October 1995

Fast concurrent dynamic linking for an adaptive operating system

Crispin Cowan

Tito Autrey

Calton Pu

Jonathan Walpole

Fast Concurrent Dynamic Linking for an Adaptive Operating System *

Crispin Cowan, Tito Autrey, Calton Pu, and Jonathan Walpole

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

October 26, 1995

Abstract

The need for customizable and application-specific operating systems has been recognized for many years. A customizable operating system is one that can adapt to some particular circumstance to gain some functional or performance benefits. Microkernels have attempted to address this problem, but suffer performance degradation due to the cost of inter-process protection barriers. Commercial operating systems that can efficiently adapt themselves to changing circumstances have failed to appear, in part due to the difficulty of providing an interface that is efficient to invoke, provides a protection barrier, and can be dynamically reconfigured.

Providing such a safe, efficient, and dynamic interface in a concurrent operating system requires an effective concurrency control mechanism to prevent conflicts between system components proposing to *execute* specialized components, and those components responsible for dynamically replacing specialized components. This paper outlines our basic approach to specialization of operating systems, and details our dynamic replacement mechanism and its concurrency control features.

1   Introduction

A key dilemma faced by operating system developers is the need to produce software that is both general-purpose and efficient. Operating systems must execute correctly under all

---

1

conditions, but must also exhibit high performance under certain common conditions. The conventional approach to this dilemma is to write code that is general-purpose, but optimized for a single anticipated common case. The result is an implementation with functionality and performance characteristics that are fixed throughout the lifetime of the operating system.

The need for configurability arises when the anticipated common case doesn't match the characteristics of some important application. This situation can arise when the application in question was developed after the operating system, or when the operating system developer simply failed to recognize the importance of this application or class of applications.

The problem can be serious when the optimizations embedded in the operating system are particularly bad for the new application. For instance, the usual paging policy provided by most operating systems is simply not appropriate for database applications [28]. Mukherjee and Schwan [20] showed that both spinlocks and blocking locks can provide superior performance under different circumstances. Thus it is important to provide operating system facilities that are appropriate to the application.

Micro-kernels have addressed this problem by providing a minimal kernel, and encapsulating the rest of OS functionality in replaceable server processes [4, 5, 9, 10, 15, 22, 27]. Such systems can be customized by replacing or providing additional servers that implement the desired policies while making use of existing mechanisms provided by the micro-kernel. Using this approach, customization is supported at a coarse granularity, through the replacement of complete servers.

The customizability that comes from restructuring operating systems as collections of user-level servers is not free. System calls that previously involved only procedure calls and accesses to shared data within the kernel now incur the overhead of virtual memory context switches and thread switches associated with message passing across protection boundaries.

In view of the fact that performance really matters, operating system researchers have explored several alternatives to the micro-kernel approach. Projects such as SPIN [3] and the Exokernel [11] provide facilities to allow applications to incorporate their own variants of OS functionality while moving the traditional microkernel protection barriers up or down so as to minimize the performance penalties of dynamic configuration. Object-oriented operating systems use objects for configurability and to provide protection.

The Synthetix approach to OS configuration is to automatically provide **specialized** implementations of various OS services. The specialized implementations are tuned to provide improved OS performance by exploiting **invariants** that are particular to the application

2

using the service.

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.

Exploiting these ideas in an operating system with concurrent processes requires an effective concurrency control mechanism to prevent conflicts between system components proposing to *execute* specialized components, and those components responsible for dynamically replacing specialized components. This paper outlines our basic approach to specialization of operating systems, and details our dynamic replacement mechanism and its concurrency control features.

Section 2 elaborates on the Synthetix notions of specialization. Section 3 describes our experiments exploiting specialization in systems. Section 4 details our mechanism for safe, concurrent replacement of specialized modules, a process we call **replugging**. Section 5 describes related work. Finally Section 6 discusses our conclusions and future work.

2   Specialization

The Synthetix project seeks to define a systematic approach to dynamic customization of an operating system. We begin with a high-level specification of customization requirements using **invariants**. A **true invariant**, like a classical invariant, is a state property of the system that is guaranteed to be true at all times. A **quasi-invariant** is a state property that is momentarily true, but may be falsified at some future point in time.

Once invariants have been established, **specialized modules** can be prepared to replace their generic counterparts in the system. A specialized module can either be a specialization of mechanism or of policy. A specialized mechanism is a more efficient implementation of the same functionality, optimized using **partial evaluation** with respect to the invariants. A

specialized policy module provides the same interface as its generic counterpart, but changes the behavior of the module to provide improved performance to the application, e.g. a specialized file pre-fetching policy.

Quasi-invariants can be falsified, potentially making their corresponding specialized modules either inefficient or invalid. Thus quasi-invariants must be **guarded**. A **guard** is a test placed at a location in the system where a quasi-invariant might be invalidated: if execution of that point invalidates the quasi-invariant, then the guard **re-plugs** all the specialized modules that depend on it with less specialized modules that do not depend on the falsified quasi-invariant. Because a specialized module that depends on quasi-invariants can be removed, possibly even before it is used, we refer to the use of such specialized modules as **optimistic specialization**.

Specialized modules can be installed when ever the appropriate set of invariants and quasi-invariants is discovered to be true. Discovering that an invariant is true requires the same set of checks as discovering that an invariant is false, and so the aforementioned guards can be used to instantiate the use of specialized modules, allowing the operating system to *infer* the specializations that should be used. Sometimes, however, invariants are discovered to be true at different points in time. In that case, the specialized module may be replaced with one that is *more* specialized than the current module. We call this **incremental specialization**. Section 3.1 describes an experimental modification of the HP-UX operating system to exploit the techniques of optimistic and incremental specialization.

The HP-UX experiment is an example of mechanism specialization. An example of policy specialization is illustrated by a fine-grained specialization technique called **software feedback**. Software feedback proposes that in a system containing producer and consumer processes, the consumer feed back properties of it's input to the producer so as to balance and optimize the data flow. Section 3.2 describes our distributed video/audio player that uses software feedback to dynamically adapt to the changing bandwidth provided by the Internet. This example serves to illustrate two concepts: The feedback messages produced by the consumer explicitly change the behavior of the system, thus feedback constitutes a policy specialization rather than a mechanism specialization.

Software feedback re-specializes the behavior of the system between individual invocations of the system call to fetch data. Thus software feedback is a much finer-grained example of specialization than has previously been discussed; instead of replacing a module once and for all, as in a microkernel, or once a specialization opportunity is discovered, as in our

4

HP-UX experiment, software feedback continuously re-specializes the system. None the less, software feedback can still be modeled using the Synthetix model for specialization: the consumer describes the properties of its input data stream as quasi-invariants, and when these quasi-invariants are violated, a feedback message is sent to the producer to correct the data stream so that the quasi-invariant will *again* be true.

3   Special Experiences

We have experimentally validated the performance benefits of specialization. This section reviews these experiments. Subsection 3.1 describes optimistic and incremental mechanism specialization in the HP-UX operating system. Subsection 3.2 describes fine-grained policy specialization through software feedback in a distributed multimedia player.

3.1   Optimistic and Incremental Specialization in HP-UX

The experiment in [24] sought to evaluate the effectiveness of mechanism specialization in a commercial operating system. Previous work [18, 26] had already shown that specialized mechanism could provide performance benefits of up to a factor of 56 [17], but this work did not clearly distinguish between the benefits provided by specialized mechanisms and benefits provided by other means, such as a kernel hand-coded in assembler.

In this experiment we produced a specialized implementation of the `read` system call mechanism. The specialized `read` implementation exploits several true invariants and quasi-invariants to produce a simpler and faster `read` mechanism. For instance, the generic `read` mechanism is forced to interpret numerous data structures that describe the type of the object being read (file, socket, etc.), the type of the file system (local or network), and the parameters of the file system (block size, etc.). However, once a specific file is opened, these values all become fixed as true invariants. Thus a faster implementation of the `read` mechanism can be created that does not check these parameters, but instead hard-codes them into the specialized `read` mechanism.

The generic `read` mechanism also acquires several concurrency locks on kernel data structures to protect against errors that may occur if more than one process concurrently accesses these data structures. However, it is possible to determine at `open` time whether there *are* any concurrent processes accessing the file. If there are not, then it is a quasi-invariant that the file is not shared, and the acquisition of the concurrency locks can be omitted from the

specialized `read` mechanism. This is an important savings, because concurrency locks can be quite slow on shared memory multiprocessors [2].

Non-sharing of files is a quasi-invariant, because at any time another process may open the file and access it. To protect against this possibility, guards are placed in all locations in the kernel where files may be opened (`open`, `creat`, etc.). If it is detected that the file being opened has been "specialized", i.e. is being accessed by a specialized implementation of `read`, then the quasi-invariant has been violated, and the specialized `read` mechanism is replaced with a more generic mechanism that does not depend on the "non-shared" quasi-invariant.

Applications are designed around the fact that OS system calls are expensive to use because of high software overhead, and thus system calls such as `read` are usually used to read large blocks of data. Our experiments show that a specialized implementation can reduce the software overhead of the `read` system call by more than a factor of three. Such a reduction in system call overhead not only improves application performance, it also enables a more flexible use of OS system calls.

3.2 Policy Specialization Through Software Feedback

Two of the hottest topics in current computer systems are the Internet and multimedia. Unfortunately, they don't work well together: multimedia presentations demand real-time performance, while the bandwidth and latency characteristics of the Internet are highly variable and impossible to control. It is therefore necessary for distributed multi-media systems to **adapt** to the changing conditions found in a distributed network. This experiment showed how the use of **feedback** to make multimedia presentations **adaptive** enables video to be played across an irregular network such as the Internet without benefit of resource reservations [8].

We use **software feedback** [18, 25], reminiscent of hardware feedback, to *adapt* multimedia presentations to the changing conditions of the Internet. Our video player has a distributed client-server architecture. The client measure various properties of its video stream its receiving from the network, and feeds them back to the server, allowing both the client and the server to adapt to changing Internet conditions.

Software feedback takes the form of quasi-invariants and guards. If a quasi-invariant is true, then the present state is within tolerance, and no feedback is required. If the quasi-invariant is violated, then some property has exceeded tolerance, and some form of feedback action is necessary. Guards detect the violation of the quasi-invariants, and induce feedback

6

events which undertake to make the quasi-invariant *again* true.

For instance, it is desirable that the server only send as many frames per second as the network can support; sending additional frames just wastes bandwidth, because these frames are either dropped by the network, or discarded by the client because they arrived to late to be useful. Thus we use a quasi-invariant that the server's frame transmission rate is within ϵ of the client's frame display rate. If a guard detects that this quasi-invariant has been violated, then a feedback message is sent to tell the server to adjust it's frame transmission rate so that the client and server's frame rates will again be within ϵ tolerance of one another.

The invariants and guards used in software feedback are similar to those used in mechanism specialization. However, the actions taken by the guards that detect violations of quasi-invariants are different. Rather than replacing one mechanism with another, the guards take explicit actions that cause components of the system to change their operational behavior, effectively changing the component's policy. Thus software feedback is a form of policy specialization.

The guards are also triggered much more frequently, and the corrective actions they take are much cheaper than replacing one mechanism with another. Thus software feedback is much finer-grained than mechanism specialization. However, it is not always the case that policy specialization is fine-grained. In future research, we will examine the prospects for larger-scale policy specializations in an operating system, such as paging policy, or file system pre-fetching policy.

4   The Replugging Algorithm

The use of specialized implementations requires a facility that supports the efficient invocation of specialized implementations while also supporting safe, concurrent replacement of specialized implementations. Concurrently executing and modifying system components presents a concurrency control problem. The purpose of this paper is to present our system for satisfying these constraints.

Supporting mechanism specialization, as in the HP-UX experiment in Subsection 3.1, basically requires a simple indirect jump, as in a C function pointer. However, an adaptive operating system that is using optimistic and incremental specialization must also deal with concurrency problems. Concurrency errors may occur in the following circumstances:

- A guard triggers replugging of a specialized function concurrent with another process

attempting to execute that specialized function.

- A guard triggers replugging of a specialized function while another process is blocked inside that specialized function, e.g. waiting for disk I/O.

- Two or more guards concurrently trigger replugging of the same specialized function.

The system must correctly handle the possibility of the above concurrent events. Furthermore, it is imperative that the execution of specialized functions not be significantly slowed down by the concurrency control of the replugging system, or else the benefits of specialization will be lost.

Policy specialization, as described in the software feedback experiment in Subsection 3.2, can also be supported using indirect jumps. However, software feedback may only require the changing of some parameters, rather than replacing implementation modules. None the less, the same concurrency errors may occur if the parameters are concurrently used by an executing process and changed by a replugging action. The same performance constraints also apply: executing specialized modules must not suffer delay due to acquiring concurrency locks.

4.1   Basic Replugging

It is possible to solve this concurrency problem using some form of wait-free synchronization [14]. However, wait-free synchronization is not always faster than simple locking, can be considerably more complex, and may require hardware support for variations of atomic test&set instructions. Instead, we implemented *asymmetric* locking: both the processes executing specialized functions, and the processes replugging specialized functions must acquire and release locks on the specialized function. The asymmetry is that the method used to acquire and release the lock for processes executing the specialized function is considerably faster than that used by replugging processes.

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 some assumptions that are true in many UNIX systems:

1. Kernel calls cannot abort, i.e. take an unexpected path out of the kernel on failure. This assumption allows us to avoid checking for an incomplete kernel call to `is_read`.

8

2. There is only one thread per process. This assumption allows us to assume that multiple kernel calls cannot concurrently access process level data structures.

3. That there can be at most one thread executing inside specialized code. This assumption simplifies the streamlined lock acquisition for executing specialized functions.

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 to the standard `read` function.

2. Set `inside-flag`.

3. Branch indirect. This branch leads to either the holding tank or the read path. The indirect address 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 the new read path pointed to by the replugger.

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 replugger synchronizes with the reader thread 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.

To generalize this algorithm so that more than one thread can execute a specialized function at the same time, the `inside-flag` should be changed to an `inside-counter` that is incremented and decremented by threads using the specialized function. Thus the fast-path lock becomes a counting semaphore, rather than a boolean semaphore. The replugging path continues to check the `inside-counter`, as before, but only proceeds with replugging when the `inside-counter` is 0. This method suffices so long as all paths *out* of the specialized read path decrement the `inside-counter`, including abnormal terminations.

The replugging procedure suffers from substantial inefficiency if a thread executing the specialized `read` blocks for disk I/O, because it will spinwait at step 4 until the I/O completes. To make the the replugging procedure more efficient, a second `blocked` flag is added. Because disk I/O is orders of magnitude slower than computation, conventional semaphores can be used for this purpose, and so `blocked` is a conventional semaphore. Before blocking on the disk I/O, the executing thread thread sets the `blocked` flag. Before entering the spinwait at step 4, the replugging algorithm checks the `blocked` flag, and if set, waits on `blocked` instead of spinwaiting.

4.2   Distributed Replugging

Replugging components in a distributed system presents additional problems. Replugging a component becomes slower because of the additional latency imposed by waiting until it can be assured that no other process will try to execute the facility being replugged on any other machine. Fault-tolerant execution on a system in which network failures can occur is especially difficult.

Software feedback addresses this problem with careful design of the policy specializations. Software feedback does not have to synchronize between replugging on one machine and execution on another machine because there is a momentary tolerance of inconsistency between the machines. After replugging has been effected on one machine, that machine *ignores* further violations of the relevant quasi-invariants for a period of time, so as to allow the system to adapt to the replugging action that has just been applied.

In classical feedback terms, this can be viewed as **damping** the software feedback mechanism so as to avoid repeating oscillations due to positive feedback. Damping further feedback for a period of time that is longer than twice the longest possible propagation delay across the distributed system suffices to guarantee that oscillation due to positive feedback will not occur.

4.3   Replugging Performance

The dynamic replugging system just described has two important performance figures:

1. The added overhead of invoking a replaceable function.

2. The cost of replacing a function.

The overhead added to the fast path in `is_read` includes checking if it is specialized and calling `read` if not, and for writing the inside-flag bit twice, and the indirect function[1] call with zero arguments otherwise. A timed microbenchmark shows this cost to be 35 cycles.

The cost of replacing a function 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

---

[1] What is really desired is an indirect branch, with no register save or stack frame overhead. It would require sophisticated compiler analysis to determine that such a specialized procedure call linkage could be used, much less construct it.

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, effectively employing a wait-free synchronization technique, 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.

5   Related Work

Our dynamic linking mechanism is motivated by the needs of optimistic and incremental specialization, and so it is subtly different from previous dynamic linking mechanisms. The dld tool [16] provides for basic dynamic linking, but is designed to be used by an application program. As such, it does not deal with concurrency issues, but does partially automate the garbage collection of un-used functions from the program's address space.

The OMOS system [23] dynamically links modules in a system in an object-oriented manner. OMOS automatically decides which version of a module will best meet the specified requirements, functioning somewhat like an interface definition language. OMOS does deal with concurrency issues, but functions at a coarser granularity than our system because it was designed to support the Flex microkernel [7].

Chorus [27] 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 [7] 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 [3] 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. Mukherjee and Schwan et al [12, 20, 21] control concurrent execution and modification of an object using attributes and object ownership. Choices [6] 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 [13] to redirect object invocations to appropriate handlers based on the type of object. The Substrate Object Model [1] supports extensibility in the AIX kernel by providing additional interfaces for passing usage hints and customizing in-kernel implementations. Similarly, the Apertos operating system [29] supports dynamic reconfiguration by modifying an object's behavior through operations on its meta-interface.

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 automatically determine whether separate extensions are composable.

Like Synthetix, Scout [19] 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.

6 Conclusions

This paper has described an efficient mechanism to support the concurrent execution and replacement of functions in an operating system. Such a facility is essential for operating systems that wish to adaptively reconfigure themselves at a fine granularity. Fine-grained adaptivity is required to use the techniques of optimistic and incremental specialization, described here and in [8, 24].

We have demonstrated the feasibility and usefulness of dynamic linking as used by incremental and 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.

In future work, we will explore more efficient and general methods of implementing concurrent dynamic linking. We will also investigate methods to automate the techniques of optimistic and incremental specialization, so as to make these techniques more accessible to the kernel developer.

7 Acknowledgements

References

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

[2] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 223–233, Boston, MA, September 1992.

[3] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Perfor-

mance in the SPIN Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[4] 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, WA, April 1992.

[5] 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.

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

[7] 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.

[8] Shanwei Cen, Calton Pu, Richard Staehli, Crispin Cowan, and Jonathan Walpole. A Distributed Real-Time MPEG Video Audio Player. In *Proceedings of the 1995 International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'95)*, New Hampshire, April 1995.

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

[10] 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.

[11] 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.

[12] Ahmed Gheith, Bodhisattwa Mukherjee, Dilma Silva, and Karsten Schwan. KTK: Kernel Support for Configurable Objects and Invocations. In *International Workshop on Configurable Distributed Systems*, March 1994. Also available as GIT-CC-94/11, `ftp://ftp.cc.gatech.edu/pub/coc/tech_reports/1994/GIT-CC-94-11.ps.Z`.

[13] 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 (SOSP'93)*, pages 69–79, Asheville, NC, December 1993.

[14] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[15] 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.

15

[16] W. Wilson Ho and Ronald A. Olsson. An Approach to Genuine Dynamic Linking. *Software — Practice and Experience*, 21:375–390, April 1991.

[17] Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis Kernel. In *Symposium on Operating Systems Principles*, 1989.

[18] Henry Massalin and Calton Pu. Fine-Grain Adaptive Scheduling Using Feedback. *Computing Systems*, 3(1):139–173, Winter 1990.

[19] 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.

[20] Bodhisattwa Mukherjee and Karsten Schwan. Experiments with Configurable Locks for Multiprocessors. Report GIT-CC-93/05, College of Computing, Georgia Institute of Technlolgy, Atlanta, GA, January 1993.

[21] Bodhisattwa Mukherjee and Karsten Schwan. Improving Performance by use of Adaptive Object: Experimentation with a Configurable Multiprocessor Thread Package. In *Second IEEE International Symposium on High-Performance Distributed Computing (HPDC-2)*, Spokane, WA, July 1993. Also available as GIT-CC-93/17, `ftp://ftp.cc.gatech.edu/pub/coc/tech_reports/1993/GIT-CC-93-17.ps.Z`.

[22] 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.

[23] Doug Orr. OMOS - an object server for program execution. In *Proc. International Workshop on Object-Oriented Operating Systems*, 1992.

[24] 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.

[25] Calton Pu and Robert M. Fuhrer. Feedback-Based Scheduling: a Toolbox Approach. In *Proceedings of Fourth Workshop on Workstation Operating Systems*, Napa Valley, CA, October 1993.

[26] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis Kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[27] 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, WA, April 1992.

[28] Michael Stonebraker. Operating system Support for Database Management. *Communications of ACM*, 24(7), 1981.

[29] 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'92)*, Vancouver, BC, October 1992.