The Flux OSKit: A Substrate for Kernel and Language Research

Bryan Ford Godmar Back Greg Benson Jay Lepreau Albert Lin Olin Shivers

University of Utah University of California, Davis Massachusetts Institute of Technology

Abstract

Implementing new operating systems is tedious, costly, and often impractical except for large projects. The Flux OSKit addresses this problem in a novel way by providing clean, well-documented OS components designed to be reused in a wide variety of *other* environments, rather than defining a new OS structure. The OSKit uses unconventional techniques to maximize its usefulness, such as intentionally exposing implementation details and platform-specific facilities. Further, the OSKit demonstrates a technique that allows unmodified code from existing mature operating systems to be incorporated quickly and updated regularly, by wrapping it with a small amount of carefully designed "glue" code to isolate its dependencies and export well-defined interfaces. The OSKit uses this technique to incorporate over 230,000 lines of stable code including device drivers, file systems, and network protocols. Our experience demonstrates that this approach to component software structure and reuse has a surprisingly large impact in the OS implementation domain. Four real-world examples show how the OSKit is catalyzing research and development in operating systems and programming languages.

1 Introduction

As operating system functionality continues to expand and diversify, it is increasingly impractical for a small group to implement even a basic useful OS core—e.g., the functionality traditionally found in the Unix kernel—entirely from scratch. Furthermore, generally only a few specific areas in an OS core are interesting for research purposes. For example, any realistic OS, in order to be useful even for research, must include many largely uninteresting elements such as boot loader code, kernel startup code, various device drivers, kernel `printf` and `malloc` code, and a kernel debugger. The necessity of writing this kind of infrastructure not only slows down larger OS research projects, but also greatly increases the cost of entry into OS research so that many small but useful experiments are simply not viable.

While it is possible to adapt existing systems, they are generally complicated and entwined with interdependencies. The OSKit, developed by the Flux research group at the University of Utah, addresses this problem by providing a framework and a set of modularized library code with straightforward and well-documented interfaces for the construction of operating system kernels, servers, and other core OS functionality. The OSKit provides functionality such as simple bootstrapping, a minimal POSIX environment usable in kernels, memory management suited for physical memory and its constraints, extensive debugging support, and higher-level subsystems such as protocol stacks and file systems. The OSKit gives a developer an immediate starting point for investigating "real" OS issues such as scheduling, virtual memory, IPC, file systems, or security. Developers can easily replace generic OSKit modules or functions with their own, guided by research interests or performance considerations. The OSKit can be used to bootstrap unconventional operating systems quickly, such as those for embedded systems and network computers.

The OSKit is heavily used in at least three OS kernels under ongoing development at different institutions. Our own microkernel-based OS, Fluke [17], puts almost all of the OSKit to use. Over half of the Fluke kernel is OSKit code, and many of the servers and user-level utilities that run on top of this kernel also rely heavily on parts of the OSKit. The OSKit has also enhanced and accelerated our OS research by allowing us to quickly create several prototype kernels in order to explore ideas before investing the effort necessary to incorporate these ideas into the much larger primary development system.

Research groups at MIT and U.C. Davis, represented by co-authors of this paper, have recently adopted the OSKit for systems-level language research. Traditional kernels distance the language from the hardware; even microkernels and other extensible kernels enforce some default policy which often conflicts with a particular language's semantics. The OSKit provides a valuable tool to allow advanced languages to be evaluated and experimented with at a low level, to explore novel OS structures enabled by such languages, and to make it possible to obtain accurate performance measurements without the interference of a full-scale OS. By implementing Standard ML [26] directly on the hardware using the OSKit, we are able to model hardware resources with the constructs of a functional programming language. Our port of SR ("Synchronizing Resources") [3], a parallel language intended for systems-level programming but never extensively used for this purpose, allows us to investigate the effectiveness of using a communication-oriented language for implementing OS functionality. Finally, using a Java [19] virtual machine running on the OSKit, we have prototyped a small network computer supporting a Java-based web server and other applications, as well as an active network router that dynamically executes Java bytecode embedded in network packets.

The rest of this paper describes the OSKit and reports on our experience using it for research in operating systems and advanced language systems. Section 2 discusses related work. Section 3 outlines the major OSKit components, and Section 4 details the OSKit's design and implementation. Section 5 describes two example network-oriented OSKit-based applications. Section 6 presents our experience with the OSKit through several case studies. Finally, in sections 7 and 8 we present status, future work, and conclusions.

2 Related Work

Many OS research projects have taken code from other existing, stable systems to reduce the startup cost of OS research: Mach [1] used device drivers from BSD and hardware vendors, the x86 port of SPIN[10] uses device drivers from FreeBSD, and VINO [30] takes its device drivers, bootstrap code, and low-level support for virtual memory from NetBSD. Although this approach certainly saves

Ford, Back, and Lepreau are at the Univ. of Utah (baford,gback,lepreau-@cs.utah.edu), Benson is at U.C. Davis (gbenson@cs.ucdavis.edu), and Shivers and Lin are at MIT (shivers,lin@ai.mit.edu).

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Dept. of the Army under contract number DABT63–94–C–0058 and Rome Laboratory under grant F30602–96–1–0343, and by the National Science Foundation under grant CCR–9633438.

To appear in the Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997, Saint-Malo, France.

time, the developer must still manually take apart the old OS, figure out all the relevant inter-module dependencies and other requirements, and find a way either to emulate these requirements in the new OS environment or change the code appropriately to adapt it to the new environment. The OSKit allows the developer to save more time by providing common components in a convenient form, already separated out and documented.

Recent research projects such as the exokernel [14], SPIN, and VINO focus on creating *extensible systems* which allow applications to modify the behavior of the core OS to suit their particular needs. However, these systems still define a particular, fixed set of "core" functionality and a set of policies by which the core can be used and extended. As the exokernel authors state, "mechanism *is* policy, albeit with one less layer of indirection" [14]. The OSKit, in contrast, makes no attempt to be a useful OS *in itself* and does not define any particular set of "core" functionality, but merely provides a suite of components from which real OS's can be built.

Many real-time, embedded operating systems, such as QNX [20] and VxWorks [35], are designed as a set of modular components that can be statically or dynamically linked with a small core kernel in various configurations. These systems apparently do provide a relatively hospitable kernel environment for a single POSIX-based application, such as a Java virtual machine, and indeed industry has recently constructed a Java-based system using VxWorks. However, the main purpose of these embedded systems' modularity is to allow them to be used in very small, tightly-constrained hardware environments as well as (or instead of) in fully-equipped workstations and PCs. All of the optional components still rely on the basic OS environment provided by the core kernel, and are neither designed with the intention of making them usable in other environments nor sufficiently documented to make it practical for users of these systems to do so. For example, the VxWorks execution environment always runs a special "exception thread" (in VxWorks terminology, a "task"), whose purpose is to field messages from distressed threads, providing a safe execution environment in which to execute cleanup code. Other VxWorks components require this thread's presence, so cannot easily be used in other environments.

Several *object-oriented operating systems* have been created, such as Choices [12], which provides a full operating system, and the Taligent [27] system, which provides OS services above an underlying microkernel. Like the OSKit, these systems attempt to provide an extensible OS structure built from customizable, replaceable components. However, these are still *operating systems* in and of themselves: they still define a basic core OS structure and a framework within which OS components are to be extended, and make no attempt to allow their components to be easily separated out and used in other widely different OS environments. Thus, these object-oriented operating systems are comparable in their basic design goals to the extensible and scalable systems described above.

3 Overview of OSKit Components

In this section we provide a brief overview of several of the most important OSKit components. For reference, Figure 1 illustrates the overall structure of the OSKit and the relationships between its components.

3.1 Bootstrapping

Most operating systems come with their own boot loading mechanisms which are largely incompatible with those used by other systems. This diversity of existing mechanisms is caused not so much by any fundamental difference in the bootstrap services required by each OS, but instead merely by the ad hoc way in which boot loaders are typically constructed. Because boot loaders are basically uninteresting from a research standpoint, OS developers generally just produce a minimal quick-and-dirty design, which results in each boot loader being unsuitable for the *next* OS due to slight differences in design philosophy or requirements. To solve this problem, the OSKit directly supports the *MultiBoot standard* [16] which was cooperatively designed by members of several OS projects to provide a simple but general interface between boot loaders and OS kernels, allowing any compliant boot loader to load any compliant OS. Using the OSKit, it is easy to create OS kernels that support a variety of existing boot loaders that adhere to the MultiBoot standard. In addition, the OSKit includes tools that allow these MultiBoot kernels to be loaded from older BSD and Linux boot loaders, and from MS-DOS.

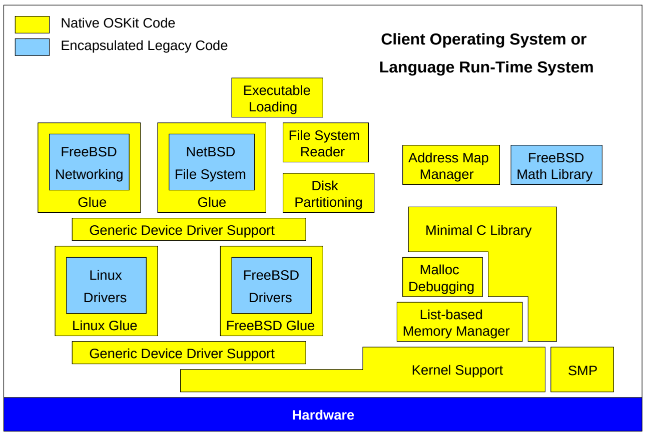

Figure 1: The structure of the OSKit. The shaded components indicate off-the-shelf code that is encapsulated in the OSKit using glue code. This figure shows only the approximate relationships between components; the sizes of the regions do not correspond to the sizes of the components.

A key feature of the MultiBoot standard that makes it highly useful to research systems is the ability of the boot loader to load additional files, or *boot modules*, at boot time along with the kernel itself. A boot module is simply an arbitrary "flat" file, which the boot loader does not interpret in any way, but instead merely loads into chunks of reserved physical memory along with the kernel image itself. Upon starting the kernel, the boot loader then provides the kernel with a list of the physical addresses and sizes of all the boot modules that were loaded, along with an arbitrary user-defined string associated with each boot module. These boot modules and their associated user-defined strings can then be interpreted by the kernel however it sees fit; typically their purpose is to ease the kernel's bootstrapping burden by providing arbitrary data that the kernel might need to get started, such as initialization programs, device drivers, and file system servers.

3.2 Kernel Support Library

The primary purpose of the OSKit's kernel support library is to provide easy access to the raw hardware facilities without adding overhead or obscuring the underlying abstractions. It contains a large collection of useful functions and symbol definitions that are highly specific to supervisor-mode code. In contrast, most of the other libraries in the OSKit are often useful in user-mode code as well, even though they are designed primarily with kernels in mind. Also unlike the rest of the OSKit, much of the kernel support code is necessarily architecture-specific; no attempt has been made to hide machine-specific details that might be useful to the client OS. For instance, on the x86, the kernel support library includes functions to create and manipulate x86 page tables and segment registers. Other OSKit components can, and often do, provide higher-level architecture-neutral facilities built on these low-level mechanisms, but the architecture-specific interfaces always remain accessible in order to provide maximum flexibility.

The OSKit's kernel support library is especially important on the x86 architecture, whose OS-level programming environment is particularly complex and obscure. The kernel support library takes care of setting up a basic 32-bit execution environment (x86 processors normally start up in a 16-bit mode for compatibility with MS-DOS), initializing segmentation and page translation tables, installing an interrupt vector table, and providing default trap and interrupt handlers. Naturally, the client OS can modify or override any of this behavior; however, by default, the kernel support library automatically does everything necessary to get the processor into a convenient execution environment in which interrupts, traps, debugging, and other standard facilities work as expected. The library also by default automatically locates all of the boot modules loaded with the kernel and reserves the physical memory in which they are located so that the application can easily make use of them later on. The client OS need only provide a `main` function in the standard C style; after everything is set up, the kernel support library will call it with any arguments and environment variables passed by the boot loader. Thus, using the OSKit, a "Hello World" kernel is as simple as an ordinary "Hello World" application in C.

3.3 Memory Management Library

Memory management code typically used in user space, such as the `malloc` implementation in a standard C library, is not generally suitable for kernels because of the special requirements of the hardware on which they run. Device drivers often need to allocate memory of specific "types" and with specific alignment properties: e.g., only the first 16MB of physical memory on PCs is accessible to the built-in DMA controller. To address these memory management issues, the OSKit includes a pair of simple but flexible memory management libraries. The *list-based memory manager*, or LMM, provides powerful and efficient primitives for managing allocation of either physical or virtual memory, in kernel or user-level code, and includes support for managing multiple "types" of memory in a pool, and for allocations with various type, size, and alignment constraints. The *address map manager*, or AMM, is designed to manage address spaces that don't necessarily map directly to physical or virtual memory; it provides similar support for other aspects of OS implementation such as the management of processes' address spaces, paging partitions, free block maps, or IPC namespaces. Although these libraries can easily be used in user space, they are designed specifically to satisfy the needs of OS kernels.

3.4 Minimal C Library

Mature OS kernels typically contain a considerable amount of code that simply reimplements basic C library functionality such as `printf` and `malloc`. This is done because the "real" C library implementations of such functions are optimized for maximum performance and functionality in the rich user-space environment provided by a full-function OS, and therefore make too many assumptions to be usable in a kernel environment. For example, a standard `printf` usually relies on the full `stdio` package, which among other complexities manages the mapping of file handles to file descriptors and dynamically allocates buffer memory.

By contrast, the OSKit provides a minimal C library designed around the principle of minimizing dependencies rather than maximizing functionality and performance. For example, locales and floating-point are not supported, and the standard I/O calls don't do any buffering, instead relying directly on underlying `read` and `write` operations. Dependencies between C library functions are minimized, and those dependencies that do exist are documented so that individual functions can be replaced as necessary in order to adapt the minimal C library to arbitrary environments.

3.5 Debugging Support

One of the OSKit's most important practical benefits is that, given an appropriate hardware setup, it immediately provides the OS developer with a full source-level kernel debugging environment. The OSKit's kernel support library includes a serial-line stub for the GNU debugger, GDB [32]. The stub is a small module that handles traps in the client OS environment and communicates over a serial line with GDB running on another machine, using GDB's standard remote debugging protocol. The OSKit's GDB stub can be used even if the client OS performs its own trap handling, and even supports multithreaded debugging if the client OS provides appropriate hooks. In the future, we plan to integrate a local debugger into the OSKit as well, which can be used when a separate machine running GDB is not available.

In addition to the basic debugging support, the OSKit also provides a memory allocation debugging library, which tracks memory allocations and detects common errors such as buffer overruns and freeing already-freed memory. This library provides similar functionality to many popular application debugging utilties, except that it runs in the minimal kernel environment provided by the OSKit.

3.6 Device Driver Support

One of the most expensive tasks in OS development and maintenance is supporting the wide variety of available I/O hardware. Devices are tricky and often have undocumented glitches and new hardware is constantly being released with incompatible software interfaces. For these reasons, the OSKit leverages the extensive set of stable, well-tested drivers developed for existing kernels such as Linux and BSD. To avoid divergence from these existing source bases and allow new and improved drivers to be easily integrated into the OSKit in the future, existing driver code is incorporated into the OSKit largely unmodified using an *encapsulation* technique described later in Section 4.7. These existing drivers are surrounded by a thin layer of OSKit glue code which allows them to be used in environments completely different from those for which the drivers were originally written. Currently, most of the Ethernet, SCSI, and IDE disk device drivers from Linux 2.0.29 are included—over fifty in all—as well as eight character device drivers imported from FreeBSD in the same way, supporting the standard PC console and serial port and various multi-serial port boards. Because of the OSKit's careful packaging of these drivers, the FreeBSD drivers work alongside the Linux drivers without a problem. In the future we expect to incorporate drivers from other sources as well, possibly even from popular commercial operating systems for which hardware vendor-supplied drivers are often available.

3.7 Protocol Stacks

The OSKit provides a full TCP/IP network protocol stack; like the device drivers, the networking code is incorporated by encapsulation so that it can easily be kept up-to-date. However, whereas the OSKit currently takes its network device drivers from Linux, which is the largest source of freely available drivers for the PC platform, the OSKit's network components are instead drawn from the 4.4BSD-derived FreeBSD [24] system, which is generally considered to have much more mature network protocols. This demonstrates a secondary advantage of using encapsulation to package existing software into flexible components: with this approach, it is possible to pick the best components from different sources and use them together—in this case, Linux network drivers with BSD networking.

3.8 File Systems

To complete our picture, the OSKit incorporates standard disk-based file system code, again using encapsulation, this time based on NetBSD's file systems. NetBSD was chosen in this case as the primary source base because its file system code is the most cleanly separated of the available systems; FreeBSD and Linux file systems are more tightly coupled with their virtual memory systems. We are currently incorporating Linux file systems as well, to support many diverse file system formats, such as those of Windows 95, OS/2, and System V.

Our development of a highly secure file server using the OSKit's file system provided an interesting experience with the use of such a component. The OSKit file system's exported COM interfaces are similar to the internal VFS interface [23] used by many Unix file systems. These interfaces are of sufficiently fine granularity that we were able to leave untouched the internals of the OSKit file system. For example, the OSKit interface accepts only single pathname components, allowing the security wrapping code to do appropriate permission checking. The fileserver itself, however, exports an interface accepting full pathnames, providing efficiency where it matters, between processes. Avoiding any modification of the main file system code greatly reduces our maintenance burden, allowing us easily to track NetBSD releases.

4 OSKit Design and Implementation

In order to make the OSKit flexible enough to be used in a wide variety of diverse environments, it was necessary to adopt a different set of design rules than would normally be used for building kernels themselves. Often this involves applying well-known and accepted software engineering principles in unconventional ways. This section describes the OSKit's design and implementation philosophy and rationale, and provides specific examples of how they are applied.

4.1 Library Structure

The most important goal of the OSKit is to be as convenient as possible for the developer to use. Although this goal has many ramifications throughout the OSKit's design, its first manifestation is in the basic layout and usage pattern of the OSKit as a whole. The OSKit is structured as a package that can be automatically built and installed, in most cases, as easily as an ordinary GNU-style application. It is *self-sufficient* in that it does not use or depend on any existing libraries or header files installed on the system; the only things the user must provide are the compiler, linker, and a few other development tools. Building and installing the OSKit causes a set of libraries to be created in a user-defined location (e.g., `/usr/local/lib` and `/usr/local/include`) from which they can then be linked into operating systems just like ordinary libraries are linked into user-level applications.

The OSKit is structured this way because developers are already familiar with libraries and know how to use them; although it is not common practice to link libraries into a kernel, this is simply because until now few libraries have been *designed* to be usable in kernels. Given a set of libraries designed for this purpose, it is much easier for a developer to link in a library and use it than to drop in a set of `.c` files, figure out what compiler options to compile them with, what header files they need, etc. Developers can define their own source tree layout and build environment rather than being required to integrate their sources into a predefined existing structure.

4.2 Modularity Versus Separability

While modularity is a standard software design goal, in the OSKit it gains a new level of importance. A primary goal of the OSKit is to allow developers to use arbitrary components in a given situation without being forced to use other parts of the OSKit; this means that the OSKit's components must not only be *modular*, but also fully *separable*. For example, the client should be able to use the OSKit's device drivers without also having to use the OSKit's memory manager, even though device drivers necessarily require some kind of memory allocation service. A traditional kernel such as BSD may be extremely clean and modular, but is still not very separable because of extensive inter-module dependencies which make it difficult, for instance, to use BSD's device drivers without BSD's memory allocator, or BSD's file system code without BSD's VFS layer.

To provide full separability between components in the OSKit, it is often necessary to introduce thin intermediate "glue" layers to provide a level of indirection between a component and the services it requires. In many cases these layers take the form of library functions with trivial default implementations, whose sole purpose is to be overridden by the client OS when the need arises. In other cases, the layer of indirection is provided through the use of function pointers and dispatch tables which allow components to be dynamically bound together by the client OS at run time. The former method is generally used for services for which there is generally only one implementation in the system, such as `putchar` and `malloc`, whereas the latter method is used when multiple implementations of a service must coexist, such as the block I/O interfaces to different disk device drivers.

4.2.1 Separability Through Overridable Functions

As an example of the first method, to allocate memory, all device driver components in the OSKit invoke a client-supplied function called `fdev_mem_alloc`. A default implementation of this function is provided which uses the OSKit's memory management library, but this default can easily be overridden by the client OS if it uses its own method of managing physical memory. This way, in simple situations where the client just uses the OSKit's defaults, everything "just works" without any special action on the client's part; however, the client OS can obtain full control over memory allocation and other services when needed.

4.2.2 Separability Through Dynamic Binding

As an example of the second method of ensuring separability, none of the OSKit's file system components have any link-time dependencies on the OSKit's device driver components, even though the file systems must invoke the block device drivers in order to access the underlying disk on which the file system resides. Instead, when the client OS initializes an OSKit device driver, the device driver returns a pointer to an *interface* to use to access the device. OSKit interfaces will be described in more detail later, but essentially are just opaque objects with dynamic dispatch tables similar to C++ virtual function tables. Later, when initializing the OSKit's file system component, the client OS passes the device driver's interface pointer and the file system henceforth uses that interface to invoke the driver's services. In this way, the client OS can bind at run time any file system to any device driver, and neither component needs to know how it is being used.

4.3 Component Granularity

The OSKit's libraries each contain a number of logical components; the client OS incorporates these components by referencing symbols defined by the libraries, and the linker determines which specific object files to pull in. However, beyond this principle there is no single, standard definition of exactly what a "component" is or how it is used. By not attempting to force all components into a single fixed design methodology, the OSKit gains a degree of flexibility that we have found to be essential to its success. In particular, the most natural size and granularity for components vary widely in different parts of the OSKit, from tiny single-function "components" such as `strcpy` to large components each consisting of many modules such as the BSD file system. To cope with such large variations in granularity, there must be some corresponding variety in implementation and usage patterns. The OSKit's libraries are roughly divided into two main categories: *function libraries* and *component libraries*.

4.3.1 Function Libraries

The OSKit's function libraries provide relatively simple, low-level services in a traditional C-language function-oriented style. They are designed for fine-grained use and control, generally on a function-by-function basis, allowing the client OS to use particular library functions while leaving out or individually overriding other functions. The dependencies between library functions are minimized, as are dependencies on other libraries; where these dependencies inevitably exist, they are well-defined and explicitly exposed to the OS developer.

For instance, the OSKit's minimal C library provides an implementation of the well-known `printf` function as well as other standard I/O services; however, these services are designed very differently from those of traditional C libraries. A standard I/O module traditionally acts as one big "black box" which implements a broad set of high-level services on top of a corresponding set of low-level services (`read`, `write`, etc.), and maintains private state to implement buffering and other common features. Making use of a single standard I/O function in an application pulls in various internal support routines, and with them many dependencies on other facilities such as memory allocation, terminal control, etc. The standard I/O services in the OSKit's minimal C library, on the other hand, minimize dependencies and internal state (e.g., they perform no buffering), and their implementations are documented so that the client OS can exercise full control over them. For instance, the OSKit's default `printf` function is implemented in terms of two other functions, `puts` and `putchar`; the default `puts`, in turn, is implemented only in terms of `putchar`. While this implementation would be a bug in a standard C library, in which overriding one function is not supposed to affect the behavior of another, in the OSKit's minimal C library it is extremely useful because it allows the client OS to obtain basic formatted console output simply by providing a `putchar` function and nothing else.

4.3.2 Component Libraries

Whereas the function libraries are designed for maximum fine-grained flexibility and controllability, the component libraries are designed to provide large chunks of functionality in one shot, as quickly and conveniently as possible. They adopt a more coarse-grained, object-oriented "black box" design philosophy with relatively fewer public entrypoints. For example, in the OSKit's device driver component libraries, each device driver is represented by a single function entrypoint which is used to initialize and register the entire driver. Most of the internal details of the driver and the hardware it controls are hidden from the client OS, which generally interacts with these components only through common, well-defined, object-oriented interfaces, giving the OS developer "plug and play" control over the overall system structure. This design increases large-scale flexibility at the expense of fine-grained controllability: by using a particular OSKit device driver, the OS developer gives up direct control over the piece of hardware the driver is controlling, but gains the ability to drop a different driver in its place later on without changing anything else.

There is no clear-cut criterion defining the appropriate granularity for particular components; in fact, for some services it may be desirable to have alternative libraries available implemented at different granularities. For example, while the OSKit's minimal C library serves the needs of kernels and simple applications well by emphasizing simplicity and flexibility over functionality, in the future we may integrate a more traditional C library, such as the BSD C library, as an alternative OSKit component, to be used in situations where more complete functionality but less fine-grained control is needed by the application.

4.4 COM Interfaces

For usability, it is critical that OSKit components have clean, well-defined interfaces, designed along some coherent set of global conventions and principles. To provide this standardization, we adopted a subset of the Component Object Model [25] as a framework in which to define the OSKit's component interfaces. At its lowest level, COM is merely a language-independent protocol allowing software components within an address space to rendezvous and interact with each other efficiently, while retaining sufficient separation so that they can be developed and evolved independently. Besides the obvious advantages of making the OSKit's interfaces more consistent with each other and with those widely used in component-oriented applications, COM also brings several technical advantages described below.

4.4.1 Implementation Hiding

COM is founded on the notion of *interfaces*, which are comparable to Java [19] interfaces: they define a set of methods that can be invoked on an object without providing any direct access to the object's internal state. COM interfaces are defined independently of the components that implement them, ensuring that implementation and interface remain well-separated; in practice many different implementations of a particular COM interface often coexist even within a single system. As represented in the C language, a COM interface is an opaque structure whose actual size and content is unknown to the client, except that its first member is a pointer to a table of function pointers, similar to a C++ virtual function table. For example, Figure 2 shows a slightly simplified but essentially complete definition of the OSKit's `blkio` interface, which is implemented by each of the OSKit's disk device drivers as well as by other components.

```
typedef struct blkio {
  struct blkio_ops *ops;
} blkio_t;

struct blkio_ops {
  error_t  (*query)(blkio_t *io,
                    const struct guid *iid,
                    void **out_ihandle);
  unsigned (*addref)(blkio_t *io);
  unsigned (*release)(blkio_t *io);
  unsigned (*getblocksize)(blkio_t *io);
  error_t  (*read)(blkio_t *io, void *buf,
                   off_t offset, size_t amount,
                   size_t *out_actual);
  error_t  (*write)(blkio_t *io, const void *buf,
                    off_t offset, size_t amount,
                    size_t *out_actual);
  error_t  (*getsize)(blkio_t *io, off_t *out_size);
  error_t  (*setsize)(blkio_t *io, off_t new_size);
};

/* Friendly macros */
#define oskit_blkio_read(io, buf, ofs, amount, out_actual) \
                        ((io)->ops->read((io),
                        (buf), (ofs), (amount), (out_actual)))
....

#define BLKIO_IID GUID(0x4aa7df81, 0x7c74, 0x11cf, \
    0xb5, 0x00, 0x08, 0x00, 0x09, 0x53, 0xad, 0xc2)
```

Figure 2: The OSKit's COM Interface for Block I/O.

The `blkio_ops` structure is the dynamic dispatch table for this interface, representing the methods that can be called. The last two lines define the Globally Unique Identifier (GUID) identifying the `blkio` interface.

4.4.2 Interface Extension and Evolution

An object can export any number of COM interfaces; each interface represents one particular "view" of the object with its own independent function table through which methods can be invoked. Interfaces are identified by algorithmically generated DCE Universally Unique Identifiers (UUIDs), so new COM interfaces can be defined independently by anyone with essentially no chance of accidental collisions. Given a pointer to any COM interface, the object can be dynamically queried for pointers to its other interfaces, providing what is known in many languages as "safe downcasting" or "narrowing." This mechanism allows objects to implement new or extended versions of existing interfaces while retaining compatibility with clients that only understand the original interface, and it allows clients to probe an object safely and take advantage of extended interfaces if available while falling back on the base interface if not. For example, the OSKit's `bufio` interface is an extension to the `blkio` interface in Figure 2, which adds methods to allow direct pointer-based access to the data stored in the object in the common case in which this data happens to be in local memory. The OSKit's raw, unbuffered disk device drivers only provide the basic `blkio` interface, since a read or write to the object translates to a disk read or write; however, an object representing a buffered disk device or a RAM disk could also support the extended `bufio` interface to provide more efficient access to clients that can take advantage of the extended interface.

4.4.3 No Required Support Code

Finally, one of the abstraction features in our use of COM that is most important for the purposes of the OSKit is that interfaces can be completely "standalone" and do not require any common infrastructure or support code that the client OS must use in order to make use of the interfaces. Contrast this, for example, with the BSD, Linux, and *x*-kernel network stacks, in which the protocols themselves are modular and interchangeable to some degree, but each of their interfaces depends on a particular buffer management abstraction with a particular concrete implementation (`mbufs`, `skbuffs`, and `Msgs`, respectively). In order to use any BSD networking code, one must also incorporate and "design around" the BSD `mbuf` code; it would be nontrivial at best to replace it with an alternative buffer management implementation that differs in more than minor details. The OSKit's corresponding interfaces, on the other hand, are purely behavioral contracts between modules that rely on no particular common implementation infrastructure.

4.5 Execution Environment

To achieve full OSKit component separability, it is necessary to define and document not only the interface implemented by a component, but also all of the interfaces the component itself uses and the execution environment on which it depends: in other words, each component must be described not only "in front" but "all around." For function libraries such as the minimal C library, this is mostly a matter of documenting each function's behavior and dependencies: for instance, the documented "environment" of the `printf` function consists of the `puts` and `putchar` functions on which it is based.

For larger components such as device drivers, however, issues such as concurrency and synchronization are important, and the reentrancy and interruptibility requirements of each component must be defined carefully. Naturally, the complexity of the execution environment required by a component varies depending on the size and complexity of the component itself in addition to other factors; however, in all cases the OSKit's design attempts to minimize the complexity of this expected execution environment. For instance, the OSKit does not require the OS to provide a notion of "interrupt priority levels" as is used in BSD, even though the OSKit incorporates BSD file system and networking code and can be made to use multiple IPLs if desired. The OSKit documentation specifies several basic execution models of varying complexity, ranging from an extremely simple concurrency model in which the component makes almost no assumptions about its environment, to the most complex model in which components must be aware of and have some control over various concurrency issues such as blocking, preemption, and interrupts. All of the OSKit's components conform to one of these documented execution models. Further, since using the OSKit in a given environment will invariably involve some adaptation to local requirements, we have also included in the documentation a number of "recipes" for using OSKit components in various common environments, such as preemptive, multiprocessor, or interrupt-model kernels.

4.6 Exposing the Implementation

Whereas hiding the implementation of a module is generally considered good software design practice, we take an approach in line with Kiczales' "Open Implementation" philosophy [22]. The OSKit often explicitly exposes the implementation of a component as part of its documented interface, in order to provide maximum power and flexibility to the client OS. For instance, the OSKit's basic memory management library exposes a number of functions that are fairly specific to its particular implementation, such as the ability to reserve particular regions of physical memory or walk through and examine the free list. The client OS is not obligated to use these low-level interfaces and in most cases can stick to the standard `malloc`-like interface, but the availability of the low-level interfaces is often important in meeting the needs of particular kernels.

The OSKit employs an open implementation philosophy even for the more coarse-grained component libraries in which it is usually desirable to hide most implementation details. However, in this case, the key point is that implementation details are hidden unless explicitly requested; they are not forced onto the client. For example, all of the OSKit's device drivers, whether derived from BSD or Linux, export a common set of basic interfaces which hide the nature and origin of each individual driver; however, each device driver can also export additional interfaces providing extended, driver-specific functionality. In fact, the COM interface extension mechanism (Section 4.4.2) provides an ideal basis for open implementation in the OSKit.

4.7 Encapsulation of Legacy Code

Much of the code in the OSKit is derived directly or indirectly from existing systems such as BSD, Linux, and Mach. For small pieces of code that aren't expected to change much in the original source base, or are expected to diverge widely from the original base anyway, we simply assimilated the code into the OSKit's source tree, modifying it as necessary, and maintaining it as part of the OSKit from then on. However, for large or rapidly-changing bodies of code borrowed from existing systems, such as device drivers, file systems, and network protocol stacks, we instead took the approach of cleanly *encapsulating* the code within its new environment. This approach generalizes the technique explored by Goel at Columbia and Utah, in which Linux device drivers were used unchanged in the Mach 3.0 kernel [18]. The OSKit defines a set of COM interfaces by which the client OS invokes OSKit services; the OSKit components implement these services in a thin layer of *glue* code, which in turn relies on a much larger mass of *encapsulated* code, imported directly from the donor OS largely or entirely unmodified. The glue code translates calls on the public OSKit interfaces such as the `blkio` interface into calls to the imported code's internal interfaces, and in turn translates calls made by the imported code for low-level services such as memory allocation and interrupt management into calls to the OSKit's equivalent public interfaces. Although sometimes tricky to implement, this design requires virtually no modifications to the encapsulated code itself, vastly simplifying the task of keeping the code up-to-date with new versions of the donor OS. Of course, the glue code still has to be updated to deal with major changes in the native environment being emulated, but this is much simpler than updating all the imported code manually, and occurs much less frequently. For example, the OSKit's Linux driver set has already tracked the Linux kernel through several versions, starting with Linux 1.3.68; the encapsulation technique has made these upgrades relatively straightforward, and they continue to become easier as the emulation mechanisms are refined. The following sections describe some of the particular techniques we employed in encapsulating legacy code in the OSKit.

4.7.1 Basic Structure

We have found it extremely useful to preserve not only the *contents* of source files imported from legacy systems, but also the directory structure they reside in. For instance, all of the encapsulated FreeBSD code is located in the OSKit subdirectory `freebsd/src`; this directory exactly mirrors the `/usr/src` tree in the actual FreeBSD distribution, except that it only contains the files the OSKit actually uses. The glue code that encapsulates the imported FreeBSD code is located in separate directories such as `freebsd/net` and `freebsd/dev`, keeping the glue well separated from the encapsulated code. This structure allows changes in a new release of the donor OS to be incorporated simply by applying an appropriate patch to the appropriate OSKit directory subtree and then fixing any resulting conflicts. Of course, if the changes in the donor OS were extensive, the conflict resolution and debugging process can take some time and thought, but it is still much simpler and quicker than updating heavily modified or restructured code.

4.7.2 Conversions and Namespace Management

The imported OS code defines and relies on a large number of symbols which create namespace management problems at both compile and link time. For instance, the imported Linux and FreeBSD kernel header files each define their own versions of many standard POSIX types such as `size_t` and `struct stat`, which may or may not happen to be equivalent to each other or to the definitions used in the OSKit component interfaces. Mismatches between types used in imported code and those used in the public OSKit interfaces, such as differences in the `stat` structure, are handled by performing conversions in the glue code surrounding the encapsulated component. However, this means that the glue code must include both the header files imported from the donor OS and the header files defining the OSKit interfaces; to prevent symbol name conflicts, all symbols defined by these OSKit headers are given prefixes (e.g., `oskit_stat`) to disambiguate them from symbols used in legacy code. Following this rule in the OSKit interface definitions also leaves a cleaner namespace for the client OS.

The link-time namespace presents another problem: although the legacy header files are never used or seen by other components or by the client OS, any global functions or variables the legacy code defines may conflict with those defined by the client OS or by other components. For example, the NetBSD file system and FreeBSD networking components use many functions with the same names but incompatible definitions, which is not surprising given the common heritage of these systems. To solve this problem we used preprocessor magic to rename these symbols: e.g., the `wakeup` function used in the FreeBSD device drivers is named `FDEV_FREEBSD_wakeup` in the compiled object files comprising the library, preventing linker conflicts with other code.

4.7.3 Exporting COM Interfaces

We have found the implementation flexibility afforded by the OSKit's COM interfaces to be critical to exporting efficient interfaces to legacy code. For example, as mentioned earlier, BSD and Linux each have internal "network packet buffer" abstractions, known as `mbufs` and `skbuffs` respectively, whose implementation details are thoroughly known throughout their respective device driver and networking code. It would be impractical to change either code base to use a different packet buffer representation, but in order to make the BSD and Linux components interoperate with each other and with client OS code that may use a different abstraction, the details of `mbufs` and `skbuffs` must be hidden within the respective components. COM interfaces allow this to be done without copying except in a few unavoidable situations.

When a Linux network driver receives a packet from the hardware, it reads it into a contiguous `skbuff` and then passes it up to higher-level networking code, which in this case is the OSKit's Linux glue code. This glue code must in turn export the packet from the component using the OSKit's common networking interfaces in which packets are represented by `bufio` interfaces (see Section 4.4.2). Because COM interfaces make essentially no restrictions on the implementation details of the objects themselves, the Linux glue code can export the `skbuff` directly as a COM `bufio` object without copying the data, merely by adding a `bufio` interface to the Linux `skbuff` structure itself. The COM interface is simply a one-word field in the `skbuff` structure in which the glue code places a pointer to a function table providing methods to access the `skbuff`'s contents; the semantics of these functions are defined by the `bufio` interface, but the functions themselves are implemented by the glue code with full internal knowledge of Linux's `skbuff` implementation. After the `skbuff` leaves the component, external code only manipulates it through its `bufio` interface.

Packets submitted to the driver component for transmission are also represented by a COM `bufio` interface, but the Linux glue code cannot assume that the object is really an `skbuff` since, for example, the packet may have been manufactured in the FreeBSD TCP/IP code where packets are instead represented as `mbufs`. The Linux glue code can easily recognize "foreign" `bufio` objects by checking their function table pointer; when it receives one, it first calls its `map` method to obtain a direct pointer to the data in the buffer if possible. This call will only succeed if the implementor of the `bufio` object happens to store the requested range of data in contiguous local memory; if it does, the Linux glue code creates a "fake" `skbuff` pointing directly to this data. Otherwise, the glue code allocates a normal `skbuff` and calls the `bufio` interface's `read` method to copy the data into the buffer. In this way, copying is avoided whenever possible while presenting a clean, abstract interface to the client OS and other components.

4.7.4 Blocking and Interrupts

Since all of the OSKit's encapsulated components currently come from systems that use the relatively simple and well-understood blocking model, the encapsulated OSKit components retain this same execution model as seen by the client OS. The model has two levels, "process level" and "interrupt level." There can be many process-level threads of control using separate stacks, but only one can run at a time and context switches only occur at well-defined "blocking" points; interrupt-level activities can run any time interrupts are enabled and always run to completion without blocking. Unlike in conventional kernels, however, the "process" and "interrupt" abstractions in the OSKit components are generally only relevant for purposes of defining the concurrency model, and do not necessarily correspond to "real" processes or hardware interrupt handlers. As long as the client OS ensures that all calls it makes into the component follow the constraints defined by the concurrency model, it can use the component in practically any environment.

For example, although the encapsulated OSKit components are not inherently multiprocessor or thread safe, they can easily be used in multiprocessor or multithreaded environments by taking a component-wide lock just before entering the component, and releasing it after the component returns and during any "blocking" calls the component makes back to the client OS. Although this allows only relatively coarse-grained concurrency, this granularity is sufficient in many situations and is clearly the best we can do without heavily modifying the imported code. Furthermore, the client OS can run different OSKit components independently—for instance, a multiprocessor OS could place separate locks around the file system and network components, allowing them to execute concurrently. This medium-grained concurrency is possible because of the clean separation and independence of the OSKit components, and would be much harder to achieve if the BSD or Linux file system and networking code was imported into the client OS directly without any of the "packaging" done by the OSKit.

4.7.5 Hiding Details of Legacy Environments

Even though the concurrency model presented to the client OS by the OSKit's encapsulated components is superficially similar to the models used in the donor OS environments, the OSKit's environment is much simpler and more limited in many ways, making as few demands as possible on the client OS in order to make the components as widely usable as possible. For example, as mentioned above, the OSKit's components generally can't assume that the client OS has any notion of a "process" in the traditional sense. However, the imported legacy code is generally riddled with code that makes assumptions about processes and often accesses the "current process" structure directly (e.g., through BSD's `curproc` or Linux's `current` pointer). This is done for many reasons, but the most common cases are permission checking and blocking on events.

To avoid having to make the large number of changes required to eliminate these dependencies, we instead structured the glue code to *emulate* the abstractions expected by the encapsulated code. For example, to emulate the current process, at every entrypoint into the component from the "outside," the glue code creates and initializes a minimal temporary process structure on the stack, and initializes the global (component-wide) `curproc` pointer to point to it. This structure then represents the "current process," as far as the encapsulated code is concerned, for the duration of this call, and automatically disappears when the call completes—in essence, the glue code manufactures processes on demand. Since other threads of control may execute in the component during any blocking calls the component makes back to the client OS, the glue code must also intercept these calls and save the `curproc` pointer on the local per-thread stack for their duration in order to prevent it from getting trashed by other concurrent activities. In this way, the glue code emulates the process abstraction expected by the legacy code while completely hiding it from the client OS.

4.7.6 Sleep/Wakeup

Another related part of the donor OS environment used throughout the imported legacy code is the sleep/wakeup mechanism, which for example the interrupt handler in a device driver uses to wake up a blocked `read` or `write` request after it has completed. Naturally, the details of the sleep/wakeup mechanism vary widely even among similarly structured kernels such as BSD and Linux. The glue code in each encapsulated component emulates the native sleep/wakeup mechanism of the donor OS on top of a single extremely simple underlying abstraction designed to be as easy as possible for the client OS to implement: namely a "sleep record," which is like a condition variable except that only one thread of control can wait on it at a time. The client OS can directly implement these "sleep records" in various ways, such as using conventional condition variables or event objects or whatever else the client OS uses internally for synchronization. In fact, in the OSKit's single-threaded example kernels, sleeping is implemented simply as a busy loop that spins on a one-bit field in the sleep record structure.

Given this minimal sleep/wakeup mechanism provided by the client OS, we found that the easiest way to emulate the more elaborate mechanisms expected by legacy OS code was to incorporate the actual sleep/wakeup code from the legacy OS and modify it slightly to use the OSKit's sleep record abstraction in place of the legacy OS's scheduler. For instance, the BSD sleep/wakeup mechanism uses a global hash table of "events," where an event is just an arbitrary 32-bit value; when `wakeup` is called on a particular event, all processes waiting on that particular value are woken. In the encapsulated BSD-based OSKit components, we retain BSD's original event hash table management code; however, the hash table is now only used within that particular component rather than throughout the entire system, and instead of all the scheduling-related fields in the emulated `proc` structure there is now only a sleep record which the BSD glue code uses to block the current "process" as necessary.

4.7.7 Memory Allocation

Another tricky aspect of the legacy environment to be emulated is the memory allocation mechanism. BSD's in-kernel `malloc` package tries to be particularly clever in a number of respects: (1) all allocated blocks are naturally aligned according to their size (e.g., 65–128 byte blocks are aligned on a 128-byte boundary); (2) blocks with a size of exactly a power of two can be allocated efficiently without wasting space; and (3) the allocator automatically keeps track of the sizes of allocated blocks. Any two of these properties can be implemented easily, but it takes special tricks to provide all three at once—and several parts of the BSD kernel, such as the `mbuf` code for networking and the `clist` code for character I/O, depend on all three properties.

The BSD kernel allocator provides these three properties by reserving on system startup a special range of kernel virtual memory for this allocator, and creating a separate table with one entry for each page in this range. Each entry records the size of the blocks allocated from that page, so that on a call to `free` the allocator can quickly look up the size of the allocated block without having to reserve space in the memory block itself for this information, which would conflict with the first two properties above. Unfortunately, this solution is not acceptable in the OSKit because OSKit components do not have any knowledge of or control over the virtual or physical memory layout of the kernel or user-mode application in which they run. In this case, there appears to be no entirely satisfactory solution that does not involve heavy modifications to the imported legacy code; however, we have adopted an imperfect but practical solution which relies on the (generally true) heuristic assumption that most memory blocks returned by the client OS will be fairly densely packed within the address space. To provide the properties above in this context, our BSD glue code uses BSD's kernel `malloc` package unmodified, layering it on top of the memory allocation facility the client OS provides, except that our glue code watches the memory blocks returned by the client OS and dynamically re-allocates and grows the allocation table as necessary to ensure that it always covers all of the addresses that the allocator has ever "seen." Naturally, this solution may become very inefficient or cease to work in unusual cases where the memory blocks returned by the client OS are very widely dispersed in the component's address space; however, it works fine in all of the situations we have so far encountered.

4.7.8 Physical Memory Access

Another common problem with legacy kernel code, particularly device driver code, is the assumptions it makes about how it can access physical memory. For example, some of the Linux device drivers assume that all physical memory is direct-mapped into their address space starting at virtual address zero, and access memory-mapped devices, BIOS information, etc., simply by "manufacturing" pointers directly from physical addresses. This makes it impossible to use these particular drivers in a client OS that does not directly map physical memory in this way; In this case there appears to be no solution other than fixing the drivers themselves. Fortunately, FreeBSD drivers use a well-known symbolic constant when accessing physical memory in this way; this "constant" can simply be changed into a variable initialized by the BSD glue code on start-up to point to a region of mapped physical memory.

5 Example OSKit-Based Systems

The examples distributed with the OSKit include Chesapeake's Test TCP (`ttcp`) benchmark [13] which measures TCP send/receive bandwidth. We implemented a second benchmark to measure latency, similar to hbench's [11] `lat_tcp`, called `rtcp`, which measures the time required for a 1-byte round trip. We will use these examples to demonstrate how applications can tie various OSKit components together, and to measure the unavoidable though fairly small performance impact caused by the interactions between components and by mismatches in the internal abstractions used by imported legacy code.

The minimal C library provided all but a few of the functions required by these examples. Since `ttcp` relies on the times reported by `getrusage` for its timing, we implemented a simple `getrusage` based on the timers kept by the FreeBSD-derived networking code. `ttcp` also uses `signal` and `select` which are currently not provided by the OSKit, but they are only used to handle exceptional conditions and can be implemented as null functions without affecting the results. Aside from these additions and the initialization of the device drivers and networking stack, we were able to compile the applications unchanged. The required initialization code looks like this:

```
fdev_linux_init_ethernet();
fdev_probe();
oskit_freebsd_net_init(&sf);
posix_set_socketcreator(sf);
fdev_device_lookup(&fdev_ethernet_iid, &dev);
oskit_freebsd_net_open_ether_if(dev[0], &eif);
oskit_freebsd_net_ifconfig(eif, IPADDR, NETMASK);
```

`fdev_linux_init_ethernet` initializes the Linux ethernet drivers, causing all supported drivers to be linked into the resulting application. (The client OS can alternatively link in only specific drivers if it chooses.) `fdev_probe` locates all devices for which a driver has been initialized. The FreeBSD networking stack is initialized with `oskit_freebsd_net_init` which returns a "socket factory" interface used to create new sockets; `posix_set_socketcreator` is then called to register that socket factory with the C library so that its `socket` function will work. The `fdev_device_lookup` call searches the device table constructed with `fdev_probe`, returning an array of handles for Ethernet devices. The first such device is then bound to the FreeBSD network stack by `oskit_freebsd_net_open_ether_if`, which returns a handle used by `oskit_freebsd_net_ifconfig` to perform BSD `ifconfig`-style configuration of the network interface.

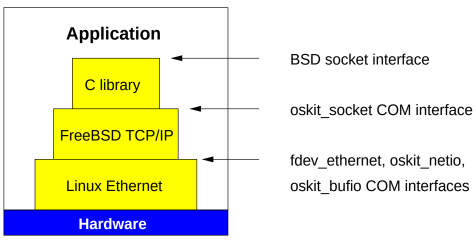

Figure 3: Structure of the `ttcp` and `rtcp` example kernels.

Figure 3 shows the structure of the `ttcp` and `rtcp` kernels when compiled with the OSKit libraries. The application uses the familiar BSD socket functions. The OSKit's C library maps these functions directly to the methods of the `oskit_socket` COM interface implemented by the FreeBSD networking component, by associating file descriptors with references to COM objects. Since the C library's `socket` call uses a client-provided socket factory interface to create new sockets, this C library code can be used with any protocol stack that provides these socket and socket factory interfaces.

	Receiver: Linux	FreeBSD	OSKit
Sender:			
Linux	**72.4**	71.2	71.3
FreeBSD	60.0	**78.6**	78.7
OSKit	56.4	68.3	**68.2**

Table 1: TCP bandwidth in MBit/s measured with `ttcp` between two Pentium Pro 200MHz PCs connected by 100Mbps Ethernet.

When the client OS binds the FreeBSD protocol stack to a Linux device driver during initialization, these components exchange callback functions which are subsequently used to pass packets back and forth asynchronously. When the driver receives a packet from the hardware, the driver calls the protocol stack's registered callback; similarly, when the protocol stack needs to transmit a packet, it calls the device driver's callback. Packets passed through these callbacks are represented as references to opaque objects implementing the `oskit_bufio` COM interface (see Section 4.7.3). In this system configuration, incoming packets are initially represented internally as `skbuffs` in the Linux network driver code; these `skbuffs` are passed directly to the FreeBSD TCP/IP component as COM `bufio` objects, which the FreeBSD glue code internally repackages as `mbufs` for the benefit of its imported FreeBSD code. Since `skbuffs` represents packets as contiguous buffers, the FreeBSD glue code is able to obtain a direct pointer to the packet data using the `map` method of the `bufio` interface, and therefore never has to copy the incoming data. Outgoing packets manufactured by the FreeBSD TCP/IP code, on the other hand, sometimes consist of multiple discontiguous buffers chained together; in this case, when the `mbuf` chain is passed to the Linux driver as a `bufio` object, the Linux glue code must read the data into its own contiguous buffer in order to present it to the encapsulated driver code as an `skbuff`. Thus, the mismatch between Linux's and BSD's internal packet representations sometimes requires extra copying on the send path, but never on the receive path.

Tables 1 and 2 compare the TCP send and receive bandwidth and latency for three environments: Linux 2.0.29, FreeBSD 2.1.5, and the OSKit using the FreeBSD 2.1.5 protocol stack and the Linux 2.0.29 device drivers. Running these tests as applications on top of Linux or FreeBSD involves system call overhead not present in the OSKit versions; to factor this out, the transmit and receive loops for `ttcp` and `rtcp` were moved into the kernel on Linux and FreeBSD and accessed via a special system call.

Table 1 presents the bandwidth measurements obtained with `ttcp`. In each case `ttcp` transmitted 131072 blocks of 4096 data bytes (52MB total). This reasonably long run of the test compensates for the relatively low 10ms granularity of the clock used for timing. The results show that the OSKit implementation receives about as fast as FreeBSD—this is due to the fact that the Linux driver always hands contiguous buffers up which can be mapped to `mbuf` clusters without copying. On the other hand, when a packet is sent, an additional copy is necessary since non-contiguous protocol `mbufs` must be copied into contiguous device driver `skbuffs`, accounting for the decrease in send performance.

Table 2 shows the latency of a 1-byte round-trip. While we can-

	Server: Linux	FreeBSD	OSKit
Client:			
Linux	**152**	168	180
FreeBSD	168	**197**	210
OSKit	180	210	**222**

Table 2: TCP one-byte round-trip time in μsec measured with `rtcp` between two Pentium Pro 200MHz PCs connected by 100Mbps Ethernet.

not interpret Linux's performance, the FreeBSD versus OSKit results indicate that the OSKit imposes significant overhead. Extra data copies are not part of the problem since this test involves small packet sizes that fit in a single protocol `mbuf`, enabling mapping into a device driver `skbuff`. Hence, the overhead is largely attributable to the additional glue code within the OSKit components: the price we pay for modularity and separability and for the ability to use existing driver and networking code unmodified in an environment for which they were not designed.

6 Experience Using the OSKit

The OSKit is already being used in several different research projects at institutions around the world, not only for "traditional" OS research but also for systems-level advanced programming language research: designing systems in which the programming language *is* the operating system. Language implementations usually have to take the operating system's interface to the hardware as a given; for languages whose semantics differ markedly from C, the match is often not ideal. Unix, for example, is tuned to provide the services required by the C run-time model, such as protected, flat address spaces and stack allocation. The OSKit, for the first time, enables advanced language systems to be easily implemented directly on the raw hardware, avoiding these mismatches created by traditional operating systems. The most striking common finding of these various research projects has been how remarkably easy it was to implement experimental kernels and advanced language systems on the raw hardware using the OSKit as a substrate.

6.1 Case Studies

In this section we first briefly describe four major research projects that have recently taken advantage of the OSKit, and the overall experience of using the OSKit in each case; the next section will describe in more detail specific aspects of the OSKit that proved to be particularly useful in these research projects.

6.1.1 The Fluke OS

In 1996 we developed an entirely new microkernel-based system called Fluke [17] to explore numerous ideas in fundamental kernel structuring, including scheduling mechanisms, execution models, IPC, and virtual memory. We had been pursuing research using the Mach microkernel, but found that none of these directions could have been explored or effectively evaluated in the context of this existing system because it was too large, inflexible, and tightly bound together to be amenable to the fundamental changes we needed to make. Therefore, we started a new system, incorporating a few pieces of Mach and BSD code but otherwise written from scratch. To ensure that Fluke would not quickly become as tightly-bound and inflexible as its predecessor, we started developing the OSKit concurrently as a set of components to be used in Fluke and other projects. Therefore, Fluke acted as the primary driving application for the OSKit, but by also using the OSKit simultaneously for other purposes, we were able to prevent it from becoming specific to Fluke.

Fluke puts almost all of the OSKit to use, and in fact over half of the Fluke microkernel is OSKit code. Most of the basic servers and other utilities that run on Fluke also use the OSKit to provide their standard C library, memory allocation, address space management, and debugging facilities. These servers include a virtual memory manager, checkpointer, file and network servers, and a process manager. Although Fluke includes a complete standard C library based on FreeBSD's C library for the use of Unix applications running on Fluke, in many situations we have found that the OSKit's minimal C library provides all the functionality needed and is much smaller, simpler, and more flexible.

6.1.2 Standard ML

Standard ML [26] is a functional programming language that includes first-class, higher-order procedures, a static polymorphic type system, exceptions, continuations, and a sophisticated module system. We built our system, called ML/OS, by porting the Standard ML of New Jersey (SML/NJ) implementation [6] to run on a PC using the OSKit. SML/NJ is a complex, Unix-based system comprising about 144,000 lines of code, in over 1000 source files. The run-time model used by SML/NJ is fairly exotic—for example, the system runs completely without a stack, using instead very aggressive heap-allocation and garbage-collection techniques to manage procedure frames. Our current research focus in the ML/OS effort is modeling concurrency using semantic features found in higher-order programming languages, in particular *continuations*. This requires the language and compiler to be intimately involved with the fundamental context switch code, something that is not possible in traditional operating systems.

At MIT, ML/OS was developed over a semester by a Master's student with the part-time assistance of an undergraduate programmer. Neither student was previously familiar with OS internals or the low-level details of the x86 architecture. Much of their effort was spent in learning and dealing with the details of the SML/NJ implementation, which was far more complex than the OSKit code to which it was being mated. In contrast to this experience, strong groups of advanced programming language researchers have been intending, for years, to implement a sophisticated functional language directly on a raw hardware platform. For example, the Fox project at CMU [5, page 214] and the Programming Principles group at Bell Labs have at different times begun efforts to port SML/NJ to run on bare hardware. But the details of doing so have been sufficiently forbidding as to prevent these efforts from ever being completed.

6.1.3 SR

SR is a language designed for writing concurrent programs, both parallel and distributed [3], for both application and systems software. It offers a flexible concurrency model, and includes as inherent parts of the language entities such as threads, synchronization, and communication mechanisms. The standard SR implementation [31] is tightly coupled to Unix I/O and Unix sockets, thus for the work described in this paper we started with a previously developed, more platform-neutral version of SR [9]. That version removes many Unix dependencies and isolates system-dependent functionality such as threads, synchronization, and network communication. Our research goal in porting SR to the raw hardware is to investigate the effectiveness of a communication-oriented language for implementing OS functionality.

Implementing SR/OS with the OSKit was accomplished by one U.C. Davis student while visiting the University of Utah. The initial implementation took approximately one week, and adding network support using the *x*-kernel protocol stack required another week. In contrast, several earlier attempts to implement SR directly on the hardware proved very difficult or were stillborn. A very early version of SR was implemented directly on PDP-11 machines, but its development and debugging were extremely tedious [28]. Later, during the Saguaro distributed operating system project, an SR implementation on the bare hardware was again considered, but abandoned due to the lack of good development tools [2].

6.1.4 Java

Finally, in a project to create a Java [19] environment on the raw hardware, we started with Kaffe [34], a freely available and portable Java virtual machine and just-in-time compiler. Kaffe is written for a standard POSIX environment, requiring support for file I/O calls such as `open` and `read`, as well as BSD's socket API. It implements its own user-level thread system, for which it relies on some minimal signal handling facilities and timer support. Our implementation goals were to provide a prototype Java-based "network computer" called Java/PC, and an active network router. Our research goals are to explore resource management issues, comparing this Java system on the bare hardware to a Java system atop the Fluke microkernel.

Building Java/PC atop the OSKit was remarkably easy: one Utah student, at that time not a major contributor to the OSKit, took just 14 hours to get the system to run a "Hello, World" Java application; large single-threaded applications, such as Sun's Java compiler, ran the next day. Less than three weeks later he had built a usable system that ran complex applications such as the Jigsaw Web Server [7], making extensive use of threads, timers, and file and network I/O. The resulting system is similar in function to Sun's JavaOS [33] but with a dramatically different implementation. Whereas almost all components in our system reuse existing C-based components provided by the OSKit, Sun's was primarily written anew in Java and took much longer to build.

6.1.5 Other Uses of the OSKit

We used an early version of the OSKit in a "DOS extender" [15], a small OS kernel that runs on MS-DOS and creates a more complete process environment for 32-bit applications; this DOS extender is now being used in commercial products. We have also used the OSKit in two small experimental kernels that we prototyped in order to test out new IPC, capability, and kernel execution environment concepts before committing to these ideas in the main Fluke kernel effort. Both of these kernels were developed in a matter of days to the point of being useful for measurement and analysis; these prototypes would not have been feasible without the OSKit. Finally, besides these experimental kernels, we have used the OSKit in several smaller utilities, such as specialized kernels to boot other kernels across the network or from existing non-MultiBoot boot loaders.

A few of the sites that have retrieved the OSKit have informed us of their use. Among these are researchers at the University of Carlos III in Spain who have built their "Off" distributed adaptable microkernel [8] on top of the OSKit, and the "bits and pieces microkernel" (*bpmk*), developed in Finland. A company, Network Storage Solutions, is using the OSKit to provide the base hardware support for a "network appliance"-style server. In the wake of the successful language-based OS projects discussed above, another Utah student recently ported the GNU Smalltalk system to the bare hardware. He implemented a complete, functional multithreaded Smalltalk system in just over seven hours, starting with little experience with operating systems, the x86 PC, the OSKit, or the Smalltalk run-time system. This system has not yet, however, been used for serious research.

6.2 Common Experiences

This section describes a few specific aspects of the OSKit that proved to be particularly useful in the above research projects, as well as OSKit features that caused problems or were commonly needed but not yet available.

6.2.1 POSIX Environment

All of the language implementations greatly benefited from the fairly complete POSIX environment provided by the OSKit's minimal C library, memory allocator, and kernel support library. This environment allowed the Unix versions of the languages to be quickly ported to the bare hardware, and then gradually specialized to take advantage of the new environment, extending their control to various hardware resources (e.g., registers, traps, interrupts) that are hidden by a normal operating system. Furthermore, the OSKit's minimal POSIX environment allowed the language research to focus on issues critical to the research, and let the POSIX environment pick up the rest: we could let unimportant code remain unimportant.

6.2.2 Bootstrap Support

A particularly notable feature of the OSKit's minimal environment is its boot module support (see Section 3.1), which provides a simple RAM-disk file system accessible immediately upon bootstrap through POSIX's standard `open`/`close`/`read`/`write` interfaces. For example, in Fluke, this boot module file system includes the first user-mode executable to be loaded and run by the kernel, and subsequently acts as the root file system for this initial server, without requiring any "real" file system or device driver components to be embedded in the kernel. ML/OS uses a boot module to hold the precompiled "initial heap image" for the ML runtime, which is over 99% of the kernel (i.e., everything written in ML); similarly, Java/PC loads its Java bytecode (`.class` files) from the initial boot module file system. Other alternatives are available in each of these cases, such as manually embedding data into `.o` files linked into the kernel, or using the OSKit's device drivers and file system components to load this data from a "real" disk-based file system, but the boot module facility invariably proved to be by far the most simple, robust, and convenient for this purpose.

6.2.3 No Imposed Process/Thread Abstraction

The absence of an OS-defined process or thread abstraction proved of great benefit to all three languages. ML/OS provides continuation-based Concurrent ML [29] threads as the machine's basic thread facility, complete with a preemptive scheduler, console I/O, and timer events. Whereas OS thread systems usually *center* on stacks, CML threads are entirely *without* stacks, running entirely in the heap. Interrupts and thread context switches use garbage collected continuations. Modeling concurrency in this way is central to our line of research; building on an infrastructure that imposed no thread abstraction meant that we could implement this model directly. It was similarly straightforward to port the built-in thread packages in Kaffe and SR to the OSKit. This contrasts with our experience porting Kaffe to a kernel providing its own thread abstraction—our Fluke microkernel. On Fluke, in order to avoid classic problems such as blocking for I/O, we needed to use native Fluke threads instead of Kaffe's built-in threads; minor mismatches between Java's and Fluke's thread semantics caused the Fluke port of Kaffe to take much longer.

6.2.4 Exposed Implementation and Hardware

In the ML port it was pleasant to discover how simple it can be to implement a high-performance functional programming language when one doesn't have to bend over backwards to accommodate the demands of an ill-suited operating system. The SML/NJ sources are larded with complex sequences of code designed to work around the limits of the Unix architecture. For example, SML/NJ allows heap allocation in signal handlers; to make this work, the run-time system must go through arcane gymnastics when it wishes to modify the processor's register state from a Unix signal handler. The trickery involved is considered sufficiently clever as to be worth reporting in the literature [4]. Since in our case the hardware is exposed in a documented manner, we just did it. No Unix; no trickery.

Java provided other examples of leveraging the exposed implementation and hardware. Kaffe relies on the delivery of `SIGSEGV` signals to detect null pointer accesses and throw the null pointer exception specified by Java semantics. The x86 architecture provides an efficient way to catch such accesses, by setting the processor's breakpoint registers appropriately. Besides allowing Java/PC direct access to this facility, the OSKit also provided a simple way for it to install its own custom trap handlers written in ordinary C, which can still fall back to the default handler for traps that are of no interest.

6.2.5 Modular, Separable Components

"Network computers" seek to minimize memory footprint, and often do not need a disk or file system; using the OSKit it proved trivial to build a version of Java/PC that included networking but no file system. We have not yet made any effort to minimize memory footprint, but the inherent modularity of the OSKit keeps the resulting system to a modest size: the static (code+data) size of our executable is 412KB, including one ethernet driver, networking (121KB), the Kaffe virtual machine and native libraries (132KB), and various glue code. Note that this system does not contain an implementation of Java's abstract windowing toolkit (AWT).

6.2.6 Mature Components with Flexible Interfaces

The networking performance of our Java/PC and SR/OS systems demonstrates the value of using mature, tuned components with flexible interfaces. Four weeks into the Java/PC project, using a measurement program written in Java, we measured a sustained TCP receive throughput of 78Mbps over a 100Mbps Ethernet, using the hardware platform described in Section 5. As expected, the TCP send throughput was lower at 59Mbps due to the extra copy. This relatively high performance is not surprising considering that the BSD network protocols have been tuned for over 15 years. In contrast, Sun's recent re-implementation of TCP/IP in Java [33] has been reported as being "more than adequate for Web browsing," but by inference is probably as yet nowhere near the performance of traditional C implementations.

6.2.7 Fully Defined Interfaces and Execution Models

The simple, well-defined execution models used by the OSKit components enabled them to be used fairly easily in very different environments from those in which they were originally designed. For example, even though the Linux and FreeBSD-based components were designed for a traditional nonpreemptive uniprocessor kernel environment, they were easily incorporated into the fully multithreaded Java/PC and SR/OS environments by surrounding them with small amounts of locking and other glue code. The Fluke kernel uses these same components in an even more exotic environment. Since Fluke can be configured as an interrupt-model kernel, with only one kernel stack per processor rather than one per thread, it is impossible to run process-model OSKit components on these kernel stacks since they are not retained across context switches. However, we were easily able to solve this problem by running the process-level activities of these components on ordinary Fluke threads running in user mode but in the kernel's address space, while interrupt handlers in the components still run in supervisor mode. Using the OSKit components in these fundamentally foreign environments is only possible because their assumptions on the surrounding environment are minimized, and the few remaining assumptions are precisely defined.

6.2.8 Library Structure

SR/OS's use of the *x*-kernel [21] protocol framework demonstrated the value of the OSKit's simple structure as a set of libraries independent of the client system. The *x*-kernel has an extremely complex build environment which is entirely different from the OSKit's. However, we got the *x*-kernel working quickly with SR/OS by working completely within the x-kernel's build environment and using the OSKit as an independent, previously-installed "package." Since all of the OSKit's functionality is provided by libraries, we just needed to add the appropriate '`-L`' and '`-l`' parameters to the main *x*-kernel makefile, and to point the main makefile to the OSKit include directories. Because the OSKit's exported structure is simple, it is easy to use it from within more complicated environments.

6.2.9 Tools: Debugging and Documentation

Although mundane from a research perspective, the practical importance of quality debugging tools and documentation should not be underestimated; this is particularly important to those whose primary interest and expertise lie elsewhere than in operating systems. The OSKit's robust source-level debugging support provided an environment familiar to application developers, contrasting sharply with the methods used during the early phases of the ML/OS project before GDB support was available: we could debug only by inserting "halt" instructions into the kernel, rebuilding, rebooting, and observing whether the kernel would quietly lock up ("success") or exhibit random behavior ("failure").

Finally, the documentation provided by the OSKit developers at Utah was a great help to the ML/OS hackers at MIT, even when it was much less complete than it is today. Had the ML project proceeded with its original intent to cannibalize Linux, we would likely still be puzzling out the code and interfaces of the kernel internals. Another more informal form of documentation were the twenty-line kernels E-mailed from Utah to MIT in answer to questions. These tiny (in source) but complete kernels were enabled by many features of the OSKit, all working together: the bootstrap/kernel support, the C/POSIX environment, the boot modules, and the component separability.

6.2.10 Deficiencies

These research projects also revealed several deficiencies in the OSKit; some of these have already been addressed, whereas others remain for future work. They include:

- Java/PC's concern with memory revealed a size disadvantage in the technique of using components built from encapsulated legacy code: the imported code was not designed with memory footprint as a primary concern, and the glue code encapsulating the imported code adds some additional overhead. In the future we may import or develop alternative implementations of these components, designed for minimal size rather than generality.

- Profiling of the benchmark kernels described in Section 5 revealed that a significant amount of time is spent in memory allocation and deallocation. This overhead is attributable to the fact that the OSKit's default memory manager library is designed for flexibility and space efficiency rather than common-case performance. For fast allocation of small data structures with no type or alignment restrictions, a more conventional high-level allocator would be more appropriate, possibly layered on top of the OSKit's existing low-level allocator. The OSKit currently does not provide a high-level allocator of this kind, but we expect to integrate one in the future.

- When an OSKit-based application "exits," the OSKit simply reboots the machine without performing any cleanup. Some applications, such as the `ttcp` benchmark in Section 5, assume that their network connections will be cleanly closed when they exit, as is done for Unix processes. On the OSKit such an application currently just leaves its peers hanging; this problem will be fixed in a future version of the OSKit's minimal C library.

- In the first released version of the OSKit, the layout of stack frames for synchronous traps was documented and visible to the client OS, but the layout of stack frames for hardware interrupts was not. This deficiency caused problems for both

ML/OS and Java/PC which needed access to the state of the interrupted code in order to handle preemption properly. Although they could have gained access to this state by replacing the OSKit's default interrupt handling mechanism with their own, doing so would have been inconvenient, so instead we modified the OSKit's hardware interrupt handler to use the same well-documented stack frame used for synchronous traps. This experience demonstrates an instance in which the OSKit initially *failed* to expose its implementation details sufficiently and had to be fixed later.

7 Status

In Table 3 we present a breakdown of the source sizes of the various OSKit components. The OSKit currently consists of over 260,000 lines of code. Of that, 32,000 lines of native and assimilated code and interfaces provide access to remaining 230,000 lines of entirely encapsulated imported code. Over 500 pages of documentation are provided; most of this documentation is in Unix `man`-page format, one module or interface per page, so it is not as forbidding as it sounds. While the OSKit currently only runs on x86 PCs, most of it is designed to be easily portable to other platforms, and two thirds of the OSKit's exported interfaces are architecture-neutral. Most of the heavily architecture-specific aspects of the OSKit are isolated in the low-level kernel support library and the bootstrap code.

Our first public release of the OSKit in July 1996 was an alpha version; two months later we made one public update, primarily adding initial device driver support. For the seven months in which at least one of those versions was available, inspection of our FTP logs shows that, excluding obvious mirror sites, 537 different external sites obtained the OSKit, including 122 at U.S. companies. The currently released version of the OSKit is available at `http://www.cs.utah.edu/projects/flux/`.

8 Conclusion

We have been pleasantly surprised at how phenomenally quickly one can develop OS and direct hardware language implementations using the OSKit, and by the widespread and disparate interest in the OSKit from both the research and commercial communities. The OSKit's evolution has been almost entirely demand-driven, and we believe that is a major reason for its success. Rather than being designed *a priori*, with some inevitably flawed vision of future needs, it has been constantly refined and augmented, driven by the demands of a wide variety of real client systems. The ability of the OSKit itself to reuse components from outside sources, unchanged, is critical to its immediate as well as long term success. From this experience, we conclude that research and development of operating systems and languages are greatly aided by the pragmatic but systematic approach to software structure and reuse that the OSKit exemplifies. We expect the growing number of OSKit clients to drive continued growth in its power and flexibility.

Acknowledgements

We are especially grateful to Mike Hibler for last minute writing help, and to the many people at the University of Utah who have contributed to the OSKit, including Steve Clawson, Mike Hibler, John McCorquodale, Bart Robinson, Steve Smalley, Pat Tullmann, Kevin Van Maren, and Jeff Turner. We thank Shantanu Goel for his important work, both at Columbia and at Utah, on the Linux device driver framework in Mach. Erich Boleyn co-authored the MultiBoot specification and developed a boot loader for it. We thank the OSKit's clients for their feedback, and especially our shepherd John Ousterhout, the anonymous reviewers, members of the Flux project, Bob Kessler, and Ron Olsson for their helpful comments. Finally, we are grateful to the thousands who have contributed to the systems which provided the base for many of our components.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.

[2] G. R. Andrews. Personal communication, Feb. 1997.

[3] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Co., Redwood City, California, 1993.

[4] A. W. Appel. A Runtime System. *Lisp and Symbolic Computation*, 3(4):343–380, Nov. 1990.

[5] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, MA, 1992.

[6] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Third International Symp. on Programming Language Implementation and Logic Programming*, pages 1–13. Springer-Verlag LNCS 528, Aug. 1991.

[7] A. Baird-Smith. Jigsaw — An Object-Oriented Web Server in Java. http://www.w3.org/pub/WWW/Jigsaw/.

[8] F. J. Ballesteros and L. L. Fernandez. The Network Hardware is the Operating System. In *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, May 1997. To appear.

[9] G. D. Benson and R. A. Olsson. A Portable Run-Time System for the SR Concurrent Programming Language. In *Proceedings of the Workshop on Run-Time Systems for Parallel Processing*. IR-417, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, April 1997.

[10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[11] A. B. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proc. of the 1997 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Seattle, WA, June 1997.

[12] R. Campbell, N. Islam, P. Madany, and D. Raila. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, Sept. 1993.

[13] Chesapeake Computer Consultants, Inc. Test TCP (TTCP). http://www.ccci.com/tools/ttcp, 1997.

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[15] B. Ford. MOSS: A DOS extender based on the Flux OS Toolkit. Available as http://www.cs.utah.edu/projects/flux/moss/, 1996.

[16] B. Ford and E. S. Boleyn. MultiBoot Standard. Available as ftp://-flux.cs.utah.edu/flux/multiboot, 1996.

[17] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, Oct. 1996. USENIX Assoc.

[18] S. Goel and D. Duchamp. Linux Device Driver Emulation in Mach. In *Proc. of the Annual USENIX 1996 Technical Conf.*, pages 65–73, San Diego, CA, Jan. 1996.

[19] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems Computer Company, 1996. Available as http://java.sun.com/doc/-language_environment/.

[20] D. Hildebrand. An Architectural Overview of QNX. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, Apr. 1992.

Library	Description	Interface MI	Interface x86	Implementation Native MI	Implementation Native x86	Implementation Encapsulated
boot	Bootstrap Support	—	67	—	2829	—
kern	Kernel Support	325	1379	476	3890	—
smp	Multiprocessor Support	6	2	—	868	—
lmm	List Memory Manager	33	—	314	—	—
amm	Address Map Manager	60	—	349	—	—
c	Minimal C library	588	4	4863	220	—
memdebug	Malloc Debugging	18	—	398	13	—
diskpart	Disk Partitioning	205	—	311	—	—
fsread	File System Reading	13	—	1581	—	—
exec	Program Loading	133	5	61	125	—
com	COM interfaces & support	1514	—	667	—	—
fdev	Device driver support	35	—	861	388	—
linux_dev	Linux drivers & support	77	—	2801	—	77023
	FreeBSD common code	—	—	524	116	8275
freebsd_dev	FreeBSD drivers & support	8	13	558	10	14755
freebsd_net	FreeBSD network stack	44	—	1318	—	17241
freebsd_m	FreeBSD Math library	—	—	—	—	7517
netbsd_fs	NetBSD file system	8	—	2465	—	18968
x11	X11-based windowing	—	—	1909	—	90182
		3067	1470	19456	8459	
	Total	4537		27915		233961

Table 3: "Filtered" source code size, of the OSKit components, classified into interface (header files) and implementation (C and assembly language source), with the latter classified into home-grown/assimilated code and encapsulated, imported code. The code is further broken down into machine-independent (MI) and machine-dependent (x86) portions. This count of source code lines filters out comments, blank lines, preprocessor directives, and punctuation-only lines (e.g., a line containing just a brace), and typically is $1/4$ to $1/2$ the size of unfiltered code. The X11-based windowing support is currently in progress.

[21] N. Hutchinson and L. Peterson. The x-kernel: An Architecture for Implementing Protocols. *IEEE Transactions on Software Engineering*, SE-17(1):64–76, Jan. 1991.

[22] G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, 13(1):8–11, Jan. 1996.

[23] S. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proc. of the Summer 1986 USENIX Conf.*, pages 238–247, Atlanta, GA, June 1986.

[24] G. Lehey. *The Complete FreeBSD*. Walnut Creek CDROM Books, 1996.

[25] Microsoft Corporation and Digital Equipment Corporation. *Component Object Model Specification*, Oct. 1995. 274 pp.

[26] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[27] W. Myers. Taligent's CommonPoint: The Promise of Objects. *Computer*, 28(3):78–83, Mar. 1995.

[28] R. Olsson. Personal communication, Feb. 1997.

[29] J. H. Reppy. CML: A Higher-order Concurrent Language. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 293–305, June 1991.

[30] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996. USENIX Assoc.

[31] The SR Programming Language, Version 2.3.1, Dec. 1995. http://www.cs.arizona.edu/sr/.

[32] R. M. Stallman and Cygnus Support. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, Inc., Boston, MA, 4.12 edition, Jan. 1994.

[33] Sun Microsystems, Inc. JavaOS: A Standalone Java Environment, Feb. 1997. http://www.javasoft.com/products/javaos/-javaos.white.html.

[34] T. Wilkinson. KAFFE - A virtual machine to run Java code. http://www.tjwassoc.demon.co.uk/kaffe/kaffe.htm.

[35] Wind River Systems, Inc., Alameda, CA. *VxWorks Programmer's Guide*, 1995.