

Symbiotic Systems Software: Fast Operating Systems for Fast Applications

Margo Seltzer, Christopher Small, and Michael D. Smith
{margo,chris,smith}@eecs.harvard.edu

Abstract

Historically, advances in compiler technology have been driven by the characteristics of applications, particularly those that comprise the SPEC benchmark suite. To achieve high performance, many of the most promising compilation techniques rely on accurate profile information to direct the optimization and instruction scheduling process. Prior studies have shown that for these applications it is possible to generate representative data sets that are suitable for profile-based compilation algorithms. Though we would like to apply sophisticated compiler techniques to operating systems (and eventually other multi-threaded software systems such as database management systems), we are presented with two significant problems. First, the process of profiling the operating system is more difficult than profiling an application, because of the complexity of the operating system and the interaction between the operating system and its applications. Second, even when one can profile the operating system, there seems to be no general agreement of what constitutes a representative data set for such a large and complex system. This implies that the optimization of the operating system in isolation is not the correct approach. A better approach for improving the combined system and application performance is to allow the compiler to *symbiotically* optimize the operating system in the context of the applications that use it. This paper proposes a structuring of the compiler and operating system so that it is possible to perform this type of symbiotic, cross-address space compile-time optimization.

1 Introduction

There has been tremendous progress in compile-time optimization in the past 15 years. Performance gains are achieved by reducing the number of cycles required to execute a piece of code. This is done by removing unnecessary instructions and by tailoring the instruction schedule to better manage the available hardware resources. Recently, sophisticated optimizations such as blocking [Lam91], global instruction scheduling [Hwu93, Lown93], code layout [Pett90], procedure inlining [Chang91], and partial-redundancy elimination

[Brig94] have been shown to improve application performance by 20-300%. Most of these studies have focused on applications such as the SPEC benchmarks, where the majority of the execution time is spent in user-level code. Yet, there is nothing intrinsic to these sophisticated compiler techniques that makes them applicable only to user-level code.

The key to these optimization techniques is an in-depth understanding of the way in which the code being optimized is used, because optimizations such as these can, in fact, degrade performance if improperly applied. For example, the aggressive movement of instructions across basic block boundaries results in the production of compensation code, the duplication of code to maintain program semantics [Fish81]. Instruction scheduling is NP-complete, and hence the global movement of code is driven by heuristics that optimize a particular path through the code, possibly at the expense of the performance of other paths. The heuristics assume that the optimized path has a much higher probability of execution than other paths, and so, the transformation will result in an overall performance win. If the assumption is incorrect, performance suffers. One can make similar arguments for the other optimization techniques mentioned above.

A compiler can attain a rudimentary understanding of program behavior through static analysis of common program structures, such as loop structures. Though excellent work has been done in program-based analysis for generation of program execution characteristics (such as branch bias [Ball93]), these methods do not yet approach the accuracy of profile-based analysis. Through profile information generated by representative training data sets, the compiler can quite accurately predict the behavior of a program [Fish92], at least for applications like those found in SPEC. In other words, profiling the behavior of the application with one data set provides a good indication of how best to optimize the application for other data sets. Though there exist many powerful tools for profiling applications, we unfortunately lack good tools for profiling the operating system. Early tools required specialized hardware [Nagle92], and only recently have more general tools

become available [Sriv94][Rose95]. Even with these tools, accurate profiling of the operating system is very difficult because of its intrinsic operation and its existing structure.

Application profiling is typically implemented by instrumenting the application code, causing *time dilation*, where the instrumented code runs more slowly, relative to external events, than the uninstrumented code. Time dilation is not usually a problem for user-level tracing because most applications are insensitive to the timing of external events. However, the *raison d'être* of an operating system is to respond to external events. Its execution path is determined by synchronous requests (from applications) and asynchronous requests (from disks, timers, and networks). Because instrumentation alters this execution path, it becomes extremely difficult to gather meaningful data. Instrumentation perturbs the behavior of the operating system functions that we are trying to measure.

It is difficult to interpret data from a profiled kernel, because kernel code is invoked by multiple applications. This sharing obscures the correspondence between kernel behavior and the application that causes it. We can insert additional instrumentation into the kernel to keep track of which process is responsible for each invocation of a traced kernel function, but this increases the perturbation of kernel code, and increases problems such as time dilation.

Even with a detailed operating system simulator, such as SimOS [Rose95], it is unclear that a representative workload exists. The application mix, i.e. the precise interleaving of those applications, and the timing of external events all determine what an operating system does and when it has to do it. The activities in the operating system therefore may vary significantly from the execution of one application to the next.

The one glimmer of hope for overcoming these obstacles results from the observation that we do not need to find a representative workload for all applications. If the overall system goal is to achieve the best application performance, the correct decisions for optimizing the kernel can only be made in the context of a particular application. This implies that the best approach is one where we design the operating system and compiler in parallel. This approach mirrors the efforts in compilation and instruction set design during the development of RISC. Previous to this development, compilers and architectures were designed in isolation, as is done with compilers and operating systems today.

RISC proponents found that the cooperative design of the compiler and the instruction set architecture led to systems that exhibited superior application performance.

We believe that this same approach should be applied to the development of system and application software. Applications and operating systems should not be optimized in isolation, but instead *symbiotically* (and possibly continuously). To maximize the performance of an application, it is necessary to optimize the operating system in the context of that application. We propose to construct an operating system, compilation tools, and profiling tools that allow us to determine which sections of the operating system contribute to the performance characteristics of an application, and specialize and optimize them to benefit that application.

The rest of this paper is organized as follows. Section 2 discusses some opportunities for compiler optimization in the operating system. Section 3 discusses operating system architectures and how well these architectures are suited to aggressive compiler optimization. Section 4 outlines the approach we are taking to symbiotic compiler and system software development, and Section 5 concludes.

2 Compile-time Opportunities

By maintaining the context of an application when execution moves into the operating system, many existing optimizations become much more fruitful. Assuming we can compile systems in the context of an application, some opportunities for performance improvements are listed below.

- *Constant propagation* across procedure calls has been shown useful within a single address space [Hall95]. Similar benefits can be derived across a system call boundary. For example, when establishing a network connection, the application specifies a socket type that dictates the code sequence that will be executed in the kernel. This socket type information remains constant throughout the life of the socket, yet the operating system repeatedly branches on its value. Once we are able to perform constant propagation from the application to the operating system, these branches will be removed. This idea is not new. Value-specific optimizations take advantage of this property to improve performance, but rely on run-time code generation [Kepp91]. A static approach has the potential to avoid the cost of dynamic code generation and to enable more aggressive optimization.

- *Global instruction scheduling* is a well-known technique for optimizing a highly probable code path. When all paths are equally likely, a good scheduler becomes much more conservative in its code motion. Preliminary research hints that an application is likely to exercise the same code paths through the operating system. Therefore a global instruction scheduler can optimize the kernel for a particular application well. However, we also find that different applications are likely to exercise different code paths. Without using the application context for kernel optimization, the opportunity for aggressive instruction scheduling is lost. The Synthesis kernel [Pu88] and Synthetix [Pu95] demonstrated that tailoring operating system code to a particular application or instance of the application does, in fact, improve performance.
- *Code layout* [Pett90] is a compile-time technique for placing code to reduce branch penalties and to avoid instruction cache conflicts. This technique relies on there being few temporal relationships between basic blocks. In an operating system, temporal relationships change with each application, but each application has few temporal dependencies. Once again, if different applications can be associated with differently laid out kernel modules, each will observe improved performance. Torrellas et. al. [Torr95] show that code layout is beneficial for improving instruction cache behavior.
- Other compiler techniques, such as *procedure inlining* [Chang91] and *global register allocation* [Wall86] can all take advantage of a global view of the entire computation.

We have listed just a few of the more obvious opportunities. These and even more sophisticated optimizations are possible because they are applied in reference to a particular application's use of the operating system, and because they are performed at compile time. Unlike code-tailoring methods that rely on run-time code generation, our approach structures the operating system so that these optimizations can be performed at compile-time, which allows use of more global information and more sophisticated optimizations.

In the next section, we discuss how different operating system architectures enable these sorts of compiler optimizations to differing levels.

3 Relationship to New OS Architectures

The current trend in operating systems research considers how best to structure the operating system to provide flexibility and improved application performance. However, little attention is being given to how best to exploit the compiler in achieving this improved performance. Many of the proposed operating system models enable the use of sophisticated compiler optimization to differing degrees.

In a conventional operating system, applications run in an address space separate from the operating system. Applications and the kernel are compiled separately and any optimization information (e.g. register usage, branch prediction buffers) known in one domain is lost when the other domain is entered. This structure provides little opportunity for optimizing the operating system for any particular application.

The Scout system addresses this problem by tailoring the operating system to the target application(s) it will run [Montz94]. By selecting the target application at compile time, Scout can optimize the operating system for a particular workload, and, in all likelihood, achieve better global optimization than is possible by a general purpose operating system. This improved optimization path is not attainable without constraints; the constraint of such statically configured systems is that they are optimized for a single workload or class of applications. This is suitable for dedicated processing nodes, but not acceptable for the machines on a user's desk or multi-purpose machines, such as our central server, which acts as a compute server, file server, network gateway, mail gateway, and Kerberos server.

Other research systems (e.g. SPIN [Bers95], Exokernel [Engl95], and VINO) present more flexible boundaries between applications and the operating system, and provide the potential for application/OS optimization. SPIN allows user code to be downloaded into the operating system's address space, removing potentially expensive calls that cross address space boundaries. The Exokernel moves code the other direction: operating system functionality is provided in libraries that are linked directly into the application. VINO [Small94] sits between the two, allowing the instantiation of kernel objects in applications as well as permitting applications to download functions into the kernel. VINO is based on a fine-grain extensibility model in which any designated kernel function can be replaced by an application-specific one.

Each of these extensible architectures provides different opportunities for compiler optimization. In the Exokernel model, applications and the operating system libraries are loaded into the same address space. Although multiple applications can use the same library, it is possible to compile a version of the library specialized for a particular application. Similarly, one might apply link-time optimizations when the application is linked with its associated operating system library.

The VINO kernel is structured as a set of component classes, which define its behavior, and instances of these classes, which make up the running kernel.

VINO kernel functionality can be reused by applications in two ways. First, an application can instantiate a kernel class at user-level, linking the kernel code into its address space. This code becomes part of the application's address space, and can be optimized just like any other application code. Second, the running kernel is comprised of a set of objects; an application can override the implementation of a method on one of its kernel objects by downloading and *grafting* a new implementation of the function into the kernel. Typically, systems that allow code to be dynamically linked into the running kernel compile the code isolated from both the application and the operating system. As described in the next section, VINO allows us to compile grafts in the context of both the application and the kernel.

Our primary motivation for supporting grafting is to allow applications to extend or modify the semantics of a kernel operation; for example, an application may wish to override the behavior of the VM page eviction algorithm used for its pages, replacing it with an implementation that is designed specifically for the application's use. However, grafting can also be used to support *symbiotic optimization*: the standard implementation of a kernel operation can be re-compiled in the context of a particular application, and grafted into the kernel for that application's use. Like the Synthesis kernel, specific versions of kernel operations can be specialized for a particular application; unlike Synthesis, the architecture of VINO is designed to allow *any* kernel operation to be symbiotically optimized and grafted into the kernel at run-time.

As we discussed above, without good profiling information, it is difficult to determine how best to apply optimizations. One barrier to gathering good profiling

data is that normally a piece of kernel code is invoked by multiple applications; profiling information gathered on this code must be separated and allocated to each application. VINO's architectural support for application-specific grafting allows us to create and run different profiled versions of a piece of kernel code for each invoking application.

4 The Plan

We are exploring symbiotic optimization of applications and operating systems in the context of the HUBE/SUIF compilation project and the VINO kernel project.

SUIF is a research compiler system distributed by Stanford [SUIF94]. The HUBE project at Harvard is enhancing SUIF through the implementation of code generators, scalar optimizations, register allocation passes, and code placement algorithms. We are currently conducting research in the areas of global instruction scheduling algorithms, branch prediction schemes, alternative memory operations, and instruction-set architecture extensions for multimedia. These optimizations are often driven by profile information.

VINO, as described above, is a new operating system architecture. Its fine-grain extensibility model is conducive to evaluating the utility of symbiotic compilation.

To explore the potential benefit of cross-domain optimization, we have selected three application domains to analyze: database management, video display, and scientific visualization. In each domain, we have selected representative applications to target. For each application, we are conducting extensive performance analysis to help us understand its performance, its demands on the operating system, and the potential for optimization.

Finally, we believe that we can apply what we learn about symbiotic optimization of applications and operating systems to symbiotic optimization of other complex, multi-threaded software systems such as database servers.

5 Summary

Much work has been done in the areas of compilation, code optimization, profiling, and operating systems restructuring for flexibility and extensibility. By bringing these tools together, we can analyze and optimize an application and portions of the operating

system as a single system, rather than as separate programs.

By considering an application's user and kernel-level behavior as a unit, we can apply today's sophisticated compile-time optimization techniques to achieve better performance from our system software. Our approach achieves this goal by giving us control over all three pieces of the problem: application, compiler, and operating system.

References

- [Ball93] T. Ball and J. Larus, "Branch Prediction for Free," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 300-313 (June 1993).
- [Bers95] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker., S. Eggers, and C. Chambers, "Extensibility, Safety, and Performance in the SPIN Operating System," *Proceedings of the 15th Symposium on Operating Systems Principles*, pp. 267-284 (December 1995).
- [Brig94] P. Briggs and K. D. Cooper, "Effective Partial Redundancy Elimination", *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 29(6), pp. 159-170 (June 1994).
- [Chang91] P. Chang, S. Mahlke, and W. Hwu. "Using Profile Information to Assist Classic Code Optimizations," *Software Practice and Experience*, 21(12), pp. 1301-1321 (December 1991).
- [Engl95] D. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc 15th Symposium on Operating Systems Principles*, pp. 251-266 (December 1995).
- [Fish81] J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *Trans. on Computers*, C-30(7) pp. 478-490 (July 1981).
- [Fish92] J. Fisher and S. Freudenberger, "Predicting Conditional Branch Directions From Previous Runs of a Program," *Proceedings of the 5th Annual Intl. Conference on Architectural Support for Programming Language and Operating Systems* pp. 85-97 (October 1992).
- [Hall95] M. Hall, B. Murphy and S. Amarasinghe. "Interprocedural Analysis for Parallelization: Design and Experience," *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing* (February 1995).
- [Hwu1993] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, Kluwer Academic Publishers, pp. 229-248 (1993).
- [Kepp91] D. Keppel, S. Eggers, and R. Henry, "A Case for Runtime Code Generation", University of Washington Technical Report UW-CSE 91-11-04 (1991).
- [Lam91] M. S. Lam, E. Rothberg and M. E. Wolf. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* pp. 63-74 (April 1991).
- [Lown93] P. Lowney et al., "The Multiflow Trace Scheduling Compiler," *The Journal of Supercomputing*, 7, pp. 51-142 (1993).
- [Montz94] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman, "Scout: A Communications-oriented Operating System". Technical Report 94-20, Department of Computer Science, University of Arizona, June 1994.
- [Nagle92] D. Nagle, R. Uhlig, and T. Mudge, "Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures," Technical Report, University of Michigan, Ann Arbor, MI (May 1992).
- [Pett90] K. Pettis and R. Hansen, "Profile Guided Code Positioning," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 16-27 (June 1990).
- [Pu88] C. Pu, H. Massalin, and J. Ioannidis, "The Synthesis Kernel," *Computing Systems I*, 1, pp. 11-32 (Winter 1988).
- [Pu95] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, K. Zhang, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," *Proc 15th Symposium on Operating Systems Principles*, pp. 314-324 (December 1995).

- [Rose95] M. Rosenblum, S. Herod, M. Witchel, A. Gupta, "Fast and Accurate Multiprocessor Simulation: The SimOS Approach," *IEEE Parallel and Distributed Technology* (3), 4 (Fall 1995).
- [Small94] C. Small and M. Seltzer, "Structuring the Kernel as a Toolkit of Extensible, Reusable Components", *Proceedings of the 4th International Workshop on Object-Oriented in Operating Systems*, pp. 134–137(August 1995).
- [Sriv94] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation* (June 1994).
- [SUIF94] Stanford SUIF Compiler Group, "SUIF: A Parallelizing and Optimizing Research Compiler," Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University (May 1994).
- [Torr95] J. Torrellas, C. Xia, and R. Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workloads," *Proceedings of the First Symposium on High-Performance Computer Architecture*, pp. 360–369 (January 1995).
- [Wall86] D. Wall, "Global Register Allocation at Link Time," Research Report 86/3, Digital Western Research Laboratory, Digital Equipment, Palo Alto, CA (October 1986)
- .