Continuous Program Optimization: A Case Study

THOMAS KISTLER and MICHAEL FRANZ
University of California, Irvine

Much of the software in everyday operation is not making optimal use of the hardware on which it actually runs. Among the reasons for this discrepancy are hardware/software mismatches, modularization overheads introduced by software engineering considerations, and the inability of systems to adapt to users' behaviors.

The obvious solution to these problems is to delay code generation until load time. This is the earliest point at which a piece of software can be fine-tuned to the actual capabilities of the hardware on which it is about to be executed, and also the earliest point at wich modularization overheads can be overcome by global optimization.

A still better match between software and hardware can be achieved by replacing the already executing software at regular intervals by new versions constructed on-the-fly using a background code re-optimizer. This not only enables the use of live profiling data to guide optimization decisions, but also facilitates adaptation to changing usage patterns and the late addition of dynamic link libraries.

This paper presents a system that provides code generation at load-time and continuous program optimization at run-time. First, the architecture of the system is presented. Then, two optimization techniques are discussed that were developed specifically in the context of continuous optimization. The first of these optimizations continually adjusts the storage layouts of dynamic data structures to maximize data cache locality, while the second performs profile-driven instruction re-scheduling to increase instruction-level parallelism. These two optimizations have very different cost/benefit ratios, presented in a series of benchmarks. The paper concludes with an outlook to future research directions and an enumeration of some remaining research problems.

The empirical results presented in this paper make a case in favor of continuous optimization, but indicate that it needs to be applied judiciously. In many situations, the costs of dynamic optimizations outweigh their benefit, so that no break-even point is ever reached. In favorable circumstances, on the other hand, speed-ups of over 120% have been observed. It appears as if the main beneficiaries of continuous optimization are shared libraries, which at different times can be optimized in the context of the currently dominant client application.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Run-time Environments*; D.3.4 [**Programming Languages**]: Processors—*Code Generation*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms: Dynamic Code Generation, Continuous Program Optimization, Dynamic Re-Optimization

---

Authors' current addresses: Thomas Kistler, Transmeta Corporation, 3940 Freedom Circle, Santa Clara, CA 95054. Michael Franz, Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92697–3425. Parts of this work are funded by a CAREER award from the National Science Foundation (CCR–97014000) and by the California MICRO Program with industrial sponsor Microsoft Research (Project No. 99-039). A precursor to the chapter on the similarity of profiling data previously appeared as a refereed contribution in the *Proceedings of the Workshop on Profile and Feedback-Directed Optimization*, Paris, France, October 1998.

1. INTRODUCTION

In the wake of dramatic improvements in processor speed, it is often overlooked that much of the software in everyday operation is not making the best use of the hardware on which it actually runs. The vast majority of computers are either running application programs that have been optimized for earlier versions of the target architecture, or, worse still, are emulating an entirely different architecture in order to support legacy code.

The first reason why hardware and software are often mismatched is linked to the speed of technology evolution. Users demand backward compatibility and are often unwilling to give up existing software when upgrading hardware. As a result of this, an immense amount of legacy code is in use every day: 16-bit software on 32-bit processors, emulated MC680x0 code on PowerPC Macintosh computers, and soon also IA32 code on IA64 hardware. Simultaneously, purely logistical constraints make it unfeasible for software vendors to provide separate versions of every program for every particular hardware implementation of a processor architecture. Just consider: there are several major manufacturers of IA32-compatible CPUs, and each of these has a product line spanning several processors—the total variability is far too great to manage in a centralized fashion.

The second reason why the capabilities of the hardware are not exploited to the fullest has to do with software engineering concerns. Increasingly, software is developed and distributed as smaller components that are linked together dynamically only at the end-users site. Unfortunately, there is a *modularization cost* associated with separate compilation—since neither the end-user's configuration nor the components' interaction schemes are known at compile-time, many traditional global code optimizations cannot be applied across component boundaries. The added benefits of component-orientation are usually so great that this drawback is readily accepted by software developers as well as users.

The obvious solution to overcoming both of these performance impediments simultaneously is to delay code generation at least until load time. Not only are the hardware characteristics of the target machine definite at this point, but load-time code generation also makes it possible to perform optimizations across the boundaries of independently-distributed software components and thereby reduce the performance penalty paid for modularization.This approach has been validated in previous work and is now common practice [Deutsch and Schiffman 1984; Franz 1994; Hölzle 1994; Hölzle and Ungar 1996; Adl-Tabatabai et al. 1998].

This paper presents a system that goes another step further, achieving a yet again better match between the software and the hardware by re-evaluating the bindings between software and hardware *at regular intervals* instead of permanently fixing them prior to execution. To this effect, a profiler constantly observes the running program and determines where the most effort is spent. Using this information as a guide, new versions of the already running software are then continuously constructed on-the-fly in the background, placing special emphasis on optimizing the performance-critical regions of the program. When the background re-optimization is complete, the new software is "hot-swapped" into the foreground and execution resumes using the new code image rather than the old one. The latter can be discarded as soon as the last thread of execution has migrated away from it.

Continuous run-time optimization not only enables an exact match between software and hardware and the ability to perform global re-optimization in reaction to the dynamic addition of further software components, but the code produced is also often of a genuinely higher quality than can be achieved using static “off-line” compilation. This is because up-to-the-minute profiling data is available to guide optimization decisions, whereas traditional compilers can at best draw on traces of previous execution runs. Consequently, a continuously optimizing system can quickly adapt to changing user-session patterns and thereby provide a higher overall performance than any static system optimized to a specific or “average” pattern. This is especially important as there is usually no single “typical” usage scenario, but rather several distinctive ones that differ widely from each other and among which a single user may alternate over the course of a single computing session. In component-oriented systems, there is not even a “typical” application, so that this effect is even more pronounced.

Continuous program optimization also doesn’t suffer the same extreme resource constraints as load-time code generation, which due to its interactive nature has severe code-generation-time limitations. This often prohibits use of the best known optimization algorithms. Systems such as the one we have built don’t have these constraints, because they perform optimization strictly in the background during idle-time (and potentially on a different processor), while an alternate version of the application program is already executing. Consequently, the speed of optimization is almost completely irrelevant; even optimization cycles that last on the order of 10 minutes are still useful.

The remainder of this paper is organized as follows. Section 2 through Section 4 present different architectural facets of our system in which code generation and code re-optimization are central system services. Section 5 discusses the issue of deciding when and what to optimize. Section 6 and Section 7 present two optimization techniques that were specifically designed to take advantage of continuous optimization and live profiling data: object layout adaptation and dynamic trace scheduling. Section 8 then evaluates the performance of the system based on these two optimization techniques. Section 9 outlines future research directions and lists some open problems that will have to be solved for continuous program optimization to become common-place in modern operating systems. Section 10 discusses related work and Section 11 concludes the paper.

2. ARCHITECTURAL OVERVIEW

We have implemented a system providing dynamic profiling, dynamic optimization, and dynamic replacement of live code and data. We view this system as a case study and possible blueprint for future efforts to provide continuous optimization capabilities.

A central trait of our architecture is extensibility. The dynamic optimizer at the heart of our system has a component structure supporting incremental modification in a plug-and-play manner. This is an essential facility for making system-level code generation useful in practice. Users migrating to a new set of hardware features then merely need the appropriate plug-in components that match the new target architecture, rather than a whole new run-time system. These plug-in components are only loosely coupled to the rest of the run-time system and communicate with it via a message bus.

The central idea behind this component-based architecture is that hardware designers know a lot about their specific product, but relatively little about run-time systems in general. A plug-in component for a run-time optimizer is akin to a device driver, except that it enables an application program to utilize the main computing engine more effectively. Just as operating systems today are shipped with a large number of device drivers for every conceivable piece of hardware that an end-user might want to install, an operating system incorporating an extensible code optimizer at its core would rely on a set of plug-in optimization components supplied by the manufacturers of the various processors. Hence, instead of today's centralized approach, in which software suppliers need to keep track of, and maintain appropriate compilers for, all the architectures for which they want to provide optimized code, our solution shifts this responsibility to the hardware providers, completely eliminating the problems of hardware variability that are one of the two main reasons for existing hardware/software mismatches mentioned above.

The overall structure of our dynamic code generation and optimization system is illustrated in Figure 1. The system, implemented on top of the Oberon System 3 [Wirth and Gutknecht 1992; Gutknecht 1994; Gutknecht and Franz 1999] for the Macintosh platform, is composed of five key constituents: a manager, a code generating loader, a profiler, an optimizer, and a replacer. This assembly of sub-systems is in turn part of the Oberon System that provides many additional services, among them dynamic loading of software modules, run-time type-tagging of dynamically allocated objects, and garbage collection.

The five constituents of our system interact as follows: When the user first launches an application, the *code-generating loader* translates the representation that programs are transported in into a sequence of native machine instructions.[1] Because this is an interactive process (the user is waiting), and because many worthwhile code optimizations are extremely time-consuming, the code-generating loader does not optimize much but instead concentrates on simply producing an executable program as quickly as possible.

[1] Our particular implementation uses the Slim Binary representation [Franz and Kistler 1997], but the architecture presented here does not depend on this fact. The same architecture could also be used with programs represented as class files for the Java Virtual Machine, or even with native code for some specific processor. This would merely have an effect on the pre-processing effort required to extract information relevant to the optimizer, such as control flow and data flow information.

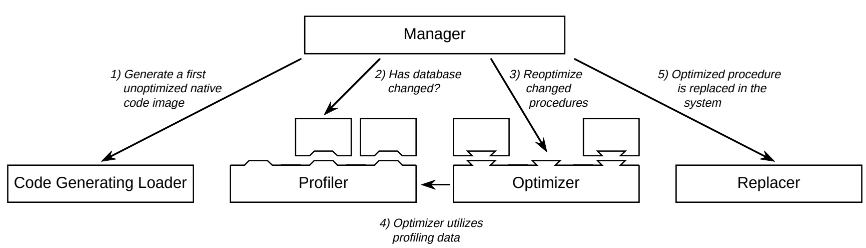

Fig. 1. General architectural overview

Name	Description
InstrCnt	Measures the sampling frequency of individual instructions using a sampling profiler
ProcCnt	Measures the execution frequency of individual procedures
ProcTime	Measures the execution time of individual procedures
EdgeCnt	Measures the execution frequency of basic block transitions [Ball and Larus 1994]
PathCnt	Measures the execution frequency of individual paths within procedures [Ball et al. 1998]
CallCnt	Measures the execution frequency of individual call-sites
CallTime	Measures the execution time of individual call-sites
CallParamVal	Monitors the most frequent actual parameters for individual call-sites

Table I. Built-in profiling components

Once the application program has begun to execute, the *profiler* starts collecting information about its behavior. This information is later used to guide optimizations. Examples of the kind of information collected by the profiler are illustrated in Table I and include the call-frequencies of individual procedures, statistics on how variables and parameters are accessed, and a catalog of which instructions stall due to misses in the data cache. The profiler runs continuously at all times. It has an extensible structure that can support a wide spectrum of profiling techniques and can be augmented as required by the plug-and-play addition of appropriate *profiling components*, such as instrumenting profilers and sampling profilers.

The *system manager* executes a low priority thread that uses application idle time to optimize the already running software in the background. It repeatedly queries the profiling database to examine whether the characteristics of the system's behavior have changed, and for which procedures they have changed. This is done by means of a similarity computation mechanism described later in Section 5. Based on this information, the system manager builds a list of procedures for potential optimization. The optimization candidates in this list are not all equally well suited for optimizations—optimizing some procedures might be more profitable than optimizing others. Therefore, the system manager additionally queries the optimization manager for a rough estimate of the profitability of optimizing each individual procedure. This information is used to sort the candidate list according to optimization priorities.

The system manager then invokes the *optimizer* on each procedure,[2] in the order in which they appear in the candidate list. As explained below, optimizations are

[2]The fact that the optimizer operates on a program using a granularity of one procedure at a time does not imply that it cannot perform interprocedural optimizations—it merely represents an architectural decision. Our implementation preserves cross-procedural state in a history database and in individual optimization components.

Optimization Components
Sparse Conditional Constant Propagation
Dead Code Elimination
Optimistic Value Numbering
Loop Invariant Code Motion
Strength Reduction
Peephole Optimization
Instruction Scheduling (Forward List Scheduling)
Hierarchical Register Allocation

Table II. Built-in optimization phases

organized as a sequence of *phases* that are executed sequentially. Table II gives an overview of the built-in optimization phases provided in our system .

Finally, after the optimizer has completed its work, the *replacer* hot-swaps the currently executing code image against the newly generated, optimized version. This process requires updating interprocedural and inter-modular dependencies and, in some cases, undoing previous optimizations.

3. THE OPTIMIZER SUBSYSTEM

The optimizer is composed of three main parts: the optimization manager, a history database, and a set of optimization components that can be dynamically added, removed, and exchanged. Figure 2 presents a schematic overview of how these parts interact. The various services offered by the optimizer operate on the program being optimized at the level of individual procedures. However, the fact that the optimizer operates on a program using a granularity of one procedure at a time does not imply that it cannot perform interprocedural optimizations—it merely represents an architectural decision. Our implementation preserves cross-procedural state in the history database and in individual optimization components.

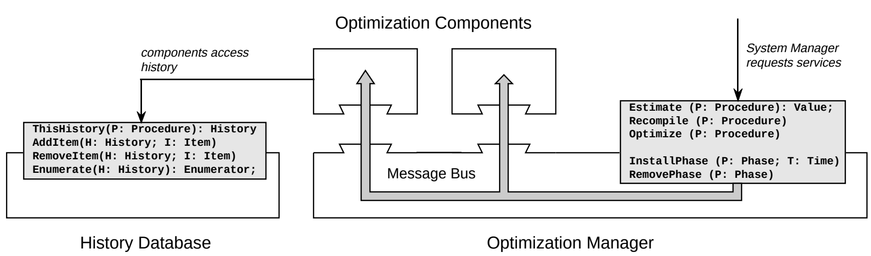

Fig. 2. Schematic View of the Optimizer

The *optimization manager* handles all requests from the system manager and coordinates the optimization process. Optimizations are organized as a sequence of *phases* that are executed sequentially. The various phases operate on a common intermediate representation of the program, *guarded single assignment form*

(GSA) [Brandis 1995], a variant of SSA [Cytron et al. 1991]. The intermediate GSA representation is not cached across separate invocations of the optimizer but generated afresh from the software transportation format[3] each time that a new optimization cycle commences—since the unit of optimization is the procedure, this keeps memory consumption within reasonable limits. Hence, the first phase of the optimizer generates GSA for a procedure of the program being optimized, while each subsequent phase retrieves this intermediate representation, performs a specific task, and then returns a possibly modified version of the procedure in the same intermediate format. If a look at actual profiling data suggests that an optimization is not profitable, the intermediate representation remains unmodified. The specific tasks that individual phases perform correspond to individual code optimizations such as dead code elimination, common subexpression elimination, and register allocation. A complete list of the built-in optimization phases provided in our system are depicted in Table II.

An *optimization component* is a container that encapsulates the implementation of one or more optimization phases. In most cases, each component implements exactly one phase. As discussed in the introduction, plug-and-play customizability makes it possible to achieve a perfect match between user software and underlying hardware platform without requiring global updates of either application programs or the run-time system. Instead, for each new member of a processor family, the hardware manufacturer merely needs to supply the specific components that perform optimizations tailored to characteristics in which the new processor differs from the generic representative of the family. As an example, there would be a unique instruction-scheduling component for each processor model. As a further example, a component supporting MMX instructions would map specific library calls and possibly further code patterns directly into multimedia instructions emitted in-line.

The *history database* is a repository that records the set of optimizations that have been performed on individual procedures. It is used for bookkeeping and for coordinating code optimizations. Since different optimization techniques may have conflicting goals (e.g., “decrease code-size to reduce misses in the instruction cache” vs. “unroll loops to reduce misses in the data cache”), an optimization phase may consult the history database to determine whether its technique interferes with previously applied optimizations. This is important because phases are independent of each other; each phase is only aware of which other phases have executed previously in the current optimization schedule for the procedure under consideration, and completely unaware about phases that might follow further downstream. The history database is kept in memory and its contents are volatile: history information is not preserved across cold-starts of our system, and even while the system is running, all global state information is “aged” periodically (see below).

Component Interaction in the Optimizer

The key to flexibility in our solution is that new phases can be registered at the optimization manager, can be removed from the optimizer, and can even replace other phases without affecting the remainder of the run-time system. This is achieved

[3] i.e., in the case of our implementation, the aforementioned Slim Binary format.

by letting the optimization manager communicate with individual phases via an open central message bus, rather than hard-coding component interfaces. When the optimization manager receives a request from the system manager, it translates the request into a sequence of messages and distributes them to the phases within installed optimization components. Each optimization phase has to conform to the message protocol depicted in Figure 3. In the following, we explain the semantics attached to the individual messages.

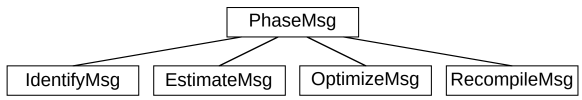

Fig. 3. Optimizer Message Protocol

When the profiler subsystem detects a substantial change in the behavior of a certain procedure, the system manager invokes the optimizer's `Estimate()` service. The `Estimate()` service assesses the profitability of applying further optimizations to the procedure. Based on this assessment, the system manager then decides whether or not to actually optimize the procedure. Since the estimate is used to eliminate unprofitable optimization candidates, it has to be computed efficiently—at least in relation to the time that would be spent optimizing those candidates. Consequently, this computation is based on simple heuristics without actually looking "into" the optimization candidate itself. This also means that the estimate can be computed without first having to generate GSA.

The `Estimate()` service is implemented by sending an `EstimateMsg` to each installed optimization phase. Each phase is responsible for computing the estimated speedup that would result if the associated optimization were added to the already existing optimization schedule for a given procedure. Hence, if the associated optimization is currently already performed on the procedure, no additional speedup can be expected and a value of zero is returned. Otherwise, a simple heuristic is used to compute the profitability of optimizing the procedure. The heuristic is based both on a hard-coded average speedup (e.g., 5% speedup for data prefetching vs. 20% speedup for common subexpression elimination) and actual speedups measured for previous applications of the same optimization (to this and other procedures). The total speedup estimate for a procedure is then derived by computing the sum of the speedup estimates of all optimization phases that are present in the system.

Once the system manager has decided which procedures to optimize, it invokes the `Optimize()` service for each optimization candidate. The `Optimize()` service creates a newly optimized version of a given procedure from scratch. To begin with, a GSA representation of the procedure is generated from the software transportation format (e.g., the original "object file" or an in-memory cache). Then, each installed optimization phase is sent an `OptimizeMsg`, instructing it to apply its respective modifications to the procedure's GSA representation (the order in which individual phases execute is discussed below). Finally, the optimized GSA representation is transcribed into native code and handed over to the replacer.

Upon receiving an `OptimizeMsg`, an optimization phase first needs to re-evaluate whether or not it would be profitable to perform its associated task. This is because the original estimate based upon which the optimization manager decided to apply this optimization was founded on inaccurate low-cost heuristics, whereas at this point, the full GSA representation and up-to-the-minute profiling data are available. This makes a much better estimate of the anticipated speedup possible. For example, a phase performing loop unrolling might have an estimated speedup of 10%. However, after looking at the loops in question, the optimization phase might determine that they exhibit too little parallelism and hence aren't profitable after all.

If the optimization had previously been applied to the given procedure, its benefit is re-evaluated on the basis of actual performance data. If the past speedup was negative or insignificant, the optimization is marked as non-profitable in the history database and is dropped from future iterations. Otherwise, it is applied again. If the particular code optimization had previously not been applied to the procedure, the optimization phase examines profiling data and decides whether to apply it or not (for example, based on whether a certain profiling counter exceeds a certain threshold). If it decides to apply the optimization, it is added to the history database and is performed.

In contrast to the `Optimize()` service that applies optimizations *optimistically*, the `Recompile()` service optimizes procedures *pessimistically*. Neither does it consider new optimizations nor does it remove unsuccessful old ones. It only re-performs the set of optimizations recorded in the history database, excluding optimizations previously determined to be non-profitable. This service is particularly useful for *de-optimizations*, a case in which optimizations have to be selectively undone. We already gave an example where this is useful, in the case where a statically bound method is overridden in a dynamically loaded extension. In such a situation, the previous optimization must be undone. This is achieved by removing it from the history database and calling the `Recompile()` service for all affected methods. The `Recompile()` service is implemented by sending a `RecompileMsg` to all installed optimization phases.

Finally, the `IdentifyMsg` is sent to an optimization phase to request meta-information, such as its name and when it wishes to be executed during the optimization process. Our architecture does not attempt to solve the general *phase-ordering problem* of compiler construction [Click and Cooper 1995]. In our current implementation, the various phases "know" their relative place in an ideal schedule, under the implied assumption that this can somehow be coordinated by extension providers. A newly loaded optimization component can inspect the set of already present phases and then install its constituent phases immediately before or immediately after any already existing phase. A single component can contain several phases that execute at different points in the time-line, and copies of the same phase can be inserted at multiple points in the time-line. Still, we acknowledge that this solution is sub-optimal, which is why the issue is revisited under the heading of "open questions" below.

4. THE PROFILER SUBSYSTEM

It has been almost thirty years since it was first realized that code quality could be improved by using feedback information (i.e., execution profiles) to guide optimizations [Ingalls 1971]. As processor complexity has been rising, the number of optimization decisions that a compiler must make has grown accordingly. Unfortunately, making the wrong optimization choices can seriously affect runtime performance. Access to profiling information makes it possible to base optimization decisions (such as which procedures to inline, which execution paths to favor during scheduling, and which variables to spill to memory) on actual measured performance data rather than on imprecise (and often ad-hoc) heuristics.

The most accurate profile of a program's execution can be obtained by *simulating* the processor under consideration as well as the relevant parts of the memory hierarchy at the gate level. However, this approach is hardly feasible in a system that aims to respond to changing user needs almost in real time. Hence, our approach considers only profiling techniques that are applicable in real-time situations. These fall into the three main categories of *instrumentation* [MIPS Computer Systems 1990; Ball and Larus 1994], *sampling* [Anderson et al. 1997; Zhang et al. 1997], and *hardware-based solutions* [Dean et al. 1997; Conte et al. 1996].

The last of these is obviously the most desirable and will become increasingly important in future microprocessors. As an example, while the 601 and 603 models of the PowerPC processor family [Motorola Inc. 1997] do not provide built-in profiling support, the PowerPC 604 is now equipped with a performance monitor. This performance monitor includes two 32-bit hardware counters that facilitate monitoring detailed events during execution, such as instruction dispatches, instruction cycles, misses in the cache, and load/store miss-latencies. The PowerPC 604e even includes four counters with augmented functionality.

For the foreseeable future, however, system builders will have to accept the fact that hardware profiling support is inadequate. Consequently, one has to rely on either or both of the other two techniques. Neither of them is fully appropriate for capturing the entire spectrum of profiling needs. On the one hand, instrumentation is well suited for generating exact path profiles [Ball and Larus 1996]; sampling techniques fail in this task because temporal information is lost in the statistical process. On the other hand, a sampling profiler can quite accurately pin down the set of instructions that miss in the data cache (the likelihood of the program counter hitting such an instruction is higher than the likelihood of hitting another instruction). An instrumenting profiler cannot usually determine whether an instruction missed in the cache, unless it is assisted by special hardware counters.

As a consequence, a sound profiling infrastructure has to support both sampling and instrumenting profilers, and be extensible to hardware-based profiling as it becomes available. This points toward an architecture that is surprisingly similar to that of the extensible optimizer introduced above. In our implementation, the profiling subsystem is composed of a profiling manager and a set of "plug-in" profiling components, as presented in Figure 4.

Just as in the optimizer, communication between the profiling manager and the installed profiling components is achieved by a broadcast mechanism via a message bus. Optimization components request profiling information by sending messages

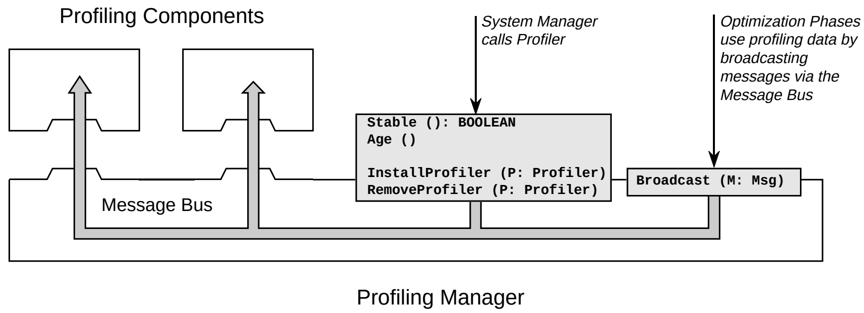

Fig. 4. Schematic View of the Profiler

to the profiling subsystem, which in turn delegates these requests to the appropriate *profiling component(s)*. As far as an optimization component is concerned, it is not relevant which profiling component actually processes its requests, as long as there is a component that does. This is the key to the evolvability offered by our solution. For example, when suitable hardware support for profiling becomes available, sampling and instrumenting profilers can be replaced by hardware-assisted ones simply by providing an appropriate profiling component that maps profiling requests directly onto the appropriate hardware counters.

Extensibility of the profiler in such a plug-and-play fashion also makes it possible to meet unanticipated requirements of future optimization phases: Suppose that a new plug-in optimization would require a specific kind of performance information that is not provided by the default profiling system. This problem can be solved easily by pairing the new optimization component with a dedicated profiling component that supplies it with the needed data.

Unlike our optimizer subsystem, the profiler doesn't provide a centralized database. This is because the kinds of data that the various profiling components collect and store are highly divergent in nature, and the availability of hardware-assisted profiling would eventually lead to an additional overhead for keeping the database in synch with the hardware counters. In our architecture, individual profiling components are autonomous and store their profiling data separately; in the current implementation, all of this information is kept entirely in memory. As is elaborated in the following, the profiling subsystem provides a centralized service for periodically and synchronously aging the information in this distributed database.

Component Interaction in the Profiler

A profiling component needs to adhere to a particular message protocol that is illustrated in Figure 5. In the following, we give an overview of the services that are associated with these messages.

The `AgeMsg` is broadcast to all profiling components when the system manager invokes the `Age()` service. This is done periodically to adjust and reduce the relevance of older profiling data. The implementation of aging is left to each individual profiling component, with exponential decay or linear decay being possible models.

The system manager also periodically checks whether the system's behavior

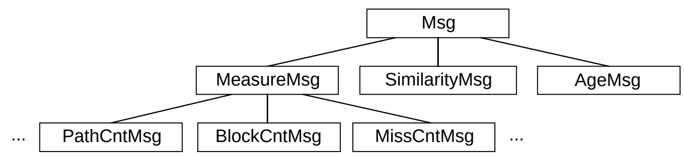

Fig. 5. Profiler Message Protocol

has shifted over time by calling the profiler subsystem's `Stable()` service. The `Stable()` service returns false if the profiling data has substantially changed since the last `Age()` request—otherwise it returns true. The profiler manager reacts to a `Stable()` request by broadcasting a `SimilarityMsg` to all profiling components. Individual profiling components react to this message by computing a similarity measure that reflects the degree of change in their profiling data for a given period of time. Section 5 discusses this similarity measure in more detail.

The primary means for optimization phases to communicate with the profiler is the message broadcast mechanism. Whenever an optimization phase requires profiling information, it creates an instance of a particular message and passes it to the profiling manager's `Broadcast()` service. In turn, the `Broadcast()` service distributes the message to all the installed profiling components. For each profiling event that needs to be monitored, a new message type is derived[4] from the common `MeasureMsg`. Examples include messages for measuring execution path frequencies (`PathCntMsg`) and data-cache miss rates (`MissCntMsg`). If a particular profiling component receives such a message and provides a service for that profiling event, it takes appropriate actions, otherwise the message is ignored.

Details about the request itself are encoded in the message and may contain one of three different request codes: (1) An optimization component may signal interest in a particular profiling event (e.g., "measure the execution frequency of path 14"), in which case the profiling component sets up auxiliary data structures to store the corresponding profiling data. It may also direct an optimization component to insert instrumentation code at the code location under consideration. (2) An optimization component may request profiling data for a particular event type (e.g., "return the current execution frequency for path 14"). And (3), an optimization component may signal that a specific set of profiling information is no longer needed (e.g., "the frequency of path 14 does not have to be measured any longer"). This is usually the case after optimizations have been performed. The profiling component then de-allocates auxiliary data structures and instructs an optimization component to remove previously installed instrumentation code.

Another significant property of our architecture is that new profiling components can easily be composed out of existing components, both vertically and horizontally. New components can share existing functionality via message forwarding. As an

[4]Note that this is an open-ended interface: the range of profiling events to be monitored cannot be determined in advance, as future optimization components might have requirements that simply cannot be anticipated. The way to solve this problem is by encapsulating the requests themselves as a "message objects" derived from a common superclass. This is also known as the *Command (233)* design pattern [Gamma et al. 1995].

example, we can construct a basic block profiler on top of a path profiler. Whenever the basic block profiler receives a `BlockCntMsg`, it creates and re-broadcasts a new `PathCntMsg` for each path that crosses the specified basic block. The basic block count is then computed by summing up the path counts for the individual paths. Neither does the new event profiler have to store additional data nor does it have to know implementation details of the path profiler.

The component-oriented architecture also facilitates a particularly elegant solution to the problem of constructing *instrumenting profilers*. Instrumentation involves modifying the actual machine code that is executed by the processor. The traditional approach has been to use binary rewriting tools [Eustace and Srivastava 1994] that insert instrumentation only after the final code images of programs have already been generated. Our solution, on the other hand, consists of structuring the instrumenting profiler as a closely coupled pair consisting of a profiling component and a corresponding optimization phase that inserts instrumentation code directly into the GSA representation of a procedure (Figure 6). If the instrumentation phase comes early enough in the optimization schedule, this has the beneficial effect that profiling instructions are automatically optimized in the context of the procedure, a very important advantage if low-overhead continuous profiling is an objective. Also note that the optimizer doesn't need to be modified in any way in order to support instrumentation—this is a natural capability provided by its component architecture.

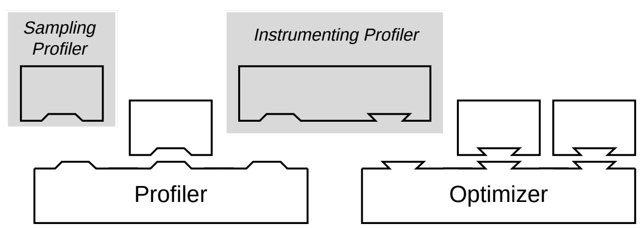

Fig. 6. Implementation of Sampling Profilers vs. Instrumenting Profilers

5. DECIDING WHEN TO OPTIMIZE, AND WHAT

One of the essential problems when performing optimizations at runtime is to decide *when* to optimize and *what* to optimize. Optimizing too little does not greatly improve runtime performance, optimizing too aggressively might lead to a situation in which the effort invested into code optimization is never fully recouped by faster-running application code [Hölzle and Ungar 1996].

This section addresses the first question, *when* to optimize. In our system, optimizations are initially performed when a program has been launched and enough profiling data has been gathered. Additionally, optimizations are reconsidered whenever the footprint of the profiling data changes substantially, i.e., when the user's behavior has shifted noticeably. In such a case, earlier optimizations may no longer align well with the current use of the system, and optimum performance may be restored only by performing optimization all over again.

In order to detect substantial changes in the user's behavior, we define a similarity measure S that reflects the degree of change of profiling data between two consecutive time steps $t-1$ and t. Each profiling component P logs n distinct values (such as a path counter or a basic block counter) that we represent as an n-dimensional vector $\vec{p}$, and is required to log these profiling values for at least the last two time steps. The similarity measure $S(P)$ can then be expressed as a function $S : P \rightarrow [0..1]$ that compares the captured data at time step $t-1$ (i.e., $\vec{p}_{t-1}$) with the captured data at time step t (i.e., $\vec{p}_t$). It returns a similarity value in the range $[0..1]$, whereas 0 denotes complete dissimilarity and 1 denotes complete data equivalence.

We first try to define $S(P)$ as a function that computes the geometric angle α between $\vec{p}_{t-1}$ and $\vec{p}_t$:

$$\alpha = \arccos \frac{\vec{p}_{t-1} \cdot \vec{p}_t}{|\vec{p}_{t-1}|\,|\vec{p}_t|}$$

This term has the advantageous property that it is independent of the time difference between $t-1$ and t since it measures the angle between the two vectors only and disregards the length of the vectors. However, it is not defined in the situation where $\vec{p}_{t-1} = \vec{0}$ and $\vec{p}_t = \vec{0}$. This is the case when the profiling database is first set up and initialized for a newly loaded application. To eliminate this problem, we adjust α by adding 1 to the denominator of the term. For simplicity reasons, we also remove the *arccos* function. The remaining term is still continuously descending and allows us to set a threshold for reconsidering new optimizations:

$$\alpha = \frac{\vec{p}_{t-1} \cdot \vec{p}_t}{|\vec{p}_{t-1}|\,|\vec{p}_t| + 1}$$

However, this function has further undesirable properties: It is very sensitive to small changes for short and low dimensional vectors $\vec{p}$. For example, if we measure the execution frequency of two paths, both paths have been executed once at time step $t-1$ ($\vec{p}_{t-1} = (1,1)$), and one path is executed once more between $t-1$ and t ($\vec{p}_t = (2,1)$), the resulting α suggests a considerable change in the profiling database—which of course is true, but an absolute change by only 1 should clearly not trigger a reoptimization. An optimal function should therefore disregard changes smaller than a given threshold. To achieve this, we define a second term β that reflects the absolute size of the change:

$$\beta = \frac{|\vec{p}_t - \vec{p}_{t-1}|}{\sqrt{n}}$$

Note that this term is independent of the dimension of $\vec{p}$ since the absolute change is divided by $\sqrt{n}$ (the unit vector of dimension n has length $\sqrt{n}$). We can now redefine the similarity function $S(P)$ as a combination of the angular component α and the length component β:

$$S(P) = e^{-\left(\frac{\beta}{c}\right)^k}(1-\alpha)+\alpha$$

As illustrated in Figure 7 for large vectors, the function still returns the geometric angle between the two vectors $\vec{p}_{t-1}$ and $\vec{p}_t$ since it strives towards α. For small vectors, however, the function strives towards 1 and is less sensitive to small changes as a result. It even completely disregards changes smaller than c—the constant c in the term was chosen to approximate the turning point of the function. By appropriately setting c, we can adjust the threshold above which changes in profiling data gets reflected in the similarity measure S (e.g., a procedure is only optimized if it has been executed at least 100 times in the last time period). Similarly to c, the constant k can be used to modify the slope of the function. We have found that a value of 8 performs quite well in practice.

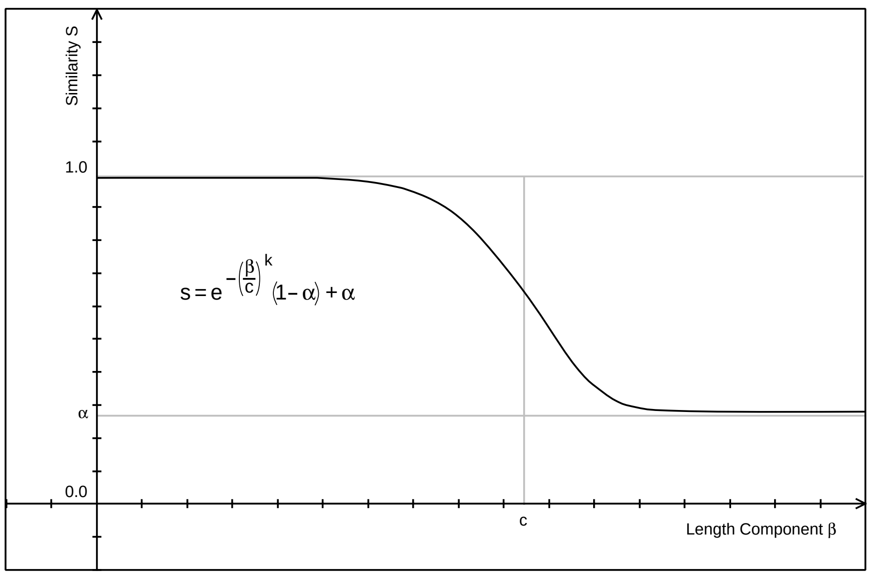

Fig. 7. Similarity Function

One more problem is still remaining, though: The function $S(P)$ always returns 1 for vectors of dimension 1. This can be circumvented elegantly by adding an additional component n to both $\vec{p}_{t-1}$ and $\vec{p}_t$ with $\vec{p}_{t-1,n} = m$, $\vec{p}_{t,n} = m$, and $m = max(\vec{p}_{t-1,0}, \ldots, \vec{p}_{t-1,n-1}, \vec{p}_{t,0}, \ldots, \vec{p}_{t,n-1})$.

In practice, we say that profiles have not changed as long as $S(P) = 1.0$. We reconsider existing optimizations when $S(P) < 0.95$.

6. DYNAMIC OBJECT LAYOUT ADAPTATION

Two comprehensive optimizations have been implemented and integrated within the framework described above. The first of these optimizations is a good representative of a newly emerging class of optimization techniques based on accurate dynamic profiling. Such techniques are almost ideally suited for a continuous optimization context, in which the available profiling information is particularly accurate because it refers to the current input set and the current user, rather than to a profile collected off-line.

Our particular optimization concerns data-cache performance. As the growth in raw processing power continues to outpace improvements in the storage hierarchy, memory performance is increasingly becoming a limiting factor of application speed. In recent years, compilers have begun to address this issue. For example, techniques have been developed to mask memory latency by fetching data ahead of time [Mowry et al. 1992], and program transformations such as cache-blocking, loop-skewing, and loop-tiling have been invented to increase data locality [Wolf and Lam 1991]. All of these optimizations are particularly effective in the domain of scientific computing, in which programs operate extensively on arrays. Unfortunately, they fare considerably worse in application domains in which most data structures are dynamically allocated and accessed via pointers. Applications of the latter kind include object-oriented and component-based programs.

The optimization presented in this section[5] increases memory performance specifically for pointer-centric applications. It is based on determining the best internal storage layout for dynamically allocated data structures and applies to programming languages that are fully type-safe. Examples of such languages include Java [Gosling et al. 1996] and Oberon [Wirth 1988]. These languages do not attach a semantic meaning to the declaration order of data members and do not expose the actual physical layout to the programmer; as a consequence, choosing an internal layout lies completely in the domain of the compiler.

The technique strives to *maximize spatial locality* of individual data members and hence is markedly different from traditional data-layout strategies that attempt to minimize the total space requirements of compound data structures [Muchnick 1997]. The traditional strategy is based on the assumption that a smaller memory footprint leads to faster applications, especially in garbage-collected environments. However, our work suggests that this assumption may be misleading. In some cases, increasing an object's size leads to a greater flexibility in placing data members, and thereby facilitates better cache performance. Our algorithm also specifically addresses the fact that there is a preferred bank-access ordering that needs to be observed to obtain optimal performance from interleaved memory. Object layout adaptation, in our implementation, is performed in two phases. First, *data member clustering* uses a simple strategy to partition the individual data members of a dynamically allocated data structure into aggregates that each fit into a single cache line. Then, after partitioning, *data member ordering* orders the data members that have already been mapped to a single cache line within the cache line to minimize load latency in case of a cache miss.

Data Member Clustering. In order to determine how to best partition the fields of an object into cache line sized aggregates, our optimization uses a simplified cost model that estimates the number of cache misses under the assumption that the number of cache misses is proportional to an execution time penalty. This

[5] A more detailed description of this optimization can be found in [Kistler 1999].

cost model is computed based on a *temporal relationship graph* (TRG) that, for a particular data object, captures information on how its fields are accessed. Similar graphs have been used in a variety of contexts [Chilimbi et al. 1999; Gloy et al. 1997]. In this graph, vertices correspond to fields, and they are connected by edges whose weights represent the degree of temporal dependency between the two connected fields. More concretely, the weights reflect the number of times that the two fields are accessed subsequently within a specific time interval. The weight is roughly proportional to the benefit of co-locating both fields on the same cache line, as this increases the probability that one field will already be in the cache as a result of accessing the other. The TRG is created by collecting path profiling information and then stepping through each program path returned by the profiler.

Once the TRG is created, the optimization searches for a multi-way graph partitioning of the temporal relationship graph such that the size of all partitions equals the size of a single cache line and such that the sum of all edges between the partitions is minimized. Finding an optimal multi-way partition for large graphs is an NP-complete problem. As such, there exists no known algorithm that solves the problem in polynomial time. However, a wide variety of heuristics-based approaches have been published in the last 30 years. One of the original papers by Kernighan and Lin describes a very efficient algorithm for *bipartitioning* (also called graph bisection) large graphs [Kernighan and Lin 1970]. Using a bipartitioning technique, we can solve the multi-way partitioning problem by recursive bisection, that is, we first obtain a 2-way partition of the graph that splits the graph into two equally-sized parts. We then further subdivide each part using 2-way partitioning. Several refinements of this algorithm have been described since, among them the improved version by Dutt [Dutt 1993] that can be implemented efficiently and which our own implementation is based on. There also exist more advanced algorithms based on multilevel partitioning schemes [Karypis and Kumar 1999]. However, since our graphs are usually small in size, the use of multilevel-partitioning algorithms does not seem justified.

Data Member Ordering. After partitioning, the data members that have already been mapped to a single cache line are ordered to minimize load latency in case of a cache miss. Two specific hardware characteristics of modern memory subsystems cause the ordering of fields on a line to be relevant, namely *memory interleaving* and *cache line-fill buffer forwarding*. Interleaving has an influence because it partitions the memory into banks that cannot be accessed equally efficiently. Modern memory controllers deliver data from a single row of memory in bursts and use a fixed sequence in which they distribute column addresses to the memory banks (two such banks in our example). There is a preferred memory bank that always receives the first column address cycle. If the read starts with a column address that is mapped to a different bank, then this first cycle is wasted. Hence, in order to achieve optimum performance, fields that have a high probability of causing a cache miss should come to lie at addresses that are mapped to the preferred memory bank. Therefore, during this ordering, a distinction is made between fields that are less likely to cause cache misses and those that are more likely to do so; the latter are placed at addresses mapped to the preferred bank of interleaved memory.

The second reason why the ordering of fields on a cache line has an influence on performance is related to the way that the cache is filled from memory. On most processors, the words on a cache line do not become simultaneously available after a cache miss has been serviced from memory. Rather, the cache line is filled in ascending memory address order, starting at the location that caused the cache miss, and "wrapping around" at the end of the cache line to load the remaining words. For example, consider a system in which the data bus is one word wide and a cache line holds eight such words. Now imagine that a read from address 003 causes a cache miss, resulting in a cache line being filled with the data stored in locations 000 through 007. The cache line would actually be filled in the order 003, 004, 005, 006, 007, 000, 001, 002; i.e., it would take at least seven additional cycles from the time at which the contents of location 003 become available until the contents of 002 become available also. On processors such as the PowerPC 604e [Motorola Inc. 1996] that forward the contents of the cache line-fill buffer to a requesting load unit immediately upon availability, it can hence make a difference whether the predominant memory access pattern is 003 followed by 002, or vice versa.

Finding an optimal ordering of fields within cache lines is done with an exhaustive search for the permutation of fields that minimizes a load latency cost associated with a particular permutation. The load latency cost considers both the effects of memory interleaving and cache line-fill buffer forwarding. Further, it also requires fields to be aligned properly. As an example, a double precision floating-point value that is not aligned to an 8-byte boundary results in a high cost value. Although our algorithm uses an exhaustive search technique, run-time is not a major problem in practice because the number of fields in a cache line is fairly small. Moreover, we use a smart branch-and-bound variant of the algorithm that is an order of magnitude more efficient than a naive implementation.

In our implementation, the optimization is fully automatic and operates at run-time on live data structures, guided by dynamic profiling data. Whenever the results of profiling suggest that a running program could benefit from data-member reordering, optimized versions of the affected procedures are constructed on-the-fly in the background. As soon as it is safe to do so, the dynamically generated code is substituted in place of the previously executing version and all affected live data objects are simultaneously transformed to the new storage layout. The program then continues its execution using the improved data arrangement, until profiling again suggests that re-optimization would be beneficial. Hence, storage layouts in our system are continuously adapted to reflect current access profiles. Since the technique presented here is fully automated, it does not involve programmers in the optimization process, but leaves them free to declare data members in any order whatsoever. It thereby elegantly de-couples software-engineering concerns from performance issues.

7. DYNAMIC INSTRUCTION RE-SCHEDULING

The optimization described in the previous section belongs to a new class of fully automatic techniques specifically designed to take advantage of live profiling data and continuous re-optimization. Many more traditional optimization techniques, however, have not specifically been designed for use in a continuous optimization infrastructure. Although it has been shown previously that traditional techniques can profit from profiling data (e.g, code placement, code scheduling, or register allocation [Pettis and Hansen 1990; Chang et al. 1991; Chang et al. 1992; Chen et al. 1994]), it is not immediately obvious that they can also noticeably benefit from continuous re-optimization. This section will present *dynamic trace scheduling*, an optimization that enables us to study the impact of our infrastructure on more traditional optimization techniques.[6]

Instruction scheduling has long been known as an effective compiler optimization technique for modern superpipelined processors. Instruction scheduling tries to exploit instruction level parallelism by statically reordering the instructions in a program but without invalidating program semantics. This is especially beneficial for in-order and VLIW processors that execute instructions in strict program order and are not very tolerant against pipeline stalls and cache misses [Chang et al. 1991]. Instruction scheduling can also be very effective for out-of-order processors. Although out-of-order processors already reorder independent instructions on-the-fly, this reordering is usually restricted in scope—the processor can only "see" a relatively small window of instructions at a time (e.g., 16 instructions on the PowerPC 604 [Motorola Inc. 1994]). Software scheduling techniques, in contrast, can be applied to whole program paths that contain several hundreds of instructions. A considerable gain in performance can therefore be expected for scheduling techniques that allow instructions to be scheduled across individual basic block boundaries.

Trace scheduling [Fisher 1981] is a technique that achieves this goal by coalescing basic blocks across branches into larger regions called *traces*. A trace is a sequence of basic blocks that is likely to be executed contiguously. However, control might leave the trace early at one or more exit points or control might enter into the middle of the trace from one or more side-entry points. Given a trace, instructions are then scheduled along this longer path rather than just on a single basic block at a time.

Since trace scheduling applies aggressive speculation to the important execution paths, possibly at the cost of degraded performance along other paths, the speed of the output code can be sensitive to the compiler's ability to accurately predict the important execution paths. Making effective use of profiling information in both the trace selection and trace compaction phase is thus extremely important for trace scheduling to be effective. In fact, our results suggest that trace scheduling based on live profiling data can increase program performance by more than 15% over trace scheduling based on static program analysis only.

Trace scheduling, in our implementation, is performed in three phases. First, the *trace selection phase* selects traces by identifying frequently executed program paths. Traces are then ordered into an execution schedule that matches the hardware resource constraints while still maintaining program semantics. This phase is called the *trace compaction phase*. Since inter-block code motions, such as moves below branches or moves above joins, might invalidate program semantics, these motions have to be made legal by compensating the code motions on the trace with insertion of copies for the moved operations in the off-trace paths. This phase is

[6] A more detailed description of this optimization can be found in [Kistler 1999].

called the *bookkeeping phase*.

Trace Selection. The trace selection phase of our optimization tries to find good sequences of basic blocks (i.e., traces) that are likely to be executed contiguously. A sequence has to fulfill two important criteria to be considered good. First, it has to be long and has to contain a large number of instructions. Only large traces provide the compactor with a potentially large enough pool of independent instructions and allow aggressive speculation. If the traces in a program are not naturally long enough, trace scheduling can benefit from trace enlargement techniques (e.g., branch target expansion, loop peeling, and loop unrolling [Chen et al. 1993]). Second, a trace is good only if the dynamic program flow often reaches the end of the trace. Traces that are likely to complete are preferable to traces that are exited before the end of the trace. This is because the most aggressive compaction algorithms aim to minimize the cycle count for the entire trace. Early exits further lower performance because instructions moved above early exits are wasted work.

Our algorithm selects traces as follows. First, it selects the most frequently executed trace by starting at the entry point of a procedure and then following the dominant fork at each branch until it reaches the end of the procedure. This is guaranteed to be the most frequently executed path since our optimizer transforms unstructured control flow into structured control flow in an earlier optimization phase [Brandis 1995]. For the same argument, it can also easily be proven that it always reaches the end of the procedure given that the control-flow graph is acyclic. The trace constructed in this manner satisfies our quality criteria; the trace is both long and the dynamic program flow always reaches the end of the trace. Once this trace is selected, the algorithm selects another start block and again follows the dominant fork at each branch until it reaches a basic block that has already been selected. At each step, the algorithm tries to pick the longest possible trace that is most likely to be executed. This process is repeated until all the basic blocks have been selected.

This algorithm can easily be extended to deal with cyclic control flow. Since the set of loops in Oberon is partially ordered under inclusion—loops are properly nested and sequenced—all the loops within a program can be scheduled separately, one at a time. For each loop L the algorithm first schedules all nested loops L' in L in a depth-first traversal and then considers the nested loops as single compacted instructions. Once nested loops are compacted, it proceeds with the same strategy outlined above for acyclic control-flow. The algorithm first finds the most frequently executed trace within the loop, usually the loop body. The algorithm then selects traces within the loop-exit paths until all the basic blocks have been selected.

Like any other heuristics-based optimization, trace scheduling relies on accurate probabilities of taking different forks for branches. Our implementation derives this information from live edge-profiles [Ball and Larus 1994] that are collected by our instrumenting edge-profiling component. Although branch probabilities could in principle also be derived from accurate path profiles (Young and Smith have shown that trace selection can be enhanced by using accurate path-profiles [Young and Smith 1998] rather than control-flow profiles [Chen et al. 1994]), this comes at an increased cost of profiling.

Trace Compaction. Once good traces have been selected, the trace compaction phase speculatively schedules instructions on each trace. This is done by moving instructions in a trace both above and below branches to achieve an efficient schedule. Our algorithm uses a *forward scheduling* approach that first schedules the roots of the dependency graph (usually load-instructions) at the earliest possible cycle and then moves downward through the graph to the leaves (typically store-instructions). Each instruction is scheduled as soon as it is data-ready, i.e., as soon as all definitions of its operands have been scheduled. If several instructions are data-ready concurrently, our algorithm applies several heuristics to select the best ordering of instructions. Among them are a dependence height heuristic (i.e., schedule instructions first that have a big dependence height), a liveness weights heuristic to avoid the problem of overscheduling [Warren 1990], a greatest uncovering heuristics (i.e., the instruction with the largest number of immediate successors is preferred), and a interlock heuristic (i.e., instructions that may cause interlocks with successors are scheduled as early as possible).

Common to these heuristics is that they are purely static and do not take advantage of live profiling information. However, profiling information can be extremely useful for scheduling instructions on traces, especially to avoid moving unnecessary instructions above high-probability exits in the trace. *Speculative yield* coupled with dependence height is a good profile-driven heuristic that addresses this problem [Fisher 1981]. It minimizes the number of instructions executed unnecessarily, by avoiding scheduling instructions early that do not contribute to high probability exits. However, one particular shortcoming of speculative yield is that there is nothing inherent in the heuristics that ensures that paths, which are shown by profiling data not to be important, do not get delayed unnecessarily. This leads to execution time degradation when those paths are really executed at run-time. Since the computation of yield values is based on single traces at a time, it cannot evaluate the effect of long-latency instructions on the trace on instructions following exit paths. In other words, it can assess whether a particular exit is delayed by moving an instruction above it but it can not assess whether the instructions following the exit are delayed by the code movement. To circumvent this problem, our implementation of trace compaction utilizes an additional heuristic that circumvents scheduling long-latency instructions above exits with a high probability. Long latency instructions include floating-point and integer division, as well as load instructions that are likely to miss in the cache.

8. EMPIRICAL EVALUATION

We have implemented the architecture described in Section 2 through Section 4 on top of Oberon System 3 [Wirth and Gutknecht 1992; Gutknecht 1994; Gutknecht and Franz 1999], and have integrated both dynamic object layout adaptation and dynamic trace scheduling into the framework for the PowerPC 604e [Motorola Inc. 1996]. The PowerPC 604e is a superscalar out-of-order processor with one branch processing unit, one condition register unit, two single cycle integer units, one multi-cycle integer unit, one floating-point unit, and one load/store unit. It contains a 32Kbyte four-way set-associative first-level data cache and first-level instruction cache, and a 1Mbyte unified second-level cache.

The empirical data presented in the following is based on a common set of bench-

Benchmark	Description	Program Size (Lines of Code)
TreeAdd	Sums the elements in a tree	166
Bisort	Sorts two disjoint bitonic sequences and then merges them	229
Health	Simulates the Columbian health care system	531
Jigsaw	Solves a jigsaw puzzle	341
BTrees	BTree library	1,343
Texts	Oberon text system	961
BLAS	Basic linear algebra subroutines (Vector/Vector, Matrix/Vector, Matrix/Matrix)	10,274
FTP	File transfer protocol client	4,690
DDD	Graphics rendering engine that implements a standard z-buffer rendering pipeline	2,250
MS	Monte carlo simulation of a constant stimulus design	2,470

Table III. Benchmark characteristics

marks, illustrated in Table III. The benchmarks include *TreeAdd*, *Bisort*, and *Health* from the Olden benchmark suite [Rogers et al. 1995] and *Jigsaw* from the WPI benchmark suite [Finkel et al. 1992]. These benchmarks have in common that each of them allocates many megabytes of data and represents frequently used operations on dynamic data structures. *BTrees* and *Texts* are fundamental shared libraries of the Oberon System 3 [Wirth and Gutknecht 1992; Gutknecht 1994; Gutknecht and Franz 1999] and are accessed by virtually every program running as part of any Oberon session. *BLAS* is a basic linear algebra reference library, providing subroutines for vector/vector, matrix/vector, and matrix/matrix operations [Dongarra et al. 1988]. In contrast to typical application programs, the latter three shared libraries are exposed to many different client contexts and various usage patterns. Reoptimization is thus particularly beneficial. [7] The remaining benchmarks represent programs from a variety of application domains: *DDD* is a 3D graphics rendering engine that implements a standard z-buffer rendering pipeline with texture mapping; and *MS* is a Monte Carlo simulation of a constant stimulus design.[8] All our benchmarks were executed multiple times, utilizing the PowerPC's performance monitor. The performance monitor includes four 32-bit hardware counters that record detailed events during execution, such as instruction

[7]Note that for optimizations such as our data layout improvement, specializing the library for different clients (i.e., creating a separate version of the library for each client) is not an option, because data structures can be shared across client boundaries, and in component-oriented systems often are shared in this manner.

[8]As others have noted [Truong et al. 1998], benchmarks such as *SpecInt95* are not ideal to measure performance gains for optimizations that target applications with poor data locality and large working sets. Consequently, we have used a non-standard suite of programs for our tests.

Benchmark	Statically Optimized	Object Layout Adaptation	Trace Scheduling
TreeAdd	0.99	1.15	1.09
Bisort	1.04	1.04	1.02
Health	1.01	1.00	1.01
Jigsaw	0.99	1.30	0.99
BTrees	1.01	1.96	1.00
Texts	1.00	1.69	1.01
BLAS	2.27	1.00	1.15
DDD	1.09	1.00	1.06
MS	1.06	1.00	1.01
Average	1.16	1.24	1.04

Table IV. Ideal performance speedup. The numbers under the heading "Statically Optimized" illustrate the performance increase of a statically optimized binary over an unoptimized binary. The numbers under the heading "Object Layout Adaptation" and "Trace Scheduling" illustrate the performance increase of a binary after performing static optimization techniques as well as the corresponding optimization technique over a purely statically optimized binaries.

dispatches, instruction cycles, misses in the cache, and load/store miss latencies.

Ideal Performance Speedup. Our first set of benchmarks (see Table IV) present an idealistic situation in which we compare a statically optimized program (performing the optimizations illustrated in Table II) with the same program after object layout adaptation and after dynamic trace scheduling—but without taking into account the costs of optimization and profiling.

Clearly, optimizing the data-layout for memory intensive programs is well worth the effort—program performance is increased by up to a factor of two over static optimizations. It is only for scientific applications and applications with no or small dynamic data structures that data-member reordering is less effective. This is in sharp contrast to trace scheduling, for which the speedups fall disappointingly short of our expectations. While scientific applications, such as *BLAS* routines, achieve a speedup of up to 15%, memory intensive applications do not profit from trace scheduling at all.[9] This can be explained by the fact that load-instructions with latencies of more than 100 cycles can seldom be scheduled sufficiently far ahead of their uses. And if instructions can be scheduled far ahead of their uses, register pressure is often increased as a result which causes additional spill-code to be inserted into the code. This cancels any potential performance gains achieved by re-scheduling. Table IV also leads to the suggestion that the static optimizer should perform trace scheduling by default rather than local scheduling. This is because trace scheduling performs well whenever static optimizations perform well, in most cases.

Profiling Costs. A more realistic picture that accounts for profiling costs is given in Figure 8 for object layout adaptation and in Figure 9 for dynamic trace schedul-

[9]These results might improve for in-order and VLIW architectures.

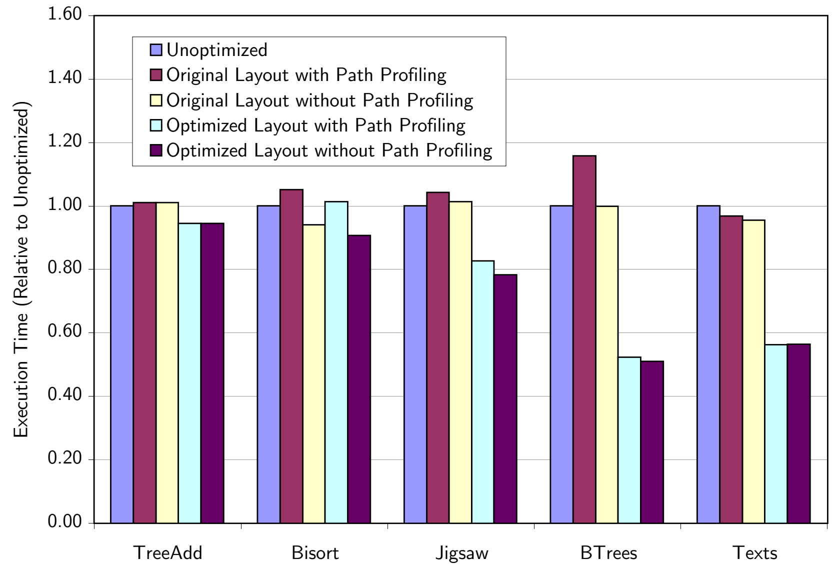

Fig. 8. Profiling costs. At load-time, the code-generating loader creates a first unoptimized version of the application. At run-time, the background optimizer performs basic code optimizations and instruments applications with path profiling—the original data layout is retained. If profiling information suggests that a different memory layout might increase performance, the storage layout of live data objects is modified.

ing. Figure 8 illustrates the different phases an executable proceeds through during object layout adaptation. As soon as the background re-optimizer commences its task, it performs a series of static optimizations on critical procedures and instruments them with path profiling code. The performance of the resulting executable is presented under the heading "Original layout with path profiling." Note that in the majority of benchmarks, performance is lowered due to the cost of profiling, which in these cases cannot be offset by static optimizations. In order to show the cost of the instrumentation, we also present the performance that this first optimized code image would have if there were no path profiling code. In actuality, the system never removes the profiling code.

After a while, the system has gathered enough path profiling information to be able to optimize the layouts of critical data structures. The resulting performance is presented as "Optimized layout with path profiling." Again, these figures include the overhead of profiling (but not the overhead of re-optimization itself), which is why we add separate numbers for timings without this overhead. In actual use, the profiling instrumentation is not removed because optimization is continuous: the system will keep monitoring the profiling data for apparent changes in system behavior, and if such a behavior is observed, will re-optimize the affected procedures. Note that this does not invalidate the original goals of continuous profiling: as can be seen in Figure 8, even with the profiling code left in, the optimized program is still faster than the optimized initial program.

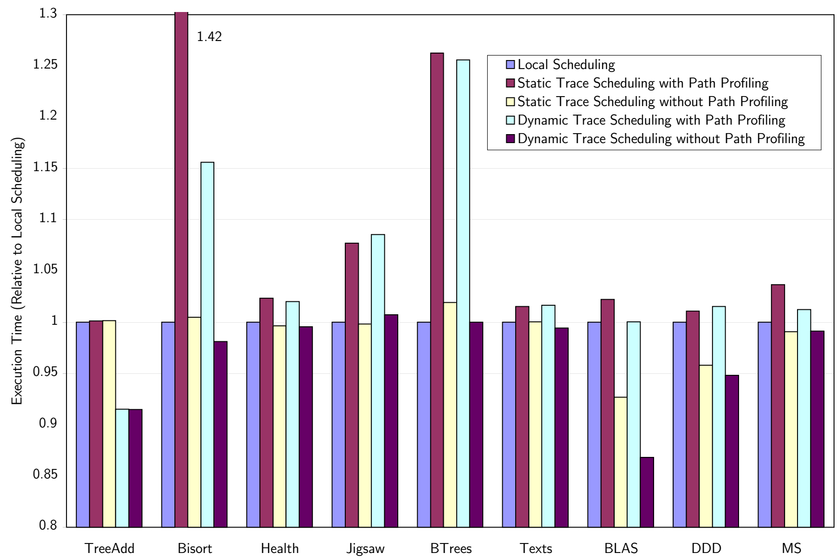

Fig. 9. Comparing performance for local scheduling, static trace scheduling, and profile-guided trace scheduling, both including and excluding profiling overhead.

Similarly, Figure 9 illustrates the cost-effects of profiling instrumentation in the case of dynamic trace scheduling. It compares program execution times of the local instruction scheduler to two different versions of trace scheduling. The first version, called "Static Trace Scheduling," is based on a static branch predictor and uses no profiling information at all. The second version, termed "Dynamic Trace Scheduling," is based on the profile-based heuristics described in Section 7. The results for the static and dynamic trace scheduler are given for both instrumented code ("... with Path Profiling") and uninstrumented code ("... without Path Profiling").

There are two important observations that can be made from Figure 9. First, profile-guided trace scheduling seems to perform noticeably better than static trace scheduling—at least in the case in which the profiling data matches the actual run-time behavior. With the exception of *Jigsaw*, the execution times for "Dynamic Trace Scheduling" are smaller than the execution times for "Static Trace Scheduling."

The second observation is more fundamental in nature and has implications on the design of our system. One of the important insights that our work has yielded (and that will be presented shortly) is that there is an added performance benefit if applications can dynamically adapt to changing user session patterns. Automatic adaptation, however, requires continuous profiling which is clearly not feasible for trace scheduling. Figure 9 illustrates that trace scheduling only pays off if profiling instrumentation is removed after performing the optimization—both trace scheduling variants cannot compete with the local scheduler if path profiling instrumentation is present in the executable. This is in direct contrast to object layout adaptation. For object layout adaptation, continuous profiling presents no major problem since the instrumented optimized code is still considerably faster than the uninstrumented unoptimized code.

So if continuous profiling is not feasible, how can we still react to behavioral changes? The solution to this problem is both simple and intuitive. Rather than profiling programs continuously, programs are profiled *periodically*. Profiling instrumentation is removed after optimization but periodically re-inserted into the executable to re-monitor the program's behavior for a short amount of time. If the profiling data suggests that the behavior has changed since the last instrumentation step, the code is re-scheduled. The instrumentation code is then removed again. Further, with the advent of cheaper or even free profiling techniques and profiling hardware support [Anderson et al. 1997; Wu et al. 1999], this problem will be of less concern in the future.

Profitability. Given the potential for both object layout adaptation and dynamic trace scheduling, is performing these optimizations at run-time worth the considerable effort? Can the time invested in code optimization ever be recouped by a faster running program? In order to answer this question, we first need to study how the "break-even" point is reached in a system such as ours—that is, if it is ever reached at all. As Figure 10 illustrates, the benefit of re-optimization is not simply the ratio of the resulting speedup and the combined overheads of profiling and code re-generation. This is because the speedup itself is achieved only after the re-optimization phase has completed: if the optimization was completed halfway through execution, then only half of its potential benefit could be realized. As shown in Figure 10, the first part of this cost is related to the fact that profiling information is not immediately available; we cannot circumvent having to execute the unoptimized version of the program for a while first to detect the hot spots in the program (O_0). This period of time is commonly referred to as "opportunity cost." Once hot spots are detected, there is a further price to pay for re-generating and fine-tuning the code. In some cases, we even need to insert additional path profiling instrumentation for these hot spots (C_0). Again, we have to run the new version of the program for a while until this information becomes available (O_1). Only then can we generate an even more optimized version of the program (C_1). Hopefully, i.e., if the program's overall run-time is sufficiently long, this cost is eventually recouped because the resulting program is significantly faster than the original.

Also note that the cost for the first optimization cycle is higher than the cost for subsequent optimization cycles. Ideally, since profiling instrumentation is never actually removed, subsequent optimization cycles do not have to pay the price for finding hot spots and inserting path profiling instrumentation any more (O_0 and C_0). In addition, the opportunity cost O_1 can be partially overlaid with the time it takes for the previous optimization cycle to pay off.

Hence, the "break-even" point of such an optimization process with $n-1$ phases can be represented by the following generic formula:

$$\textit{break-even point} = \frac{S_n \sum_{i=0}^{n-1} (C_i + O_i) - \sum_{i=0}^{n-1} O_i S_i}{S_n - 1}$$

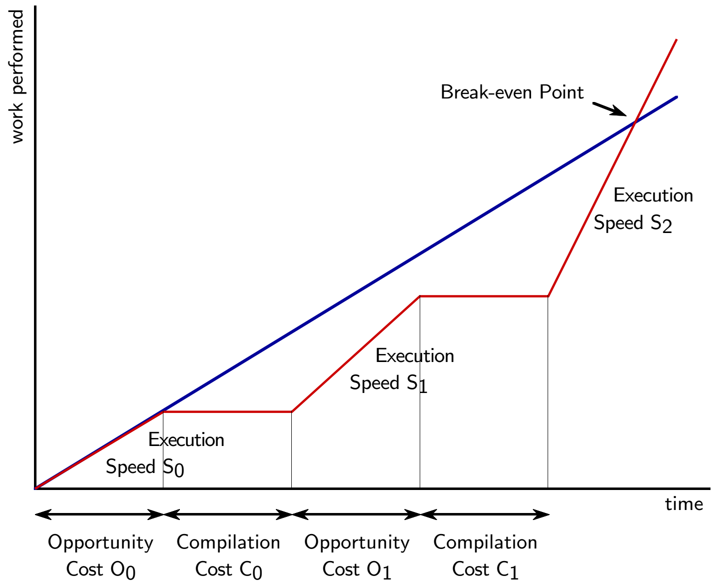

Fig. 10. Computing the break-even point

In the above formula, O_i denotes the opportunity cost at phase i (i.e., the time required to detect hot spots in the program), C_i denotes the optimization costs at phase i (i.e., the time required to optimize a hot spot), and S_i denotes the execution time ratio of the unoptimized unprofiled program over the current version of the program at phase i.

Based on the above formula, Table V and Table VI attempt to answer the question whether performing object layout adaptation and dynamic trace scheduling at run-time pays off. It lists the times required to optimize the individual benchmarks and the resulting "Break-even Points" for various opportunity costs. For example, assuming that the system requires one minute to collect enough profiling information before it can start the optimization, optimizing the storage layout of the Oberon shared text subsystem pays off after invoking text services for a total of 3.3 minutes. Similarly, if the system collects profiling information for one minute, re-scheduling the code for *TreeAdd* pays off after only 2.5 minutes. For larger shared libraries, such as the *BLAS* routines, re-scheduling pays off after 16.8 minutes of continuous execution. This is already a quite substantial period of time and re-scheduling might hence only be feasible in the context of long-running computationally intensive tasks. In addition, the break-even point for trace scheduling is—not surprisingly—too large to be practical for most other benchmarks. Consequently, our optimization performs a static analysis of the source program prior to performing dynamic optimizations. By limiting object layout adaptation to programs with dynamic data structures and by limiting trace scheduling to programs with a high number of nested loops and a high number of floating-point operations, optimizing unprofitable sections of code is avoided.

	Compilation Costs (s)		Break-even point (s) for various opportunity costs		
	C_0	C_1	O_i=60s	O_i=120s	O_i=180s
TreeAdd	0.4	16.2	430.0	561.0	692.0
Bisort	1.3	5.8	∞	∞	∞
Jigsaw	3.2	9.5	205.0	336.0	468.0
BTrees	19.1	50.9	276.0	404.0	533.0
Texts	5.4	30.2	198.0	316.0	433.0

Table V. Break-even point (in seconds). Illustrates the time required for the object layout adaptation to pay off. If the unoptimized program version ran longer than the break-even point, performing the data layout technique first and then running the optimized program version would perform better. The compilation cost C_0 includes the cost for applying standard optimizations to the application and inserting instrumentation utilized later by the memory optimization. C_1 includes the cost for reading the collected path profiling data and creating the TRG graph, computing the new memory layout and changing the layout of all live objects, as well as the cost for generating code for the new memory layout.

	Compilation Costs (s)		Break-even point (s) for various opportunity costs		
	C_0	C_1	O_i=60s	O_i=120s	O_i=180s
TreeAdd	1.1	1.2	148.0	269.0	349.0
Bisort	3.6	4.4	1462.0	2500.0	3192.0
Health	3.9	4.7	2254.0	2665.0	2938.0
Jigsaw	2.9	3.7	∞	∞	∞
BTrees	0.9	1.1	∞	∞	∞
Texts	5.5	6.8	2374.0	2649.0	2831.0
BLAS	36.5	79.9	1009.0	1138.0	1223.0
DDD	56.6	63.5	2439.0	2571.0	2659 .0
MS	19.6	24.3	5384.0	5743.0	5982.0

Table VI. Break-even point (in seconds): Illustrates the time required for the optimization to pay off. If the unoptimized program version ran longer than the break-even point, performing trace scheduling first and then running the optimized program version would perform better overall. See accompanying text for an explanation on how this values are computed. The compilation cost C_0 includes the cost for applying standard optimizations to the application and inserting instrumentation utilized later by the dynamic trace scheduler. C_1 includes the cost for reading the collected path profiling data and re-optimizing the application using the trace scheduler that is guided by the path profiles.

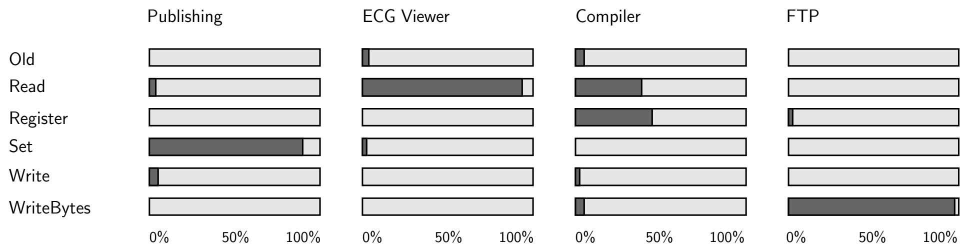

Fig. 11. Procedure execution times for the Oberon file service

Behavioral Mismatches. Another important insight that our work has yielded is that continuous, rather than do-it-once, optimization yields an added benefit. This is because a single piece of code is often put to several quite distinct uses over the course of a single user session lasting several hours, while at each moment the user's attention is usually focused on a relatively small number of current tasks. Figure 11 illustrates this in the case of the Oberon file system. It shows the usage pattern for various client applications, among them a publishing document editor, a medical application that displays patient images and image sequences taken by an echocardiograph (ECG), an Oberon compiler for the PowerPC, and a file transfer protocol client (FTP).

Clearly, different client applications place different loads on the shared library functionality. The *ECG Viewer*, as an example, primarily reads from files and does so in a sequential way. *Publishing* on the other hand spends most of its time positioning the reader in the source document whenever the user scrolls and moves the cursor to different positions in the document. Moreover, *FTP* primarily writes to files but never accesses any of the read interfaces.

Similarly, the distribution of execution paths and basic blocks within single procedures can change considerably for different execution scenarios. As an example, Listing 1 on the following page and Figure 12 depict the procedure *OpenFinder* from the text library and its control flow graph, respectively. The distribution of paths for various client applications is given in Figure 13. Here, the distribution of executed paths strongly varies with different client applications. While the Web-Browser falls straight through the procedure (path 1-11-24-34-46-13), other applications spend most of their execution time in the innermost loop of the procedure (path 24-34-44).

This raises the question of how a library optimized for one particular client performs when it is used with another one. To this effect, we performed a series of experiments that are summarized in Figure 14 and Figure 15. In the first experiment, we took an "original" data-structure layout in which the fields were arranged strictly in the order specified by the programmer in the source text with four layouts that were automatically obtained by our optimizer for four different uses of the *Texts* library, and correlated their performance across these four different usage scenarios. In order to simplify the comparison, the cost of re-generating the code itself has been disregarded. This is because this cost varies greatly depending on the order in which the layout of different types is modified. The cost of the first optimization cycle differs from that of subsequent ones, because the first cycle

```
(** Open Finder at position pos in T. The finder is automatically
advanced to the next object in text. *)
PROCEDURE OpenFinder* (VAR F: Finder; T: Text; pos: LONGINT);
  VAR p: Piece; org: LONGINT;
BEGIN
  FindPiece(T, pos, org, p);
  WHILE (p.f # Wfile) & (p.lib IS Fonts.Font) DO
    org := org + p.len; p := p.next
  END;
  F.pos := org; F.ref := p; F.eot := FALSE
END OpenFinder;
```

Listing 1. Text system: OpenFinder

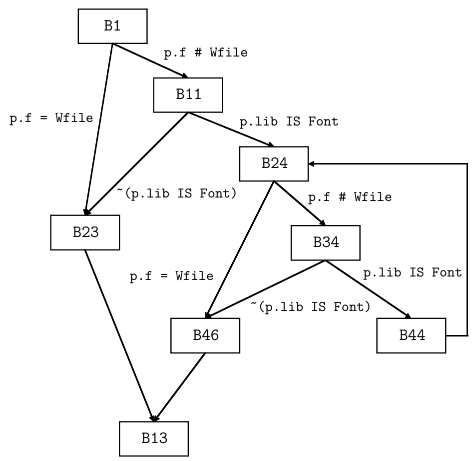

Fig. 12. Basic block diagram for Texts.OpenFinder

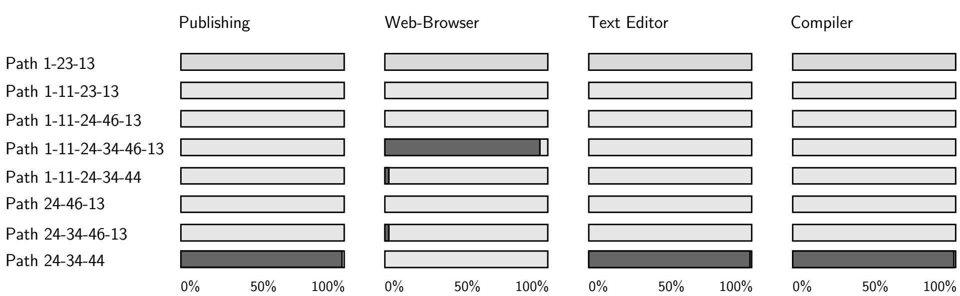

Fig. 13. Path frequencies for Texts.OpenFinder

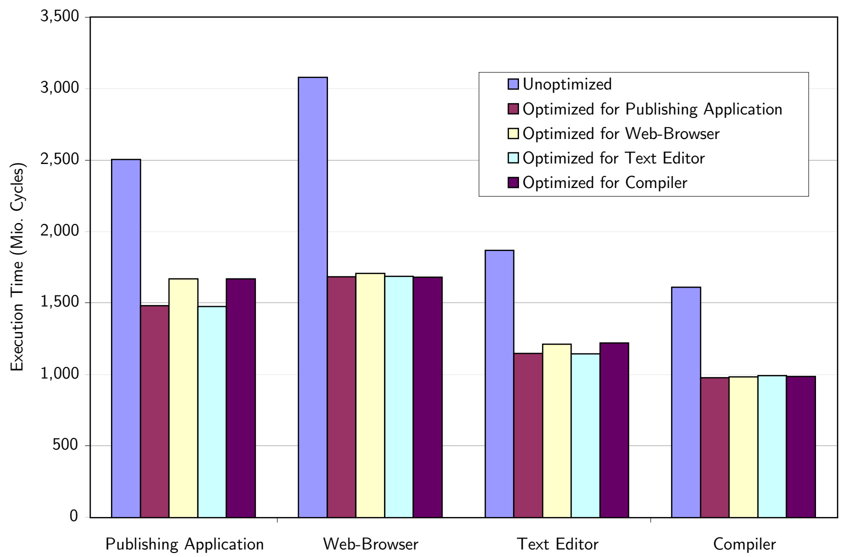

Fig. 14. Optimizing the *Texts* benchmark for different predominant access patterns.

additionally needs to insert profiling code whereas subsequent cycles do not.

As can be seen in Figure 14, there is clearly a difference in optimizing for different text services. For example, the publishing application is 13% faster using a text library that is custom-tailored for it rather than a library tailored for the compiler or the Web browser. Similarly, the text editor is 7% faster with its custom-tailored version of the library versus the compiler's custom-tailored version. These results also confirm the expected result that dynamic compilation is superior to static compilation, because it can adapt to multiple behavior patterns instead of just a single one.

The reason why custom tailoring yields an additional benefit in this particular case is that the text service supports not only plain sequences of characters, but also enriched documents that have elements such as images, buttons, and hyperlinks embedded within them. The fields that support these additional "floating text elements" are accessed relatively frequently when dealing with Web pages and using the publishing application. But source programs rarely contain any embedded elements, and hence the program editor and the compiler access the corresponding fields more infrequently than other clients of the text service. This results in two very different usage scenarios, one in which the corresponding fields are placed with other frequently accessed ones on the same cache line, and another in which they are "demoted" in favor of other data members.

In the second experiment, we evaluated the effect of profiling mismatches on dynamic trace scheduling. To this effect, we performed a series of tests with the procedure $DTBSV$ from the $BLAS$ benchmark for which trace scheduling evidently yields an added benefit. $DTBSV$ solves one of the equations $Ax = b$ or $A^T x = b$ where A is an $n \times n$ band matrix with multiple diagonals. Depending on the

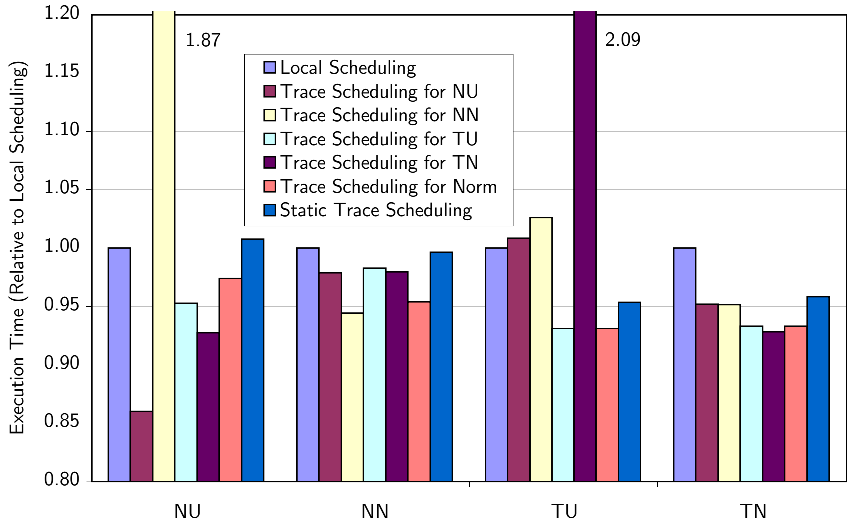

Fig. 15. Optimizing the procedure *DTBSV* from the *BLAS* library for different predominant execution patterns.

actual input parameters, A can either be a unit or a non-unit matrix, as well as upper or lower triangular. In order to evaluate the impact of profiling data mismatches, we compared the performance of the local instruction scheduler to the performance obtained by trace scheduling code for four different uses of *DTBSV*. We also correlated the performance across these four different usage scenarios. The four usage scenarios are solving the equation $Ax = b$ for unit triangular matrices ("NU"), solving $Ax = b$ for non-unit triangular matrices ("NN"), solving $A^T x = b$ for unit triangular matrices ("TU"), and solving $A^T x = b$ for non-unit triangular matrices ("TN"). The results of our experiments are summarized in Figure 15. For simplicity reasons, the overhead of path profiling and optimization is again not included in the benchmarks.

Figure 15 clearly shows that there is a benefit in optimizing for different path profiles. Throughout the results, the version optimized for the currently predominant execution pattern is noticeably faster than the versions optimized for another execution pattern. As an example, *DTBSV* executes roughly twice as fast with input parameters "NU" if it has been optimized for input parameters "NU" rather than for input parameters "NN." Similarly, *DTBSV* executes about twice as fast with input parameters "TU" if it has been optimized for input parameters "TU" rather than for input parameters "TN."

The second insight that Figure 15 yields is that profiling data that combines data from several different execution scenarios is inferior to accurate profiling data. The execution times for "Trace Scheduling for Norm" are regularly larger than the execution times for the variants that generate code for one particular execution scenario ("Trace Scheduling for NU/NN/TU/TN"). The former results are based on a scenario in which there is no predominant execution pattern and all parameters are equally likely to occur.

In any case, Figure 15 confirms our intuition that profile-guided trace scheduling performs better than static trace scheduling. Using profiling data—whether specialized for a particular set of parameters or not—performs better than using no profiling data at all ("Static Trace Scheduling").

9. FUTURE WORK

One of the essential problems of dynamic re-optimization is to decide whether the effort of optimization can be recouped by the faster running program in a reasonable amount of time. The last section has yielded some evidence that this might not always be the case. In certain situations, a system is better off not to optimize a given piece of code. Currently, our architecture assigns a hard-coded benefit estimate to each optimization phase (e.g., 5% speedup for data prefetching, 20% speedup for common subexpression elimination) upon which the system decides whether to perform optimization or not. This situation, however, is not always optimal as the benefit values are only estimates and thus inaccurate in many instances. The values also consider the program structure only to a limited degree, which often influences the outcome of optimization techniques. Loop unrolling, for example, increases the performance of loop-intensive programs by several orders of magnitude but barely affects straight-line programs.

In order not to haphazardly perform optimizations, more sophisticated solutions are desirable for future generations of architectures. *Program metrics* might emerge as one such solution. Program metrics reflect the structure of a program and can thus better be used to guide optimization decisions. Potential metrics include the ratio of load to store instructions, the ratio of floating point operations to integer operations, the number and depth of loops, or the frequency of procedure calls. Metrics might also capture information about memory accesses; whether data is accessed through arrays or dynamic data structures.

For memory optimizations, metrics about the program behavior rather than the program structure might be favorable. For example, certain types of optimizations (e.g., data-linearization prefetching) are only useful if dynamic data structures remain relatively constant at run-time. Beneficial metrics about memory behavior would hence yield information about whether a given data structure is predominantly static or dynamic. A good metric might also lead to insights into whether changes in the data structure mainly involve moving existing data objects around rather than adding new objects to it. Classification of live data structures into lists, trees, and dags might also improve on optimization decisions.

Several questions about program metrics remain to be explored. What is a minimal set of metrics that are beneficial for a wide variety of optimization decisions? Can program metrics be deduced by static program analysis, by dynamic program profiling, or only by a combination of the two?

It also remains to be explored how program metrics can be used to predict the potential of different optimization techniques in the case of a particular program. For independent optimizations O, we might be able to compute the potential benefit B_o based on a mapping of metrics values $v_0 \in V_0, \ldots, v_m \in V_m$ onto potential speedups, e.g.,:

$$B_o : V_0 \times \ldots \times V_m \rightarrow [0..1]$$

However, not all optimizations are mutually independent. Some optimizations have to be performed concurrently to yield a positive net speedup. For example, trace scheduling is very likely to perform much better in the presence of loop-unrolling. Likewise, some optimizations may disable other optimizations and must hence not be performed concurrently. In the presence of independent optimizations the above formula might have to be generalized to *sets of optimizations*.

$$B : \wp(\{O_0, \ldots, O_n\}) \times V_0 \times \ldots \times V_m \rightarrow [0..1]$$

Given a concrete implementation of B, the system favors the set of optimizations for which B is maximal. In theory, this calculation is of exponential complexity. In practice, however, we might be able to take advantage of the fact that a given optimization usually depends on only a few other optimization techniques. Hence, we might find the maximum by partitioning the set of optimizations $O_0, \ldots, O_n$ into partitions $P_0, \ldots, P_k$, where the P_i contain mutually independent optimization techniques, and by computing their maxima individually:

$$\max_{o \in \wp(\{O_0 \ldots O_n\})} B(o, v_0, \ldots, v_m) = \sum_{i=0}^{k} \max_{o \in P_i} B(o, v_0, \ldots, v_m)$$

Given the fact that compiler construction has always been an engineering discipline rather than an exact science, it is very unlikely that the above-mentioned problem will ever be solved using a general approach that applies equally well to all sorts of optimizations. More likely, particular solutions will evolve around carefully engineered solutions that are based on years of experience and collections of benchmark results.

10. RELATED WORK

Since 1996, when the project described in this paper was started, dynamic optimization has become a very active field of research with several approaches, different from our work, being investigated simultaneously.

The first fully automated system for runtime code optimization was described by Hansen [Hansen 1974]. Although it bore many structural similarities to our system, as well as to today's Java just-in-time compilers, it was markedly different from the more recent systems in that it used profiling data only to decide when to optimize and what to optimize, but not how to optimize. Consequently, the speedups achieved were inherently limited and could not exceed the speedups achieved with traditional static optimization techniques. In contrast, our system allows optimization techniques to take advantage of live profiling-data and to adapt to the user's behavior.

With the advent of object-oriented programming languages, several similar research projects were initiated with the explicit goal of making dynamic dispatches faster, reducing the overhead of garbage collection, and minimizing the overhead of thread synchronization; among them are the Smalltalk-80 system [Deutsch and Schiffman 1984], the Self-93 system [Hölzle 1994; Hölzle and Ungar 1996], the

HotSpot system, the Intel VTune system [Adl-Tabatabai et al. 1998], and the Jalapeño system [Alpern et al. 1999; Alpern et al. 1999]. Typical optimizations performed by these systems include run-time type feedback [Hölzle and Ungar 1994], message inlining [Dean and Chambers 1994], message splitting [Chambers 1992], polymorphic inline caches [Hölzle et al. 1991], customization [Chambers and Ungar 1989], and escape analysis [Choi et al. 1999]. In contrast, our work focuses primarily on traditional optimizations and on novel optimizations that specifically exploit live profiling data. Our work differs from these system in another aspect: Although all of these systems perform dynamic optimization, they do not (yet) perform unlimited dynamic re-optimization. Consequently, these systems can only adapt to changing user session patterns to a limited degree. For example, in the SELF system, methods were only re-optimized if not yet "optimal" (i.e., if a method contained sends that could not be inlined because of missing type information), but once a certain level of optimization had been achieved, the counters were removed and no further enhancements were possible. One of the main findings of this paper, however, is that recompiling even fully optimized code images in response to changes in profiling-data can give rise to real performance improvements.

Most of the above-mentioned systems—including ours—are based on source code optimizations. In contrast, *binary translation systems* operate on binary images directly. This prevents certain highly aggressive optimization techniques to be performed at runtime (e.g., the memory optimization technique described in this paper) but allows optimizing legacy applications whose source code is no longer available. An example of such a system is Hewlett Packard's Dynamo project [Bala et al. 1999] that optimizes HP PA-RISC binaries in-flight. Often, however, binary translation systems not only optimize a binary for a given instruction set, but translate it to or emulate it on an entirely different instruction set. Digital FX!32 [Hookway and Herdeg 1997] uses such an approach to enable the execution of Intel x86 applications on Alpha microprocessors. Similarly, Hewlett Packard's Aries [Zheng and Thompson 2000] system optimizes and translates Intel x86 code into Intel IA64 code, IBM's BOA system [Gschwind et al. 2000] translates PowerPC code into instructions for a smaller but faster BOA processor, and Transmeta's code morphing software [Klaiber 2000] facilitates the execution of Intel x86 code on a fast but low-power VLIW processor.

There are many differences between binary translation systems and our work. First, our system is based on a type-safe transportation and intermediate format. This allows for much more aggressive optimizations since limiting issues such as self-modifying code or precise exception handling do not arise. Second, for some of the binary translation systems (e.g., Digital FX!32), profiling information is used only to determine which program parts to translate, but not to guide optimizations. The optimized code can hence never surpass statically optimized code. Third, the flexibility of binary translation systems is often limited by the fact that code images are only optimized once and cannot be undone or redone. Since the profiles encountered in the first run of the application may considerably deviate from the profiles collected in successive application runs, code is not necessarily optimal for the predominant execution patterns. Even worse, Digital FX!32, as an example, cannot immediately react to critical performance bottlenecks at all since code is rewritten only after an application quits. For applications that run very infrequently or only once, this is a stringent limitation.

Continuous optimization has also been studied for systems providing incremental ("staged") specialization of an already executing program at run-time [Engler et al. 1996; Lee and Leone 1996; Marlet et al. 1999; Grant et al. 1999], based on manual annotation of the source program by a skilled programmer. In these approaches, a static compiler constructs a dedicated run-time code-generator that is able to dynamically create variants of the program to be executed, specialized depending on actual input data. In contrast to our background re-optimization engine, which is a full-fledged optimizing compiler, these dedicated code generators are much simpler and operate on the complexity level of macro expansion. Value-specific optimizations also have the disadvantages of involving the programmer in that he or she has to explicitly identify the bottlenecks in the application. The potential benefits of such optimizations hence are highly dependent on the skill level of the programmer. To this date, it is not clear how making ill-chosen annotations affects run-time performance. Further, value specific optimizations seem to be limited to a very narrow application domain that includes simulations, code interpretation, and event dispatching.

Finally, several optimization techniques have been described that are similar in spirit to either object layout adaptation or dynamic trace scheduling. Among them are field reorganization techniques [Truong et al. 1998; Chilimbi et al. 1999], object co-location techniques [Chilimbi and Larus 1998], object clustering and coloring techniques [Chilimbi et al. 1999], object placement techniques [Calder et al. 1998], and profile-driven scheduling strategies [Chen et al. 1993; Chen et al. 1994; Deitrich and Hwu 1996; Chekuri et al. 1996].

11. CONCLUSION

This paper has presented a study of a system that provides code generation and continuous code optimization as a central system service. The system constantly monitors the system's state and re-performs optimizations as needed to achieve a closer match between the executing software and the available hardware resources. The system not only continuously adapts to the user's behavior but also eliminates some of the most severe performance problems found in today's software systems caused by hardware/software mismatches, software components, and portable code.

This paper has also presented two optimization techniques that are early representatives of an emerging class of code optimizations that are applicable to programs that are already running. *Object layout adaptation* improves the storage layout of dynamically allocated data structures. It is based on a two-tiered strategy that first assigns fields to cache lines and then optimizes the order of fields within individual cache lines. *Dynamic trace scheduling* improves the instruction level parallelism for predominant execution patterns by continuously adapting the instruction schedule to the most frequently executed program paths.

Our results have shown that—because of the profiling feedback loop—object code produced by continuous optimizations is often of a higher quality than can be achieved using static "off-line" compilation. Optimizations at runtime, if performed judiciously, can often surpass optimizations performed at compile-time, independent of whether the latter are guided by profiling information or not. Our results have also given evidence that re-optimizing an already running program in response to changes in user behavior can give rise to real performance improvements. The main beneficiaries of such re-optimizations are shared libraries, which at different times can be optimized in the context of the currently dominant client application.

ACKNOWLEDGEMENTS

We would like to thank Peter Fröhlich, Ziemowit Laski, Craig Snider, and Christian Stork who provided many helpful comments.

REFERENCES

Adl-Tabatabai, A.-R., Cierniak, M., Lueh, C.-Y., Parikh, V. M., and Stichnoth, J. M. 1998. Fast, Effective Code Generation in a Just-in-Time Java Compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*. Montreal, Canada, 280–290.

Alpern, B., Attanasio, C. R., Barton, J. J., Cocchi, A., Hummel, S. F., Lieber, D., Ngo, T., Mergen, M., Shepherd, J. C., and Smith, S. 1999. Implementing Jalapeño in Java. In *Proceedings of the ACM SIGPLAN '99 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*.

Alpern, B., Cocchi, A., Lieber, D., Mergen, M., and Sarkar, V. 1999. Jalapeño—a Compiler-Supported Java Virtual Machine for Servers. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99)*. Atlanta, Gorgia.

Anderson, J. M., Berc, L., Dean, J., Ghemawat, S., Henzinger, M., Leung, S.-T. A., Sites, D., Vandevoorde, M., Waldspurger, C., and Weihl, W. E. 1997. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. St. Malo, France. Also published as Technical Note 1997-016. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, July 1997.

Bala, V., Duesterwald, E., and Banerjia, S. 1999. *Transparent Dynamic Optimization: The Design and Implementation of Dynamo*. Tech. Rep. HPL-1999-78, Hewlett Packard Laboratories. June.

Ball, T. and Larus, J. 1994. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16,* 3, 1319–1360.

Ball, T. and Larus, J. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Paris, France.

Ball, T., Mataga, P., and Sagiv, M. 1998. Edge Profiling versus Path Profiling: The Showdown. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. San Diego, California, 134–148.

Brandis, M. 1995. *Optimizing Compilers for Structured Programming Languages*. Ph.D. thesis, Institut für Computersysteme, ETH Zürich.

Calder, B., Krintz, C., John, S., and Austin, T. 1998. Cache-Conscious Data Placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, California, 129–149.

Chambers, C. 1992. *The Design and Implementation of the* Self *Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.D. thesis, Stanford University.

Chambers, C. and Ungar, D. 1989. Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI)*. Portland, Oregon, 146–160.

Chang, P. P., Chen, W. Y., Mahlke, S. A., and Hwu, W.-M. W. 1991. Comparing Static and Dynamic Code Scheduling for Multiple-Instruction-Issue Processors. In *Proceedings of the 24th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Albuquerque, New Mexico, 25–33.

Chang, P. P., Mahlke, S. A., Chen, W. Y., and Hwu, W.-M. W. 1992. Profile-Guided Automatic Inline Expansion for C Programs. *Software—Practice and Experience 22,* 5 (May), 349–369.

Chang, P. P., Mahlke, S. A., and Hwu, W.-M. W. 1991. Using Profile Information to Assist Classic Code Optimizations. *Software—Practice and Experience 21,* 12 (Dec.), 1301–1321.

Chekuri, C., Johnson, R., Motwani, R., Natarajan, B. K., Rau, B. R., and Schlansker, M. 1996. Profile-Driven Instruction Level Parallel Scheduling with Applications to Super Blocks. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 58–67.

Chen, W. Y., Mahlke, S. A., Warter, N. J., Anik, S., and Hwu, W.-M. W. 1994. Profile-Assisted Instruction Scheduling. *International Journal of Parallel Programming 22,* 2 (Apr.), 151–181.

Chen, W. Y., Mahlke, S. A., Warter, N. J., Hank, R. E., Bringmann, R. A., Anik, S., and Hwu, W.-M. W. 1993. Using Profile Information to Assist Advanced Compiler Optimization and Scheduling. In *Advances in Languages and Compilers for Parallel Processing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Pitman Publishing, London.

Chilimbi, T. M., Davidson, B., and Larus, J. R. 1999. Cache-Conscious Structure Definition. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*. Atlanta, Georgia, 13–24.

Chilimbi, T. M., Hill, M. D., and Larus, J. R. 1999. Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*. Atlanta, Georgia, 1–12.

Chilimbi, T. M. and Larus, J. R. 1998. Using Generational Garbage Collection To Implement Cache-Conscious Data Placement. In *Proceedings of the First International Symposium on Memory Management*. Vancouver, 37–48.

Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S. 1999. Escape Analysis for Java. In *Proceedings of ACM SIGPLAN '99 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. Denver, Colorado.

Click, C. and Cooper, K. D. 1995. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS) 17,* 2 (Mar.), 181–196.

Conte, T. M., Menezes, K. N., and Hirsch, M. A. 1996. Accurate and Practical Profile-Driven Compilation Using the Profile Buffer. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Paris, France, 36–45.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (Oct.), 451–490.

Dean, J. and Chambers, C. 1994. Towards Better Inlining Decisions Using Inlining Trials. In *Conference on Lisp and Functional programming*. LISP Pointers, 273–282.

Dean, J., Hicks, J. E., Waldspurger, C. A., Weihl, W. E., and Chrysos, G. 1997. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 292–302.

Deitrich, B. L. and Hwu, W.-M. W. 1996. Speculative Hedge: Regulating Compile-Time Speculation Against Profile Variations. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Paris, France, 70–79.

Deutsch, L. P. and Schiffman, A. M. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages (POPL)*. Salt Lake City, Utah, 297–302.

Dongarra, J. J., Croz, J. D., Hammarling, S., and Hanson, R. J. 1988. An Extended Set of Fortran Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Transactions on Mathematical Software 14*, 18–32.

Dutt, S. 1993. New Faster Kernighan-Lin-Type "Graph-Partitioning Algroithms". In *Proceedings of the IEEE/ACM International Conference on CAD*.

Engler, D. R., Hsieh, W. C., and Kaashoek, M. F. 1996. 'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. St. Petersburg Beach, Florida, 131–144.

Eustace, A. and Srivastava, A. 1994. *ATOM: A Flexible Interface for Building High Performance Program Analysis Tools.* Tech. Rep. TN-44, Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA. July.

Finkel, D., Kinicki, R., Lehmann, J., and CaraDonna, J. 1992. *Comparisons Of Distributed Operating System Performance Using The WPI Benchmark Suite.* Tech. Rep. CS-TR-92-2, Worcester Polytechnic Institute.

Fisher, J. A. 1981. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers C-30,* 7, 478–490.

Franz, M. 1994. *Code-Generation On-the-Fly: A Key to Portable Software.* Ph.D. thesis, Institut für Computersysteme, ETH Zürich.

Franz, M. and Kistler, T. 1997. Slim Binaries. *Communications of the ACM 40,* 12 (Dec.), 87–94. Also published as Technical Report TR 96–24, Department of Information and Computer Science, University of California, Irvine, June 1996.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. 1995. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Gloy, N., Blackwell, T., Smith, M. D., and Calder, B. 1997. Procedure Placement Using Temporal Ordering Information. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 303–313.

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

Grant, B., Philipose, M., Mock, M., Chambers, C., and Eggers, S. J. 1999. An Evaluatioin of Staged Run-Time Optimization in DyC. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI).* Atlanta, Georgia, 293–304.

Gschwind, M., Altman, E. R., Sathaye, S., Ledak, P., and Appenzeller, D. 2000. Dynamic and Transparent Binary Translation. *IEEE Computer Magazine 33,* 3 (Mar.), 54–59.

Gutknecht, J. 1994. Oberon System 3: Vision of a Future Software Technology. *Software—Concepts and Tools 15,* 1, 26–33.

Gutknecht, J. and Franz, M. 1999. Oberon With Gadgets: A Simple Component Framework. *Object-Oriented Application Frameworks 2.*

Hansen, G. J. 1974. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs.* Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University.

Hölzle, U. 1994. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming.* Ph.D. thesis, Department of Computer Science, Stanford University.

Hölzle, U., Chambers, C., and Ungar, D. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP).* Geneva, Switzerland, 21–38.

Hölzle, U. and Ungar, D. 1994. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI).* Orlando, Florida, 326–336.

Hölzle, U. and Ungar, D. 1996. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. *ACM Transactions on Programming Languages and Systems 18,* 4 (July), 355–400.

Hookway, R. J. and Herdeg, M. A. 1997. DIGITAL FX!32: Combining Emulation and Binary Translation. *Digital Technical Journal 9,* 1 (Jan.), 3–12.

Ingalls, D. 1971. The Execution Time Profile as a Programming Tool. In *Design and Optimization of Compilers.* Prentice-Hall, 107–128.

Karypis, G. and Kumar, V. 1999. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing 20,* 1 (Jan.), 359–392.

Kernighan, B. W. and Lin, S. 1970. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 291–307.

Kistler, T. 1999. *Continuous Program Optimization.* Ph.D. thesis, Department of Information and Computer Science, University of California, Irvine.

Klaiber, A. 2000. The Technology behind Crusoe Processors. Transmeta Corportation.

Lee, P. and Leone, M. 1996. Optimizing ML with Run-Time Code Generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*. Philadelphia, Pennsylvania, 137–148.

Marlet, R., Consel, C., and Boinot, P. 1999. Efficient Incremental Run-Time Specialization for Free. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*. Atlanta, Georgia, 281–292.

MIPS Computer Systems. 1990. *UMIPS-V Reference Manual (Pixie and Pixstats)*.

Motorola Inc. 1994. *PowerPC 604: RISC Microprocessor User's Manual*.

Motorola Inc. 1996. *PowerPC: Addendum to PowerPC 604 RISC Microprocessor User's Manual: PowerPC 604e Microprocessor Supplement and User's Manual Errata*.

Motorola Inc. 1997. *PowerPC Microprocessor Family: The Programming Environments*.

Mowry, T. C., Lam, M. S., and Gupta, A. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Boston, Massachusetts, 62–73.

Muchnick, S. S. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufman.

Pettis, K. and Hansen, R. C. 1990. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI)*. White Plains, New York, 16–27.

Rogers, A. M., Carlisle, M. C., Reppy, J. H., and Hendren, L. J. 1995. Supporting Dynamic Data Structures on Distributed-memory Machines. *ACM Transactions on Programming Languages and Systems (TOPLAS) 17,* 2 (Mar.), 233–263.

Truong, D. N., Bodin, F., and Seznec, A. 1998. Improving Cache Behavior of Dynamically Allocated Data Structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Paris, France, 322–329.

Warren, Jr., H. S. 1990. Instruction Scheduling for the IBM RISC System/6000 Processor. *IBM Journal of Research and Development 34,* 1 (Jan.).

Wirth, N. 1988. The Programming Language Oberon. *Software Practice and Experience 18,* 7 (July), 671–690.

Wirth, N. and Gutknecht, J. 1992. *Project Oberon*. Addison-Wesley.

Wolf, M. E. and Lam, M. S. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Ontario Canada, 30–44.

Wu, Y., Lee, Y.-F., and Wang, H. 1999. An Efficient Software-Hardware Collaborative Profiling Technique for Wide-Issue Processors. In *Proceedings of the Workshop on Binary Translation*. Newport Beach, CA.

Young, C. and Smith, M. D. 1998. Better Global Scheduling Using Path Profiles. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 115–126.

Zhang, X., Wang, Z., Gloy, N., Chen, J. B., and Smith, M. D. 1997. System Support for Automatic Profiling and Optimization. In *Proceedings of the 16th ACM Symposium of Operating Systems Principles (SOSP)*.

Zheng, C. and Thompson, C. 2000. PA-RISC to IA-64: Transparent Execution, No Recompilation. *IEEE Computer Magazine 33,* 3 (Mar.), 47–52.