the ACM*, vol. 26, no. 6, pp. 419–429, 1983.

[131] T. M. Chilimbi, B. Davidson, and J. R. Larus, “Cache-conscious structure definition,” *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 13–24, May 1999, published as part of the proceedings of PLDI’99.

[132] T. P. Kistler and M. Franz, “Continuous program optimization: Design and evaluation,” *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 549–566, June 2001.

[133] T. Kistler and M. Franz, “Automated data-member layout of heap objects to improve memory-hierarchy performance,” *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 3, pp. 490–505, 2000.

[134] R. M. Rabbah and K. V. Palem, “Data remapping for design space optimization of embedded memory systems,” *ACM Transactions on Embedded Computing Systems*, vol. 2, no. 2, pp. 1–32, May 2003.

[135] T. M. Chilimbi and J. R. Larus, “Using generational garbage collection to implement cache-conscious data placement,” *ACM SIGPLAN Notices*, vol. 34, no. 3, pp. 37–48, Mar. 1999.

[136] Y. Shuf, M. Gupta, H. Franke, A. Appel, and J. P. Singh, “Creating and preserving locality of Java applications at allocation and garbage collection times,” *ACM SIGPLAN Notices*, vol. 37, no. 11, pp. 13–25, Nov. 2002, proceedings of the 2002 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA’02).

[137] S. Rubin, R. Bodík, and T. Chilimbi, “An efficient profile-analysis framework for data-layout optimizations,” *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 140–153, Jan. 2002, proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL’02).

[138] M. Frigo and S. G. Johnson, “FFTW: An adaptive software architecture for the FFT,” in *1998 IEEE International Conference on Acoustics Speech and Signal Processing*, vol. 3. IEEE, 1998, pp. 1381–1384.

[139] R. C. Whaley and J. J. Dongarra, “Automatically tuned linear algebra software,” University of Tennessee, Tech. Rep. UT-CS-97-366, 1997.

[140] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, “Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology,” in *1997 International Conference on Supercomputing*, 1997, pp. 340–347.

[141] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A comparison of empirical and model-driven optimization," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 63–76, May 2003, published as part of the proceedings of PLDI'03.

[142] J. Bogda and A. Singh, "Can a shape analysis work at run-time?" in *Usenix Java Virtual Machine Research and Technology Symposium (JVM'01)*, Apr. 2001, pp. 13–26.

[143] V. C. Sreedhar, M. Burke, and J.-D. Choi, "A framework for interprocedural optimization in the presence of dynamic class loading," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 196–207, May 2000, published as part of the proceedings of PLDI'00.

[144] I. Pechtchanski and V. Sarkar, "Dynamic optimistic interprocedural analysis: a framework and an application," *ACM SIGPLAN Notices*, vol. 36, no. 11, pp. 195–210, Nov. 2001, proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[145] F. Qian and L. Hendren, "Towards dynamic interprocedural analysis in jvms," in *Usenix 3rd Virtual Machine Research and Technology Symposium (VM'04)*, May 2004.

[146] M. Hirzel, A. Diwan, and M. Hind, "Pointer analysis in the pressence of dynamic class loading," in *18th European Conference on Object-Oriented Programming*, June 2004.

[147] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta, "Quicksilver: a quasi-static compiler for Java," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 66–82, Oct. 2000, proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00).

[148] M. Philipose, C. Chambers, and S. J. Eggers, "Towards automatic construction of staged compilers," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 113–125, Jan. 2002, proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'02).

[149] A. Zaks, V. Feldman, and N. Aizikowitz, "Sealed calls in Java packages," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 83–92, Oct. 2000, proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00).

[150] J. Choi and H. Srinivasan, "Deterministic replay of Java multithreaded applications," in *SIGMETRICS Symposium on Parallel and Distributed Tools*, Aug. 1998, pp. 48–59.

[151] D. L. Bruening, "Systematic testing of multithreaded Java programs," Master's thesis, Massachusetts Institute of Technology, 1999.

[152] D. R. Engler, "VCODE: A retargetable, extensible, very fast dynamic code generation system," *ACM SIGPLAN Notices*, vol. 31, no. 5, pp. 160–170, May 1996, published as part of the proceedings of PLDI'96.

[153] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 895–913, Sept. 1999.

[154] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño dynamic optimizing compiler for Java," in *ACM 1999 Java Grande Conference*, June 1999, pp. 129–141.

[155] N. Shaylor, "A just-in-time compiler for memory-constrained low-power devices," in *Usenix Java Virtual Machine Research and Technology Symposium (JVM'02)*, Aug. 2002, pp. 119–126.

[156] M. Chen and K. Olukotun, "Targeting dynamic compilation for embedded environments," in *Usenix Java Virtual Machine Research and Technology Symposium (JVM'02)*, Aug. 2002, pp. 151–164.

[157] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue,

T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani, "Effectiveness of cross-platform optimizations for a Java Just-in-Time compiler," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2003, pp. 187–204.

[158] L. Eeckhout, A. Georges, and K. De Bosschere, "How Java programs interact with virtual machines at the microarchitectural level," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2003, pp. 169–186.

[159] Transmeta Corporation, "Transmeta Corporation: Crusoe," http://www.transmeta.com/crusoe.

[160] N. Vijaykrishnan, M. Kandemir, S. Tomar, S. Kim, A. Sivasubramaniam, and M. J. Irwin, "Energy behavior of Java applications from the memory perspective," in *Usenix Java Virtual Machine Research and Technology Symposium (JVM'01)*, Apr. 2001, pp. 207–220.

[161] C. Krintz, B. Calder, H. B. Lee, and B. G. Zorn, "Overlapping execution with transfer using non-strict execution for mobile programs," *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 159–169, Nov. 1998.

[162] C. Krintz, B. Calder, and U. Hölzle, "Reducing transfer delay using Java class file splitting and prefetching," *ACM SIGPLAN Notices*, vol. 34, no. 10, pp. 276–291, Oct. 1999, published as part of the proceedings of OOPSLA'99.

[163] "Jikes Research Virtual Machine (RVM)," http://www.ibm.com/developerworks/oss/jikesrvm.

[164] "Mono," http://go-mono.com.

[165] "Open Runtime Platform," http://orp.sourceforce.net.

PLACE
PHOTO
HERE

Matthew Arnold received the B.S. degree from Rensselaer Polytechnic Institute in 1995, and the M.S. and Ph.D. degrees from Rutgers University in 1998 and 2002, respectively. For his thesis work he developed low-overhead techniques for performing online profiling and feedback-directed optimization in a Java virtual machine. In 2002, Matthew became a Research Staff Member in the Software Technology Department at the IBM T.J. Watson Research Center, where he is continuing his research in language-level profiling techniques, as well as developing profiling and visualization tools for distributed web services applications. He is a member of ACM.

PLACE
PHOTO
HERE

Stephen Fink received the B.S. degree from Duke University in 1992 and the M.S and Ph.D. degrees from the University of California, San Diego in 1994 and 1998, respectively. In 1998, Stephen became a Research Staff Member in the Software Technology Department at the IBM T.J. Watson Research Center. He was a member of the team that produced the Jikes Research Virtual Machine, and is currently investigating the application of static program analysis to Enterprise Java Beans. His research interests include programming language implementation techniques, program analysis, and parallel and scientific computation. He is a member of ACM.

PLACE
PHOTO
HERE

David Grove received the B.S. degree from Yale College in 1992, and the M.S. and Ph.D. degrees from the University of Washington in 1994 and 1998, respectively. In 1998, David became a Research Staff Member in the Software Technology Department at the IBM T.J. Watson Research Center. He is a member of the Jikes RVM core team and helped develop its adaptive optimization system, optimizing compiler, and runtime system. His research interests include program language design and implementation, virtual machines, and adaptive optimization. He is a member of ACM.

PLACE
PHOTO
HERE

Michael Hind received the B.A. degree from the State University of New York at New Paltz in 1985 and the M.S. and Ph.D. degrees from New York University in 1991. From 1992-1998, Michael was a professor of computer science at the State University of New York at New Paltz. In 1998, Michael became a Research Staff Member in the Software Technology Department at the IBM T.J. Watson Research Center, working on the Jalapeño project, the project that produced the open source Jikes RVM. In 2000, he became the manager of the Dynamic Optimization Group at IBM Research. His research interests include program analysis, adaptive optimization, and memory latency issues. He is a member of ACM.

PLACE
PHOTO
HERE

Peter F. Sweeney (Member, IEEE) received the B.S. and M.S degrees in Computer Science from Columbia University in 1984 and 1989, respectively. In 1985, Peter became a Software Engineer in the Software Technology Department at the IBM T.J. Watson Research Center. In 2000, he became a Research Staff Member. His research interests include understanding the behavior of object-oriented programming languages to reduce their space and time overhead. He is a member of ACM.