

Microlanguages for Operating System Specialization*

Calton Pu, Andrew Black, Crispin Cowan, Jonathan Walpole

Dept. of Computer Science and Engineering

Oregon Graduate Institute

P.O. Box 91000, Portland, OR 97291-1000

email: {calton, black, crispin, walpole}@cse.ogi.edu

Charles Consel

Dept. of Computer Science

University of Rennes/IRISA

email: consel@irisa.fr

Abstract

Specialization is a technique that has the potential to provide operating system clients with the performance and functionality that they need, while still retaining the advantages of a simple generic code base for the operating system maintainer. However, at present the specialization process is labor-intensive and requires the knowledge of an expert in the domain of application behavior. In order to realize the full advantages of specialization, we believe that the process must be automated. This means building tools for specialization, and also making the domain knowledge explicit in some form or other.

A specialization toolkit has been developed jointly at the Oregon Graduate Institute and IRISA, as part of the Synthetix project. This paper discusses our preliminary ideas on the use of *Microlanguages* to describe application behavior and to make that information available to the specialization tools.

1 Introduction

Modern operating systems have been growing in size and complexity due to constant pressure for additional functionality. As the variety of applications widens, and hardware platforms become increasingly powerful, operating systems are required to take on increasingly diverse roles. Consequently, operating system implementations have been stretched far beyond their original intent. For example, file systems designed for sequential access to fixed-size blocks are compelled to support access to variable-sized frames of compressed video data in situations where only two out of three frames are actually required by the application.

In an attempt to keep some bound on the size of the operating system kernel, application-specific functionality is often implemented by *middleware*: libraries and user-level servers. However, to support the necessary functionality on top of a kernel designed to meet other goals, the middleware is forced either to duplicate system functionality in a more efficient or controllable way, or to use the system-call interface in ways that were never intended by its designers and for which it may be poorly optimized. These tactics have appropriately been called “hematomas of duplication” and “coding between the lines” [16]. These forces result in uncontrolled growth in the size and complexity of the kernel and the middleware.

Specialization of the operating system kernel has emerged as a promising technique for addressing these problems; Synthesis [20, 18], the *x*-kernel [19], SPIN [1, 2], and Synthetix [7, 21] all allow the introduction of custom or specialized code into the kernel. However, there are two limitations to the wide application of specialization. First, although it has been applied successfully in specific domains, each domain seems different enough to require a large new effort. Second, attempts to make customization easy, such as SPIN, have yet to address the concerns of quality control, interference with other kernel modules, maintainability, and system evolution.

*This work supported in part under DARPA grant N00014-94-1-0845, DARPA contract F19628-95-C-0193, Air Force contract F19628-93-C-0069, and by the Intel corporation.

This paper takes the position that specialization can be directed through *microlanguages*, which exist at several levels of abstraction. At the highest level, they are application domain-specific but kernel and hardware independent: they characterize how the application will use the kernel interface. At a lower-level, they are specific to the implementation of a particular kernel; a low-level microlanguage may be thought of as a form of meta-interface language that controls a particular system component, e.g., file system, network protocol stack, or memory allocator. From the application programmer’s point of view, a sufficiently rich microlanguage provides a high-level declarative way to request system customization. On the systems side, kernel designers determine the scope of the corresponding specializations during the design and implementation of a microlanguage.

Programs written in the microlanguage are called, naturally, microprograms. Microprogramming is a well-known term used in computer architecture; we feel that the concepts introduced here are the software analogs of the corresponding hardware microprogramming concepts. Our microlanguages are small and specialized languages, as are hardware micro-instruction sets. However, there is an important difference: although hardware micro-instruction sets may have large numbers of primitives, they normally have small and simple semantics. In contrast, microlanguages are designed to encapsulate information about specific application domains, so even though a microlanguage syntax may be small and simple, it typically has deep domain-specific semantics.

The *x*-kernel [19] and Horus [26] both achieved clean modular decompositions of the kernel’s communication function. The *x*-kernel carefully analyzed network protocols, dividing them into microprotocols that were composed at runtime to process messages in a flexible and efficient way. However, because the composition of the protocol stack was specified by the application, the application programmer needed to be deeply aware of the protocol decomposition inside the kernel. Horus has implemented group communication primitives using a similar decomposition. The module composition languages of the *x*-kernel and Horus might be considered to be low-level microlanguages restricted to the description of the internal structure of the kernel. In this paper we also envisage higher level microlanguages that are designed to describe the needs of the application.

A alternative approach to customization has been taken by the SPIN project [1, 2], which provides application programmers with a general-purpose programming language in which they write extensions that are loaded into the kernel. Although the language is type-checked and pointer safe, its expressiveness makes it difficult to guarantee the safety, overall performance, reliability and fault tolerance of the resulting kernel, and to protect the kernel and other users from interference caused by the newly inserted code. In our approach, the expressive power of the microlanguage is restricted to the fundamental concepts of the application domain. As a consequence, verifying that a program written in a microlanguage will not break the system is a much easier task.

Initially, we will focus on microlanguages that describe the way that a client program will use (or not use) certain functionality of the kernel. Most of the time, the microlanguage does not change the kernel’s functional interface. For example, a file system interface will still support `read` and `write`. Rather, the microlanguage will address the meta-interface (in the meta-object protocol sense [17]) of a system component. Using the microlanguage, the client will tell the operating system how the base functionality will be used. When several alternative specialized implementations exist, the microlanguage will control which should be chosen.

2 Overview of our Methodology

Our work on microlanguages is part of a larger research program designed to improve operating system maintainability and performance. Our approach can be divided into four stages: microlanguage design, microlanguage implementation, evaluation, and refinement. In this paper we will focus on design and implementation.

2.1 Microlanguage Design

A microlanguage is designed for a specific domain—some particular area of system functionality. Our goal is a microlanguage that uses a minimal set of primitives, parameters, and constraints to express application behavior and needs in a simple and controlled way. We expect that most of the design effort will be expended in domain analysis: effective

microlanguage design requires a thorough understanding of the commonalities and variabilities in application behavior, for this is what the language must seek to express.

We intend that our high-level microlanguages be portable (from one kernel to another) and extensible. Our belief is that we can achieve portability by designing microlanguages to characterize *application behavior*, and letting the implementation define the kernel's response. This is in contrast to a microlanguage that provides explicit control over what the kernel should do, such as specifying which file system pages should be pre-fetched into the kernel's buffers at what time. Although this might be adequate to obtain improved performance, such a language would be specific to a particular kernel implementation: changing the file system block size or the size of the buffer cache would invalidate the microprogram.

The need for extensibility is perhaps less obvious. A microlanguage will describe an application's needs and behavior in a manner appropriate to the state-of-the-technology at its design time. As new classes of applications are developed, the microlanguage will need to be maintained and updated. Typically, it is easier to extend a language based only on declarations and constraints than one that is imperative or computationally unrestricted. We plan to define microlanguages rigorously, so that microprograms are amenable to accurate program analyses to determine properties that are critical to operating systems.

2.2 Implementing Microlanguages

The second stage of our research is to implement a microcompiler and the run-time microengine (a better word might have been *microkernel*, but that term is taken). The microcompiler translates the high-level microprogram into a low-level microprogram, which is then executed by the microengine. Like a conventional compiler, a microcompiler is driven both by the semantics of the language that it implements and by its target architecture. In our example, the microcompiler will understand the way in which the kernel manages its buffer cache, and will translate a high-level description of the application's I/O needs into a low-level microprogram that tells the kernel when and what to prefetch and to flush.

Microengines execute the low-level microprograms. Whereas "execution" of a series of prefetch and flush requests might require no more than making calls to an appropriate kernel meta-interface, execution of a microprogram can be significantly more complicated. In particular, we intend that low-level microprograms be used to control the Synthetix specialization machinery.

Specialization is an implementation technique that has been shown effective in the optimization of kernel calls [1, 2, 20, 21]. However, undisciplined use of specialization can increase the complexity of kernel code significantly. The kernel maintainer is faced with what could become a software engineering nightmare: many versions of a module that are supposed to have identical effects and which must be maintained in tandem, but each of which relies on different constraints for its correctness.

We believe that the solution to this problem is automation. In Synthetix, the relationship between the unspecialized general-case code and the various specializations is made explicit. We use invariants and quasi-invariants to describe when a specialized module is applicable, and guards and repluggers to detect the violation of these invariants and switch specialization accordingly. The microcompiler generates a low-level microprogram that directs the underlying kernel specialization; the microengine interprets this program and if necessary generates the specialized operating system code at run-time.

2.3 Evaluation and Refinement

The third stage is to use the microlanguage to describe the activities of higher level software and evaluate the appropriateness of the microlanguage and the improvements achieved. Note that it is unlikely that automated customization will obtain all of the performance benefits that could be obtained by inserting new, hand crafted code directly into the operating system kernel. That is not our goal: rather, we aim to retain a simple and straightforwardly maintainable code base for the operating system, while at the same time gaining most (but not all) of the benefits of custom code. This goal is similar to that achieved when high-level system implementation languages replaced assembly language in

system kernels. The protagonists of assembly language claimed that high-level languages would never be as efficient. They were right, but it did not matter: programmer productivity was much more important than machine efficiency.

In the past, the specialization of an operating system kernel has been primarily evaluated using microbenchmarks that compare kernel calls before and after optimization. However, such microbenchmarks only show the effects on the particular kernel call, isolated from the rest of the system. In reality, it is the “whole system” or “end-to-end” performance that is of most interest to the user. Since microlanguages can facilitate cross-layer optimization, we plan to use benchmarks that will compare the performance of whole microlanguage-based systems to those without customization.

The fourth stage is to use the evaluation results to refine the microlanguage and its attendant tools. Having done the experimental evaluation of systems with microlanguages, we will use the results to refine the language, the experiment and the system.

3 An Example Microlanguage

To illustrate these concepts, we present an example microlanguage designed to describe the needs of a family of applications that use file systems, an area where we have significant experience [21]. We start with an application-level microlanguage, and then use a microcompiler to translate this to a system-level microlanguage primarily concerned with customizing OS implementation details.

Our goal is to specify the application’s data needs to the file system in order to maximize effective use of the file system’s buffer cache. Prior knowledge of an application’s data needs enables the file system to prefetch the data, thus minimizing I/O latency.

The meaning of a microprogram in the file-system microlanguage can be given using a trace semantics. The trace is a bounded or unbounded sequence of operations on the file-system interface, intended to represent the operations that the microprogram *predicts* that the application will carry out. In the following examples, we generate this trace explicitly using an imperative microprogram. This program shares variables with the parent application program; this provides the connection between them. The `will` keyword prefixes an action that the microprogram predicts the application program will take, and has the effect of adding this action to the trace. The implementation of the microprogram should be such that correct predictions enhance performance; miss-predictions degrade performance, but do not otherwise affect the correctness of the application program.

To inform the system that the application will be reading and seeking in a regular pattern, we might write:

```
forever do {
    will read(fd, *, 512);
    will lseek(fd, 8192, SEEK_CUR);
}
```

The variable `fd` is declared in the parent program and denotes an open file object (a UNIX file descriptor). The microprogram specifies that the application will start reading `fd` from position 0 (the default action), read 512 bytes, and then seek ahead 8192 bytes before reading again. The `*` indicates that the second argument to `read ()` (the application buffer to copy into) is not known to the microprogram.

This microprogram can be viewed as denoting an unbounded sequence of `will` statements that inform the kernel about the pattern of the parent program’s system calls involving `fd`. A scientific applications may have a more complex striding pattern:

```
for i = 1 to infinity by 1 do {
    will read(fd, buffer, 512);
    will lseek(fd, 4096 * (i % 8), SEEK_SET);
}
```

It may also be the case that the application buffer is known to be static, indicated above by the use of the parent program variable `buffer`. Given a static application buffer, the kernel’s file system buffers can be chosen to avoid

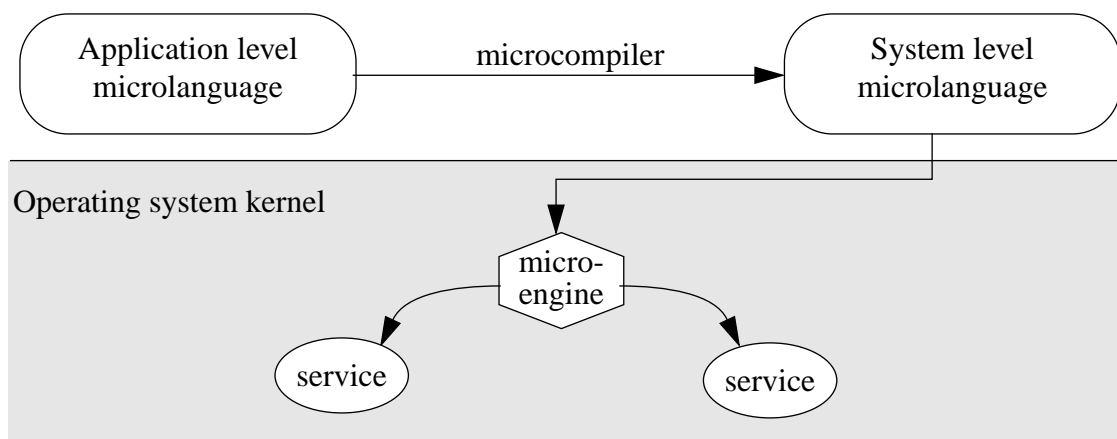


Figure 1: File System Microcompiler translates application-level microprograms into system-level microprograms, which then drive a microengine that customizes the behavior of the system

collisions in the CPU cache when the data is copied by the `read()` system call, yielding up to ten times performance improvements in data copying bandwidth [12, 13].

Simple striding patterns like these could just as easily be expressed using a grammar or a recurrence relation, which would obviate the need for the explicit loops. However, this is not always the case. Multimedia applications have more complex access patterns. For instance, a video player may be fast-forwarding through a movie, skipping over some frames and displaying others. Because the frames are variable-sized (depending on the complexity of the image) the access pattern cannot be described using a simple expression; an index file must be used to predict the read size and the stride size. Furthermore, to guarantee Quality-of-Service, frames must be available at specific times [24]. Hence, this microprogram also specifies *when* the data will be read. The following microprogram uses the file descriptor `index_fd` to specify the sequence of frames to read, their offsets in the file `fd`, and the times at which the read requests will be made. The parent program variable `frame_interval` is used to compute when the next read will occur.

```

int offset, stride;
int time = 0;
forever do {
    read(index_fd, &offset, sizeof(offset));
    will read(fd, buffer, offset) at time;
    read(index_fd, &stride, sizeof(stride));
    will lseek(fd, stride, SEEK_CUR);
    time = time + frame_interval;
}

```

The preceding high-level microprograms express the needs of the application program, but say nothing about the resources managed by the file system. We use a microcompiler to translate these microprograms into lower-level microprograms that pertain to system resources, as shown in Figure 1. The microcompiler for the file system microlanguage knows about the particular file system architecture used in the kernel, and generates a low-level microprogram that specifies the loading and freeing of particular blocks; this might be nothing more than a sequence of statements like

```
prefetch(fd, offset, size);  
release(fd, offset, size);
```

Such a microprogram would be executed by a microengine that maintains high- and low-water marks for the number of pre-fetched blocks in the file system buffer cache. It is thus able to choose a prefetching strategy appropriate to the application's goals that nevertheless avoids wasting system resources. The number of pre-fetched blocks to be kept in the buffer cache is a function of the size of the cache, the timing requirements of the application and the specific performance of the system I/O devices. The microengine manages the cache by making requests through an appropriate meta-interface while the application program is running. Naturally, it must also monitor the application program's use of the file system interface.

Note that the activities of the microengine have no effect on the correct execution of the application program. At worst, an inappropriate microprogram will degrade performance; it will not crash the kernel or the application.

An alternative approach to the implementation of the low-level microprogram is to use it to drive the Synthetix specialization machinery. That is, rather than interpreting the microprogram in tandem with the application's use of ordinary system calls, we generate specialized system calls that perform prefetching in addition to their normal function. In previous research, we have generated such specialized system calls for simple striding patterns. Microlanguages will provide us with the information about program behavior that we need to generalize this work.

4 Microlanguage Research Issues

The example microlanguages in Section 3 should be regarded as initial attempts at languages that might be adequate for the domain of file system specialization. We intend to explore design issues for such languages much more thoroughly. The systematic design of microlanguages for targeted system components brings up questions not only about the application area (file systems in our example), but also about the interactions between the file system microlanguage and other microlanguages in related areas.

For example, in the Synthetix experiment with the HP-UX Unix File System, we have found a close relationship between large (bigger than 64kB) block read performance and cache management algorithms in the PA-RISC hardware. It is plausible that a microlanguage dedicated to the description of hardware cache management could help kernel designers to smooth the interactions between virtual memory management and file system.

Besides variables of the parent program, declarations and constraints over them, and predictions of program behavior, our microlanguages contain other elements of a "normal" programming language: data structures and executable statements. However, we intend to avoid making microlanguages computationally complete programming languages, since we believe that such expressive power is unnecessary. In fact, it may be preferable to reduce the scope of a microlanguage rather than increasing its expressive power.

Typical microlanguage constructs will be a combination of declarations and constraints on the use of specific types of objects that abstract from kernel functionality. It may be useful to make negative as well as positive statements about the functionality that the program will use, e.g., we may include `wont` statements as well as `will` statements. Since the building blocks are relatively simple and static, their combination is the main source of expressiveness.

We see microlanguages as a special kind of meta-interface in Meta-Object Protocols [17]. While the purpose of meta-object protocols is basically the same as microlanguages, i.e., to give clients more control over the underlying implementation, microlanguages are typically more restrictive by design. Meta-object protocols are defined to allow clients to direct the control flow through the underlying implementation, either through declarations or through imperative statements. There is no consensus on the appropriate style for the definition of meta-object protocols: should they be highly restricted or should they allow generic programming? Arguments from both sides have been presented in a recent workshop on Open Implementation [16].

By calling our approach *microlanguage* instead of meta-interface or meta-object protocol, we are taking a clear position on the question of meta-interface style. We believe that, at least for critical code such as operating system kernels, application program use of meta-interfaces should be carefully restricted. This does not mean that a more computationally complete meta-interface is not valuable. It simply says that a complete meta-interface defined over a

critical system component requires great knowledge and care in its use, since the potential for abuse is great and the consequences of abuse are heavy.

Just as we favor the restricted use of meta-interfaces in microlanguages, we will avoid the indiscriminate proliferation of many microlanguages. We envision families of microlanguages, each devoted to an important function. For example, we see the control of I/O as an important area, where an entire family of microlanguages will arise. But instead of creating one microlanguage for file systems and another completely different one for network protocols, we will design a core I/O microlanguage, and appropriate extensions for file systems as well as network protocols. The core microlanguage will capture the essential data flow aspects of I/O, while the extensions will represent the peculiarities of each I/O device and usage.

5 Related Work

Microprotocols—*x*-kernel, SPIN: The SPIN project [1, 2] is using a “safe” language for customizing the kernel. Program fragments written in this language are loaded into the kernel address space dynamically, and can affect the kernels behavior in general ways. This facility is a powerful one, but has the potential to damage the system (or other systems) in unforeseen ways. SPIN also imposes a considerable software burden on application programmers wishing to customize the operating system interface. Microlanguages can be seen as a disciplined, secure, and simple way for the users to customize the kernel, using application-level primitives that were built into the systems software by trusted programmers. Microlanguages are a restricted and high-level form of SPIN’s “safe” language, specialized to an application domain so that they can be safe, implementable and powerful.

The main idea of the *x*-kernel is to analyze the network protocol stack, dividing it into micro-protocols. By implementing micro-protocols in fine-grain modules and composing them into actual protocol code at run-time, the *x*-kernel is both elegant and efficient. While the *x*-kernel designers have been very successful in the network protocol area, the generalization of their technique to other operating system kernel components has proved to be elusive. Our tools for the expression of micro-protocols allow the use of a similar kind of inter-module interfacing in a broader range of application domains. Thus we see that the *x*-kernel used what amounts to a single implementation of a low-level microlanguage.

Meta-Object Protocols: Microlanguages are also related to Meta-Object Protocols [17] and the recent movement towards Open Implementation [16]. In an Open Implementation system, a meta-interface is added to the system’s functional interface. The meta-interface is used to direct the underlying implementation to take the most suitable execution path. Microlanguages can be seen as a systematic and disciplined way to develop meta-interfaces and to link the meta-interfaces to the underlying implementation. Rather than directing the implementation to take one particular path or another through a meta-interface, which would require knowledge of the implementation, an application can simply describe its own behavior with a microprogram, which is independent of the operating system implementation. Thus the microlanguage approach can be viewed as a particular discipline in the construction of Open Implementations.

Modular Operating Systems—Mach, Chorus, Choices, Apertos: In response to the saturation of monolithic kernels, one of the most important developments in modern operating systems is the movement towards micro-kernels such as Mach [3] and Chorus [22], and more recently, object-oriented operating systems such as Choices [5] and Apertos [27]. All of these systems have a high degree of organization and modularity. However, they also suffer performance penalties for this modularity. More recent research on these systems has been focused on reducing the overhead of this modularity, typically by composing modules into coarser grain modules, and co-locating operating system servers in the kernel address space [11].

Partial Evaluation: Traditional partial evaluation [6, 15] specializes a program with respect to some known parts of its input, providing the basis for a simple and automatic approach to program optimization through specialization. However, existing approaches to partial evaluation identify only two stages at which information becomes available: compile time and run time. This problem is shared by existing partial evaluators such as Mix [14], Schism [8], and Silimix [4]. In an operating system, many more stages can be identified, including boot time, compile time, and link time. Therefore, we are working on progressive and incremental specialization using information about invariants gathered

from a wide range of sources at many stages. The research described in this paper makes extensive use of the principles of partial evaluation, but in the context of imperative C-based programming languages, and in a more dynamic and flexible manner. Most of these aspects are integrated in a partial evaluator for C, named Tempo, which enables programs to be specialized both at compile time and run time [9, 10].

6 Summary

We have described an approach to designing *microlanguages* that allow applications to declare their needs to the operating system and to inform the system of their future behavior. The approach enables an operating system to use as little or as much of the information present in the microprogram as is relevant to that particular system. We presented a microlanguage designed to support optimization of file system performance by predicting application data needs. We have described how this approach may be generalized to allow applications in various domains to specify their needs to the operating system, and outlined some research issues that lie ahead in designing future microlanguages and deploying operating system facilities that exploit them.

References

- [1] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E.G. Sirer. SPIN - An Extensible Microkernel for Application-specific Operating System Services. In *SIGOPS 1994 European Workshop*, February 1994. UW Technical Report 94-03-03.
- [2] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.
- [3] D.L. Black, D.B. Golub, D.P. Julin, R.F. Rashid, R.P. Draves, R.W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and mach. In *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, Seattle, April 1992.
- [4] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 70–87. Springer-Verlag, 1990.
- [5] R.H. Campbell, N. Islam, and P. Madany. Choices, frameworks, and refinement. *Computing Systems*, 5(3), Summer 1992.
- [6] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.
- [7] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, June 1993.
- [8] Charles Consel. A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages. In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 66–77, Copenhagen, Denmark, June 1993.
- [9] Charles Consel, Luke Hornoff, Jacque Noye, Francois Noël, and Eugen-Nicolae Volanschi. A Uniform Approach for Compile-Time and Run-Time Specialization. In *International Workshop on Partial Evaluation*, Dagstuhl Castle, Germany, February 1996. Springer-Verlag LNCS.

- [10] Charles Consel and Francois Noël. A general approach to run-time specialization and its application to C. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburg Beach, FL, January 1996.
- [11] M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. A second-generation micro-kernel based unix: Lessons in performance and compatibility. In *Proceedings of the Winter Technical USENIX Conference '91*, Dallas, 1991.
- [12] Jon Inouye, Ravindranath Konuru, Jonathan Walpole, and Bart Sears. The Effects of Virtually Addressed Caches on Virtual Memory Design & Performance. *Operating Systems Review*, 24(4):896–908, October 1992. Also published as OGI technical report CSE-92-010, <ftp://cse.ogi.edu/pub/tech-reports/1992/92-010.ps.gz>.
- [13] Jon Inouye, Jonathan Walpole, and Ke Zhang. Fast Byte Copying: A Re-Evaluation of the Opportunities for Optimization. Report CSE-95-010, Oregon Graduate Institute, Portland, Oregon, June 1995. <ftp://cse.ogi.edu/pub/tech-reports/1995/95-010.ps.gz>.
- [14] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
- [15] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.
- [16] Gregor Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, January 1996. <http://www.parc.xerox.com/spl/projects/oi/ieee-software/>.
- [17] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [18] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 191–201, Arizona, December 1989.
- [19] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.
- [20] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.
- [21] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.
- [22] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–69, Seattle, April 1992.
- [23] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
- [24] Richard Staehli, Jonathan Walpole, and David Maier. Quality of Service Specifications for Multimedia Presentations. *Multimedia Systems*, 3(5/6):251–263, November 1995.
- [25] R. D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981.
- [26] Robbert van Renesse, Takako M. Hickey, and Kenneth P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report TR94-1442, Cornell University, Ithaca, New York, August 1994.
- [27] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *OOPSLA '92 Conference Proceedings*, Vancouver, BC, Canada, October 1993.