Customizable Operating Systems *

Jonathan Walpole, Crispin Cowan,
Andrew Black, Jon Inouye, Calton Pu, and Shanwei Cen

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

November 15, 1995

Abstract

A customizable operating system is one that can adapt to improve its functionality or performance. The need for customizable and application-specific operating systems has been recognized for many years, but they have yet to appear in the commercial market. This paper explores the notion of operating system customizability and examines the limits of existing approaches. The paper begins by surveying system structuring approaches for the safe and efficient execution of customizable operating systems. Then it discusses the burden that existing approaches impose on application software, and explores techniques for reducing this burden. Finally, support for customizability in the Synthetix project is described and illustrated through two examples: a dynamically specialized file system `read` call, and an adaptive Internet-based MPEG video player.

Restructuring Operating Systems for Customizability

A key dilemma faced by operating system developers is the need to produce software that is both general-purpose and performance-critical. Operating systems must execute correctly under all conditions, but must also exhibit high performance in common circumstances. 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.

*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.

The need for customizability arises when the anticipated common case doesn't match the characteristics of some important application. This can occur when the application 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 [23].

An important lesson for operating system developers is that their systems must perform well for many common cases. Some of these cases can be anticipated, but others – such as those that arise because of new applications – can not. Moreover, optimizations for one case are likely to be particularly bad for some other case. Hence, the conventional approach of optimizing for a small number of common cases is not viable.

A basic principle that helps to improve the customizability of operating systems is the separation of mechanism from policy. Policies embedded in the operating system are often the cause of the poor performance of applications for which they are inappropriate. One solution to this problem is to allow applications to specify their own policies, in the form of specialized operating system components. In general, this approach requires that interfaces, previously hidden within the kernel, be defined and exposed to application developers.

Hydra [16] was an early example of a customizable operating system. The goal of the Hydra kernel was to implement mechanisms, while allowing application-level software to define policies. The invocation of application-level software on each policy decision was deemed to be impractical due to the high cost of protection domain crossing. Therefore, Hydra implemented parameterized policies such that application-level software could *select* the appropriate parameters but leave the enforcement of the application-specified policy to the kernel. The success of this approach depends on the ability to anticipate appropriate policies for future applications.

Micro-kernel operating systems take an alternative approach by encapsulating some operating system functionality in application-level servers [3, 13, 14, 19, 22]. The interfaces within such operating systems are implemented using message passing facilities provided by the micro-kernel. These systems can be customized by providing additional or replacement servers that implement the desired policies while making use of existing mechanisms provided by the micro-kernel or other servers. Using this approach, customization is supported at a coarse granularity, through the replacement of entire servers. Depending on whether the kernel needs to be recompiled and rebooted to load new servers, customization can be characterized as static or dynamic.

The customizability that comes from restructuring operating systems as collections of application-level servers is not free, however. 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, thread switches, and marshaling and unmarshalling of data, all of which are associated with message passing across protection boundaries. This inter-server communication overhead leads to the use of coarse grain servers and hence coarse grain customizability. In view of the fact that fine-grain adaptivity is desirable and performance really does matter, operating system researchers have explored several alternatives to the micro-kernel approach.

Second-generation micro-kernels reduce message passing overhead by moving performance-critical servers back down into the kernel address space [5, 22]. In this way, inter-server communication can be highly optimized because there is no longer any address space or protection domain crossing when invoking another kernel-resident server.

Some researchers believe that the problem with the micro-kernel approach is that functionality was split at the wrong level, and so micro-kernels require too much communication among servers. Proponents of this position argue that more kernel functionality should be moved up to a higher level, but that the right destination is application-level shared libraries rather than server processes [7, 10]. In this way, operating system code is accessible for customization at a fine granularity, but the cost of invoking a customized service is the cost of a procedure call rather than a system call or message. We refer to this approach as the shared library approach.

Both shared libraries and second-generation micro-kernels optimize performance at the expense of protection. In the second-generation micro-kernel approach, neither the micro-kernel itself nor other kernel-resident servers are protected from downloaded application-specific servers. In the shared library approach, elevated operating system code runs in a write-protected section of the application's address space, but its data is not protected from regular application code.

A key problem for operating system developers is how to support customizability without losing either performance or protection. Application-specific code should execute with performance at least comparable to generic kernel code, but should not be able to read or write arbitrary locations in the kernel address space, unfairly consume system resources, or compromise the integrity of other operating system components.

To meet these challenges, operating system designers are revisiting language-level solutions to protection and encapsulation. For example, the SPIN operating system [2] allows components to be downloaded into the kernel, but they must be written in Modula-3 and compiled using the SPIN compiler to ensure protection. Other systems use lower level software-based protection techniques such as sandboxing to isolate new components without incurring a large performance overhead [11, 24].

Object-oriented operating systems support protection through encapsulation [4]. All operating system components are defined as objects. New custom objects are defined using inheritance and specialization. An added benefit of object-oriented approaches is that they can provide some guidance for customization by requiring that the type of a new custom object conforms to the type of the object that it replaces.

All of the approaches outlined above address customization from the standpoint of operating system structure: how should operating systems be structured so that specialized components can be added in a controlled way? In early systems, the ability to add a specialized component in a clean way was considered useful, even if it required the system to be rebuilt and rebooted. More recently, however, the need for dynamic customization has been recognized.

Gopal et al [12] categorize systems as customizable, extensible, or adaptable, according to the following criteria. A *customizable* operating system allows applications to specify their requirements so that appropriate specialized operating system components can be used for the application. An *extensible* operating system allows new, unforeseen customizations to be incorporated into a running system without requiring it to be rebuilt and rebooted. An *adaptable* operating system allows the customizations to change dynamically during execution to match changing application requirements. Restructuring alone does not support adaptable or extensible operating systems. Such systems require mechanisms for detecting when specialized components are no longer appropriate, and for replacing them dynamically.

Supporting Adaptable and Extensible Operating Systems

In many of the systems discussed above, applications that wish to customize the system must either add their own specialized components or ensure that appropriate specialized components have been installed in advance. Typically, the addition of a specialized component requires that the application download the appropriate code into the kernel. We call such approaches *low level* and *explicit* because applications explicitly specify the changes they need, and they do so by providing kernel code rather than a high level description of the behavior that they would like to see. In such systems the responsibility for tuning the operating system's performance has effectively been abdicated to the application.

While approaches based on low level and explicit customization allow precise tuning to application needs, they also have several problems. First, the fact that customization requirements are specified in the form of kernel code means that a high degree of kernel programming expertise is required at the application level. Second, an individual application may not have the global system view necessary to implement specialized components successfully in the presence of conflicting customizations from other applications. For this reason, the approach does not scale well to large systems with many applications. Third, supporting adaptable operating systems is difficult because it requires the application to respond dynamically to changes in the system, which may be caused by events external to the application. Again, the lack of a global system view by any particular application makes it difficult to provide such support. Finally, explicit customization does not support "dusty-deck" applications, or applications that are unwilling or unable to take on the responsibility for tuning operating system performance.

An alternative to explicit customization is *inferred* customization. Operating systems that support inferred customization generate and select appropriate specialized components dynamically and automatically using information that is available through the normal system call interface. Such systems provide some support for dusty deck applications; however the limited information used to drive customization means that many opportunities for optimization are missed. An early example of a system based on inferred customization was the Synthesis kernel [17, 21]. Synthesis was a precursor to Synthetix, which is discussed later.

In order to gain the benefits of both explicit and inferred customization, it is possible to combine the techniques in a single system. For example, a system based on inferred customization could infer customizations solely from the system call behavior of applications, or it could use additional hints passed to it from the application via a *meta-interface* [15, 25].

Meta-interfaces can take many different forms. They can support abstract specifications of an application's intended use of a system, or they can provide the means for applications to download code directly into the kernel. We call the former a high-level meta-interface and the latter a low-level meta-interface. Orthogonally, meta-interfaces may allow applications to *inform* the operating system of their intentions, or they may allow applications to *direct* the operating system's behavior [15].

Table 1 summarizes the various attributes of customizable operating systems. Table 2 summarizes the approaches taken by the systems that we have described. Table 2 also lists the Synthetix project. In the following sections we outline the Synthetix model for building fine-grain adaptable operating systems that support inferred customization, but also use a high-level hint-based meta-interface. The practical application of this model is described in two side bars that outline the implementation of a dynamically specialized `read` system call in HP-UX, and an adaptable Internet-based video player.

The Synthetix Specialization Model

The Synthetix project seeks to define a systematic approach to building adaptable operating systems. We begin by establishing a high-level specification of system properties that are exploitable by customization 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 become false at some future time.

Once invariants have been established, *specialized components* can be prepared to replace their generic counterparts in the system. A specialized component can be either 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 [8]. A specialized policy component provides the same interface as its generic counterpart, but changes the behavior of the component to provide improved performance to the application. An example of this approach would be a file system pre-fetching policy specialized for the access patterns of a particular application.

Quasi-invariants can become false, potentially making their corresponding specialized components 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 invalidates the quasi-invariant, then the guard *re-plugs* all the specialized components that depend on that quasi-invariant with less specialized components that do not depend on it. Because a specialized component that depends on quasi-invariants can be removed, possibly even before it is used, we refer to the use of such specialized components as *optimistic specialization.*

Specialized components can be installed whenever the appropriate set of invariants and quasi-invariants is discovered to be true. Discovering that an invariant is true requires

Generator of customization

Label	Description
select	The application selects among choices offered by the operating system.
replace	The application replaces a module within the operating system.
infer	The operating systems replaces its own modules. Decisions are transparent to the application.

Location of customization

Label	Description
kernel	The specialized module resides in the same address space as the operating system.
library	The specialized module resides in an application-level library. Application has access to module via procedure call and memory references.
server	The specialized module resides in an application-level server process. Application has access to module via messages.

Granularity of customization

Label	Description
fine	Allows specialization of procedures and small objects.
coarse	Restricts specialization to entire servers or libraries.

Protection Enforcement

Label	Description
native	Protection enforced by existing protection mechanisms such as virtual memory, IPC, capabilities, and system calls.
super-user	Only kernel programmer or super-user can place the customization into the kernel. Similar to policy used with UNIX third-party device drivers.
low-level	Protection maintained by low-level mechanisms such as sandboxing and transactions.
language	Protection maintained by type-safe languages in conjunction with a secure compiler, linker, and loader.

Table 1: Attributes of Customizable Operating Systems

System	Generator of Customization	Location of Customization	Granularity of Customization	Protection
Aegis	replace	library[1]	fine	low-level
Apertos	replace	kernel	fine	super-user
Cache Kernel	replace	server/library	fine	native
Choices	replace	kernel	fine	super-user
Chorus	replace	kernel[2]	coarse	super-user
Flex	replace	kernel[2]	coarse	super-user
Hydra	select	kernel	coarse	native
Lipto	replace	library	fine	native
SPIN	replace	kernel	fine	language
Spring	replace	server	fine	native
Synthetix	infer	kernel	fine	native

[1] Aegis also allows sandboxed code to be downloaded into the kernel.
[2] Servers can be run outside the kernel for debugging purposes.

Table 2: Customizable Operating Systems

the same set of checks as discovering that an invariant is false, and so the aforementioned guards can be used to trigger the use of specialized components, 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 component may be replaced with one that is *more* specialized than the current component. We call this approach *incremental specialization.* Sidebar 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. In contrast, a policy specialization is a customization of the *behavior* of an operating system component so as to improve the performance provided to an application. For instance, if some particular properties of an application's locality of reference are known, then the virtual memory system can be specialized to cater to that reference pattern. Policy specialization encompasses any form of adaptation of the function of a component.

The Synthetix project is examining a particular form of policy specialization called *software feedback* [18] in which policy is specialized according to a feedback mechanism. In a system containing producer and consumer processes, software feedback proposes that the consumer feed back properties of its input to the producer so as to balance and optimize the data flow. Sidebar 2 describes our distributed video/audio player, which uses software feedback to adapt dynamically to the changing bandwidth provided by the Internet. This example illustrates two concepts. First, 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.

Second, software feedback re-specializes the behavior of the system between 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 component once and for all, as in a microkernel, or once a specialization opportunity is discovered, as in our HP-UX experiment, software feedback continuously re-specializes the system. Nonetheless, software feedback can still be understood using the Synthetix model for specialization: the consumer describes the properties of its input data stream as quasi-invariants; when these quasi-invariants are violated a feedback message is sent to the producer to correct the data stream so that the quasi-invariants will *again* be true.

The techniques outlined so far enable the implementation of an adaptable operating system that preserves an existing interface: no explicit specifications of desired customizations are necessary, and thus "dusty deck" applications can experience performance improvements without any knowledge of customization. However, there are limits to the invariants that the operating system can infer from the behaviour of the application.

To extend the ability of the operating system to specialize itself, we propose to extend the operating system's interface with *microlanguages*. A microlanguage is a small, application-specific, mostly declarative specification of the invariants that the application would like the operating system to use. This approach allows the application to state its desired properties *without* any knowledge of the internal structure of the operating system: specified invariants that are not relevant to a particular operating system implementation can simply be ignored. Microlanguages are intended to be small and have simple syntax, but deep semantics.

In summary, Synthetix defines a model for supporting both inferred and high-level explicit customization in an adaptable operating system. Guards are used to manage conflicts among specialized components and support optimistic specializations. Invariants and microlanguages constitute a high-level meta-interface through which applications can specify the specialized behavior that they would like the operating system to exhibit. The two sidebars outline our experience using this model in a commercial operating system and in a distributed Internet-based application. These examples show that the Synthetix model is not limited to coarse-grain, infrequent specializations, but is suitable even when respecialization must take place at a finer grain than a system call. Finally, the customization techniques outlined here are orthogonal to operating system structure. One of our case studies has been performed in a monolithic kernel; however, we could easily have applied the same approach in a micro-kernel or an object oriented operating system.

Sidebar 1: Specializing HP-UX

The experiment presented in [20] sought to evaluate the effectiveness of mechanism specialization in a commercial operating system. Previous work [18, 21] had already shown that specializing operating system mechanisms 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 in HP-UX. Figure 1 shows the flow graph for the standard HP-UX implementation of `read`, and Figure 2 shows the specialized implementation of `read`. 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, specialized for the file being opened, can be created at `open` time. Hence, rather than checking these parameters, it hard-codes them directly.

The generic `read` mechanism also acquires several concurrency locks on kernel data structures to protect against interference 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. The quasi-invariant that the file is not shared characterizes this situation; when it holds, the acquisition of the concurrency locks can be omitted from the specialized `read` mechanism. This is an important saving, because lock acquisition can be expensive on shared memory multiprocessors [1].

Non-sharing of files is a *quasi*-invariant because at any time another process may open

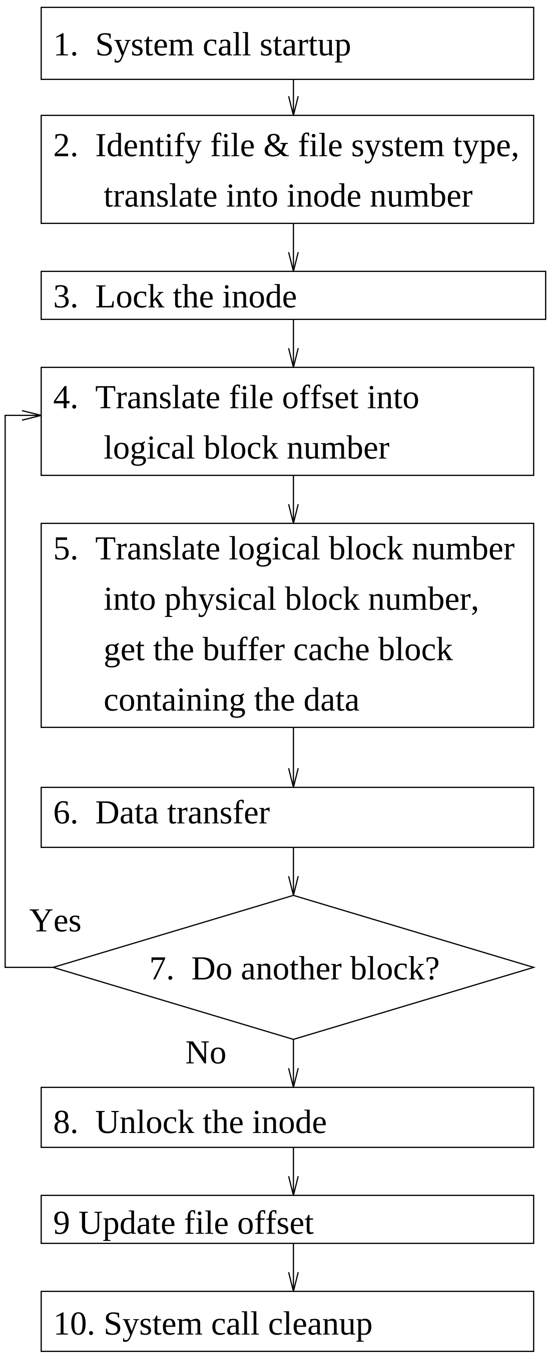
1. System call startup
2. Identify file & file system type, translate into inode number
3. Lock the inode
4. Translate file offset into logical block number
5. Translate logical block number into physical block number, get the buffer cache block containing the data
6. Data transfer
7. Do another block?
Yes
No
8. Unlock the inode
9 Update file offset
10. System call cleanup

Figure 1: HP-UX `read` Flow Graph

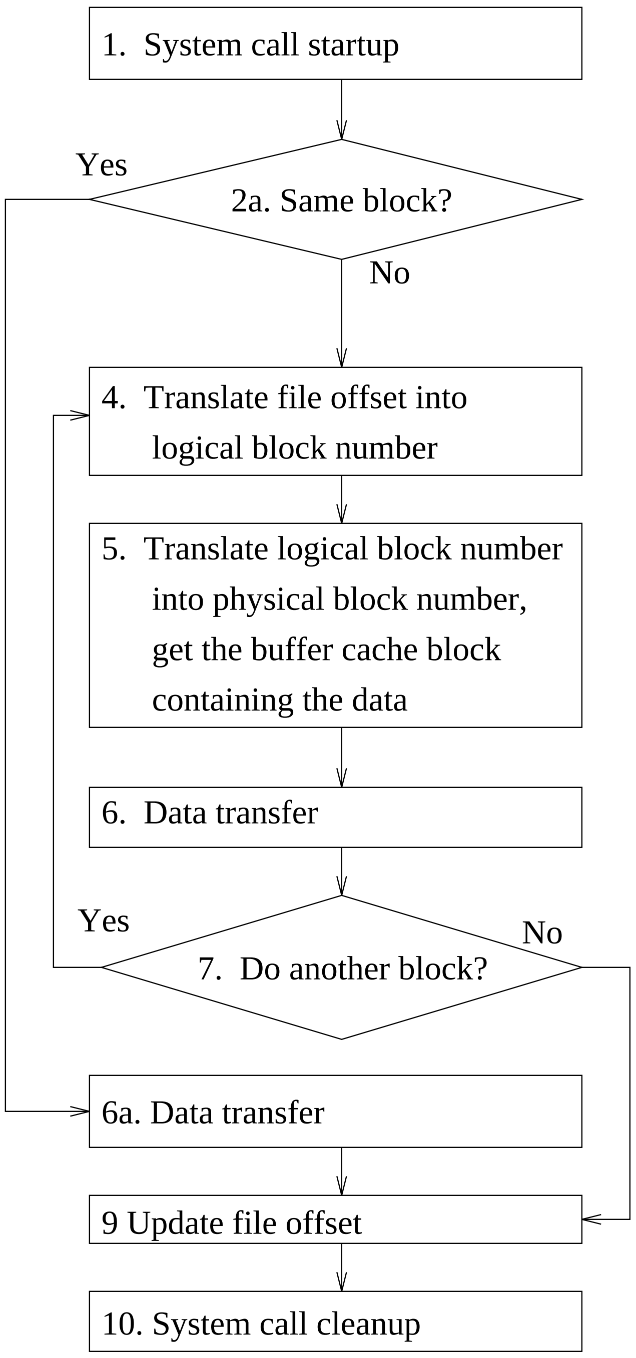
1. System call startup
2a. Same block?
Yes
No
4. Translate file offset into logical block number
5. Translate logical block number into physical block number, get the buffer cache block containing the data
6. Data transfer
7. Do another block?
Yes
No
6a. Data transfer
9 Update file offset
10. System call cleanup

Figure 2: Specialized `read` Flow Graph

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 a specialized `read` mechanism associated with it 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.

This approach to customization requires the system programmer to identify common cases, such as common access patterns to files, to represent them using invariants and quasi-invariants, and then to place the appropriate guards to support automatic replacement of specialized components. It also requires support for dynamic replacement of kernel components that may be executing [9]. The performance improvements that result from the approach depend on the ability to move interpretation code out of the operating system's commonly accessed "fast paths"; the necessary guard code is placed in other, less frequently accessed, code paths. Our experiments show that, in the case of `read`, this technique can reduce the software overhead of a system call by more than a factor of three, even in an optimized commercial operating system. Such a reduction in system call overhead not only improves application performance, it also enables a more flexible use of operating system calls.

Sidebar 2: Policy Specialization Through Software Feedback

Two of the hottest topics in 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 reservation [6].

We use *software feedback* [18], reminiscent of hardware feedback, to adapt multi-media presentations to the changing conditions of the Internet. Our video player has a distributed client-server architecture as shown in Figure 3. The client measures various properties of the video stream it is 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 the present state is within tolerance, a quasi-invariant is true and no feedback is required. If the quasi-invariant

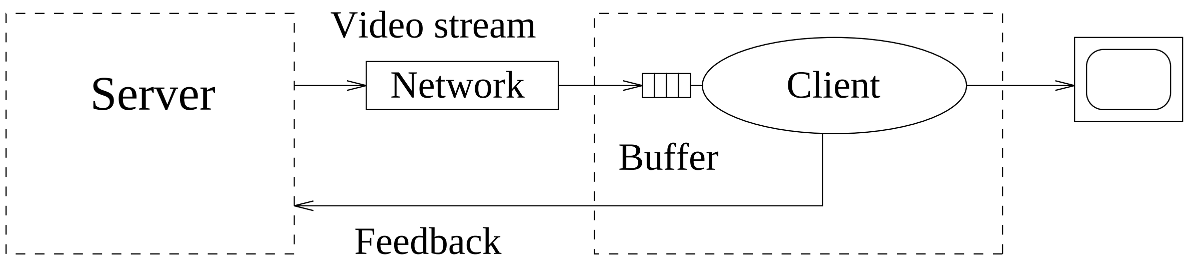

Figure 3: Architecture of the player

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

For instance, it is desirable that the server send only 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 its frame transmission rate so that the client and server's frame rates will again be within ϵ of one another.

A more involved example of policy specialization is the use of a software feedback system to adapt simultaneously to changes in network latency and network jitter. Network jitter is short-term variation in the inter-arrival time of frames: the client must buffer a sufficient number of frames to mask jitter, so as to present the frames to the user in a smooth, regular fashion. Network latency is the delay between the server sending a frame and the client receiving the frame: network delay is an important factor in determining how far ahead the server should be working from the client's current play position so as to keep the client's buffer at an optimum fill level. Note that latency typically changes more slowly than jitter.

Both network jitter and changes in network latency are manifested as changes in the arrival time of frames at the client. However, the policy required to adapt to each is different: rising jitter requires allocating additional buffer space in the client, while changes in network latency require changes in the work-ahead position of the server. The feedback system determines which of these two policies to apply by using *filters* on the feedback data. Both network jitter and changes in network latency are measured using an aging average of frame arrival time, but different aging factors are used to identify the two different phenomena.

Selecting policies in this way can be viewed as specialization of a specialization: the particular policy specialization to be applied is selected adaptively based on current circumstances.

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.

References

[1] 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.

[2] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[3] D.L. Black, D.B. Golub, D.P. Julin, R.F. Rashid, R.P. Draves, R.W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel Operating System Architecture and Mach. In *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, Seattle, WA, April 1992.

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

[5] 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.

[6] Shanwei Cen, Calton Pu, Richard Staehli, Crispin Cowan, and Jonathan Walpole. A Distributed Real-Time MPEG Video Audio Player. In *Proceedings of the 1995 Inter-*

national Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'95), New Hampshire, April 1995.

[7] 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.

[8] Charles Consel and Francois Noël. A general approach to run-time specialization and its application to C. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburgh Beach, FL, January 1996. To appear.

[9] Crispin Cowan, Tito Autrey, Calton Pu, and Jonathan Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. Report CSE-95-019, Dept. of Computer Science and Engineering, Oregon Graduate Institute, Portland, OR, October 1995. Submitted for review.

[10] Peter Druschel. Efficient Support for Incremenal Customization of OS Services. In *Proceedings of Third International Workshop on Object Orientation in Operating Systems (IWOOOS-III)*, pages 186–190, Asheville, NC, December 1993.

[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] Ajei Gopal, Nayeem Islam, Beng-Hong Lim, and Bodhi Mukherjee. Structuring Operating Systems using Adaptive Objects for Improving Performance. In *Proceedings of the Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS '95)*, pages 130–133, Lund, Sweden, August 1995.

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

[16] R Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/Mechanism Separation in Hydra. In *Proceedings of the 5th Symposium on Operating System Principles (SOSP'75)*, pages 132–140, November 1975.

[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] 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.

[20] 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.

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

[22] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the Chorus Distributed Operating System. In *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–69, Seattle, WA, April 1992.

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

[24] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles (SOSP'93)*, pages 203–216, Asheville, NC, December 1993.

[25] 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.