Flexibility in Object-Oriented Operating Systems: A Review

Vinny Cahill

Department of Computer Science
Trinity College Dublin

Abstract

This report presents a review of recent research into flexible operating systems. In this context, flexible operating systems are taken to be those whose designs have been motivated to some degree by the desire to allow the system to be tailored, either statically or dynamically, to the requirements of specific applications or application domains. We begin by presenting a review of recent research into flexible system software with particular emphasis on the motivations for providing flexibility and the different approaches to achieving flexibility that are available. We then provide an overview of the main technologies for achieving flexibility in system software that have been employed. As it turns out, the use of object orientation is a common feature of many flexible operating systems. Thus, in order to more fully illustrate the use of object-orientation to achieve flexibility, we review a number of the most influential object-oriented operating systems in detail.

Document Identifier	TCD-CS-96-05
Document Status	Technical Report
Created	12 July 1996
Revised	24 May 1996
Distribution	Public

© 1996 TCD CS

Permission to copy without fee all or part of this material is granted provided that the TCD copyright notice and the title of the document appear. To otherwise copy or republish requires explicit permission in writing from TCD.

This report presents a review of recent research into flexible operating systems. In this context, flexible operating systems are taken to be those whose designs have been motivated to some degree by the desire to allow the system to be tailored, either statically or dynamically, to the requirements of specific applications or application domains.

Section 1 presents a review of past research into flexible operating systems with particular emphasis on those that have employed object-oriented techniques to achieve flexibility. As will be seen, the use of object orientation is a common feature of many flexible operating systems. Section 2 continues from section 1 by reviewing a number of the most influential object-oriented operating systems in detail. Finally, Section 3 summarises the report and provides some perspectives on the use of object-orientation to achieve flexibility in system software.

It should be noted that this report presents only a summary of our review. A more complete presentation of this review is available on the World Wide Web (WWW) and includes links to WWW pages describing groups, projects, and individuals currently doing research in this area as well as an annotated bibliography containing several hundred entries [17].

1 Towards Flexible Operating Systems

Operating-systems research has always been driven by developments in technology as well as by goals such as improved performance, increased modularity, and the desire to improve flexibility. Technological developments such as the introduction of reduced instruction set architectures, the increased availability of shared-memory multi-processors, or the widespread deployment of high-speed and high-bandwidth networks continue to have an enormous impact on the design of new operating systems. Improving the performance of the various services traditionally provided by operating systems such as thread management, virtual memory, or the file system has likewise always been at the heart of operating-systems research. Recently, perhaps in response to the accelerating pace of advances in technology, increased modularity has also become an important goal both to improve portability and to make system development and maintenance easier. Finally, flexibility – the ability to design systems that can be tailored to the requirements of specific applications or application domains – has always been something of a holy grail for the designers of operating systems. This has never been more so than at present as is evident from recent workshops and conferences such as [3] or [4].

1.1 Why Flexibility?

In [24], Draves identifies a number of problems that can be addressed by making operating system software more flexible:

- feature deficiency: the operating system does not provide some feature required by the application;
- performance: (some) operating system services do not provide performance that is acceptable to the application;
- version skew: the application is dependent on a different version of the operating system for its correct operation.

To these might be added "feature abundance". Feature abundance occurs when the operating system provides superfluous features that, although not used by the application, result in some (avoidable) runtime overhead.

Clearly, increased flexibility should allow operating systems to be more easily tailored to provide (only) the features required by the applications to be supported. It is perhaps not so obvious how increasing flexibility might improve performance. The essential observation is that every operating system embodies particular trade-offs concerning the way in which the services that it provides are implemented. Typically, in general-purpose operating systems, these trade-offs are made to suit the requirements of what are perceived to be typical applications. The resulting trade-offs will certainly not be the right ones for every application resulting in sub-optimal performance for some applications. There is a substantial body of evidence showing that, for some applications at least, exploiting knowledge of the application in making these trade-offs can substantially improve performance. Kiczales et al. [48] use the term *mapping dilemma* to refer to these trade-offs and characterise a mapping dilemma as a *"crucial strategy issue whose resolution will invariably bias the performance of the resulting implementation"*. Decisions as to how to resolve mapping dilemmas are called *mapping decisions* and a *mapping conflict* occurs when the application performs poorly as a result of an inappropriate mapping decision. Increased flexibility is intended to allow mapping decisions to be more easily made on an application-specific basis.

1.2 Approaches to Achieving Flexibility

Two basic approaches to achieving flexibility can be distinguished: static flexibility and dynamic flexibility. *Static flexibility* allows system software to be tailored to one application, set of applications, or some particular hardware configuration, as required, at its build time, i.e., when it is compiled, linked, or loaded (depending on the software development process used). *Dynamic flexibility* allows system software to be tailored to the needs of current applications at run time.

Three major approaches to building dynamically flexible system software can be identified. One approach is to build general-purpose systems that provide services supporting a range of different policies, which are suitable for the requirements of a wide range of applications, together with a set of service-specific interfaces allowing applications to have input into the choice of policies to be used according to their particular requirements. Such systems have been variously referred to as supporting *selection* [49] or *parametric variation* [8] and are referred to in this report as being *adaptable*. Examples of adaptable systems include the SunOS operating system, which allows applications to provide advice to the virtual memory system via the `madvise` system call, and the Mach microkernel whose thread scheduling subsystem also accepts hints from applications as described in [11]. Although few systems support it, a further step in this direction would be to have the system itself choose the appropriate policies to be used for particular applications from those available – perhaps based on analysis of their past behaviour. Such systems are referred to in this report as being *adaptive*.

Another approach to building dynamically flexible system software is to allow the application to be involved in some way in the implementation of a service, perhaps by having the service make an *upcall* to a module, provided by the application, that implements some policy required by the service [49], or by allowing the application to interpose code at the interface to the service [8]. Such systems are referred to in this report as being *modifiable*. Examples of systems supporting this kind of flexibility include both the Mach and Chorus microkernels, via their external pager interfaces [67, 1], and the system call interposition toolkit for Mach 2.5 described in [39].

The other major approach to building dynamically flexible system software is to provide a means of allowing (some) system services to be added or replaced at run time in order to support new or different functionality for particular applications. Such systems are referred to in this report as being *configurable* and also *extensible* if they support the addition of new system services as well as replacement of existing services. Most microkernel-based operating systems are configurable in that they allow those parts of the operating system implemented as user-mode servers to be installed or replaced at runtime. The Spin microkernel also allows services to be installed dynamically into the kernel [9]. (Obviously, supporting multiple mechanisms and policies simultaneously or the ability to dynamically add or replace system services incurs some runtime overhead.)

To build statically flexible system software requires a system architecture that can be implemented in a number of different ways in order to include the mechanisms and policies required by different applications. The system architecture typically describes a number of subsystems of which different implementations exist. A system is built by choosing from the available subsystem implementations at build time. New implementations of particular subsystems can obviously be provided as required. Such systems are referred to in this report as being *customisable*. Examples of customisable systems include the Choices [19] and Peace [65] operating systems. In the case of static flexibility, the system architecture may be said to be *extensible* if it has been designed to facilitate the introduction of further subsystems, which provide additional functionality that is not already provided for by the architecture.

The various approaches to achieving flexibility are not mutually exclusive and different systems may employ more than one. For example, the Peace operating system can be configured dynamically [60]. It is also worth noting that the distinction between a customisable system and a configurable one may be small in practice. A configurable system that provides minimal mandatory functionality but allows the system services required by a particular application to be loaded dynamically is very close to a customisable system. The former has at least some fixed component and may incur overheads in accessing system services that are dynamically loaded.

1.3 Qualities of Flexible System Software

Flexibility, however it is supported, is not a binary attribute; different systems offer different degrees of flexibility. While there is no absolute metric by which to judge the degree of flexibility of a system, Kiczales and Lamping [47] have identified a number of "qualities" that can be used to discuss the flexibility of a system:

- *incrementality* refers to the degree to which the effort necessary to specialise an implementation is proportional to the amount of change required [49];
- *scope control* refers to the degree to which the effect of modifications to the implementation of a service affect all or only some of the clients of that service – ideally such modifications should only affect those clients that desire it;

- *interoperability* is the degree to which applications that use non-standard mapping decisions can interact with other applications; and finally,
- *robustness* is the degree to which the implementation is graceful in the face of bugs in extensions.

1.4 Technologies for Achieving Flexibility

Recent developments in operating system architecture have been dominated by the transition from so-called monolithic operating systems to those based on microkernel technology. The introduction of microkernels has been largely motivated by the promise of better support for distributed and multi-processor systems (i.e., a response to recent technological advances) as well as improved modularity and flexibility albeit at the cost of poorer performance in some cases [12].

However, the use of microkernel technology is not the only possible approach to improved flexibility. In fact, five distinct, although not mutually exclusive, technologies that have been or are being employed to achieve flexibility can be identified:

- microkernel technology;
- application-specific operating systems;
- object orientation;
- program families; and
- open implementation and reflection.

Most operating systems that have flexibility as a goal employ some combination of these technologies as illustrated in figure 1.

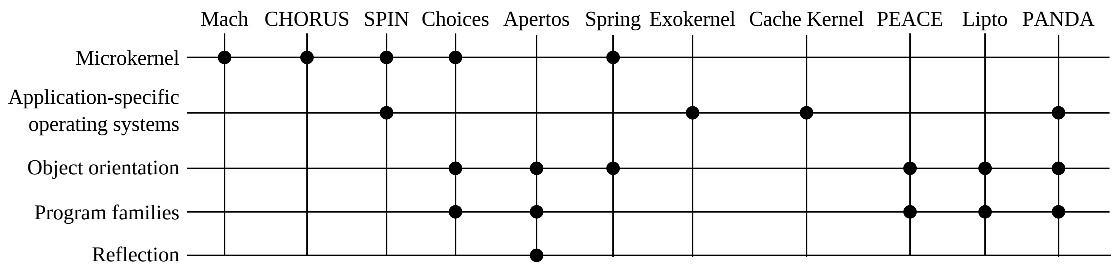

Figure 1: Technologies employed in flexible operating systems.

Microkernel Technology

Compared to traditional monolithic operating system designs, microkernels primarily offer increased flexibility by allowing large parts of the operating system to be implemented as one or more (user-level) servers. Microkernels typically provide low-level abstractions such as tasks, threads, virtual address spaces, and ports for local inter-process communication (IPC). Higher-level abstractions such as particular process abstractions as well as services such as file management and network communication are typically implemented by servers. The effect of this separation is that the kernel can support different operating system personalities, simultaneously or at different times, thereby providing the flexibility necessary to address feature deficiency and version skew problems. For example, the Mach microkernel has been used to support various operating system personalities ranging from MS-DOS to 4.3BSD UNIX and VMS [12]. Likewise, the CHORUS microkernel supports both UNIX and an object-support operating system personality known as COOL [50]. In general, multiple versions of the same personality can coexist.

This separation also allows application developers the choice of implementing directly on the microkernel interface as well as on an appropriate operating system personality thereby avoiding feature abundance. For example, real-time or embedded applications that have no need for the full generality of a traditional operating system might be implemented directly on the microkernel interface thereby optimising their use of system resources and improving performance.

Flexibility is also improved because modifying components of the operating system implemented as servers is simpler than modifying integrated components of a monolithic kernel. Thus, new implementations of particular services can be developed and installed more easily. However, it is worth noting that the granularity of such changes, i.e., the granularity of scope control, is quite coarse since the unit of replacement is typically a server and all clients of that server are affected. Likewise, the degree of incrementality is quite low since there is typically no specific support for modifying a single server.

Microkernels have also attempted to provide increased flexibility in other ways. For example, as described previously, Mach provides an interface to its thread scheduling mechanisms that allows applications to influence their scheduling policy. Moreover, microkernels usually allow some aspects of the abstractions that they provide to be implemented in cooperation with applications; the best known example is probably the use of external pagers to support the microkernel's virtual memory system. Provision of an external pager allows an application to control paging operations on its data, often at quite a fine granularity of scope control (since different pagers can be used for different regions of a single address space). Extensions to the external paging interface also allow applications to be involved in page replacement policy decisions [51]. Again, incrementality is poor since, in the absence of additional support, every pager has to be written from scratch.

Notwithstanding the considerable advantages that microkernels offer over traditional operating system designs in terms of flexibility, it is still worth noting that with the exception of some of the mechanisms outlined in the previous paragraph, the kernels themselves tend to be rather inflexible. Internal modularity is usually emphasised in order to isolate machine dependencies and thereby improve portability. However, the range of policies and mechanisms supported is usually quite static and not capable of being easily altered to respond to the characteristics of particular applications. Moreover, the division of functionality between kernel and servers is typically fixed. The CHORUS microkernel addresses this issue to some extent by allowing privileged servers to be loaded into the kernel's address space [57]. In that sense, CHORUS may be said to provide an extensible kernel. This deficiency of previous microkernels was also addressed explicitly in the design of the SPIN kernel [9]. The main goal of the SPIN design was to make the kernel extensible and configurable by allowing applications to dynamically install so-called *extensions* into the kernel; such extensions may change both the interface and implementation of the kernel. In contrast to other microkernels, SPIN provides only a default set of low-level, or *core*, services that are responsible for managing memory and processor resources. Applications are expected to implement higher-level kernel abstractions to suit their own needs by means of extensions built from these core services. For example, SPIN provides only very primitive support for multiplexing the processor among multiple threads of control and an application can provide its own thread package and scheduler as an extension that runs within the kernel. Language and link-time mechanisms are used to protect operating system code running in the kernel address space while providing extensions with fine-grained protected access to core services and services provided by other extensions (often at the cost of only a single procedure call). In particular, the SPIN kernel and extensions are implemented in a type-safe language (Modula-3). As a result, extensions cannot access memory or execute privileged instructions unless they have been given explicit access through an interface.

Application-Specific Operating Systems

In [5], Anderson put the case for what he termed "application-specific operating systems". The goal of an application-specific operating system is to deliver high-performance, combined with the ease of sharing resources and data among applications and the simpler programming model found in general-purpose operating systems, without requiring the application programmer to (re)write large parts of the operating system as had traditionally been the case. In Anderson's proposal as much of the operating system as possible is pushed out into runtime libraries linked with each application. The role of the operating system kernel is then reduced to mediating application requests for physical resources and enforcing hardware protection boundaries between applications. The runtime library linked with each application is responsible for providing the traditional operating system interface to applications.

While Anderson did not describe a specific operating system implemented according to this principle, a number of other researchers have proposed operating system designs conforming to this general approach, most notably PANDA [7], V++ [21], and ExOS/Aegis [29]. In each case, the designers have proposed kernels that are responsible only for mediating application requests for physical resources and enforcing hardware protection boundaries between applications as suggested by Anderson. All higher level operating-system mechanisms and policies, which could benefit from application-oriented specialisation, are intended to be implemented in user-level servers and libraries.

Anderson's position was probably motivated by his earlier experience with scheduler activations [6] in which the implementation of kernel threads in Topaz was modified so that the kernel was only responsible for the allocation of processors to applications while the applications themselves were made responsible for actually scheduling threads (in user mode). However, his approach can also be seen as a natural evolution of the microkernel-based approach to operating system design. In this context, the main innovation can be seen as the provision of kernel interfaces that expose the management of physical resources to applications rather than providing higher-level interfaces as in traditional microkernels. The task of building the application-specific operating system outside of the kernel remains. Interestingly, the designers of most of the existing application-specific operating systems have suggested using object-oriented techniques to provide customisable implementations of the user-level components of the system. In addition, in most cases the supervisor-mode component of the operating system, although small, is fixed. Aegis allows application code to be dynamically loaded into, and run, in the kernel address space using a variety of techniques to protect other code running in the same address space from malicious interference. The SPIN microkernel mentioned previously falls somewhere between an application-specific operating system, as defined by Anderson, and a traditional microkernel in that it provides a small set of core kernel services from which application-oriented services can be built. However, in the case of SPIN those services are expected to be implemented by in-kernel extensions rather than by runtime libraries.

Object Orientation

A complementary approach to the development of flexible system software is that based on the use of object-oriented techniques. While both traditional and microkernel-based operating systems have been implemented in object-oriented programing languages, Russo [58] characterises an object-oriented operating system as one that is both implemented using object-oriented techniques and provides its services via invocations on system objects. Object-oriented operating systems are attractive for a number of reasons. Object orientation is currently the software engineering discipline of choice in the computer industry. The advantages – from the software engineering point of view – of using object-oriented analysis, design, and implementation techniques are well known [13] and include code reuse, improved portability, easier maintenance, and incremental extensibility. For these reasons, it seems only natural to use these same techniques in the design and implementation of operating systems. Indeed, this observation has lead some researchers to conclude that object-oriented operating systems are "boring" as a research topic [34]. However, object-oriented operating systems offer a number of other advantages in addition to those already mentioned. The use of object-oriented techniques to design an operating system provides the opportunity to build highly customisable systems. Object-oriented operating systems, such as Choices, that are based on frameworks support code reuse and customisation through the use of inheritance [19]. Moreover, system classes may be exposed to applications so that they can take advantage of inheritance and polymorphism to provide specialised implementations of some system services for their own needs as in PANDA [7]. Choices is reviewed in detail in section 2.1 as being representative of this approach.

Recent multi-server microkernel-based operating systems have also made extensive use of object orientation, in particular, to define the interfaces between the various servers making up the (distributed) operating system [32, 52]. Use of object-oriented interfaces allows a clean separation of interface from implementation permitting multiple implementations of a service to coexist and, in particular, allowing applications to provide their own implementations of system services. As [34] points out, such separation of interface and implementation blurs the distinction between applications and system services allowing users to acquire and install "shrink-wrapped" system services from multiple vendors and install them (possibly dynamically) as long as they conform to the appropriate interfaces.

Furthermore, the use of object orientation allows location-transparent access to distributed services to be provided via remote object invocation (ROI) and hence provides a good basis for so-called "single system image" distributed operating systems. Location transparency allows the configuration of the system (including the distribution of servers) to be adjusted dynamically. It is interesting to note that microkernel-based operating systems are now employing the kind of support offered by object-support operating systems – such as ROI, dynamic linking, and garbage collection – to implement their servers [61]. Section 2.4 describes Spring as being representative of the application of object-orientation in a distributed multi-server operating system.

Notwithstanding the availability of support for location-transparent object invocation, most microkernel based operating systems distinguish services provided by the microkernel (and accessed via system calls) and those provided by servers (and perhaps accessed via object invocation). Moreover, most still map a server to a protection domain. A notable exception that emphasises orthogonality between modularity and protection is Lipto [28], which is reviewed in detail in section 2.2.

Program Families

An alternative approach to the design of (statically) flexible system software, which predates the recent interest in microkernels and object orientation, is based on the use of program families. Parnas [53] defines program families as *"sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analysing individual members"*. The members of a family might differ only in that they run on different hardware configurations, because they embody different mapping decisions due to their intended use, or because some members of the family provide only a subset of the features of other members so that their clients do not have to pay the costs associated with the resources consumed by unused features [54].

In order to support fine-grained customisation of the members of a program family, its design begins by identifying *a minimal subset of system functions* that might conceivably perform a useful service. This minimal subset may be sufficient for the needs of some clients – in which case it is referred to as *a minimal but perfect subset*. Thereafter, further members of the family are implemented by making *minimal extensions*, which provide additional functionality, to the system. Essentially, each extension defines a new virtual machine that is available to clients. [54] claims that maximum flexibility is achieved when the smallest possible extensions are made in each iteration. However, as discussed in [65] there is a trade-off between taking such a pure bottom-up approach to the design of a program family – in which a number of extensions may be needed before a system meeting the needs of any client emerges – and taking a more pragmatic approach which is driven top-down by application requirements – leading to a design in which extensions are dictated by the immediate needs of some clients.

Operating system designs based on the use of program families were proposed as early at the 1970s [33]. However, this approach was largely ignored by operating-systems researchers until the widespread adoption of object-oriented software engineering techniques and their application to operating systems in the late 1980s. As described in [40], the minimal subset of system functions is naturally implemented as a set of superclasses and minimal system extensions can be introduced by means of subclassing. The design of the PEACE object-oriented operating system is based on this principle and is reviewed in detail in section 2.3.

Open Implementation and Reflection

Recently Kiczales has proposed a new model of abstraction for software engineering known as the *open implementation* model [45, 48, 49, 47]. The basis of this model is the argument that while the traditional information hiding/black-box model of abstraction shields clients of an abstraction from having to know the details of how that abstraction is implemented, it also prevents them from altering those details when desirable. In preference, clients of an abstraction should be able to exercise some control over the mapping decisions made in its implementation. Kiczales advocates distinguishing the *base interface* of an abstraction, which offers the usual functionality of the abstraction, from the *meta interface*, which allows clients to control how certain aspects of the abstraction are implemented. Rather than exposing all the details of the implementation, the meta interface should provide an abstract view of the implementation that allows clients to adjust the implementation in well-defined ways.

Open implementation provides a model for how flexible software systems could be built as well as a common basis for discussing the designs of existing flexible systems. Indeed much of the terminology used to describe flexible systems in this chapter – mapping dilemmas, scope control and incrementality – has its roots in the open implementation community. Kiczales claims that many existing approaches to providing flexibility in system software can be understood in terms of the open implementation model and the separation between base and meta interfaces. For example, the Mach external pager interface can be seen as a meta interface to the Mach virtual memory system.

The model of open implementation was an outcome of work on the use of *metaobject protocols* in the implementation of object-oriented programming languages [46]. A metaobject protocol is essentially a meta interface to the implementation of a language's object model that allows the application programmer to control how certain aspects of the object model – such as object invocation, dispatching, or inheritance – are implemented. The use of metaobject protocols is itself a synthesis of earlier work on reflection and object orientation. A reflective system is essentially one which is implemented in terms of itself, for example, in the case of programming languages, a reflective language is one where the implementation of the language uses the language itself. Parallel work has applied these same ideas to object-oriented operating systems leading to the development of reflective object-oriented operating systems, most notably Apertos [66]. While these systems offer the potential of being highly configurable, to date their performance has been poor. However, on-going research into the use of techniques such as dynamic code generation may alleviate these problems in the future [55].

2 Influential Object-Oriented Operating Systems

Section 1 reviewed a number of different approaches to achieving flexibility in systems software. This section continues by reviewing a number of object-oriented operating systems that have been particularly influential or are otherwise noteworthy.

2.1 Choices – An Object-Oriented Operating System Framework

Choices [18, 19] is an object-oriented operating system under development at the University of Illinois at Urbana-Champaign since 1987. Native implementations of Choices exist for the Sun SPARCstation 1 and 2, the Encore Multimax, IBM-compatible Intel 386- and 486-based personal computers (PCs) and the Intel Hypercube. In addition, a version of Choices, known as VirtualChoices, runs above SunOS 4.1.

The Choices project was perhaps the first to explicitly undertake the design and implementation of a family of operating systems using object-oriented techniques. In particular, Choices pioneered the use of frameworks as the basis for the design of an operating system family. As such, Choices has been highly influential and has inspired much of the recent interest in object-oriented operating systems.

Goals and Approach

The primary goal of the Choices project was to explore the use of object-oriented techniques for the design of a family of operating systems for distributed- and shared-memory multi-processors. The Choices design is intended to allow the construction of customised operating systems by permitting a high degree of application-specific specialisation including the provision of support for real-time and embedded systems [18]. A design methodology based on object-orientation and the use of frameworks was adopted in order to support customisation as well as to facilitate both design and code reuse between members of the operating system family through the use of inheritance. The design in intended to encourage modifications and extensions, to support porting, and to facilitate maintenance.

The design of Choices is captured as a framework that describes the fundamental (abstract) components of the system and the ways in which they interact. At runtime, a Choices system consists of a collection of objects that implement all aspects of the system including the application and hardware interfaces as well as all operating system resources, mechanisms, and policies.

In fact, Choices was designed as a hierarchy of frameworks. The top-level framework defines abstract classes representing the fundamental components of the system and constraints to which its sub-frameworks must conform. The constraints imposed by the framework on its sub-frameworks ensure that they may be integrated in a consistent manner to form a Choices system. The sub-frameworks represent subsystems of the operating system and introduce specialisations of the abstract classes defined by the top-level framework. The sub-frameworks may also introduce additional abstract classes and constraints. Each (sub-)framework defines interfaces as methods on abstract classes but restricts the implementation of those interfaces by the constraints that it imposes. The components, interfaces, and constraints introduced by the framework and its sub-frameworks are common to all members of the operating system family and represent the common architecture shared by all family members. In this sense, the framework supports design reuse across all the members of the operating system family. A particular operating system, i.e., a particular instantiation of the Choices framework, is built by introducing concrete classes implementing the components defined by the framework.

The Choices Framework

In Choices, the top-level framework introduces three fundamental abstractions: *processes* (i.e., threads), *domains* (i.e., virtual address spaces) and *memory objects* as illustrated by its entity relationship diagram shown in figure 2. Choices sub-frameworks describe various subsystems including process management, memory management, persistent storage, message passing, communication protocols, and device management. For example, figure 3 shows the entity relationship diagram illustrating the main abstractions of the memory-management sub-framework.

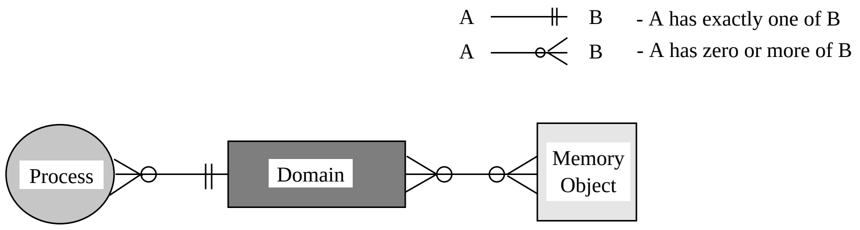

Figure 2: The Choices framework.

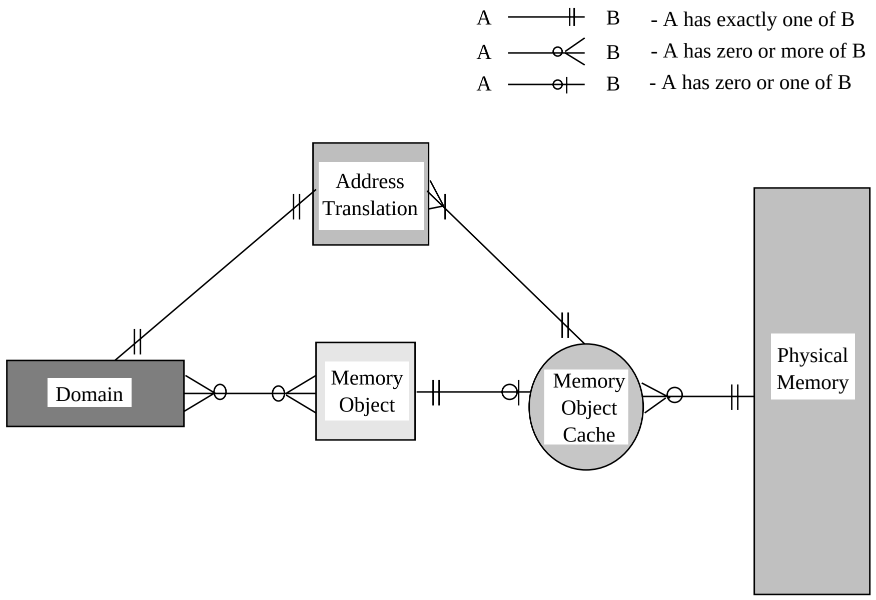
A B - A has exactly one of B
A B - A has zero or more of B
A B - A has zero or one of B
Address Translation
Domain
Memory Object
Memory Object Cache
Physical Memory

Figure 3: The Choices memory-management sub-framework.

A Choices system includes a single *kernel domain* and a number of *user domains*. Both the user and kernel domains may support multiple processes. *System processes* run in the kernel domain in supervisor mode while *application processes* run in user domains (and may run in the kernel domain as a result of a system call). The framework supports the use of numerous scheduling policies as well as gang scheduling for tightly-coupled, shared-memory multi-processor applications.

The memory-management sub-framework supports multiple 32-bit virtual address spaces, one- and two-level paging, and shared memory. Memory objects, backed by different data sources, can be mapped into different regions of a virtual address space and can have their own individual page replacement policies. Memory objects can be used to implement shared memory, since they can be mapped into multiple virtual address spaces simultaneously, as well as memory-mapped files, since they can be backed by a file. Later versions of Choices support persistent objects as a further specialisation of the memory object abstraction and distributed shared memory (DSM) by allowing memory objects to be mapped at multiple nodes simultaneously.

The persistent storage sub-framework supports a number of different record and stream-oriented file systems as well as a persistent object store (POS). This includes the 4.2BSD, System V UNIX, and MS-DOS file system interfaces and disk formats. The design allows any selection or combination of the available storage systems to be included in a single Choices system.

The message-passing sub-framework supports a range of options for each of the major functional components of a message-passing system including the choice of message transport mechanism, synchronisation mechanism, and reliability semantics. Likewise, the communications sub-framework supports a number of communications protocols including the Internet protocol stack. Finally, the device management sub-framework provides device abstractions and the ability to dynamically load device drivers.

In Choices, the interface to the kernel is provided by a collection of kernel objects. Applications that need to call the kernel must first lookup the required kernel object in a privileged name server. If the caller is authorised to invoke the requested object, the name server allocates a proxy for the target object in read-only memory in the caller's address space and registers the proxy with the kernel. The proxy acts as a capability for the kernel object. The caller can subsequently invoke the proxy as if it were the target object. When invoked, the proxy code will trap into the kernel to invoke the kernel object that it represents (after appropriate protection checks have been carried out to verify the source of the request).

Support for Distributed and Persistent Programming

Choices was originally developed as an operating system for distributed- and shared-memory multi-processors providing such parallel programming functionality as multiple threads per address space, various synchronisation mechanisms, gang scheduling, and message passing. Later versions of Choices added additional support for distributed and persistent programming in C++ including support for ROI and DSM.

Page-based DSM is supported by a specialisation of the memory-management sub-framework that allows a single memory object to be mapped in multiple address spaces on different nodes of a network. The DSM protocol employed supports a multiple-reader/single-writer policy using write invalidation [38, 20].

Access to remotely created C++ objects is also supported by an extension of the kernel object proxy mechanism that allows access to remote objects using the underlying Choices message-passing primitives [22].

Persistent C++ objects are supported as a specialisation of the file system sub-framework. User-defined subclasses of the class `AutoLoadPersistentObject` support automatic activation (and deactivation) on demand [19]. References between such persistent C++ objects are implemented as instances of subclasses of the class `Reference`. An instance of `Reference` stores a descriptor for the target persistent object that includes its location in the file system. The `Reference` class hierarchy mirrors the `AutoLoadPersistentObject` class hierarchy allowing almost all uses of persistent objects to be type checked at compile time. Use of `Reference` also allows the use of references to persistent C++ objects to be syntactically identical to the use of references to ordinary C++ objects. Garbage collection of acyclic persistent data structures based on reference counting is also supported [20]. Finally, DSM can be used to access remote persistent objects.

2.2 Lipto – Orthogonality of Modularity and Protection

Lipto [27, 28] is an object-oriented operating system developed at the University of Arizona between 1991 and 1993. Lipto was developed as a prototype implementation of an object-oriented architecture for a family of portable distributed operating systems.

Goals and Approach

The primary goal of the Lipto project was the development of a new operating system architecture supporting both a high degree of application-specific customisability and configurability, and, a high degree of portability. The underlying philosophy was that applications should be provided only with the functionality that they need from their operating systems and should not have to pay a performance penalty due to unused functionality.

Central to the Lipto approach are two basic principles. The first principle is that operating systems should be built from collections of composable and configurable *services*. The second principle is that modularity – the division of a program (in this case the operating system) into separate loosely-coupled units exporting well-defined interfaces – is orthogonal to protection and should therefore be decoupled from protection in the operating system architecture. Such decoupling is a prerequisite for fine-grained and efficient system customisation and configuration.

While the importance of modularity in operating system design is well known, most operating system architectures have tended to tightly couple modularity and protection. For example, in conventional microkernel architectures the unit of modularity is a "server" and servers are mapped one-to-one on to protection domains. In such architectures, modularity is necessarily coarse-grained due to the cost of crossing protection boundaries. The operating system designer is forced to make a trade-off between modularity and performance. Moreover, the division of functionality between different servers, as well as between servers and the kernel, is normally static and made as an early design decision. The result is the traditional debate within the operating systems research community as to what should be "in the kernel" and what should be implemented in user space. Due to the coarse granularity of modules, application-specific customisation – in particular incremental customisation – of the system is difficult.

The Lipto argument is that, by decoupling modularity and protection, decomposition of the system into *modules* can proceed according to best-practice software engineering principles without concern for the cost of crossing protection domains. The assignment of modules to protection domains then becomes a matter of configuration. In this way, the trade-off between modularity and performance traditionally made early in the design of an operating system becomes a trade-off between protection and performance made at system configuration time. Moreover, the granularity at which the system can be customised is much finer and the assignment of modules to protection domains can be done on an application-specific basis.

Given a fine-grained decomposition of the system into modules, operating system services can then be composed dynamically from appropriate collections of modules without regard for the eventual configuration of the modules with respect to protection domains.

The major requirement for supporting these goals is the provision of location-transparent operation invocation between modules that is optimised for the common case of collocated modules. Given location-transparent operation invocation, modules can use other modules without knowledge of whether they are currently located within the same or a different protection domain. Efficient local invocation allows the efficient implementation of services composed of modules located within a single protection domain – comparable to monolithic operating system architectures – without the need to sacrifice modularity.

Architecture and Infrastructure

Lipto's architecture (see figure 4) consists of: a fixed nucleus, referred to as the *nugget*, which provides only functionality that *must* be implemented in supervisor mode; infrastructure supporting modules and objects, in particular, location-transparent operation invocation and dynamic binding; and, finally, a configurable and extensible collection of modules implementing other operating system services. The nugget provides a minimal set of trusted services including low-level process and memory management (i.e., the implementation of address spaces including the kernel address space). As well as supporting location-transparent operation invocation, the module and object infrastructure supports dynamic loading of module implementations. Modules can be assigned arbitrarily to address spaces (i.e., protection domains) including the kernel address space.

In Lipto, a module provides the implementation of one or more types of object. Objects are passive, encapsulate state, and export a set of operations. The composition of services from modules is governed by abstract interface definitions called *service classes*. A service class defines a downcall interface and an upcall (or callback) interface for each of the object types used to provide a service in that class. Each module implements exactly one service class. A service is composed by building a graph of modules in a bottom-up fashion starting from modules in service classes that need no underlying services. Service classes form a hierarchy according to a conformity relationship and Lipto supports both inheritance of interfaces and polymorphism. Modules that implement the same service class, i.e., that provide the same abstract interface, are interchangeable. Moreover, a client requiring a module that implements a particular service class can also use any module in a service class that is a descendent of the required service class in the inheritance graph.

The main feature of the infrastructure supporting modules and objects is location-transparent operation invocation. In order to support fine-grained modularity, the invocation mechanism is optimised for the common case of collocated modules. Virtual addresses are used as object references and proxies are used to represent non-local objects and to forward invocation requests to the objects that they represent. Binding to a service is accomplished using the so-called

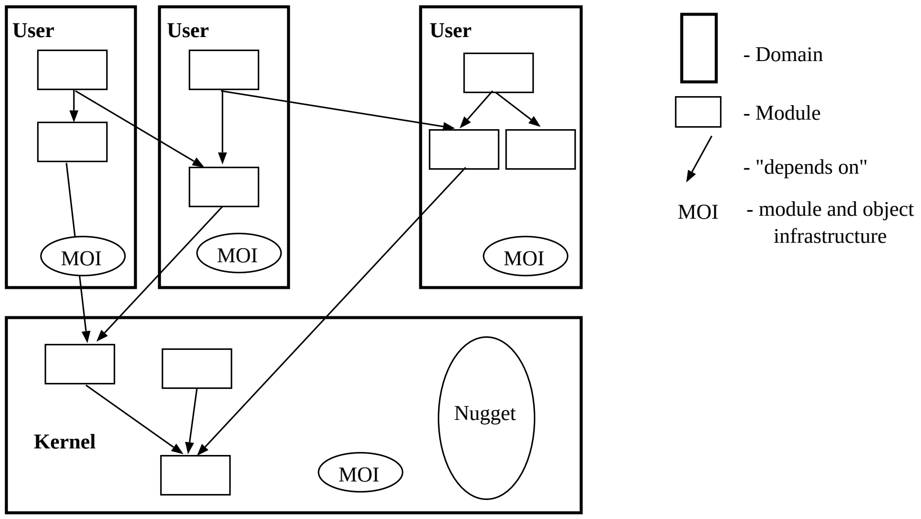

Figure 4: The architecture of Lipto.

system directory. Clients lookup services in the system directory using service identifiers and are returned references to the appropriate server objects. Object location, authentication, and binding all happen as a side effect of service lookup, effectively establishing a connection between client and server, and further optimising invocation performance. The underlying remote procedure call (RPC) service used for operation invocation is itself configurable depending on the location of client and server. Different RPC protocols can be used for cross-address-space and remote invocations. A protocol called *user-kernel* RPC is used when objects in user address spaces invoke objects located in the local kernel address space and a protocol called *kernel-user* RPC when objects in the kernel address space invoke objects in local user address spaces.

It is important to note that Lipto does not require that modules be written in any particular language, object-oriented or otherwise, as long as the implementation of objects respects the encapsulation required by the architecture. Likewise, the Lipto architecture does not support inheritance of module implementations relying instead on composition to achieve code reuse. Moreover, Lipto specifically does not provide object support mechanisms such as persistence or migration, preferring to leave that task to higher-level subsystems. However, the designers do advocate that distributed applications be implemented in languages that support distributed and persistent objects [26].

Finally, although Lipto supports incremental configuration of operating system services via composition, actually carrying out such configuration requires that applications be provided with a meta-level interface allowing them to control the composition of the services that they use [25].

2.3 Peace – An Object-Oriented Operating System Family

Peace [65] is the name given to a family of operating systems developed at the German National Research Center for Information Technology between 1990 and 1993. Peace, which stands for Process Execution And Communication Environment (and also for the key design goals of the project Portability, Efficiency, Adaptability, Configurability, and Extensibility), is a second generation *object-oriented* operating system family which evolved from a previous *microkernel-based* operating system family developed between 1986 and 1990, and also known as PEACE.

The PEACE family of operating systems is targeted primarily at distributed-memory parallel computers, in particular, *massively* parallel machines with (potentially) thousands of processing nodes. PEACE runs as a native operating system on the MANNA supercomputer and PowerPC-based massively parallel computers, and as a guest system above PARIX, SunOS 4.1, and WINDOWS-NT.

Goals and Approach

Perhaps the major goal of PEACE was to develop an operating system architecture capable of delivering good communication performance to parallel applications. In particular, one of the main goals of the design was to minimise both the *message-startup time* and *communication latency* associated with message-passing operations. Message-startup time is described as the processing time lost to a program as a result of it issuing a request to transmit a message. Communication latency is the one-way, end-to-end time required to transmit and deliver a message including both the protocol overhead at each end and the network latency. The PEACE designers assert that message-startup time is the dominant factor since communication-latency–hiding techniques (such as the use of multiple threads) are available. Experience with the previous microkernel-based implementation of PEACE showed that the major source of message-startup overhead is operating system software, particularly where message startup requires a protection boundary between kernel and user space to be crossed. Figures given in [64] show that multi-tasking overhead for nodes running a single application task contributed 74% of the message-startup time in that version of PEACE.

The fundamental tenets underlying the design of PEACE are that the *design* of an operating system for a massively parallel machine should be based on a family of operating systems (in which each member of the family is tailored to the needs of specific applications) and that the *implementation* of a family of operating systems should be based on the use of object orientation. Thus, applications do not have to pay for unnecessary functionality, in particular, unnecessary isolation of kernel and user space where only a single application task per node is required.

As described in section 1.4, the design of an operating system family begins by defining *a minimal subset of system functions* providing a collection of fundamental abstractions. This minimal subset may be sufficient for the needs of some applications – in which case it is referred to as *a minimal but perfect subset* – or may simply serve as the basis for implementing minimal system extensions towards the requirements of specific applications. In fact, a particular application can be seen as the final system extension. While the design of an operating system family does not imply any particular implementation technique, the PEACE designers consider that it is *"almost natural to construct program families by using an object-oriented framework"* [65]. The minimal subset of system functions is implemented as a set of superclasses and minimal system extensions are introduced by means of subclassing. Moreover, polymorphism allows different implementations of the same interface to coexist. Finally, providing an object-oriented interface allows applications to specialise operating system abstractions by means of inheritance.

The designers of PEACE rejected a design based on the use of an underlying microkernel. Microkernels provide multi-tasking support by default and thus introduce both unnecessary functionality and resulting overhead for many parallel applications in which only a single task per node is required. Moreover, microkernels introduce mandatory isolation of kernel and user space, often making access to the network extremely heavyweight (due to the need to cross the kernel/user protection boundary) and resulting in poor message-startup time.

Although realising that support for kernel isolation and multi-tasking are required by some parallel applications, in particular to allow *scaling transparency* whereby the number of tasks used by the application is independent of the number of nodes, the PEACE philosophy dictates that they should not be imposed on all applications unnecessarily. Thus, the PEACE family of operating systems includes members supporting nodes with different operating modes including single-user/single-tasking (SUST), single-user/multi-tasking (SUMT), multi-user/single-tasking (MUST) and multi-user/multi-tasking (MUMT) nodes. The PEACE family-based approach to operating system design therefore subsumes the traditional microkernel-based approach.

The same philosophy is applied to the design of the PEACE communications subsystem. Realising that the choice of protocol depends on the operating mode of the system, the nature of the application, the communication paradigm used at the application programming interface, and the interconnection network provided by the parallel machine, the designers of PEACE based the design of the communications subsystem on a protocol suite from which appropriate protocol implementations may be chosen.

In addition, while applications should be provided with transparent access to file, input/output (I/O), and other system services, the PEACE design recognises that not all nodes need to provide these services. Moreover, such services need only be loaded as used. Thus, the design supports a dynamically alterable operating system structure based on incremental loading (and replacement) of services as required.

In summary, PEACE supports both *horizontal extension* and *vertical extension* where the former refers to the introduction of new abstractions and the latter to the provision of new implementations of an abstraction by customisation or component replacement.

The PEACE Family

A member of the PEACE family is composed of three major functional components known as the *nucleus*, the *kernel*, and POSE (the Parallel Operating System Extension) as shown in figure 5.

The nucleus acts as the minimal subset of system functions and provides a runtime executive for thread processing and IPC. The nucleus is node-dependent, present on every node of the parallel machine, and acts as the software backplane connecting all the components running on the machine. The nucleus interface defines an abstract data type of which several implementations exist depending primarily on the desired operating mode of the node on which the nucleus will run. The implementations of this abstract data type define a nucleus family on top of which the PEACE parallel operating system family is implemented.

The second major functional component is the kernel providing thread management (including the ability to create and delete threads), device abstractions, and propagation of TRAPs and interrupts. The kernel need not be present on every node and hence its services may be accessed

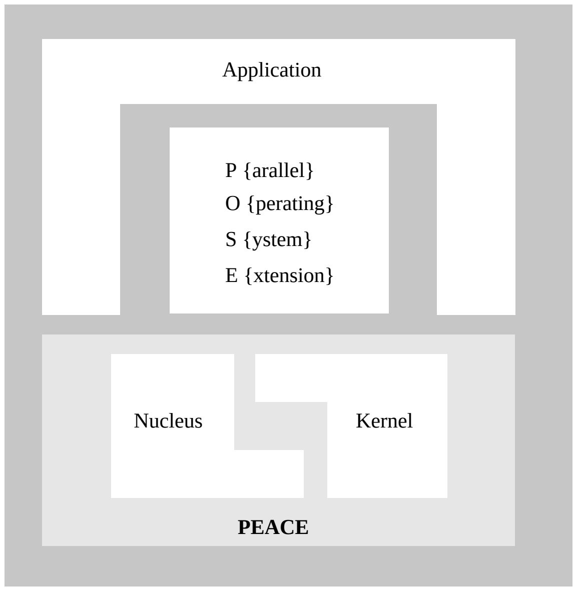

Figure 5: The major components of PEACE.

remotely. The kernel is however node-dependent since it may need access to specific devices or perform other functions that, although not being hardware dependent, must be executed locally at the appropriate node, for example, creation of a new thread.

Finally, POSE provides application-oriented operating system services including memory management, file management and I/O. Again, POSE need not be present on every node and its services may be accessed remotely. POSE is neither hardware- nor node-dependent.

The nucleus and kernel are bundled together as a single *entity*, known as the *kernel entity*, which can be loaded to bootstrap a node. In general, entities correspond to tasks and form the units of distribution in the system. Entities can be loaded dynamically and are used to represent system extensions. POSE is composed of a number of entities that can be loaded on any appropriate node as required. Thus, POSE is a dynamically alterable component supporting incremental loading of the operating system. Whether an entity shares a node with another entity is a matter of configuration rather than design.

All of the components described above are represented by objects (active objects in the case of the kernel and POSE) and their services accessed via local, cross-address-space or remote invocation as appropriate. As will be described below, such an object invocation might result in loading of the entity providing the required service.

The nucleus family (see figure 6) encompasses members implementing a number of variants of the four different operating modes outlined above. The most primitive SUST family member provides network communication and non-preemptive thread processing. The next member provides preemptive thread scheduling based on a timer and thereby allows both kernel and application threads to share a node. The final SUST member provides the basis for higher-level multi-user and multi-tasking family members by providing support for *nucleus separation*. This requires that TRAPs be used to call the nucleus although no memory protection is provided. Nucleus separation also allows the operating mode of a node to be changed dynamically by replacing the kernel entity with a different version and changing the TRAP vector table entries that point to nucleus functions [41].

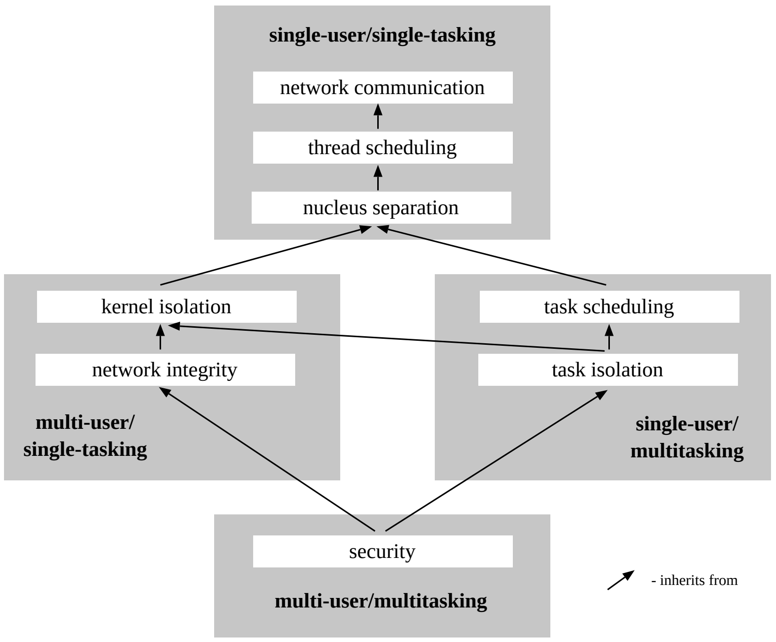

Figure 6: The PEACE nucleus family.

The MUST family members allow a multi-node machine to be partitioned among different users at the same time. Each node is assigned to a single user at a time and multiple users must be able to share the machine's interconnection network safely. This requires that the network interface of each node be protected so that one user's application cannot interfere with another's. Thus, the basic MUST family member supports memory protection and vertical isolation of the kernel entity forcing applications that wish to access the network to do so via the nucleus. A further MUST family member adds support for capability-based object addressing so that one application cannot use communication endpoint identifiers referring to objects belonging to another application.

The SUMT family members naturally support multiple tasks per node where each task consists of a collection of threads. The basic SUMT family member adds support for task scheduling while a further member adds support for horizontal isolation of tasks based on memory protection (as well as vertical isolation of the kernel).

Finally, the MUMT family member combines features of both the MUST and SUMT members to provide both network security and protected address spaces.

Based on the architecture and different operating modes outlined above, PEACE distinguishes three different types of nodes, each of which supports a number of different configurations (see figure 7). *User nodes* are dedicated to the execution of user applications. A user node runs a nucleus and may run a kernel. The nucleus may provide single- or multi-tasking with fixed numbers of threads and tasks. If the kernel is present, thread and task creation are supported. *System nodes* provide globally accessible operating system services, always run a kernel, and may run POSE. A system node that does not run POSE only provides access to local devices (i.e., it acts as a device server), while a node that runs POSE may provide higher-level operating system services. Finally, *generic nodes* run a nucleus, kernel, and POSE and support user and operating-system applications in multi-tasking mode. POSE may run in supervisor mode, yielding a monolithic kernel organisation, or user mode, yielding a microkernel-like kernel organisation. All of these configurations can coexist in the same system. PEACE can also be layered above an existing operating system (see figure 8) to support parallel processing on workstation clusters.

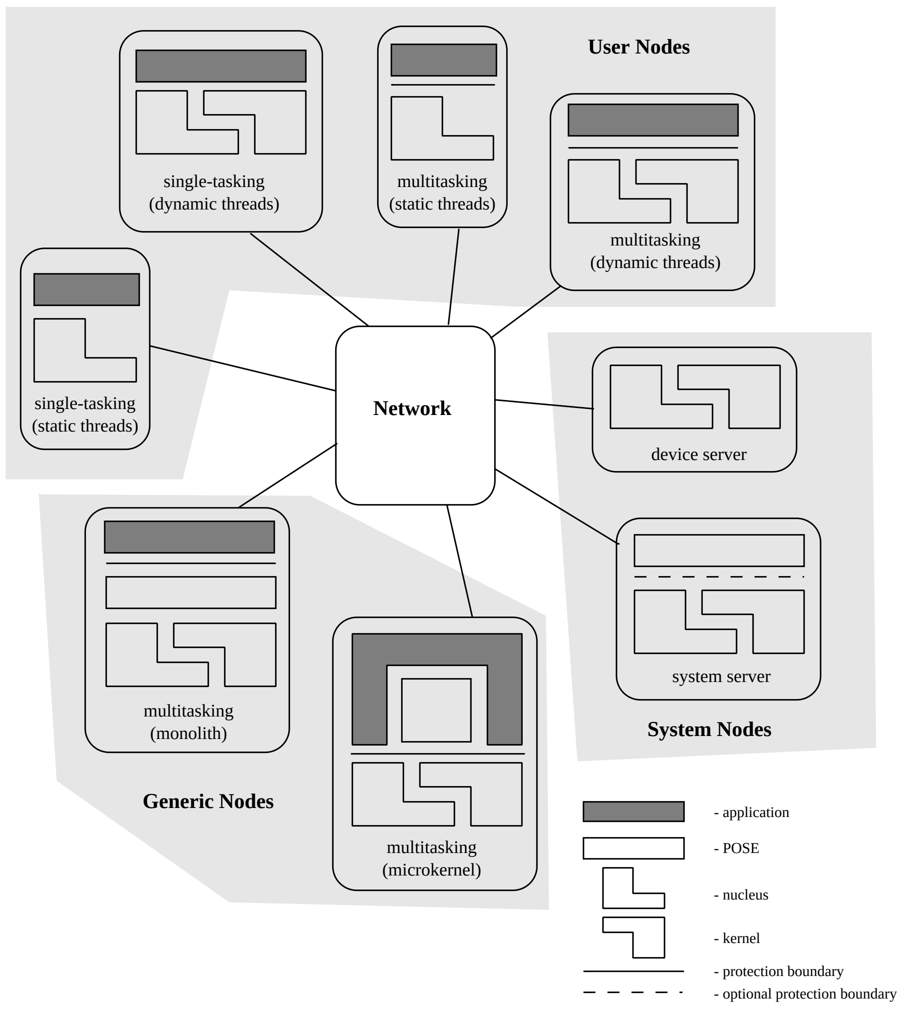
User Nodes
single-tasking (dynamic threads)
multitasking (static threads)
multitasking (dynamic threads)
single-tasking (static threads)
Network
device server
system server
System Nodes
multitasking (monolith)
multitasking (microkernel)
Generic Nodes
- application
- POSE
- nucleus
- kernel
- protection boundary
- optional protection boundary

Figure 7: Native PEACE configurations.

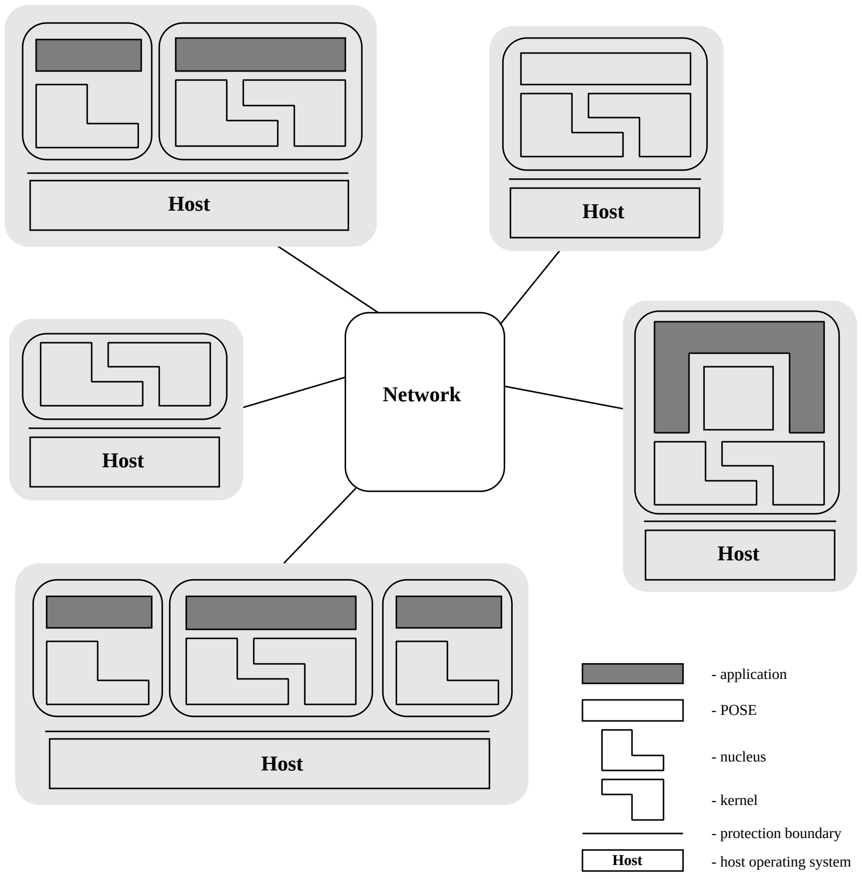

Figure 8: Hosted PEACE configurations.

Using PEACE

The PEACE nucleus implements a software backplane for parallel applications that provides thread processing and inter-process communication. Technically, the nucleus is provided as a number of C++ class hierarchies. The question then arises as to how applications actually use the nucleus. Two different approaches are possible yielding two different views of PEACE.

In the *object-oriented* view, the class hierarchies comprising the nucleus are fully exposed to application programmers allowing them to customise the system to their own needs. Thus a completely application-oriented operating system results. While providing maximum flexibility, this approach has a number of drawbacks. For example, application programmers may be overwhelmed by the complexity of the full class hierarchy. More practically, application programmers are limited to using the system implementation language to develop their applications.

The alternative view is a *class-based* one in which the internals of the system are hidden behind a number of interface classes implementing application-oriented abstractions and supporting different language bindings. Of course, these interface classes are replaceable and may be different for different types of applications. The interface classes are derived from the appropriate system classes by means of inheritance. In this view, the software backplane is seen as providing a number of building blocks – thread location, communication, and scheduling – from which different application-oriented *execution models* may be provided. Each execution model is characterised by a specific process and communication model. In most cases, the interface classes can be implemented very efficiently in a small number of C++ statements. Moreover, it is still possible for application programmers to implement higher-level abstractions starting from those provided by the interface classes, using inheritance or otherwise, in a user-level runtime library.

Incremental Loading and Reconfiguration

Another goal of PEACE was to overcome the performance problem associated with bootstrapping a massively parallel machine consisting of very many nodes. To tackle this problem PEACE supports *incremental bootstrapping* of a parallel machine by allowing individual nodes to be bootstrapped only as required. A similar approach is applied to the bootstrapping of the operating system. In fact, most services provided by PEACE are only loaded on demand. Such *incremental loading* of the operating system results in a completely application-oriented operating system configuration and avoids overhead due to the presence of unnecessary functionality. Furthermore, services that have been loaded previously, but are no longer required, may be discarded.

Kernel and POSE services are provided by active objects that may be installed and discarded dynamically and are hence referred to as *transient objects*. These objects are always accessed via object invocation and incremental loading is triggered by the object invocation system when it discovers (for example, from the communications subsystem) that the entity encapsulating the required object is not currently loaded. Entity loading (including loading of the kernel entity at a node to be bootstrapped) is supported by fundamental POSE services that must be present at some node in the system [60]. Transient objects may be discarded explicitly by their clients or may shut themselves down as appropriate.

Incremental loading can also be applied to a subset of the functions provided by the nucleus if required. Since nucleus functions are provided by passive, as opposed to active, objects, a nucleus that supports such incremental loading must include mechanisms to detect attempts to use functions that are not currently loaded. The responsibility for actually loading the required functions rests with POSE.

Dual Objects

To support the development of parallel (and distributed) applications as well as operating system services, PEACE provides language support for distributed objects based on a novel object model supporting *dual objects* [42], which is reminiscent of the fragmented object model of FOG [30]. A dual object is a distributed object represented by a single *prototype* and one or more *likenesses*. The prototype maintains both the public and private state of the dual object and is typically located at a single node although it may migrate between nodes dynamically. A likeness acts both as a (possibly inconsistent) replica of the public state of the dual object and as a proxy for the prototype. Invocations on a dual object may be performed locally on the likeness (if they only need access to the public state of the object) or remotely on the prototype via the likeness. *Vertical consistency* between likeness and prototype is maintained by (optional) *unification* and *extraction* during invocations. Unification replaces the public state of the prototype with the state maintained by the likeness. Extraction generates a new likeness from the public state of the prototype. Maintaining *horizontal consistency* between likenesses is the responsibility of the application programmer.

Dual objects are supported by *clerks* where a clerk is a thread responsible for creating prototypes of some type and executing remote invocations on them. Clerks may be created at many nodes and requests to create prototypes must be directed to an appropriate clerk and may lead to dynamic loading of the clerk's code. In a typical scenario, a binding between a likeness and a clerk is established when the likeness's constructor is executed and a corresponding prototype is created at the clerk. Future invocations on the likeness result in remote invocation requests for the prototype being executed by the clerk.

A number of different parameter passing modes for references and pointers are supported including standard modes such as call-by-value, call-by-result, and call-by-value-result but not call-by-reference. For references, call-by-value-result is the default. In addition to these standard parameter passing modes, call-by-copy and a variant of Emerald's call-by-move [10], referred to as call-by-vanish, are supported. Call-by-vanish moves the target object but does not maintain local references to it at the source node. Finally, call-by-likeness allows a likeness to be sent in a remote invocation request. Since possession of a likeness allows methods to be invoked on a remote object, this mode subsumes call-by-reference.

Language support for dual objects is provided by the P++ preprocessor for C++. Classes marked as `global` are interpreted as dual object classes and appropriate prototype, likeness, and clerk classes generated. Both the prototype and likeness classes can be individually named and are introduced as formal types. Hence it is possible to derive a dual object class from a likeness class resulting in a truly "distributed" object.

2.4 Spring – An Object-Oriented, Multi-Server, Operating System

Spring [52, 35] is an object-oriented multi-server operating system originally developed at Sun Microsystems Laboratories between 1989 and 1993 with subsequent development being undertaken at SunSoft. At the time of writing Spring runs on Sun SPARCstations 2, 5, 10, and 20.

Spring was conceived as a next-generation operating system designed specifically for a networked world. The design of Spring consequently emphasises support for the development of distributed applications and services, and support for a high degree of location transparency. Another major concern was to provide strong support for security. Finally, the design stresses modularity as a means of improving the flexibility and extensibility of the system, as well as its maintainability and the ease with which new applications and services can be installed.

Goals and Approach

The rationale for the Spring project is based on the premise that, while existing operating systems provide many useful and indeed necessary features, they also lack a number of features that are necessary in a modern operating system. Chief among these are adequate support for distributed applications (including sophisticated distributed system services), strong support for security (especially in the distributed case), and support for multi-threading and multi-processing. Other problems with current operating systems identified by the designers of Spring include the difficulty of delivering, maintaining, and evolving the system. Thus Spring was intended to improve "dramatically" on current systems and, in particular, to make distributed programming "significantly" easier while still supporting existing applications and interworking with existing networked systems [23].

The architecture of Spring is characterised by a strong emphasis on modularity. All the components of the system are defined by tightly-specified fully-abstract interfaces allowing multiple implementations of any component to coexist transparently to its clients. Most components of the operating system are thus regarded as replaceable parts. Interfaces are defined in an interface definition language (IDL) that is both similar to, and a forerunner of that standardised by the Object Management Group (OMG) in its Common Object Request Broker Architecture (CORBA) [31]. Use of an IDL makes interfaces language-independent and hence contributes to the openness of the system. Spring IDL also supports interface inheritance. Hence, at run-time a client expecting to use a component supporting some given interface may actually use a component supporting a derived interface.

In addition to the emphasis on modularity, the designers of Spring adopted object orientation as the basis for providing location transparency. Almost all components of the system are implemented as objects managed by a set of object managers that are themselves objects. Access to all services is via location-transparent secure object invocation. Any given object can be located in the same address space as its clients, in a different address space, or on a remote machine. From a programmer's point of view there is no distinction between local and remote objects. Moreover, there is no distinction between applications and system services. Both are implemented as objects and accessed via object invocation. Thus, the distribution of system services is transparent and, in this sense, Spring is inherently distributed.

The Spring Architecture

The architecture of Spring is fairly conventional in that it is based on the use of an underlying microkernel that provides the low-level support for the other components of the system, which are implemented by object managers running in user mode (see figure 9). Most components are provided as dynamically loadable modules. At boot time the operating system makes the decision as to which components are loaded into which address spaces so that related components can be grouped together [35].

The Spring microkernel consists of two major subcomponents: the *nucleus* and the *virtual memory manager* (VMM). The Spring nucleus provides *domains* (i.e., tasks) and *threads*, and supports secure cross-domain object invocation based on *doors*. A door is essentially a software capability for an entry point to a domain. A door is represented by a program counter value (to which control is transferred by the nucleus during an invocation via the door) and a value specified by the domain to which the door refers (which might be used to identify a particular object within the domain). Doors are unforgeable and are maintained by the nucleus. Domains refer to doors via domain-local door identifiers. Possession of a door confers the right to carry out an invocation via the door. A door can only be obtained with the permission of the domain to which it refers or of a third party that already possesses the door. The target domain can

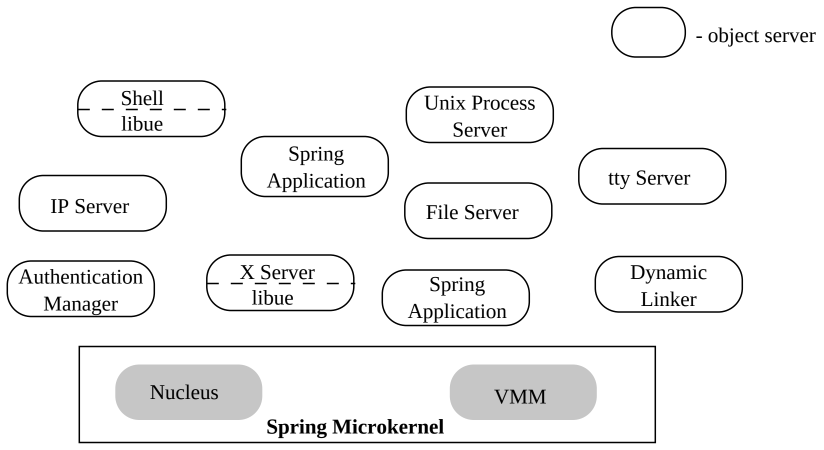

Figure 9: The architecture of Spring.

revoke access to a door at any time. The implementation of doors is optimised for invocations having only a small number of parameters (which can be transferred via registers) while the Spring virtual memory system provides support for the transfer of large volumes of page-aligned data in door invocations. Although users perceive both sides of a cross-domain invocation as being executed by different threads, in fact the nucleus supports hand-off scheduling based on a scheduling abstraction known as a *shuttle*. Invocations destined for remote domains are forwarded via user-mode network proxies that implement different protocol stacks.

The VMM is also fairly conventional supporting demand-paged virtual memory and providing *address spaces* and *memory objects* representing memory that can be mapped into one or more address spaces [44]. The VMM manages local physical memory and page replacement, and caches pages of memory objects implemented by (external) pagers. The VMM allows different memory objects supporting different views of (for example, different access rights for) the same underlying memory to coexist on a node while sharing the same physical memory cache. The design of the VMM distinguishes between the memory object itself and the object on which paging operations relating to the memory object are invoked by the VMM – the *pager object* for the memory object. This allows the external pager to implement the memory object and the pager object in different domains. The VMM's interface to the external pager is provided by a *cache object* for each mapped memory object. The VMM also supports copy-on-write and move operations for page-aligned data transfer. These are, in turn, used in the implementation of door invocation.

Among the various user-mode services provided by Spring are file services, name services and a UNIX emulation service. Spring provides a distributed file service supporting an on-disk UNIX compatible file system. The Spring file service also supports memory-mapped files (by virtue of the fact that *file objects* are derived from memory objects) and a caching layer implements local caching of remote files thereby minimising network access.

Spring supports general-purpose name services allowing symbolic names to be associated with objects of arbitrary types. A name service is build up from a collection of *context objects*. A context stores a collection of name bindings between names and object references analogous to a conventional directory. Since contexts are objects, a name binding for a context can be stored in any other context allowing arbitrary naming graphs to be created. The naming services are also used to support persistence as described below.

Finally, in line with the goal of supporting existing applications, Spring provides a UNIX emulation service implemented by a library and associated server [43]. Although binary compatibility with UNIX was not a goal of Spring, and is not supported, the Spring UNIX emulation supports a large subset of the system calls provided by SunOS 4.1 and runs dynamically-linked SunOS 4.1 executables without modification. The goal of the UNIX subsystem design was to reuse as many of the standard Spring services, such as the file service and naming service, as possible and only provide the additional support necessary to map between the interfaces provided by these services and that of UNIX. Hence, the UNIX emulation library mainly keeps track of the relationship between UNIX file descriptors and the corresponding Spring objects, delivers signals, and supports program startup. The UNIX server, known as the Process Manager, maintains the parent-child relationship between processes, keeps track of process and group identifiers, forwards signals, and implements sockets and pipes. System call redirection is based on replacing `libc` with the Spring UNIX emulation library, `libue`, at `exec` time.

Support for Distributed and Persistent Programming

As described previously, support for distributed programming was at the heart of the design of Spring. The interfaces to all the components of the system are defined in Spring IDL and almost all components are accessed using location-transparent object invocation. Spring supports transparent access to objects whose implementations are local to their clients' domains as well as those implemented by object managers in other, possibly remote, domains. Object invocation is supported by an IDL compiler that generates the client and server stubs used by the object invocation mechanism. At this level the Spring object invocation mechanism is similar to many RPC implementations. However, in Spring the interface-specific stub code generated by the IDL compiler makes use of a so-called *subcontract* to actually carry out an invocation [36]. Subcontracts are (replaceable) modules that implement the basic mechanisms to support object invocation and argument passing depending on the desired semantics of the invocation. For example, different subcontracts might be used to carry out a a simple remote invocation, an invocation on a replicated object, or an invocation on a persistent object. Any set of interface-specific stubs can work with any subcontract and vice versa. Thus, all subcontracts provide a standard interface to client stubs including routines to marshal the current object, unmarshal an object, and carry out an invocation as well as some other miscellaneous routines.

In Spring, objects are always conceived of as existing at their clients; if objects are manager-based then their manager only maintains their underlying state. At the client, an object consists of a method table, a subcontract-operations vector, and a representation. The method table is bound to the code for the object's stubs while the subcontract-operations vector points to the subcontract operations to be used by the stubs. The representation will typically include one of more doors providing access to the underlying state of the object wherever it may be located.

Spring also supports the notion of *compatible* subcontracts since interface-specific unmarshalling code may receive objects that use different subcontracts. In this case, dynamic linking of the correct subcontract code is used to recover and allow the stubs to continue correctly.

The designers of Spring tried to take a very general approach to support for persistence. They identified a number of desirable attributes of such support including provision for allowing objects of any type to be made persistent, for new persistent types to be added to a running system, for persistent objects to be shared between users, for naming and object services to be separated, for persistent and transient name spaces to be supported, and, finally, for objects to contain references to both persistent and transient objects. As a result, they defined a framework allowing an object manager to control how its objects are made persistent based on the concepts of *freezing* (generating a persistent representation of an object known as a *freeze token*) and *pickling* (storing an object's state stably), and the use of a name service to support persistence. When an object is bound to a name in the persistent part of the name space, the name service freezes the object automatically. A name server that supports this feature is known as a persistent name server and the name space that it maintains as a persistent name space. The framework defines the interface between the name service and an arbitrary object manager that allows persistence to be managed securely. However, as described in [56], the framework does not provide support for handling inter-object references in persistent objects nor for garbage collection.

3 Perspectives on Object-Oriented Operating Systems

Choices is probably the best known of the various object-oriented operating systems that have been developed to date and has consequently been highly influential. The Choices project demonstrated that the well-known advantages of object-oriented design and implementation could be obtained in the context of an operating system without loss of performance [59]. The use of an object-oriented design allowed Choices to provide customisability and extensibility as expected. Moreover, the Choices experience has provided valuable guidance to the developers of other object-oriented operating systems concerning the design, organisation, and use of frameworks. On the other hand, Choices is clearly deficient in a number of respects. Notwithstanding the fact that some support for embedded applications was provided [18], Choices essentially provides a family of multi-user/multi-tasking operating systems based on a kernel architecture that is somewhere between a traditional monolithic kernel and a microkernel. Thus, although the design is based on a program family, it can be said that "the minimal subset of system functions" is rather large and hence the resulting potential for customisation rather modest.

Lipto's main contribution is in showing that modularity and protection can be decoupled in the design of an object-oriented operating system. Lipto uses location-transparent object invocation to allow operating system services, which are modelled as collections of objects, to be configured in different ways with respect to protection domains. In Lipto, the kernel/user boundary is configurable and the trade-off between protection and performance that is made when deciding what functionality to put in the kernel and what functionality to put in user space can be made dynamically. On the down side, Lipto includes a fixed mandatory supervisor-mode component that, as far as can be determined, implements multiple address spaces per node. Hence, although Lipto is more configurable, like Choices, it essentially provides a multi-user/multi-tasking operating system.

In contrast to Choices, the PEACE project took a more purely family-based approach to the design of an object-oriented operating system. The result is a system that can be customised to provide instantiations supporting a number of different operating modes ranging from those supporting only a single user and single task per node to those supporting multiple users and multiple tasks per node. The designers of PEACE were motivated primarily by the desire to provide optimal system support for different (sets of) parallel applications running on massively-parallel computers. Like Lipto, PEACE also supports a high degree of system configurability. The use of location-transparent object invocation allows many services to be configured to run either in supervisor mode or user mode transparently to their clients. In addition, operating system services modelled as collections of objects can be loaded on demand as they are invoked.

Spring is reviewed here because it represents the state of the art in the design of general-purpose object-oriented operating systems. Spring represents a synthesis of previous work on object-oriented and microkernel-based operating systems. While the overall architecture of Spring is similar to that of other multi-server microkernel-based operating systems such as CHORUS or Mach-US [61], Spring is most interesting in that it employs a general-purpose object-support infrastructure, which provides location-transparent object invocation, to access (distributed) services. Thanks to its use of location-transparent object invocation, Spring supports the same degree of configurability of service placement as Lipto and PEACE. However, it is based on a traditional multi-tasking microkernel.

References

[1] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic virtual memory management for operating system kernels. In *Proceedings of the* 12th*Symposium on Operating Systems Principles*, pages 123–136. ACM Special Interest Group on Operating Systems, December 1989. Also Technical Report CS/TR-89-18.2, Chorus Systèmes.

[2] ACM Special Interest Group on Operating Systems. *Proceedings of the* 14th*Symposium on Operating Systems Principles*, December 1993. Also Operating Systems Review, 27(5).

[3] ACM Special Interest Group on Operating Systems. *Proceedings of the* 6th*SIGOPS European Workshop*, September 1994.

[4] ACM Special Interest Group on Operating Systems. *Proceedings of the* 15th*Symposium on Operating Systems Principles*, December 1995. Also Operating Systems Review, 29(5).

[5] Thomas E. Anderson. The case for application-specific operating systems. In *Proceedings of the* 3rd *Workshop on Workstation Operating Systems*, pages 92–94. IEEE Computer Society, IEEE Computer Society Press, April 1992.

[6] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the* 13th*Symposium on Operating Systems Principles*, pages 95–109. ACM Special Interest Group on Operating Systems, October 1991. Also Operating Systems Review, 25(5).

[7] Holger Assenmacher, Thomas Breitbach, Peter Buhler, Volker Huebsch, and Reinhard Schwarz. The PANDA system architecture - a pico-kernel approach. In *Proceedings of the*

4th *Workshop on Future Trends of Distributed Computing Systems*, pages 470–476. IEEE Computer Society Press, September 1993.

[8] Arindam Banerji and David L. Cohn. An infrastructure for application-specific customisation. In *Proceedings of the* 6th *SIGOPS European Workshop* [3], pages 154–159.

[9] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the* 15th *Symposium on Operating Systems Principles* [4], pages 267–284. Also Operating Systems Review, 29(5).

[10] Andrew Black, Norman C. Hutchinson, Eric Jul, Henry M. Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[11] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[12] David L. Black, David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Florin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and Mach. In *Proceedings of the* USENIX *Workshop on Microkernels and Other Kernel Architectures* [63], pages 11–30.

[13] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, 1994.

[14] Luis-Felipe Cabrera and Norman C. Hutchinson, editors. *Proceedings of the* 3rd *International Workshop on Object-Orientation in Operating Systems.* IEEE Computer Society, IEEE Computer Society Press, December 1993.

[15] Luis-Felipe Cabrera and Eric Jul, editors. *Proceedings of the* 2nd *International Workshop on Object-Orientation in Operating Systems.* IEEE Computer Society, IEEE Computer Society Press, September 1992.

[16] Luis-Felipe Cabrera, Vincent Russo, and Marc Shapiro, editors. *Proceedings of the* 1st *International Workshop on Object-Orientation in Operating Systems.* IEEE Computer Society, IEEE Computer Society Press, October 1991.

[17] Vinny Cahill. Research projects in object-support, object-oriented, and extensible operating systems. WWW page http://www.dsg.cs.tcd.ie/research/os_relwork.html, November 1995.

[18] Roy H. Campbell, John H. Hine, and Vincent F. Russo. Choices for mission critical computing. Studies in Computer and Communication Systems, chapter 2, pages 11–20. IOS Press, Amsterdam, Netherlands, 1992.

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

[20] Roy H. Campbell and Peter W. Madany. Considerations of persistence and security in choices, an object-oriented operating system. In John Rosenberg and J. Leslie Keedy, editors, *Security and Persistence*, Workshops in Computing, pages 289–300. Springer-Verlag,

May 1990. Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information.

[21] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the* 1^{st}*Symposium on Operating Systems Design and Implementation*, pages 179–193. USENIX Association, November 1994.

[22] Amitabh Dave, Mohlalefi Sefika, and Roy H. Campbell. Proxies, application interfaces, and distributed systems. In Cabrera and Jul [15], pages 212–220.

[23] Thomas W. Doeppner. The Spring operating system: Internals overview. Tutorial presented at 1^{st}Symposium on Operating Systems Design and Implementation, November 1994.

[24] Richard P. Draves. The case for run-time replaceable kernel modules. In *Proceedings of the* 4^{rd} *Workshop on Workstation Operating Systems* [37], pages 160–164.

[25] Peter Druschel. Efficient support for incremental customization of OS services. In Cabrera and Hutchinson [14], pages 186–190.

[26] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Lipto: A dynamically configurable object-oriented kernel. *IEEE Technical Committee on Operating Systems and Application Environments Newsletter*, 5(1):11–16, Spring 1991.

[27] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Service composition in Lipto. In Cabrera et al. [16], pages 108–111.

[28] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proceedings of the* 12^{th}*International Conference on Distributed Computing Systems*, pages 512–520. IEEE Computer Society Press, June 1992.

[29] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the* 15^{th}*Symposium on Operating Systems Principles* [4], pages 251–266. Also Operating Systems Review, 29(5).

[30] Yvon Gourhant and Marc Shapiro. FOG/C++: A fragmented-object generator. In *Proceedings of the* USENIX *C++ Conference* [62], pages 63–74.

[31] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, July 1995. Revision 2.0.

[32] Paulo Guedes and Daniel P. Julin. Object-oriented interfaces in the Mach 3.0 multi-server system. In Cabrera et al. [16], pages 114–117.

[33] A.N. Habermann, Lawrence Flon, and Lee Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, May 1976.

[34] Graham Hamilton, Yousef A. Khalidi, and Michael N. Nelson. Why object oriented operating systems are boring. In Cabrera et al. [16], pages 118–119.

[35] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the 1993 Summer* USENIX *Conference*. USENIX Association, June 1993.

[36] Graham Hamilton, Michael L. Powell, and James G. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the* 14th *Symposium on Operating Systems Principles* [2], pages 69–79. Also Operating Systems Review, 27(5).

[37] IEEE Computer Society. *Proceedings of the* 4rd *Workshop on Workstation Operating Systems*. IEEE Computer Society Press, October 1993.

[38] Gary M. Johnston and Roy H. Campbell. An object-oriented implementation of distributed virtual memory. In *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 39–57. USENIX Association, 1989.

[39] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the* 14th *Symposium on Operating Systems Principles* [2], pages 80–93. Also Operating Systems Review, 27(5).

[40] Jörg Cordsen and Wolfgang Schröder-Preikschat. Object-oriented operating system design and the revival of program families. In Cabrera et al. [16], pages 24–28.

[41] Jörg Cordsen and Wolfgang Schröder-Preikschat. Towards a scalable kernel architecture. In *Proceedings of the Autumn '92 Openforum Technical Conference*, pages 15–33, November 1992.

[42] Jörg Nolte. Language level support for remote object invocation. Technical Report 654, GMD, June 1992.

[43] Yousef A. Khalidi and Michael N. Nelson. An implementation of UNIX on an object-oriented operating system. In *Proceedings of the 1993 Winter* USENIX *Conference*, pages 469–479. USENIX Association, January 1993.

[44] Yousef A. Khalidi and Michael N. Nelson. The Spring virtual memory system. Technical Report SMLI TR-93-9, Sun Microsystems Laboratories, Inc., February 1993.

[45] Gregor Kiczales. Towards a new model of abstraction in software engineering. In Cabrera et al. [16], pages 127–128.

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

[47] Gregor Kiczales and John Lamping. Operating systems: Why object-oriented? In Cabrera and Hutchinson [14], pages 25–30.

[48] Gregor Kiczales, John Lamping, Chris Maeda, David Keppel, and Dylan McNamee. The need for customisable operating systems. In *Proceedings of the* 4rd *Workshop on Workstation Operating Systems* [37], pages 165–169.

[49] Gregor Kiczales, Marvin Theimer, and Brent Welch. A new model of abstraction for operating system design. In Cabrera and Jul [15], pages 346–349.

[50] Rodger Lea, Christian Jacquemot, and Eric Pillevesse. COOL: System support for distributed programming. *Communications of the ACM*, 36(9):37–46, September 1993.

[51] Dylan McNamee and Katherine Armstrong. Extending the Mach external pager interface to support user-level page replacement policies. In *Proceedings of the* USENIX *Mach Workshop*, pages 17–29. USENIX Association, March 1990.

[52] James G. Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia. An overview of the Spring system. In *Proceedings of* 39^{th} *IEEE International Computer Conference*. IEEE Computer Society Press, February 1994.

[53] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[54] David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–137, March 1979.

[55] 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 *Proceedings of the* 15^{th}*Symposium on Operating Systems Principles* [4], pages 314–324. Also Operating Systems Review, 29(5).

[56] Sanjay R. Radia, Peter W. Madany, and Michael L. Powell. Persistence in the Spring system. In Cabrera and Hutchinson [14], pages 12–23.

[57] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. Overview of the Chorus distributed operating system. In *Proceedings of the* USENIX *Workshop on Microkernels and Other Kernel Architectures* [63], pages 39–69.

[58] Vincent F. Russo. Object-oriented operating system design. *IEEE Technical Committee on Operating Systems and Application Environments Newsletter*, 5(1):34–38, Spring 1991.

[59] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: a Case Study. In *Proceedings of the* USENIX *C++ Conference* [62], pages 103–114.

[60] Henning Schmidt. Making PEACE a dynamic alterable system. In *Proceedings of the* 2^{nd} *European Distributed Memory Computing Conference*, volume 487 of *Lecture Notes in Computer Science*, pages 422–431. Springer-Verlag, April 1991.

[61] J. Mark Stevenson and Daniel P. Julin. Mach-US: UNIX on generic OS object servers. In *Proceedings of the* USENIX *Technical Conference*, pages 119–130. USENIX Association, January 1995.

[62] USENIX Association. *Proceedings of the* USENIX *C++ Conference*, April 1990.

[63] USENIX Association. *Proceedings of the* USENIX *Workshop on Microkernels and Other Kernel Architectures*, April 1992.

[64] Wolfgang Schröder-Preikschat. PEACE – a software backplane for parallel computing. *Parallel Computing*, 1993. To appear.

[65] Wolfgang Schröder-Preikschat. *The Logical Design of Parallel Operating Systems.* Prentice Hall, London, 1994.

[66] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In Andreas Paepcke, editor, *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 414–434. ACM Special Interest Group on Programming Languages, ACM Press, October 1992. Also SIGPLAN Notices 27(10), October 1992.

[67] Michael Young, Avadis Tevanian Jr., Richard F. Rashid, David B. Golub, Jeffrey Eppinger, Jonathan J. Chew, William J. Bolosky, David L. Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the* 11^{th}*Symposium on Operating Systems Principles*, pages 63–76. ACM Special Interest Group on Operating Systems, November 1987. Also Technical Report CMU-CS-87-155.