A Survey of Customizability in Operating Systems Research

G. DENYS

CoWare

AND

F. PIESSENS AND F. MATTHIJS

Katholieke Universiteit Leuven

An important goal of an operating system is to make computing and communication resources available in a fair and efficient way to the applications that will run on top of it. To achieve this result, the operating system implements a number of policies for allocating resources to, and sharing resources among applications, and it implements safety mechanisms to guard against misbehaving applications. However, for most of these allocation and sharing tasks, no single optimal policy exists. Different applications may prefer different operating system policies to achieve their goals in the best possible way. A *customizable* or *adaptable* operating system is an operating system that allows for flexible modification of important system policies. Over the past decade, a wide range of approaches for achieving customizability has been explored in the operating systems research community. In this survey, an overview of these approaches, structured around a taxonomy, is presented.

Categories and Subject Descriptors: C.4 **Performance of Systems**: *design studies*; D.4.6 **Operating Systems**: Security and Protection; D.4.7 **Operating Systems**: Organization and Design

General Terms: Design, Performance, Security

Additional Key Words and Phrases: Customizability, microkernels, operating systems, software protection mechanisms

1. INTRODUCTION

The main purpose of a *customizable* or *adaptable* operating system is to provide flexible mechanisms and policies to its clients, so that they can achieve their goals in a better way. The clients of an operating system are the applications, but also

---

F. Piessens is a Postdoctoral Fellow of the Belgian National Fund for Scientific Research (N.F.W.O.)

Author's addresses: G. Denys, CoWare, 2121 North First Street, San Jose, CA 95131; email: geert@coware.com; F. Piessens and F. Matthijs, Department of Computer Science, K. U. Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium; email: {frank,frankm}@cs.kuleuven.ac.be.

application developers and administrators. Depending on the context, different goals will be driving the need for customizability. Often, performance is the driver for customizability in general-purpose operating systems. For embedded systems, the power consumption or software footprint may be the main driver. When an operating system supports multiple clients, conflicting needs are likely to arise. An important goal of making the system customizable is to adapt gracefully to these conflicting needs.

Customizability (or adaptability) has always been a concern for operating system designers. Even in the very early operating systems, some form of customization was possible, often during the system generation phase. However, in recent years, customizability has become a major topic in operating system research, and a number of different techniques have been explored to achieve build-time, installation-time and even run-time customizability. While working on a project that required a run-time customizable protocol stack, and thus studying the literature on customizability, the authors felt that the variety of approaches made the literature quite inaccessible, and hindered a quick dissemination of results. This article provides a survey and a taxonomy of the various approaches to customizability that have been proposed and experimented with over the past decade, and we hope this paper will be a good entry point into the literature on customizability.

The article is structured as follows: we start in Section 2 with a discussion of some typical design issues in designing customizable operating systems. This discussion forms the motivation for our taxonomy of customizable operating systems, presented in Section 3. The structure of the rest of this article mirrors the presented taxonomy: various research projects are discussed according to their classification in the taxonomy. Of course, the list of projects we discuss is not exhaustive, but it includes at least one representative example of each interesting class in the taxonomy.

2. DESIGN ISSUES IN CUSTOMIZABLE OPERATING SYSTEMS

Introducing customizability in operating systems invariably leads to confrontations with other operating system characteristics. In this section, we describe important considerations and issues that designers have to take into account.

2.1. Performance

When designing a customizable operating system, or any customizable system for that matter, an important trade-off that designers have to make is between customizability and performance.

Often customizability is introduced to improve system or application performance, as noted in the introduction. However, the customizability mechanisms themselves may introduce performance overheads, which have to be weighed against the benefits the designers are pursuing with the customizable system.

In early, noncustomizable, operating systems this trade-off has led to mostly monolithic solutions where functionality and policies were fixed. The designers determined what functionality to offer and what policies to use, so that "most" applications could be supported with reasonable efficiency. Applications that matched the operating system's design decisions would run well, applications with nonstandard resource usage patterns, such as databases, were less well supported, resulting in suboptimal performance.

2.2. Spread

The customizable system may support customization throughout the operating system so that every important policy can be adapted (*widespread* customization). Alternatively, the system may only support focused customization, in which only a certain set of policies can be adapted, for instance the scheduling policy.

2.3. Granularity

For each customizable policy, the operating system can set the granularity of customization.

With *fine-grained* customization, all aspects of the policy can be adapted in a seemingly unrestricted way. In a *coarse-grained* approach, only certain aspects of the policy can be changed, in a restricted way; for instance, a choice between predefined scheduling algorithms.

2.4. Integrity

Making an operating system customizable should not introduce other defects in the system. Customizability can compromise the integrity of an operating system, especially if applications are allowed to introduce new functionality or policies in the system. Therefore, an important requirement for a good design is integrity protection.

We can define the integrity of an operating system as the set of invariants which are established and maintained by the operating system in order to ensure its correct internal operation and to fulfill its contract toward its clients. When discussing integrity, it is essential to realize that this may cover a lot more than just memory protection—ensuring that no application can access data or execute code in an unauthorized way. There are various other ways in which applications can misbehave: resource hoarding (infinite looping, excessive memory allocation, etc.), denial of service or unauthorized use of kernel interfaces. The integrity can be enforced through verification or protection (see Section 6 for a detailed discussion).

The level and complexity of measures that need to be enforced by an operating system is related to both the *granularity* and the *spread* of customization that is supported by the operating system. The finer-grained and the more widespread the customizability is in an operating system, the more complex the safety mechanisms will need to be.

2.5. Purpose

The operating system is either intended for a well-known set of applications (a *specific* or *special-purpose* operating system), or it makes no assumptions about the expected functionality and variety of the applications both present and future (a *general-purpose* operating system).

A general-purpose operating system will increase the need for fine-grained and widespread customization, whereas a special-purpose system may succeed by having coarser-grained and less widespread customization.

2.6. Paradigm

Another important design choice is the paradigm supported by the operating system. The underlying paradigm profoundly affects the way the operating system is structured and the way applications must be built on top.

The main choice is whether to support a *traditional*, POSIX-like, paradigm or a non-traditional paradigm. The traditional paradigm is supported by the vast majority of the installed base of production operating systems, such as Linux, Solaris and Windows. Similarly, the majority of customizable operating system projects discussed in this survey is structured according to this traditional paradigm, or at least supports this paradigm.

A nontraditional paradigm, such as the reflective operating systems discussed in Section 5.3, or the data-path oriented systems discussed in Section 4.1, may better support the needs of a customizable system. On the other hand, it is clear that the paradigm used by an operating system has a strong impact on how applications are to be developed. Special tools may be required to build applications, which may seriously complicate matters for application developers.

3. TAXONOMY

The design issues mentioned in the previous section have forced operating system designers to adopt a large number of different solutions. In order to classify the various approaches, we introduce a taxonomy of customizable operating systems.

The two main criteria in our taxonomy are:

(1) The initiator of adaptation.
   —A *human* administrator or operating system designer.
   —An *application*.
   —The *operating system* itself. We refer to this as *automatic adaptation*.
(2) The time of adaptation.
   —During design, build or installation time of the operating system. We refer to this as *static adaptation*.
   —During boot or run time of the operating system. We refer to this as *dynamic adaptation*.

These criteria are well suited to classify the current research projects on customizable operating systems. Because these two criteria are orthogonal, six categories are identified by this taxonomy. However, only five of the six categories make sense: application-initiated adaptation can not happen statically. We discuss each category in more detail. We use the first criterion as the primary structure for this discussion and the structure of the rest of this survey as well.

3.1. Human-Initiated Adaptation

When using *static adaptation*, the systems in this category are referred to as operating system *frameworks* which the designer can customize to create an operating system.

If the application domain is known in advance, one can make a *specific* operating system, which supports the envisioned functionality but nothing more. This results in a high-performance, application-specific operating system. The specific operating system is built from a framework, which can be customized by the operating system designer at design time.

To produce an optimal specific operating system, it must be possible to customize the framework thoroughly and with fine granularity. The system designer selects from a set of predefined parts those parts that best match the target application(s). He may introduce new parts or customized versions of predefined parts. Once the operating system is built from these parts, its functionality and policies are fixed.

This form of adaptation can be applied to embedded operating systems for devices with a specific functionality. The operating system requirements of a cellular phone differ from those of a network router, but their respective operating systems may be derived from the same operating system framework.

For specific operating systems, the expected functionality is well understood and more or less stable in time. Hence, designing a specific operating system to be very flexible (run-time customizable) would be of little value for the supported applications. It is preferable to design the operating system to optimally fit the need of the envisioned functionality. Flexibility in this respect would only result in a performance penalty.

Operating system frameworks may be sufficiently general that they can even be used to derive general-purpose operating systems.

Dynamic adaptation is possible in most production operating systems: a system administrator can initiate adaptations at boot or run time. At boot time, the kernel can be passed parameters and configuration settings to adapts its behavior; at run time the administrator can install and deploy new kernel modules, or can observe performance parameters and consequently *tune* the operating system.

Human-initiated adaptation is further discussed in Section 4.

3.2. Application-Initiated Adaptation

Applications are the main clients of an operating system. Since they only exist during the operational phase of the operating system, application-initiated adaptation can only be performed dynamically and not statically. Hence, the second criterion is not relevant here.

Application-initiated adaptation is particularly useful for more general-purpose operating systems. Whereas traditional operating systems had great difficulty incorporating new needs, like those of multimedia applications, customizable

operating systems should do better. The application often knows its own needs and usage patterns. With this knowledge, it can instruct the operating system to perform customizations on its behalf.

Mechanisms for accomodating application-specific customizations at run-time introduce a performance overhead. However, the goal of a dynamically customizable operating system is often to achieve better overall performance by allowing customizations whose benefits outweigh this overhead.

Devices such as personal digital assistants (PDA) and set-top boxes, whose applications change, but not frequently nor abruptly, can benefit from an operating system supporting this dynamic adaptation.

The category of application-initiated dynamic adaptation is quite broad, and therefore, we divide it in two subcategories. In Section 5, we discuss operating systems that allow applications to customize the operating system services by introducing code at the user or nonprivileged level. These operating systems are typically structured around a microkernel. Operating systems that allow the kernel itself to be extended or adapted by applications are discussed in Section 6. These *extensible kernels* use innovative mechanisms to ensure the integrity of the kernel.

3.3. Automatic Adaptation

Automatic adaptation is adaptation initiated by the operating system itself. Here, no intervention is needed to customize the operating system.

Portability may be considered to be a *static* form of automatic adaptation. By detecting on which platform the operating system is being built, it can configure itself by using, for instance, the C preprocessor.

However, the most interesting category is when the operating system automatically and *dynamically* adapts itself to the running applications. To this end, the operating system must be able to independently observe and analyze the applications and automatically change its behavior to support the applications in better and more efficient ways. Ideally, such operating systems always offer an optimal execution environment for the applications. This strategy could underly general-purpose systems where a variety of applications are running and interacting in unforeseen ways.

Production operating systems typically support a limited form of dynamic automatic adaptation in specific and well-understood subsystems, for example in the file system, that can monitor the behavior of user applications to optimize its performance.

Automatic dynamic general-purpose systems might be considered the ultimate goal of customizable operating systems. However, it is clear that the problems involved in building such a system are very difficult. We are not aware of any truly automatic systems. Nevertheless, useful lessons can be learned from the few projects discussed in Section 7 which apply automatic adaptation in a limited way.

4. HUMAN-INITIATED ADAPTATION

4.1. Static Designer-Initiated Adaptation

For operating systems in this category, all system functions and policies are determined at design time and all system services are typically incorporated in the kernel. The main use of a build-time adaptable operating system is as a *specific* operating system. A specific operating system is designed for a specific function, or even for a single application, whereas a traditional operating system is designed to support a much broader set of applications.

The initiator of adaptation is the system designer. In a specific operating system, the functionality and the requirements are well known and understood at design time. This enables the creation of a lean and high-performance system, in which only the strictly required functionality is present, and all services have been optimized statically for the given application.

The obvious drawback is that new functionality or other applications cannot be

supported by a specific operating system. For devices with a single, well-known, function (for instance, a router or a video camera) this is not an issue. It does mean however that a new operating system has to be designed and implemented for each application, and that a single device or computer can support only one (or a limited number of) application(s) [Kiczales et al. 1993]. To avoid designing a new operating system from scratch each time, specific operating systems should be designed using a general-purpose framework. This framework must be able to produce optimal operating systems for a wide variety of applications. We will now discuss four projects which provide a framework for system development.

4.1.1. The Flux OSKit. The OSKit [Ford et al. 1997] is a straightforward approach to produce specific operating systems. It consists of a framework and a module library with clean and documented interfaces. Its focus lies in language and kernel research. It provides a way for the designer to focus on the issue of interest, while the OSKit fills in the voids via an extensive library which contains common system services taken and adapted from other other operating systems (Linux, FreeBSD, NetBSD).

The library contains modules for simple bootstrapping, a minimal POSIX environment, memory management, debugging support and a number of high-level components, such as protocol stacks and file systems. Each generic component is easily replaced by the designer.

The OSKit framework is a collection of libraries which are linked in to create an operating system kernel. The different components are modular and separable, thanks to the use of *glue* layers which form an indirection between a component and the services it uses.

In general, the components of the library are rather coarse-grained. The components are basically *subsystems*, such as the file system, the network protocol stack or the collection of device drivers. As a result, the granularity of adaptation of the OSKit framework is rather coarse.

4.1.2. Scout. The Scout approach [Montz et al. 1995; Mosberger and Peterson 1996] is directed toward communication applications. It is meant for networked devices.

Scout is a communication-oriented operating system framework. An operating system designed with Scout consists of a number of modules that implement well-defined and independent functions. The *module graph* captures the connections between the individual modules. Connected modules must provide a common interface. The interfaces are typed and enforced by Scout. The module graph is fixed at design time.

A *path* conceptualizes a data flow from an I/O source, through the system, to an I/O sink. It can be viewed as a fixed sequence of routing decisions through the system's modules. A path consists of two components:

—The sequence of modules which determines the semantics (reliability, security and timing) of the path.

—The required resources necessary to process and route the data.

Paths through the module graph are created and destroyed in a dynamic fashion. A path is well suited for resource allocation and performance optimization, because it provides nonlocal context, not available within a single module. This global information is used for optimal resource allocation and code specialization along a path.

Generally, a path is automatically created from the module graph. Using module-specific knowledge, a maximum length path is created. Through global transformation rules, this path is transformed (optimized).

In Spatscheck and Peterson [1997], a safety architecture is defined that allows the designer to specify the safety policy for a Scout operating system. This safety architecture also adds multiple protection

domains to Scout, which otherwise runs in a single address space.

4.1.3. Choices. Choices [Campbell et al. 1987, 1993] is an object-oriented operating system that directly applies the framework idea. Its various subsystems, such as memory management, process management, file storage, exceptions, etc. are directly built from object-oriented frameworks. System resources, mechanisms, and policies are represented as instances of classes that belong to a class hierarchy, where customization is done through the use of inheritance. Thus, a specific operating system is created by specializing classes in the various hierarchies, and by instantiating a specific set of objects which together form the operating system. The application interface is a collection of kernel objects exported through the application/kernel protection layer.

Choices exhibits a minor form of dynamic adaptation. The adaptation is not directly controlled by the applications, however. Only when an application needs a specific service, a class can be dynamically loaded to implement the service. Every possible behavior is still statically determined.

4.1.4. Pebble. Pebble [Magoutis et al. 2000] is described as a *component-based* operating system by its authors. Pebble is both a portal-based microkernel (as we will describe in Section 5.1), and a general-purpose operating system framework. It provides a set of operating system abstractions, implemented by trusted user-level operating system components. Those components can be augmented, replaced or layered, allowing alternate abstractions to coexist or override the default set. Pebble allows the construction of modular operating systems for appliances out of reusable components. In contrast to the other systems discussed in this section, the system services are not integrated in the kernel. System services are offered by trusted server components, which run in user-level protection domains.

The approaches discussed in this section can lead to excellent results for devices or computers whose functionality is well known in advance.

4.2. Dynamic Administrator-Initiated Adaptation

This class of operating systems supports adaptation at boot or run time by an administrator, or possibly even a user of the system. Adaptation can be achieved in at least two ways.

First, the *loadable kernel module* technique allows a trusted person (typically the administrator or *root* of the system) to load modules of new code into the kernel. This new code can then interface against a documented set of APIs. In that way, the functionality of the operating system can be changed or extended. The advantage of this technique is its simplicity: it is very similar to dynamic class loading for ordinary applications. The main disadvantage is that adding arbitrary code to the kernel can break the safety mechanisms of the system, and can cause the system to crash. A stable system can become unstable after loading a malicious (or just buggy) kernel module.

Second, adaptation can be done through *tuning* of the system. Typically, the operating system parameterizes its policies, and these parameters can be changed by an administrator or user. To allow an administrator to make an informed decision, the operating system also makes available *performance counters*, that is, counters that indicate how well a particular policy is performing. The advantage of this technique is that it does not compromise the protection mechanisms. The disadvantage is that the spread and granularity of the customization are more limited.

The techniques discussed in this section are not extremely interesting from a research point of view, but they are widely deployed in all production operating systems (e.g., Linux, Solaris, Windows NT). Windows 2000, for example, has hundreds of performance counters and related system variables [Solomon and Russinovich 2000]. The Linux kernel makes extensive use of loadable modules.

It is interesting to note that the context and the motives for extensibility or adaptability in production operating systems are very often different than in research operating systems. In the research community, performance has been the most important driver, often in an *untrusted* context. The initiator of the adaptation cannot be trusted and hence the operating system integrity must be protected. In production operating systems, extensibility has been mainly used to add various forms of functionality, in a *trusted* context, where the initiator of adaptation is trusted (f.i. the system administrator). The loadable kernel module technique is used to extend the kernel with new device drivers, with support for new filesystems, with new authentication techniques (e.g., the Pluggable Authentication Module approach [Samar and Lai 1996] is supported by all major operating systems) and various other kinds of functionality. Performance enhancing modules as well as integrity protection mechanisms are very rare in practice. As a consequence, many of the research results on adaptability and extensibility have not found their way into production operating systems.

As long as production operating systems can assume a trusted context for extensibility, their extensibility mechanisms will remain less sophisticated than those addressed in the research community. It is our belief that more sophisticated mechanisms will be needed in the future. The fact that performance has not been driving extensibility in production operating systems may be an indication that the research projects are solving the "wrong" problem. However, we believe that most of the extensibility approaches discussed in this survey can also be used to achieve other goals than enhancing the performance.

5. APPLICATION-INITIATED ADAPTATION AT USER LEVEL

This section and the next section describe application-initiated adaption. This adaptation happens at run time, during the regular operation of the system. In this section, we focus on adaptation performed at the user (nonprivileged) level.

5.1. Microkernel Systems

Microkernel-based systems are characterized by a *minimal* kernel. The system services are at user level and, possibly, provided by the applications. The services are called from the kernel via *upcalls*.

It is important to note that the microkernel approach has a *principal adaptability*. Since a *pure* microkernel aims to be minimal, it doesn't impose any service or policy. Hence, each application can provide its own services tuned for its specific needs.

The microkernel approach is the oldest discussed in this survey. It is clear that the early generations of microkernels did not live up to expectations. Due to inferior performance and coarse-grained services, the principal adaptability was not realised in practise. Substantial operating system libraries, like a UNIX file system, were reintegrated in the kernel or given privileges. This enhanced the performance but didn't help the customizability at all.

The new generation of microkernels is a better match for the goals of customizability. Inherently, a microkernel has a large amount of domain crossings—between user and kernel level and between address spaces. The cost of these domain crossings influences the overall performance and flexibility of the system. Hence, the mechanism to switch back and forth between user and kernel level and the interprocess communication (IPC) mechanism, which is used frequently to call the functions of a user-level system service, need to have efficient implementations. Another critical performance aspect is the memory *locality* of the microkernel. Naive microkernel implementations may take page faults or TLB flushes on domain crossings, resulting in terrible performance, even with efficient mechanisms [Chen and Bershad 1993; Liedtke 1993].

The chosen set of abstractions integrated in the kernel profoundly affects both performance and flexibility. Therefore, the new generation tries to minimize

the number of abstractions fixed in the kernel. The fewer abstractions, the more flexible the operating system remains for the applications. The designer tries to find the minimal set of abstractions that are necessary for proper operation. This set is then integrated in the kernel and the applications are free to implement the other services. This is embodied by L4 [Liedtke 1995, 1996]; the abstractions imposed by the kernel are address spaces, threads, IPC, and unique identifiers. With these abstractions, L4 supports the recursive construction of address spaces—the initial address space represents the entire memory and I/O ports and is owned by the initial subsystem or application; it supports activity within and communication between address spaces. The careful design and implementation of L4 and its predecessors have established new standards for interprocess communication times [Liedtke 1993].

When pursuing the microkernel philosophy to its logical extreme, the result is an operating system in which *all* system services are in user space, as a library, and the kernel is merely an abstraction of the underlying hardware. This extreme is pursued in the Exokernel [Engler et al. 1995; Engler and Kaashoek 1995]. The Exokernel tries to eliminate all abstractions from the kernel and lowers the kernel interface to just above the raw hardware. The only function of the Exokernel is to allocate, revoke and multiplex the physical resources (memory pages, processor time slices, disk blocks, etc.) in a secure manner. Processes are given direct access to critical data structures; this comes at the cost of exposing a rather complex interface to the application. The design of generally usable interfaces in the Exokernel remains a challenge [Shapiro et al. 1999].

5.1.1. Portal-Based Systems. Portal-based systems are microkernels and share the philosophy of running as little code in privileged mode as possible. A portal-based system provides *protection domains* in which user code can operate in a safe manner. A protection domain consists of a set of pages, and a set of portals. A protection domain may share both pages and portals with other protection domains. *Portals* are used for communication between protection domains.

Kea [Veitch and Hutchinson 1996] is a portal-based microkernel which provides the low-level concepts needed for constructing high-level services. These low-level concepts are *domains* (virtual address spaces), *inter-domain calls* (IPC) and *portals*. A portal is associated with a specific service; it is an entry point for an interdomain call. An application that owns a portal identifier can access the service portal.

A service must register its interface to the kernel. The kernel then controls access to this interface. The interface defines the allowed interactions between the application and the service's implementation. In contrast to a pure microkernel, Kea doesn't allow complete freedom for the design and implementation of services. However, this allows Kea to introduce *dynamic reconfiguration*.

When a service is called, through its portal, the kernel can decide which implementation is chosen. For instance, when an application uses a file service, the administrator can decide to interpose a compression service, which first compresses the data before passing it to the file service. The compression service must support the same file service interface, however.

A more complex form of reconfiguration happens in an indirect manner. An application can associate a new portal with the identifier for a specific service. For instance, when a virtual memory manager uses a portal identifier for its page eviction service, some applications may choose their own page eviction service and have the virtual memory manager use it for pages belonging to the application.

The implementation of services will determine the granularity of adaptation in Kea. If the virtual memory manager fixes the page eviction policy (instead of using a portal for it, as in the example), nobody can change it, except by replacing the entire service.

In SPACE [Probert et al. 1991; Probert and Bruno 1996], the only abstractions present in the kernel are a generalization of the exception or interrupt handling mechanism. This generalized exception handling mechanism could be implemented in hardware; the result would be a truly "kernel-less" operating system. Pebble [Gabber et al. 1999] shares with SPACE the idea of cross-domain communication as a generalization of interrupt handling. As Kea, it provides protection domains, inter-domain calls via portals and portal reconfiguration. Portals are implemented as generalized interrupt handlers. Pebble's kernel includes only code to transfer threads from one protection domain to another and a small number of support functions that require kernel mode. As with the Exokernel, Pebble moves the implementation of resource abstractions to user level, but unlike the Exokernel, it also provides a set of higher-level abstractions, implemented by user-level operating system components, as discussed in Section 4.1. Each user-level component runs in its own protection domain, isolated by means of hardware memory protection.

5.1.2. Capability Systems. Two systems, Fluke and EROS, are capability systems structured around a microkernel.

Fluke's architecture combines a microkernel with virtual machines. The Fluke kernel [Ford et al. 1996] supports the construction of *recursive virtual machines*. The kernel provides basic services and an interface that describes the high-level services. The virtual machines use this interface and re-export it to the next layer. Each layer completely simulates the environment for the layer above: the interface between each layer is always the same. Hence, services can be composed by stacking virtual machines. Each layer only affects the performance of the subset of the operating system interface it implements.

Because of this stacking or layering, Fluke supports a "vertical" decomposition of services, whereas a microkernel supports a "horizontal" decomposition, by moving traditional kernel functionality into user-level servers, side-by-side. Fluke's *nested process architecture* is the common interface between each layer in the system. This interface has three components:

(1) the basic instruction set, which is implemented by the processor, allows processes to directly execute a safe subset of the machine instructions, without emulation.
(2) the low-level system services, which are implemented by the kernel.
(3) the *common protocols*, which is implemented in each layer, wholly or partially (for non-implemented functionality, the functionality of the layer below is reused). The kernel also implements this interface and offers this environment to the first process (virtual machine).

Maintaining the consistency of the interfaces between the virtual machines limits the addition of new methods or parameters.

Pebble's architecture is close in spirit to the nested process architecture of Fluke. The Fluke model requires that system functionality be replaced in groups; a memory management "nester" must implement all of the funtions in the virtual memory interface specification. Pebble permits finer-grained extensibility through portal replacement.

Fluke is a capability system, similar to EROS [Shapiro et al. 1999]. Both systems also provide persistence. A *capability* is an unforgeable pair made up of an object identifier and a set of authorized operations (an interface) on that object. UNIX file descriptors are an example of capabilities. In a capability system, each process holds capabilities, and can perform only those operations authorized by its capabilities. In Fluke, all references between low-level objects are represented as kernel-mediated capabilities.

The EROS architecture consists of a kernel that implements a small number of primitive capability types. It further includes a collection of system services that are implemented by user-level applications and provide higher-level

abstractions such as files and memory objects. The kernel presents a fairly direct abstraction of the underlying hardware, as in the Exokernel; storage allocation, scheduling and fault handling policies are exported from the kernel to allow application-customized resource management.

The set of capabilities accessible to a subsystem makes up a *protection domain*. Applications as well as the operating system reside in their own protection domain. EROS has applied the ideas of L4 for transfer of control between protection domain boundaries.

EROS provides transparent persistence using a single level storage model. The definitive representation for all operating system objects is the one that resides on the disk. The objects are cached in software at different levels by the kernel, to enhance performance.

Each resource access is ultimately performed by capability invocation. If authorized by the capability, the invocation causes the named object to perform some operation specified by the invoker. The majority of capability invocations are IPC operations.

5.1.3. Discussion. While the low-level kernel concepts are fixed in a microkernel, each application can build its own high-level concepts using the kernel concepts. Although an application can always ignore the operating system services and implement them in its own way, in a monolithic system it will not be able to *avoid* the services and their overhead, whereas in a microkernel it can completely avoid them.

The actual flexibility of a microkernel is affected by the low-level concepts fixed in the kernel. A drawback of the minimal microkernel approach is that applications are faced with a very low-level kernel interface. Even the most simple application will have a substantial amount of work implementing the necessary high-level concepts. It must be assumed that on most systems, simple applications will use a library which provides common high-level services. In that case, it is the design of the library that determines the actual customizability of the operating system, rather than the design of the kernel. The design of the library is unrestricted; the microkernel does not impose any paradigm at all for the construction of services on a higher level.

Inherently, a microkernel performs a large amount of domain crossings. The major improvement in the new generation shows that the biggest cost of these domain crossings is in software, and that it can be overcome by solid kernel design. The minimum cost possible is attained when all software steps are eliminated. It is still an open question whether at that point microkernels will perform better and be more flexible than other systems, for example, those based on pure software protection. The minimal concepts exhibited by a microkernel still have to prove whether they are powerful enough to support all kinds of functionality.

The next two sections discuss caching and reflective operating systems. Unlike "pure" microkernels, their kernel is not minimal, but provides support for the reflection or cache paradigm. This paradigm defines the way in which services are to be built in user space. Consequently, the construction of system services is supported by the kernel. The downside is that applications must be specifically designed (or retrofitted) to suit the paradigm of the operating system.

5.2. Caching Operating Systems

The Cache Kernel is the kernel of the V++ operating system [Cheriton and Duda 1994]. In this architecture, applications run on top of *application kernels*, either in the same or in a different address space. The application kernels implement the operating system services. They run in user level and manage the loading and unloading of operating system objects (threads, address spaces and other application kernels) according to their own policies. The Cache Kernel itself functions as a cache for these objects. The objects present in the kernel are those that are in active use.

A small number of objects can be locked in the kernel. This mechanism is used to ensure that schedulers and exception handlers do not generate exceptions (e.g., page errors) themselves. The kernel interface has operations for loading and unloading of system objects and it has signals to indicate whether a certain object must be loaded or unloaded.

The Cache Kernel uses a caching approach in some ways similar to EROS. Where EROS writes operating system objects back to protected structures, the Cache Kernel writes this state back to untrusted application kernels.

5.3. Reflective Operating Systems

Reflective operating systems introduce an explicit paradigm which enables applications to dynamically customize their execution environment. An application must be explicitly structured as a set of objects; the operating system services are supplied by a set of *meta-objects* which support the application objects.

MetaOS [Horie et al. 1997, 1998] is an object-oriented reflective operating system, using the Java [Arnold and Gosling 1996] language. It improves on the concepts of Apertos [Yokote 1992, 1993]. Its architecture consists of three levels. The application objects reside at the *base level*. The level below consists of meta-objects, dynamically grouped into *meta-spaces*. Each metaspace supports a number of applications with similar requirements. This level is called the *meta-level*. At the bottom is the meta-meta level, which comprises a single meta-space, the *master meta-space*. This meta-space allocates the resources to the meta-spaces according to a dynamically replaceable policy.

MetaOS uses an *open implementation* [Kiczales et al. 1997; Maeda et al. 1997] to support the definition and construction of objects, meta-objects and their interfaces. Via those interfaces, meta-spaces can be adapted and extended in a dynamic and secure manner.

When an application starts running, it chooses the best suited meta-space available. It can try to initiate a number of fine-grained changes in the meta-space. If this would fail or prove insufficient, the application can clone the meta-space and migrate to it. It then has full control over the cloned meta-space, and thus over its own execution environment.

6. APPLICATION-INITIATED ADAPTATION AT KERNEL LEVEL

In this section, we discuss operating systems which allow application-initiated changes by importing application code into the kernel. Hence, we call these systems *extensible kernels*.

Extensible kernels are able to accept and execute user code in privileged mode, in a secure manner. Consequently, the behavior of the kernel itself can be dynamically changed. This practice eliminates crossings between user and kernel mode and between address spaces.

An *extension*, code introduced in the kernel by an application, cannot be trusted. The kernel must be protected from incorrect or bad behavior from the various extensions present. The employed safety mechanism is crucial in the design and organization of the kernel. We will treat this mechanism as the primary focus in the following discussion of extensible kernels. It's worth to mention that these mechanisms are also relevant in other, increasingly important domains, such as *mobile code* (Java Applets, ActiveX) and *active networks*, where "active" packets contain code to be incorporated into a network router or other active component of a network to modify its behavior in an application-specific manner.

The design of the kernel determines where and how extensions can be introduced. As mentioned in Section 2.4, a design which allows fine-grained adaptivity will require complex safety mechanisms. This may incur performance penalties.

Besides accessing memory in an unauthorized way, there are various other ways in which an extension can misbehave: resource hoarding (infinite looping, network flooding, excessive memory allocation, acquiring kernel locks, etc.), denial of service

(when the system is relying on the extension's correct execution to make forward progress) and use of unauthorized kernel interfaces. One consequence is that an extension cannot be allowed to hold any limited kernel resource for an arbitrarily long period of time [Seltzer et al. 1996].

The safety policy of the operating system can be enforced through *verification* or *protection*. *Verification* ensures the correct behavior of an extension prior to installation and deployment. We consider two kinds of verification: verification of origin, and verification of behavior. With verification of origin, code is considered safe if it is brought into the system by a trusted party (e.g., the administrator). This is a very common practise in production operating systems, such as UNIX and Windows NT, where it is called the loadable kernel module technique. The technique is used both in a human-initiated way, discussed in Section 4.2, and in a simple application-initiated way: applications with root privileges can initiate the loading of modules, or even applications with user privileges can be allowed to load a kernel module as long as it has been signed by a trusted third party (e.g., the operating system manufacturer).

With verification of behavior, the operating system tries to analyze whether a given piece of code behaves in a conforming way. Automatic verification of behavior is very appealing, but is also very hard. We will discuss an example of this technique in Section 6.2.

Protection tries to ensure the correct behavior of an application after its installation and otherwise tries to limit the damage done. Protection can be achieved by hardware means. This is typically done in microkernels. The kernel is protected from the customizations because they reside at user (nonprivileged) level. The hardware ensures that these customizations can never directly access or modify the kernel's data. When a customization fails, only the applications using it will fail as well. However, the failure of a critical service, such as a file system, may bring down the entire system. Alternatively, protection can be achieved by software means. The use of software means is more flexible and the performance of domain crossings is potentially faster. We explore software protection in Section 6.1.

Extensible kernels based on software protection logically converge to single address space operating systems, like most Java-based operating systems, where all protection mechanisms are in software [Back et al. 1998].

6.1. Software Protection

An extensible kernel cannot rely on hardware protection because the untrusted code, the extension, has the same privilege level as the kernel. Software protection tries to enforce the correct behavior of an extension while it is executing and tries to limit the damage that can be done by a misbehaving extension.

Software protection is potentially faster and more flexible than hardware mechanisms. It can be faster, as discussed in Wallach et al. [1997], and more flexible, because the safety policy need not be fixed. Hence, extensible kernels are potentially faster and more flexible than microkernels.

Software protection is pursued in two approaches discussed: *software fault isolation* and *safe languages*.

6.1.1. Software Fault Isolation (SFI). SFI [Wahbe et al. 1993] is a technique to ensure memory protection within a single address space (for instance, within the kernel). It consists of two steps. First, the untrusted code is loaded into its own *fault domain*, a logically separate part of (kernel) memory. Second, the code is changed so that the module cannot write or jump to an address outside of its fault domain. One way of implementing this is through *sandboxing*: a fault domain consists of one code segment and one data segment. A segment has the same pattern of upper address bits (the *segment identifier*). Now, before every unsafe instruction in the code segment, instructions are inserted that overwrite the upper bits of the address used in the unsafe instruction with the segment identifier.

Modules in different fault domains cannot access each other's data or code, except via an explicit RPC-interface. This RPC between fault domains performs substantially better than RPC between address spaces (as is frequently used in microkernels).

As mentioned, security involves more than just memory protection. To build a working system with SFI, as is done in the VINO project [Seltzer et al. 1996], other concerns must be addressed. In VINO, every extension in the kernel has its own stack and heap. SFI ensures the memory protection. Further, VINO uses a *lightweight transaction system* to control the execution and resource use of the extensions.

Extensibility in VINO can be done in two ways:

—An application can replace the implementation of a method of a specific kernel object (resource), if it is allowed so. This allows the standard behavior of resources to be changed.

—An application can register a handler to the kernel for a specific event, such as a network connection on a specific port. This allows new services to be installed in the kernel.

A drawback of SFI is that overhead instructions must be executed each time an unsafe instruction is encountered.

6.1.2. Safe Languages. An interesting way to ensure the kernel's integrity is by enforcing the constraints of a programming language's abstractions. The most interesting mechanism available is *type checking*. This implies that safe languages are *typed* and *type-safe*.

Languages used in research projects are Modula-3, Java and ML. ML has a formal *type system*, the well-known Hindley-Milner system, and allows for static type checking. In contrast, Modula-3 and Java are less formal, and must perform a substantive amount of runtime checking to enforce the safety policy. For instance, array boundaries are runtime checked in Java. On the other hand, ML, as a declarative (functional) language, introduces other runtime inefficiencies.

SPIN [Bershad et al. 1994, 1995] is based on Modula-3. Every interaction between an application and the kernel happens via extensions (either an extension provided by the application or the standard extension that defines the standard behavior). Every extension is associated with an event. An extension must be explicitly registered at the *dispatcher*, which installs the extensions and passes events to the extensions. Multiple extensions can be associated with the same event. The dispatcher will contact the extensions currently associated with an event and ask for permission to replace it or to add another extension.

Modula-3 is mainly used for ensuring memory safety. Additional constraints are enforced by the dispatcher and by the standard extensions. The dynamic linker ensures that only the authorized events are visible for a given extension.

An obvious drawback of a system based on a safe language is its ties to that language. All system code and extensions must be written in that language. Another drawback of a safe language is that the safety policy is fixed and determined by the semantics of the chosen language. Also, a lot of checks are performed at runtime and thus a performance cost is incurred.

6.2. Automatic Verification

In Proof-Carrying Code (PCC) [Necula and Lee 1996; Necula 1997], a proof is statically verified. This proof guarantees that the code respects an agreed-upon safety policy. When the proof is verified, the program can run at full speed with no runtime checks involved.

This approach requires the code to be presented in a specific format, called PCC. A PCC-module includes a formal proof of the compliance of the code with a given safety policy. The validity of the proof guarantees the safety of the code.

The kernel must specify and publish a safety policy. This includes a predicate in first-order logic for each instruction to

indicate in what circumstances the instruction will be safe to execute. The safety policy also includes axioms and derivation rules, which can be used in the proof.

An application will use the predicates of the safety policy to compute the *safety predicate*. Next, it will prove the safety predicate, using the rules of first-order logic and the axioms and derivation rules of the safety policy. This proof is attached to the extensions.

The kernel, at its turn, will also compute the safety predicate. Then, it checks whether the associated proof is indeed a valid proof of this safety predicate. This validation can be done through simple and efficient type checking.

PCC is a very promising approach, both in terms of flexibility (the safety policy can be specified) and efficiency (no runtime checks). But a lot of problems remain to date. One of them is the automatic generation of proofs. Other concerns are the size of the proof and the use of proofs on high-level languages.

7. AUTOMATIC ADAPTATION

As discussed in Section 3, portability through conditional compilation can be considered as a form of static automatic adaptation in our taxonomy. However, it is clear that from a research point of view, dynamic automatic adaptation is more interesting. Hence, we focus in this section on automatic dynamic adaptation, in which the operating systems adapts itself at run time to the needs of the applications.

Through automatic adaptation, the behavior of the services is changed in an implicit way. The operating system itself observes and analyzes the application. Based on this information, it adapts its policies and behavior to support the application in the best possible way.

The system behavior in this approach is analogous to the behavior of a traditional operating system. The application doesn't need to be concerned with anything other than its own functionality. The operating system is responsible for providing an optimal execution environment for the application.

Automatic adaptation is conceptually a very promising approach. Applications do not need to know anything about their own performance or requirements. They do not have to specify a certain policy or build their own execution evironment. The operating system will autonomously determine the best behavior to support the application. All information the system needs will be derived by observing and analyzing the behavior and the functionality of the application. This approach lays all responsibility in the hands of the operating system. The application cannot choose an inappropriate policy.

It is clear that automatic adaption requires a lot of knowledge and intelligence from the part of the operating system. The problem encountered here is threefold:

—The operating system must be able to extract sufficient information by simply observing the application's behavior.
—Based on this information, optimal policies must be chosen for the system's services, such as paging, scheduling, etc.
—The goal is to optimize the overall performance, for a general-purpose system. As is known from other engineering disciplines, the overall optimum performance is not necessarily obtained by selecting the local optimum performances. The interaction between the different policies is complex and not readily understood.

A substantial research effort will be necessary to overcome these issues and achieve a fine-grained, automatically adapting operating system. However, few research projects on customizable operating systems have pursued automatic adaptation. The systems discussed in this section only implement automatic adaptation in a limited way. We will now briefly describe two approaches.

7.1. Synthetix

The core idea of Synthetix [Cowan et al. 1996a, 1996b; Pu et al. 1995] is to provide specialized implementations of

operating system services, generated on the fly, through *partial evaluation*. Synthetix borrows this idea of *run-time code synthesis* from its predecessor Synthesis [Massalin 1992].

The granularity and spread of the customizability are limited: the designer decides which services can be specialized and chooses the parameters for the specialization. As with specific operating system, the kinds of applications to be supported must be known in advance to achieve sufficient adaptivity.

The service parameters are introduced by means of *invariants*. These invariants are protected by *guards*. Whenever an invariant is validated or invalidated, it is noticed by its guard. Then, the module which provides the service is replaced by a more, or less, specialized one.

For example, a system call to open a file can return specialized code to read from the file. This code could be based on invariants such as disk block size, sequential access, exclusive access, etc. When the same file is subsequently opened by a second application, the exclusive access invariant will be invalidated (by a guard in the *open* system call).

Basically, what Synthetix does is move code away from the *fast* path. In the previous example, interpretation associated with each read is moved to the open call. The implicit assumption is that a file is read more then it is opened. These policy decisions must be made by the system designer.

7.2. VINO

The general-purpose automatic approach is explored in the VINO-project [Seltzer and Small 1997], which was already discussed in Section 6. However, automatic adaptation has only been explored, not implemented in VINO.

The information needed to automatically adapt the system behavior stems from three sources:

(1) The periodic retrieval of the statistics maintained by each subsystem in VINO.
(2) The specialized compiler.
(3) *Traces* and *logs* that register incoming requests and produced results.

All information is gathered in real circumstances, during the regular operation of the application. This information is used for analysis purposes. The *online* analysis is responsible for detecting emergencies—situations where the resource consumption is bigger than expected, and rising. At this time, an adaptation is performed by applying a known heuristic. Failing an appropriate heuristic, a trace is installed on the resources and the *off-line* analysis is responsible for finding and simulating new algorithms. Obviously, this is a very challenging task which requires some learning capabilities. Nevertheless, the VINO approach is interesting just by using the heuristics for well-known cases.

As an example, consider an application that pages excessively. A trace is generated for the concerned pages. The resulting traces are investigated for well-known access patterns, such as linear memory access. If a known pattern is matched, the appropriate algorithm is installed.

VINO differs from Synthetix in that its customizability is more general-purpose in nature. Synthetix can adapt only the functions and parameters as determined by the designer. VINO supports a mechanism in which the system could develop and test new algorithms for new problems. This means that, when new needs arise, VINO could be able to cope and adapt its behavior through newly found algorithms.

8. TAXONOMY OF RESEARCH PROJECTS

In this section, we revisit the taxonomy as described previously in Section 3. In Table I, we categorize each of the research projects discussed into our taxonomy.

As can be seen from Table I, the two main directions in customizable operating systems research are:

(1) Static customization by the designer, who customizes a (general-purpose) operating system framework to a (specific) operating system. Note that a framework can also be used to create a

Table I. Schematic Representation of the Taxonomy used to Classify the Research Projects. The horizontal criterion is the initiator of customizability. The vertical criterion is the time of customization

	Human	Application	Operating System
Static	OSKit Choices Scout Pebble components		(portability through conditional compilation)
Dynamic	production operating systems	Exokernel, L4 Kea, Pebble, SPACE EROS, Fluke V++, MetaOS SPIN, VINO	Synthetix (VINO)

general-purpose operating system. For instance, Fluke is built with the OSKit framework.

(2) Dynamic customization by the applications, which request the operating system to change its policies on their behalf. This can be achieved by having the application provide operating system services as a user-level service or even as an extension directly introduced into the extensible kernel.

For dynamic or run-time adaptation, the ideal would be that the operating system itself can automatically produce an optimal execution environment. The research projects discussed in Section 7 perform modest steps in this direction. However, the majority of research projects accepts that automatic adaptation requires a high level of intelligence from the operating system, and that the applications for now know best what their needs and requirements are.

9. CONCLUSION

We presented a survey of the approaches to achieve customizability in operating systems. The survey was structured along a taxonomy used to classify the different approaches. The main line of division in the taxonomy is defined by the initiator of the customization. Customizability can be automatic (the operating system adapts itself), initiated by applications (the running applications adapt the operating system), or by a human (the operating system designer or administrator makes specific customizations). Automatic adaptation is the most ambitious approach, and can currently only be achieved in a limited form. Designer-initiated adaptation gives very good results for specific operating systems, but lacks direct support for general-purpose operating systems. The dynamic application-driven approach has the longest tradition: microkernels, having a natural customizability, have been studied for many years. More recently, dynamic customization has been tried using modern microkernels and extensible kernels.

In our opinion, a combination of dynamic and automatic techniques can lead to the most flexible customization in a *general-purpose* operating system. It is generally assumed that resource-critical applications, like database or transaction systems, will be able to assist the operating system in creating an appropriate execution environment. Hence, such applications will use application-driven techniques. It would be unrealistic however to require even the most simple application to build or specify its own execution environment. For run-of-the-mill applications, the operating system can try to create the optimal execution environment in an automatic way, using heuristics.

For *special-purpose* operating systems, we believe that designer-initiated adaptation of a fine-grained customizable operating system framework leads to the best results, provided that the applications are well-known. These conditions are typically fulfilled in case of embedded operating systems for mobile appliances.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous referees of the first version of this paper. Their detailed and interesting feedback has greatly improved the coverage and quality of the article.

REFERENCES

ARNOLD, K. AND GOSLING, J. 1996. *The Java programming language*. Addison-Wesley.

BACK, G., TULLMANN, P., STOLLER, L., HSIEH, W. C., AND LEPREAU, J. 1998. Java operating systems: Design and implementation. Tech. Rep. UUCS-98-015, Dept. of Computer Science, University of Utah. 6,.

BERSHAD, B. N., CHAMBERS, C., EGGERS, S. J., MAEDA, C., MCNAMEE, D., PARDYAK, P., SAVAGE, S., AND SIRER, E. G. 1994. SPIN - An extensible microkernel for application-specific operating system services. In *ACM SIGOPS European Workshop*. 68–71.

BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*.

CAMPBELL, R., ISLAM, N., RAILA, D., AND MADANY, P. 1993. Designing and implementing Choices: an object-oriented system in C++. *Commun. ACM 36,* 9, 117–126.

CAMPBELL, R., RUSSO, V., AND JOHNSTON, G. M. 1987. The design of a multiprocessor operating system. In *Proceedings of the USENIX C++ Workshop*. 109–125.

CHEN, J. AND BERSHAD, B. 1993. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles*.

CHERITON, D. R. AND DUDA, K. J. 1994. A caching model of operating system kernel functionality. In *Operating Systems Design and Implementation*. 179–193.

COWAN, C., AUTREY, T., KRASIC, C., PU, C., AND WALPOLE, J. 1996a. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*. 108–15.

COWAN, C., BLACK, A., KRASIC, C., PU, C., WALPOLE, J., CONSEL, C., AND VOLANSCHI, E. 1996b. Specialization classes: An object framework for specialization. In *Proceedings of the 5th International Workshop on Object-Orientation in Operating Systems (IWOOOS '96)*.

ENGLER, D. R., AND KAASHOEK, F. 1995. Exterminate all operating system abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*. 78–85.

ENGLER, D. R. KAASHOEK, F., AND O'TOOLE, J. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. 251–266.

FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A., AND SHIVERS, O. 1997. The flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. 38–51.

FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. 1996. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*. 137–151.

GABBER, E., SMALL, C., BRUNO, J., BRUSTOLONI, J., AND SILBERSCHATZ, A. 1999. The pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*. 267–282.

HORIE, M., PANG, J., MANNING, E., AND SHOJA, G. 1998. Using meta-interfaces to support secure dynamic system reconfiguration. In *Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS'98)*.

HORIE, M., PANG, J. C., MANNING, E. G., AND SHOJA, G. C. 1997. Designing meta-interfaces for object-oriented operating systems. In *Proceedings of the 1997 IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*. 989–992.

KICZALES, G., LAMPING, J., LOPES, C. V., MAEDA, C., MENDHEKAR, A., AND MURPHY, G. C. 1997. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*. 481–490.

KICZALES, G., LAMPING, J., MAEDA, C., KEPPEL, D., AND MCNAMEE, D. 1993. The need for customizable operating systems. In *Proceedings of the 4th Workshop on Workstation Operating Systems*.

LIEDTKE, J. 1993. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*.

LIEDTKE, J. 1995. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*.

LIEDTKE, J. 1996. Toward real microkernels. *Commun. ACM 39,* 9, 70–77.

MAEDA, C., LEE, A., MURPHY, G., AND KICZALES, G. 1997. Open implementation analysis and design. In *Proceedings of the 1997 Symposium on Software Reusability*.

MAGOUTIS, K., BRUSTOLONI, J. C., GABBER, E., NG, W. T., AND SILBERSCHATZ, A. 2000. Building appliances out of reusable components using pebble. In *Proceedings of the 9th ACM SIGOPS European Workshop*.

MASSALIN, H. 1992. Synthesis: An efficient implementation of fundamental operating system services. Ph.D. thesis, Columbia University, Department of Computer Science.

MONTZ, A. B., MOSBERGER, D., O'MALLEY, S. W., PETERSON, L. L., AND PROEBSTING, T. A. 1995.

Scout: A communications-oriented operating system. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*.

MOSBERGER, D. AND PETERSON, L. L. 1996. Making paths explicit in the scout operating system. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*. 153–167.

NECULA, G. C. 1997. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 106–119.

NECULA, G. C. AND LEE, P. 1996. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*. 229–243.

PROBERT, D. AND BRUNO, J. 1996. Efficient cross-domain mechanisms for building kernel-less operating systems. Tech. Rep. TRCS96-06, Department of Computer Science, University of California at Santa Barbara.

PROBERT, D., BRUNO, J., AND KARZAORMAN, M. 1991. Space: a new approach to operating system abstraction. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*. 133–137.

PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUYE, J., KETHANA, L., WALPOLE, J., AND ZHANG, K. 1995. Streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*.

SAMAR, V. AND LAI, C. 1996. Making login services independent from authentication technologies. In *Proceedings of the SunSoft Developer's Conference*.

SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*. 213–227.

SELTZER, M. I. AND SMALL, C. 1997. Self-monitoring and self-adapting operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*.

SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. 1999. EROS: a fast capability system. *Operating System Review 34,* 5, 170–185.

SOLOMON, D. A. AND RUSSINOVICH, M. E. 2000. *Inside Microsoft Windows 2000*. Microsoft Press.

SPATSCHECK, O. AND PETERSON, L. 1997. Escort: a path-based os security architecture. Tech. Rep. TR97-17, Dept. of Computer Science, University of Arizona.

VEITCH, A. AND HUTCHINSON, N. 1996. Kea—a dynamically extensible and configurable operating system kernel. In *Proceedings of the 3d Conference on Configurable Distributed Systems*.

WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. 1993. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27,* 5 (December), 203–216.

WALLACH, D. S., BALFANZ, D., DEAN, D., AND FELTEN, E. W. 1997. Extensible security architectures for Java. In *16th Symposium on Operating Systems Principles*. 116–128.

YOKOTE, Y. 1992. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Addison-Wesley, 414–434.

YOKOTE, Y. 1993. Kernel structuring for object-oriented operating systems: The Apertos approach,. In *Proceedings of the International Symposium on Object Technologies for Advanced Software*. Vol. 742. Springer-Verlag, 145–162.

Received March 2000; revised July 2001, March 2002; accepted June 2002