Optimistic Incremental Specialization: Streamlining a Commercial Operating System *

Calton Pu, Tito Autrey, Andrew Black, Charles Consel[†] Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
(synthetix-request@cse.ogi.edu)

Abstract

Conventional operating system code is written to deal with all possible system states, and performs considerable interpretation to determine the current system state before taking action. A consequence of this approach is that kernel calls which perform little actual work take a long time to execute. To address this problem, we use *specialized* operating system code that reduces interpretation for common cases, but still behaves correctly in the fully general case. We describe how specialized operating system code can be generated and bound *incrementally* as the information on which it depends becomes available. We extend our specialization techniques to include the notion of *optimistic incremental specialization*: a technique for generating specialized kernel code optimistically for system states that are likely to occur, but not certain. The ideas outlined in this paper allow the conventional kernel design tenet of "optimizing for the common case" to be extended to the domain of adaptive operating systems. We also show that aggressive use of specialization can produce in-kernel implementations of operating system functionality with performance comparable to user-level implementations.

We demonstrate that these ideas are applicable in real-world operating systems by describing a re-implementation of the HP-UX file system. Our specialized `read` system call reduces the cost of a single byte read by a factor of 3, and an 8 KB read by 26%, while preserving the semantics of the HP-UX `read` call. By relaxing the semantics of HP-UX `read` we were able to cut the cost of a single byte `read` system call by more than an order of magnitude.

1 Introduction

Much of the complexity in conventional operating system code arises from the requirement to handle all possible system states. A consequence of this requirement is that operating system code tends to be "generic", performing extensive interpretation and checking of the current environment before taking action. One of the lessons of the Synthesis operating system [25] is that significant gains in efficiency can be made by replacing this generic code with *specialized* code. The specialized code performs correctly only in a restricted environment, but it is chosen so that this restricted environment is the common case.

By way of example, consider a simplified UNIX File System interface in which `open` takes a path name and returns an "open file" object. The operations on that object include `read`, `write`, `close`, and `seek`. The method code for `read` and `write` can be specialized, at open time, to read and write that particular file, because at that time the system knows, among other things, which file is being read, which process is doing the reading, the file type, the file system block size, whether the inode is in memory, and if so, its address, etc. Thus, a lot of the interpretation of file system data structures that would otherwise have to go on *at every read* can be done once at open time. Performing this interpretation at open time is a good idea if `read` is more common than `open`, and in our experience with specializing the UNIX file system, loses only if the file is opened for read and then never read.

Extensive use of this kind of specialization in Synthesis achieved improvement in kernel call performance ranging from a factor of 3 to a factor of 56 [25] for a subset of the UNIX system call interface. However, the performance improvements due directly to code specialization were not separated from the gains due to other factors, including the design and implementation of a new kernel in assembly language, and the extensive use of other new techniques such as lock-free synchronization and software feedback.

This paper describes work done in the context of the Synthetix project which is a follow-on from Synthesis. Synthetix extends the Synthesis results in the following respects. First, Synthetix defines a conceptual model of specialization. This model defines the basic elements and phases of the specialization process. Not only is this model useful for deploying specialization in other systems, it has also proved essential in the development of tools to support the specialization process.

Second, Synthetix introduces the idea of *incremental* and *optimistic* specialization. Incremental specialization allows specialized modules to be generated and bound as the information on which they depend becomes available. Optimistic specialization allows modules to be generated for system states that are likely to occur, but not certain.

Finally, we show how optimistic, incremental specializa-

* This research is partially supported by ARPA grant N00014-94-1-0845, NSF grant CCR-9224375, and grants from the Hewlett-Packard Company and Tektronix.

[†] Also University of Rennes/IRISA (France).

tion can be applied to existing operating systems written in conventional programming languages. In this approach, performance improvements are achieved by customising existing code without altering its semantics. In contrast, other extensible operating systems allow arbitrary customizations to be introduced at the risk of altering the system semantics.

The long term goal of Synthetix is to define a methodology for specializing existing system components and for building new specialized components from scratch. To explore and verify this methodology we are manually specializing some small, but representative, operating system components. The experience gained in these experiments is being used to develop tools towards automating the specialization process. Hence, the goal of our experiments is to gain an understanding of the specialization concept rather than simply to optimize the component in question.

This paper illustrates our approach applying specialization in the context of a commercial Unix operating system and the C programming language. Specifically, it focuses on the specialization of the `read` system call, in HP-UX [12] while retaining standard HP-UX semantics. Since `read` is representative of many other Unix system calls and since HP-UX is representative of many other Unix systems, we expect the results of our study to generalize well beyond this specific implementation.

The remainder of the paper is organized as follows. Section 2 elaborates on the notion of specialization, and defines incremental and optimistic specialization. Section 3 describes the application of specialization to the HP-UX `read` system call. Section 4 analyses the performance of our implementation. Section 5 discusses the implications of our results, as well as the key areas for future research. Related work is discussed in section 6. Section 7 concludes the paper.

2 What is Specialization?

Program specialization, also called partial evaluation (PE), is a program transformation process aimed at customizing a program based on parts of its input [13, 30]. In essence, this process consists of performing constant propagation and folding, generalized to arbitrary computations.

In principle, program specialization can be applied to any program that exhibits interpretation. That is, any program whose control flow is determined by the analysis of some data. In fact, this characteristic is common in operating system code. Consider the `read` example of Section 1. At each invocation, `read` does extensive data structure analysis to determine facts such as whether the file is local or remote, the device on which it resides, and its block size. Yet, these pieces of information are invariant, and can be determined when the file is opened. Hence, the data structure analysis could be factorized by specializing the `read` code at open time with respect to the available *invariants*. Since the specialized `read` code only needs to consist of operations that rely on varying data, it can be more efficient than the original version.

Operating systems exhibit a wide variety of invariants. As a first approximation, these invariants can be divided into two categories depending on whether they become available before or during runtime. Examples of invariants available before runtime include processor cache size, whether there is an FPU, etc. These invariants can be exploited by existing PE technology at the source level. Other specializations depend on invariants that are not known until runtime, and hence rely on a specialization process that can take place at runtime. In the context of operating systems, it is useful for specialization to take place both at compile time and at run time.

Given a list of invariants, which may be available either statically or dynamically, a combination of compile-time and run-time PE should be capable of generating the required specialized code. For example, the Synthesis kernel [28] performed the (conceptual) PE step just once, at runtime during `open`. It is in principle possible to apply the run-time partial evaluator again at every point where new invariants become known (i.e., some or all of the points at which more information becomes available about the bindings that the program contains). We call this repeated application of a partial evaluator *incremental specialization* [15].

The discussion so far has considered generating specialized code only on the basis of known invariants, i.e., bindings that are known to be constant. In an operating system, there are many things that are *likely* to be constant for long periods of time, but may occasionally vary. For example, it is *likely* that files will not be shared concurrently, and that reads to a particular file will be sequential. We call these assumptions *quasi-invariants*. If specialized code is generated, and used, on the assumption that quasi-invariants hold most of the time, then performance should improve. However, the system must correctly handle the cases where the quasi-invariants do not hold.

Correctness can be preserved by guarding every place where quasi-invariants may become false. For example, suppose that specialized `read` code is generated based on the quasi-invariant "no concurrent sharing". A *guard* placed in the `open` system call could be used to detect other attempts to open the same file concurrently. If the guard is triggered, the `read` routine must be "unspecialized", either to the completely generic `read` routine or, more accurately, to another specialized version that still capitalizes on the other invariants and quasi-invariants that remain valid. We call the process of replacing one version of a routine by another (in a different stage of specialization) *replugging*. We refer to the overall process of specializing based on quasi-invariants *optimistic specialization*. Because it may become necessary to replug dynamically, optimistic specialization requires incremental specialization.

If the optimistic assumptions about a program's behavior are correct, the specialized code will function correctly. If one or more of the assumptions become false, the specialized code will break, and it should be replugged. This transformation will be a win if specialized code is executed many times, i.e., if the savings that accrue from the optimistic assumption being right, weighted by the probability that it is right, exceed the additional costs of the replugging step, weighted by the probability that it is necessary (see Section 4 for details).

The discussion so far has described incremental and optimistic specialization as forms of runtime PE. However, in the operating system context, the full cost of code generation must not be incurred at runtime. The cost of runtime code generation can be avoided by generating code *templates* statically and optimistically at compile time. At kernel call invocation time, the templates are simply filled in and bound appropriately [14].

3 Specializing HP-UX `read`

To explore the real-world applicability of the techniques outlined above, we applied incremental and optimistic specialization to the HP-UX 9.04 `read` system call. `read` was chosen as a test case because it is representative of many other UNIX system calls because it is a variation of the BSD file system [24]. The HP-UX implementation of `read` is also representative of many other UNIX implementations. Therefore, we expect our results to be applicable to other UNIX-like systems. HP-UX `read` also poses a serious challenge for our technology because, as a frequently used and well-understood piece of commercial operating system code, it has already been highly optimized. To ensure that our performance results are applicable to current software, we were careful to preserve the current interface and behavior of HP-UX `read` in our specialized `read` implementations.

3.1 Overview of the HP-UX `read` Implementation

To understand the nature of the savings involved in our specialized `read` implementation it is first necessary to understand the basic operations involved in a conventional UNIX `read` implementation. Figure 1 shows the control flow for `read` assuming that the read is from a normal file and that its data is in the buffer cache. The basic steps are as follows:

1. System call startup: privilege promotion, switch to kernel stack, and update user structure.
2. Identify the file and file system type: translate the file descriptor number into a file descriptor, then into a vnode number, and finally into an inode number.
3. Lock the inode.
4. Identify the block: translate the file offset value into a logical (file) block number, and then translate the logical block number into a physical (disk) block number.
5. Find the virtual address of the data: find the block in the buffer cache containing the desired physical block and calculate the virtual address of the data from the file offset.
6. Data transfer: Copy necessary bytes from the buffer cache block to the user's buffer.
7. Process another block?: compare the total number of bytes copied to the number of bytes requested; goto step 4 if more bytes are needed.
8. Unlock the inode.
9. Update the file offset: lock file table, update file offset, and unlock the file table.
10. System call cleanup: update kernel profile information, switch back to user stack, privilege demotion.

The above tasks can be categorized as either *interpretation*, *traversal*, *locking*, or *work*. Interpretation is the process of examining parameter values and system states and making control flow decisions based on them. Hence, it involves activities such as conditional and case statement execution, and examining parameters and other system state variables to derive a particular value. Traversal can be viewed as

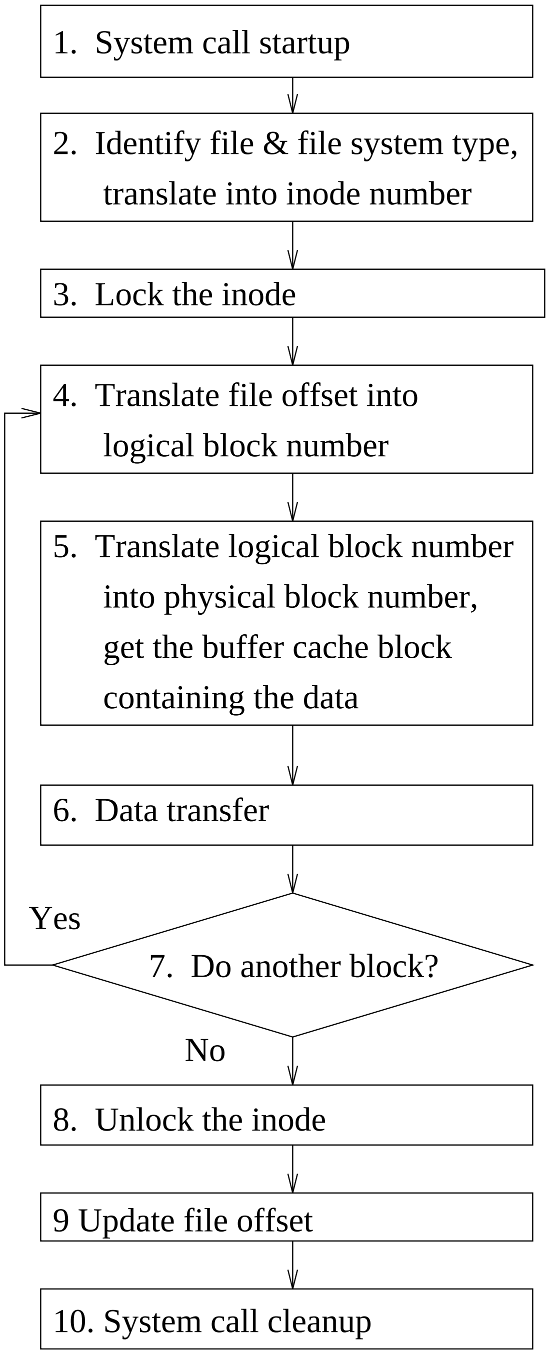

Figure 1: HP-UX `read` Flow Graph

preparation for interpretation since it is concerned with discovering system state information that has been stored previously (i.e., in data structures). Traversal is basically a matter of dereferencing and includes function calling and data structure searching. Locking includes all synchronization-related activities. Work is the fundamental task of the call. In the case of the `read`, the only work is to copy the desired data from the kernel buffers to the user's buffer.

Ideally, all of the tasks performed by a particular invocation of a system call should be in the *work* category. If the necessary information is available early enough, code in all other categories can be factored out prior to the call. Unfortunately, in the case of `read` steps 1, 2, 4, 5, 7, and 10 consist mostly of interpretation and traversal, and steps 3, 8, and most of 9 are locking. Only step 6 and a small part of 9 can be categorized as work.

Section 3.2 details the specializations applied to `read` to reduce the non-*work* overhead. Section 3.3 outlines the code necessary to guard the optimistic specializations. Section 3.4 describes the mechanism for switching between various specialized and generic versions of a system call implementation.

3.2 Invariants and Quasi-Invariants for Specialization

Figure 2 shows the specialized flow graph for our specialized version of `read` (referred to here as `is_read`). Steps 2, 3, and 8 have been eliminated, steps 4 and 5 have been eliminated for small sequential reads, and the remaining steps have been optimized via specialization. `is_read` is specialized according to the invariants and quasi-invariants listed in table 1. Only FS_CONSTANT is a true invariant; the remainder are quasi-invariants.

The FS_CONSTANT invariant states that file system constants such as the file type, file system type, and block size do not change once the file has been opened. This invariant is known to hold because of UNIX file system semantics. Based on this invariant, `is_read` can avoid the traversal costs involved in step 2 above. Our `is_read` implementation is specialized, at open time, for regular files residing on a local file system with a block size of 8 KB. It is important to realize that the `is_read` code is enabled, at open time, for the specific file being opened and is transparently substituted in place of the standard `read` implementation. Reading any other kind of file defaults to the standard HP-UX `read`.

It is also important to note that the `is_read` path is specialized for the specific process performing the `open`. That is, we assume that the only process executing the `is_read` code will be the one that performed the `open` that generated it. The major advantage of this approach is that a private per-process per-file `read` call has well-defined access semantics: reads are sequential by default.

Specializations based on the quasi-invariant SEQUENTIAL_ACCESS can have huge performance gains. Consider a sequence of small (say 1 byte) reads by the same process to the same file. The first `read` performs the interpretation, traversal and locking necessary to locate the kernel virtual address of the data it needs to copy. At this stage it can specialize the next `read` to simply continue copying from the next virtual address, avoiding the need for any of the steps 2, 3, 4, 5, 7, 8, and 9, as shown in Figure 2. This specialization is predicated not only on the SEQUENTIAL_ACCESS and NO_FP_SHARE quasi-invariants, but also on other quasi-invariants such as the assumption that the next `read` won't cross a buffer boundary, and the buffer cache replacement code won't have changed the data that resides at that virtual memory address. The next section shows how these assumptions can be guarded.

The NO_HOLES quasi-invariant is also related to the specializations described above. Contiguous sequential reading can be specialized down to contiguous byte-copying only for files that don't contain holes, since hole traversal requires

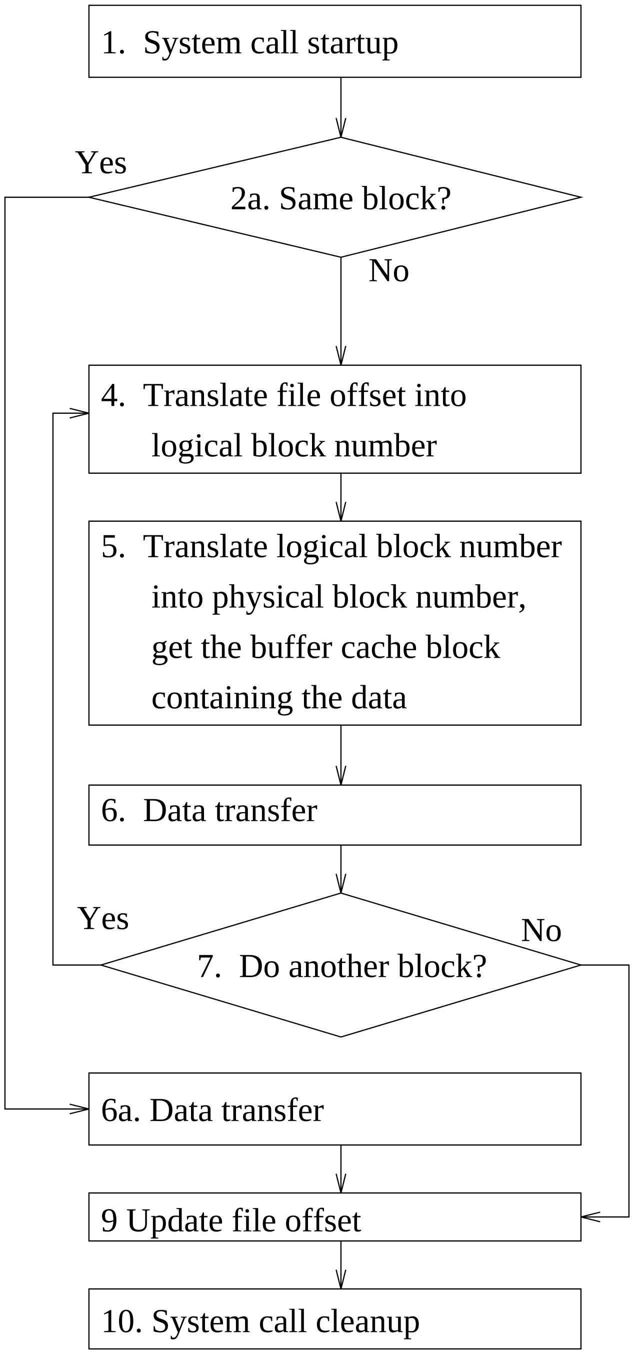

Figure 2: `is_read` Flow Graph

the interpretation of empty block pointers in the inode.

The NO_INODE_SHARE and NO_FP_SHARE quasi-invariants allow exclusive access to the file to be assumed. This assumption allows the specialized `read` code to avoid locking the inode and file table in steps 3, 8, and 9. They also allow the caching (in data structures associated with the specialized code) of information such as the file pointer. This caching is what allows all of the interpretation, traversal and locking in steps 2, 3, 4, 5, 8 and 9 to be avoided.

In our current implementation, all invariants are vali-

(Quasi-)Invariant	Description	Savings
FS_CONSTANT	Invariant file system parameters.	Avoids step 2.
NO_FP_SHARE	No file pointer sharing.	Avoids most of step 9 and allows caching of file offset in file descriptor.
NO_HOLES	No holes in file.	Avoids checking for empty block pointers in inode structure.
NO_INODE_SHARE	No inode sharing.	Avoids steps 3 and 8.
NO_USER_LOCKS	No user-level locks.	Avoids having to check for user-level locks.
READ_ONLY	No writers.	Allows optimized end of file check.
SEQUENTIAL_ACCESS	Calls to `is_read` inherit file offset from previous `is_read` calls	For small reads, avoids steps 2, 3, 4, 5, 7, 8, 9.

Table 1: Invariants for Specialization

Quasi-Invariant	HP-UX system calls that may invalidate invariants
NO_FP_SHARE	creat, dup, dup2, fork, sendmsg,
NO_HOLES	open
NO_INODE_SHARE	creat, fork, open, truncate
NO_USER_LOCKS	lockf, fcntl
READ_ONLY	open
SEQUENTIAL_ACCESS	lseek, readv, is_read itself, and buffer cache block replacement

Table 2: Quasi-Invariants and Their Guards

dated in `open`, when specialization happens. A specialized `read` routine is not generated unless *all* of the invariants hold.

3.3 Guards

Since specializations based on quasi-invariants are optimistic, they must be adequately guarded. Guards detect the impending invalidation of a quasi-invariant and invoke the replugging routine (section 3.4) to unspecialize the `read` code. Table 2 lists the quasi-invariants used in our implementation and the HP-UX system calls that contain the associated guards.

Quasi-invariants such as READ_ONLY and NO_HOLES can be guarded in `open` since they can only be violated if the same file is opened for writing. The other quasi-invariants can be invalidated during other system calls which either access the file using a file descriptor from within the same or a child process, or access it from other processes using system calls that name the file using a pathname. For example, NO_FP_SHARE will be invalidated if multiple file descriptors are allowed to share the same file pointer. This situation can arise if the file descriptor is duplicated locally using `dup`, if the entire file descriptor table is duplicated using `fork`, or if a file descriptor is passed though a UNIX domain socket using `sendmsg`. Similarly, SEQUENTIAL_ACCESS will be violated if the process calls `lseek` or `readv`.

The guards in system calls that use file descriptors are relatively simple. The file descriptor parameter is used as an index into a per-process table; if a specialized file descriptor is already present then the quasi-invariant will become invalid, triggering the guard and invoking the replugger. For example, the guard in `dup` only responds when attempting to duplicate a file descriptor used by `is_read`. Similarly, `fork` checks all open file descriptors and triggers replugging of any specialized `read` code.

Guards in calls that take pathnames must detect collisions with specialized code by examining the file's inode. We use a special flag in the inode to detect whether a specialized code path is associated with a particular inode[1].

The guards for the SEQUENTIAL_ACCESS quasi-invariant are somewhat unusual. `is_read` avoids step 5 (buffer cache lookup) by checking to see whether the read request will be satisfied by the buffer cache block used to satisfy the preceding read request. The kernel's buffer cache block replacement mechanism also guards this specialization to ensure that the preceding buffer cache block is still in memory.

With the exception of `lseek`, triggering any of the guards discussed above causes the `read` code to be replugged back to the general purpose implementation. `lseek` is the only instance of respecialization in our implementation; when triggered, it simply updates the file offset in the specialized `read` code.

To guarantee that all invariants and quasi-invariants hold, `open` checks that the vnode meets all the FS_CONSTANT and NO_HOLES invariants and that the requested access is only for `read`. Then the inode is checked for sharing. If all invariants hold during `open` then the inode and file descriptor are marked as specialized and an `is_read` path is set up for use by the calling process on that file. Setting up the `is_read` path amounts to allocating a private per-file-descriptor data structure for use by the `is_read` code which is sharable. The inode and file descriptor markings activate all of the guards atomically since the guard code is permanently present.

3.4 The Replugging Algorithm

Replugging components of an actively running kernel is a non-trivial problem that requires a paper of its own, and is the topic of ongoing research. The problem is simplified here for two reasons. First, our main objective is to test the feasibility and benefits of specialization. Second, specialization has been applied to the replugging algorithm itself. For kernel calls, the replugging algorithm should be specialized, simple, and efficient.

The first problem to be handled during replugging is synchronization. If a replugger were executing in a single-

[1] The in-memory version of HP-UX's inodes is substantially larger than the on-disk version. The specialized flag only needs to be added to the in-memory version.

threaded kernel with no system call blocking in the kernel, then no synchronization would be needed. Our environment is a multiprocessor, where kernel calls may be suspended. Therefore, the replugging algorithm must handle two sources of concurrency: (1) interactions between the replugger and the process whose code is being replugged and (2) interactions among other kernel threads that triggered a guard and invoked the replugging algorithm at the same time. Note that the algorithm presented here assumes that a coherent read from memory is faster than a concurrency lock: if specialized hardware makes locks fast, then the specialized synchronization mechanism presented here can be replaced with locks.

To simplify the replugging algorithm, we make two assumptions that are true in many UNIX systems: (A1) kernel calls cannot abort[2], so we do not have to check for an incomplete kernel call to `is_read`, and (A2) there is only one thread per process, so multiple kernel calls cannot concurrently access process level data structures.

The second problem that a replugging algorithm must solve is the handling of executing threads inside the code being replugged. We assume (A3) that there can be at most one thread executing inside specialized code. This is the most important case, since in all cases so far we have specialized for a single thread of control. This assumption is consistent with most current UNIX environments. To separate the simple case (when no thread is executing inside code to be replugged) from the complicated case (when one thread is inside), we use an "inside-flag". The first instruction of the specialized `read` code sets the inside-flag to indicate that a thread is inside. The last instruction in the specialized `read` code clears the inside-flag.

To simplify the synchronization of threads during replugging, the replugging algorithm uses a queue, called the *holding tank*, to stop the thread that happens to invoke the specialized kernel call while replugging is taking place. Upon completion of replugging, the algorithm activates the thread waiting in the holding tank. The thread then resumes the invocation through the unspecialized code.

For simplicity, we describe the replugging algorithm as if there were only two cases: specialized and non-specialized. The paths take the following steps:

1. Check the file descriptor to see if this file is specialized. If not, branch out of the fast path.
2. Set inside-flag.
3. Branch indirect. This branch leads to either the holding tank or the read path. It is changed by the replugger.

Read Path:

1. Do the read work.
2. Clear inside-flag.

Holding Tank:

1. Clear inside-flag.
2. Sleep on the per-file lock to await replugger completion.
3. Jump to standard read path.

Replugging Algorithm:

1. Acquire per-process lock to block concurrent repluggers. It may be that some guard was triggered concurrently for the same file descriptor, in which case we are done.
2. Acquire per-file lock to block exit from holding tank.
3. Change the per-file indirect pointer to send readers to the holding tank (changes action of the reading thread at step 3 so no new threads can enter the specialized code).
4. Spinwait for the per-file inside-flag to be cleared. Now no threads are executing the specialized code.
5. Perform incremental specialization according to which invariant was invalidated.
6. Set file descriptor appropriately, including indicating that the file is no longer specialized.
7. Release per-file lock to unblock thread in holding tank.
8. Release per-process lock to allow other repluggers to continue.

The way the replugger synchronizes with the reader thread is through the inside-flag in combination with the indirection pointer. If the reader sets the inside-flag before a replugger sets the indirection pointer then the replugger waits for the reader to finish. If the reader takes the indirect call into the holding tank, it will clear the inside-flag which will tell the replugger that no thread is executing the specialized code. Once the replugging is complete the algorithm unblocks any thread in the holding tank and they resume through the new unspecialized code.

In most cases of unspecialization, the general case, `read`, is used instead of the specialized `is_read`. In this case, the file descriptor is marked as unspecialized and the memory `is_read` occupies is marked for garbage collection at file `close` time.

4 Performance Results

The experimental environment for the benchmarks was a Hewlett-Packard 9000 series 800 G70 (9000/887) dual-processor server [1] running in single-user mode. This server is configured with 128 MB of RAM. The two PA7100 [16] processors run at 96 MHz and each contains one MB of instruction cache and one MB of data cache.

Section 4.1 presents an experiment to show how incremental specialization can reduce the overhead of the `read` system call. Sections 4.2 through 4.4 describe the overhead costs of specialization. Section 4.5 describes the basic cost/benefit analysis of when specialization is appropriate.

4.1 Specialization to Reduce `read` Overhead

The first microbenchmark is designed to illustrate the reduction in overhead costs associated with the `read` system call. Thus the experiment has been designed to reduce all other costs, to wit:

- all experiments were run with a warm file system buffer cache [3]

[2] Take an unexpected path out of the kernel on failure.

[3] The use of specialization to optimize the device I/O path and make better use of the file system buffer cache is the subject of a separate study currently underway in our group.

	1 Byte	8 KB	64 KB
`read`	3540	7039	39733
`is_read`	979	5220	35043
Savings	2561	1819	4690
% Improvement	72%	26%	12%
Speedup	x3.61	x1.35	x1.13
`copyout`	170	3400	NA[4]

Table 3: Overhead Reduction: HP-UX `read` versus `is_read` costs in CPU cycles

Figure 3: Overhead Reduction: 1 Byte HP-UX `read` versus `is_read` costs in CPU cycles

- the same block was repeatedly read to achieve all-hot CPU data caches
- the target of the `read` is selected to be a page-aligned user buffer to optimize `copyout` performance

The program consists of a tight loop that opens the file, gets a timestamp, reads N bytes, gets a timestamp, and closes the file. Timestamps are obtained by reading the PA-RISC's interval timer, a processor control register that is incremented every processor cycle [20].

Table 3 numerically compares the performance of HP-UX `read` with `is_read` for reads of one byte, 8 KB, and 64 KB, and Figures 3 through 5 graphically represents the same data. The table and figures also include the costs for `copyout` to provide a lower bound on `read` performance[5]. In all cases, `is_read` performance is better than HP-UX `read`. For single byte reads, `is_read` is more than three times as fast as HP-UX `read`, reflecting the fact that specialization has removed most of the overhead of the `read` system call.

Reads that cross block boundaries lose the specialization of simply continuing to copy data from the previous position in the current buffer cache block (the SEQUENTIAL_ACCESS quasi-invariant), and thus suffer performance loss. 8 KB reads necessarily cross block boundaries, and so the specialization performance improvement for the 8 KB `is_read` has a smaller performance gain than the 1 Byte `is_read`. For larger reads, the performance gain is not so large because the overall time is dominated by data copying costs rather than overhead. However, there are performance benefits from specialization that occur on a per-block basis, and so even 64 KB reads improve by about 12%.

4.2 The Cost of the Initial Specialization

The performance improvements in the `read` fast path come at the expense of overhead in other parts of the system.

[4]Large reads break down into multiple block-sized calls to `copyout`.

[5]`copyout` performs a safe copy from kernel to user space while correctly handling page faults and segmentation violations.

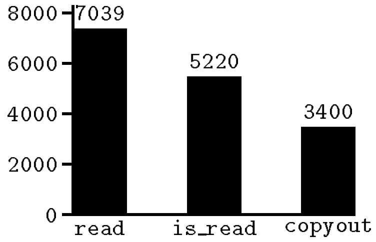

Figure 4: Overhead Reduction: 8 KB HP-UX `read` versus `is_read` costs in CPU cycles

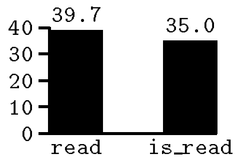

Figure 5: Overhead Reduction: 64 KB HP-UX `read` versus `is_read` costs in Thousands of CPU cycles

The most significant impact occurs in the `open` system call, which is the point at which the `is_read` path is generated. `open` has to check 8 invariant and quasi-invariant values, for a total of about 90 instructions and a lock/unlock pair. If specialization can occur it needs to allocate some kernel memory and fill it in. close needs to check if the file descriptor is or was specialized and if so, free the kernel memory. A kernel memory `alloc` takes 119 cycles and `free` takes 138 cycles.

The impact of this work is that the new specialized `open` call takes 5582 cycles compared to 5227 cycles for the standard HP-UX `open` system call. In both cases, no inode traversal is involved. As expected, the cost of the new `open` call is higher than the original. However, notice that the increase in cost is small enough that a program that opens a file and reads it once can still benefit from specialization.

4.3 The Cost of Nontriggered Guards

The cost of guards can be broken down into two cases: the cost of executing them when they are not triggered, and the cost of triggering them and performing the necessary replugging. This sub-section is concerned with the first case.

Guards are associated with each of the system calls shown in Table 2. As noted elsewhere, there are two sorts of guards. One checks for specialized file descriptors and is very cheap, the other for specialized inodes. Since inodes can be shared they must be locked to check them. The lock expense is only incurred if the file passes all the other tests first. A lock/unlock pair takes 145 cycles. A guard requires 2 temporary registers, 2 loads, an add, and a compare, 11 cycles, and then a function call if it is triggered. It is important to note that these guards do not occur in the data transfer system calls, except for `readv` which is not frequently used.

In the current implementation, guards are fixed in place (and always perform checks) but they are triggered only when specialized code exists. Alternatively, guards could be inserted in-place when associated specialized code is generated. Learning which alternative performs better requires further research on the costs and benefits of specialization mechanisms.

4.4 The Cost of Replugging

There are two costs associated with replugging. One is the overhead added to the fast path in `is_read` for checking if it is specialized and calling `read` if not, and for writing the inside-flag bit twice, and the indirect function call with zero arguments otherwise. A timed microbenchmark shows this cost to be 35 cycles.

The second cost of replugging is incurred when the replugging algorithm is invoked. This cost depends on whether there is a thread already present in the code path to be replugged. If so, the elapsed time taken to replug can be dominated by the time taken by the thread to exit the specialized path. The worst case for the `read` call occurs when the thread present in the specialized path is blocked on I/O. We are working on a solution to this problem which would allow threads to "leave" the specialized code path when initiating I/O and rejoin a replugged path when I/O completes, but this solution is not yet implemented.

In the case where no thread is present in the code path to be replugged, the cost of replugging is determined by the cost of acquiring two locks, one spinlock, checking one memory location and storing to another (to get exclusive access to the specialized code). To fall back to the generic `read` takes 4 stores plus address generation, plus storing the specialized file offset into the system file table which requires obtaining the File Table Lock and releasing it. After incremental specialization two locks have to be released. An inspection of the generated code shows the cost to be about 535 cycles assuming no lock contention. The cost of the holding tank is not measured since that is the rarest subcase and it would be dominated by spinning for a lock in any event.

Adding up the individual component costs, and multiplying them by the frequency, we can estimate the guarding and replugging overhead attributed to each `is_read`. If we assume that 100 `is_read` happen for each of the guarded kernel calls (`fork`, `creat`, `truncate`, `open`, `close` and replugging), then less than 10 cycles are added as guarding overhead to each invocation of `is_read`.

4.5 Cost/Benefit Analysis

Specialization reduces the execution costs of the fast path, but it also requires additional mechanisms, such as guards and replugging algorithms, to maintain system correctness. By design, guards are located in low frequency execution paths and in the rare case of quasi-invariant invalidation, replugging is performed. We have also added code to `open` to check if specialization is possible, and to `close` to garbage collect the specialized code after replugging. An informal performance analysis of these costs and a comparison with the gains is shown in Figure 6.

In Equation 1, *Overhead* includes the cost of guards, the replugging algorithm, and the increase due to initial invariant validation, specialization and garbage collecting for all file opens and closes. Each $Guard^i$ (in different kernel calls) is invoked $f^i_{syscall}$ times. Similarly, *Replug* is invoked f_{Replug} times. A small part of the cost of synchronization with the replugger is born by `is_read` (the setting and resetting of inside-flag), but overall `is_read` is much faster than `read` (Section 4). In Equation 2, f_{is} is the number of times specialized `is_read` is invoked and $f_{TotalRead}$ is the total number of invocations to read the file. Specialization wins if the inequality in Equation 2 is true.

The following sections outline a series of microbenchmarks to measure the performance of the incrementally and optimistically specialized `read` fast path, as well as the overhead associated with guards and replugging.

5 Discussion

The experimental results described in Section 4 show the performance of our current `is_read` implementation. At the time of writing this implementation was not fully specialized: some invariants were not used and, as a result, the measured `is_read` path contains more interpretation and traversal code than is absolutely necessary. Therefore, the performance results presented above are conservative. Even so, the results show that optimistic specialization can improve the performance of both small and large reads.

At one end of the spectrum, assuming a warm buffer cache, the performance of *small reads* is dominated by control flow costs. Through specialization we are able to remove, from the fast path, a large amount of code, concerned with interpretation, data structure traversal and synchronization. Hence, it is not surprising that the cost of small reads is reduced significantly.

At the other end of spectrum, again assuming a warm buffer cache, the performance of *large reads* is dominated by data movement costs. In effect, the control-flow overhead, still present in large reads, is amortized over a large amount of byte copying. Specializations to reduce the cost of byte copying will be the subject of a future study.

Specialization removes overhead from the fast path by adding overhead to other parts of the system: specifically, the places at which the specialization, replugging and guarding of optimistic specializations occur. Our experience has shown that generating specialized implementations is easy. The real difficulty arises in correctly placing guards and making policy decisions about what and when to specialize and replug. Guards are difficult to place because an operating system kernel is a large program, and invariants often correspond to global state components which are manipulated at numerous sites. Therefore, manually tracking the places where invariants are invalidated is a tedious process. However, our experience with the HP-UX file system has brought us an understanding of this process which we are now using to design a program analysis capable of determining a conservative approximation of the places where guards should be placed.

Similarly, the choice of what to specialize, when to specialize, and whether to specialize optimistically are all nontrivial policy decisions. In our current implementation we made these decisions in an *ad hoc* manner, based on our expert knowledge of the system implementation, semantics and common usage patterns. This experiment has prompted us to explore tools based on static and dynamic analyses of kernel code, aimed at helping the programmer decide when the performance improvement gained by specialization is likely to exceed the cost of the specialization process, guarding, and replugging.

$$Overhead = \sum_i f^i_{syscall} * Guard^i + Open + Close + f_{Replug} * Replug \quad (1)$$

$$Overhead + f_{is} * \texttt{is\_read} < (f_{TotalRead} - f_{is}) * \texttt{read} \quad (2)$$

Figure 6: Cost/Benefit Analysis: When is it Beneficial to Specialize?

5.1 Interface Design and Kernel Structure

From early in the project, our intuition told us that, in the most specialized case, it should be possible to reduce the cost of a `read` system call that hits in the buffer cache. It should be little more than the basic cost of data movement from the kernel to the application's address space, i.e., the cost of copying the bytes from the buffer cache to the user's buffer. In practice, however, our specialized `read` implementation costs considerably more than copying one byte. The cost of our specialized `read` implementation is 979 cycles, compared to approximately 235 cycles for entering the kernel, fielding the minimum number of parameters, and safely copying a single byte out to the application's address space.

Upon closer examination, we discovered that the remaining 744 cycles were due to a long list of inexpensive actions including stack switching, masking floating point exceptions, recording kernel statistics, supporting `ptrace`, initializing kernel interrupt handling, etc. The sum of these actions dominates the cost of small byte reads.

These actions are due in part to constraints that were placed upon our design by an over-specification of the UNIX `read` implementation. For example, the need to always support statistics-gathering facilities such as `times` requires every `read` call to record the time it spends in the kernel. For one byte reads, changing the interface to one that returns a single byte in a register instead of copying data to the user's address space, and changing the semantics of the system call to not support statistics and profiling eliminates the need for many of these actions.

To push the limits of a kernel-based `read` implementation, we implemented a special one-byte `read` system call, called `readc`, which returns a single byte in a register, just like the `getc` library call. In addition to the optimizations used in our specialized `is_read` call, `readc` avoids switching stacks, omits `ptrace` support, and skips updating profile information. The performance of the resulting `readc` implementation is 65 cycles. Notice that aggressive use of specialization can lead to a `readc` system call that performs within a factor of two of a pure user-level `getc` which costs 38 cycles in HP-UX's `stdio` library. This result is encouraging because it shows the feasibility of implementing operating system functionality at kernel level with performance similar to user-level libraries. Aggressive specialization may render unnecessary the popular trend of duplicating operating system functionality at user level [2, 19] for performance reasons.

Another commonly cited reason for moving operating system functionality to user level is to give applications more control over policy decisions and operating system implementations. We believe that these benefits can also be gained without duplicating operating system functionality at user level. Following an open-implementation (OI) philosophy [22], operating system functionality can remain in the kernel, with customization of the implementation supported in a controlled manner via meta-interface calls [23]. A strong lesson from our work and from other work in the OI community [22] is that abstractly specified interfaces, i.e., those that do not constrain implementation choices unnecessarily, are the key to gaining the most benefit from techniques such as specialization.

6 Related Work

Our work on optimistic incremental specialization can be viewed as part of a widespread research trend towards adaptive operating systems. Micro-kernel operating systems [5, 6, 11, 9, 21, 27, 29] were an early example of this trend, and improve adaptiveness by allowing operating system functionality to be implemented in user-level servers that can be customized and configured to produce special-purpose operating systems. While micro-kernel-based architectures improve adaptiveness over monolithic kernels, support for user-level servers incurs a high performance penalty.

To address this performance problem Chorus [29] allows modules, known as supervisor actors, to be loaded into the kernel address space. A specialized IPC mechanism is used for communication between actors within the kernel address space. Similarly, Flex [8] allows dynamic loading of operating system modules into the Mach kernel, and uses a migrating threads model to reduce IPC overhead.

One problem with allowing applications to load modules into the kernel is loss of protection. The SPIN kernel [4] allows applications to load executable modules, called *spindles*, dynamically into the kernel. These spindles are written in a type-safe programming language to ensure that they do not adversely affect kernel operations.

Object-oriented operating systems allow customization through the use of inheritance, invocation redirection, and meta-interfaces. Choices [7] provides generalized components, called frameworks, which can be replaced with specialized versions using inheritance and dynamic linking. The Spring kernel uses an extensible RPC framework [18] to redirect object invocations to appropriate handlers based on the type of object. The Substrate Object Model [3] supports extensibility in the AIX kernel by providing additional interfaces for passing usage hints and customizing in-kernel implementations. Similarly, the Apertos operating system [31] supports dynamic reconfiguration by modifying an object's behavior through operations on its meta-interface.

Other systems, such as the Cache kernel [10] and the Exo-kernel [17] address the performance problem by moving even more functionality out of the operating system kernel and placing it closer to the application. In this "minimal-kernel" approach extensibility is the norm rather than the exception.

Synthetix differs from the other extensible operating systems described above in a number of ways. First, Synthetix infers the specializations needed even for applications that have never considered the need for specialization. Other extensible systems require applications to know which specializations will be beneficial and then select or provide them.

Second, Synthetix supports optimistic specializations and uses guards to ensure the validity of a specialization and automatically replug it when it is no longer valid. In contrast, other extensible systems do not support automatic replugging and support damage control only through hardware or software protection boundaries.

Third, the explicit use of invariants and guards in Synthetix also supports the composability of specializations: guards determine whether two specializations are composable. Other extensible operating systems do not provide support to determine whether separate extensions are composable.

Like Synthetix, Scout [26] has focused on the specialization of existing systems code. Scout has concentrated on networking code and has focused on specializations that minimize code and data caching effects. In contrast, we have focused on parametric specialization to reduce the length of various fast paths in the kernel. We believe that many of the techniques used in Scout are also useful in Synthetix, and vice versa.

7 Conclusions

This paper has introduced a model for specialization based on invariants, guards, and replugging. We have shown how this model supports the concepts of incremental and optimistic specialization. These concepts refine previous work on kernel optimization using dynamic code generation in Synthesis [28, 25]. We have demonstrated the feasibility and usefulness of incremental, optimistic specialization by applying it to file system code in a commercial operating system (HP-UX). The experimental results show that significant performance improvements are possible even when the base system is (a) not designed specifically to be amenable to specialization, and (b) is already highly optimized. Furthermore, these improvements can be achieved without altering the semantics or restructuring the program.

Our experience with performing incremental and optimistic specialization manually has shown that the approach is worth pursuing. Based on this experience, we are developing a program specializer for C and a collection of tools for assisting the programmer in identifying opportunities for specialization and ensuring consistency.

8 Acknowledgements

Bill Trost of Oregon Graduate Institute and Takaichi Yoshida of Kyushu Institute of Technology performed the initial study of the HP-UX read and write system calls, identifying many quasi-invariants to use for specialization. Bill also implemented some prototype specialized kernel calls that showed promise for performance improvements. Luke Hornoff of University of Rennes contributed to discussions on specialization and tools development. Finally, we would like to thank the SOSP referees for their extensive and thorough comments and suggestions.

References

[1] Thomas B. Alexander, Kenneth G. Robertson, Dean T. Lindsey, Donald L. Rogers, John R. Obermeyer, John R. Keller, Keith Y. Oka, and Marlin M. Jones II. Corporate Business Servers: An Alternative to Mainframes for Business Computing. *Hewlett-Packard Journal*, 45(3):8–30, June 1994.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[3] Arindam Banerji and David L. Cohn. An Infrastructure for Application-Specific Customization. In *Proceedings of the ACM European SIGOPS Workshop*, September 1994.

[4] 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.

[5] 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.

[6] F. J. Burkowski, C. L. A. Clarke, Crispin Cowan, and G. J. Vreugdenhil. Architectural Support for Lightweight Tasking in the Sylvan Multiprocessor System. In *Symposium on Experience with Distributed and Multiprocessor Systems (SEDMS II)*, pages 165–184, Atlanta, Georgia, March 1991.

[7] Roy H. Campbell, Nayeem Islam, and Peter Madany. Choices: Frameworks and Refinement. *Computing Systems*, 5(3):217–257, 1992.

[8] John B. Carter, Bryan Ford, Mike Hibler, Ravindra Kuramkote, Jeffrey Law, Lay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson. FLEX: A Tool for Building Efficient and Flexible Systems. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 198–202, Napa, CA, October 1993.

[9] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[10] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–193, November 1994.

[11] David R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager. Thoth, A Portable Real-Time Operating System. *Communications of the ACM*, 22(2):105–115, February 1979.

[12] Frederick W. Clegg, Gary Shiu-Fan Ho, Steven R. Kusmer, and John R. Sontag. The HP-UX Operating System on HP Precision Architecture Computers. *Hewlett-Packard Journal*, 37(12):4–22, December 1986.

[13] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[14] C. Consel and F. Noël. A general approach to run-time specialization and its application to C. Report 946, Inria/Irisa, Rennes, France, July 1995.

[15] 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.

[16] Eric DeLano, Will Walker, and Mark Forsyth. A High Speed Superscalar PA-RISC Processor. In *COMPCON 92*, pages 116–121, San Francisco, CA, February 24-28 1992.

[17] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[18] Graham Hamilton, Michael L. Powell, and James G. Mitchell. Subcontract: A flexible base of distributed programming. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 69–79, Asheville, NC, December 1993.

[19] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 187–197, Boston, MA, October 1992.

[20] Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, second edition, September 1992.

[21] Dan Hildebrand. An Architectural Overview of QNX. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–123, Seattle, WA, April 1992.

[22] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proc. of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. See `http://www.xerox.com/PARC/spl/eca/oi.html` for updates.

[23] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[24] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, Reading, MA, 1989.

[25] 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.

[26] David Mosberger, Larry L. Peterson, and Sean O'Malley. Protocol Latency: MIPS and Reality. Report TR 95-02, Dept of Computer Science, University of Arizona, Tuscon, Arizona, April 1995.

[27] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba — A distributed Operating System for the 1990's. *IEEE Computer*, 23(5), May 1990.

[28] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[29] 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.

[30] P. Sestoft and A. V. Zamulin. Annotated bibliography on partial evaluation and mixed computation. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[31] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 414–434, Vancouver, BC, October 1992.