Using the OSKit as a base for L4 applications

Michael Hohmuth

Dresden University of Technology
Department of Computer Science
D-01062 Dresden, Germany
email: hohmuth@inf.tu-dresden.de

1 Increasing the number of L4 applications

The OSKit is a framework and set of modularized components and library code, together with extensive documentation, for the construction of operating-system kernels, servers, and other OS-level functionality [3]. The DROPS project at Dresden University of Technology has been using the OSKit since the Flux group of Utah University released OSKit 0.60. We used it as a library of pieces useful for operating-systems development: a C runtime library, basic console input–output, low-level support for PC peripherals, some data structures—for all kinds of small applications for which we didn't want to write code ourselves because doing so would be tedious, boring, inefficient, or uninteresting.

With the advent of version 1.0[1], however, the OSKit has reached a level of completeness that makes it useful for other uses. It now contains a lot of device drivers, file systems, and a networking stack, and provides a well-componentized free-standing base powerful enough to support Unix applications without Unix—or any other operating system. As an impressive example, the Flux group has ported the Kaffe Java virtual machine to the OSKit, building a stand-alone Java operating system.

Naturally, we increasingly wished to leverage that power for applications running on our L4-based systems. In the past, we have often found ourselves porting Unix programs or Linux device drivers to L4, and we gladly would have taken any help that was offered from elsewhere. The OSKit represents such an offer—in fact, it not only offers help, it contains complete solutions for many problems. However, the OSKit didn't seem to be easily accessible: Dozens of megabytes of source code and 500+ pages of documentation seemed scary at first.

With this paper I aim at making L4 developers more accustomed with the OSKit. I describe my port of the OSKit to L4, and I show ways of using the OSKit and assess their difficulty. However, I will not give my judgement about the OSKit's applicability for particular purposes like construction of real-time systems; I just aim at making it easier to develop for L4 in order to make L4-based systems more useful.

In short, there are three main motivations for using the OSKit under L4:

1. Port standard Unix applications to L4 with relatively small development investment
2. Derive new L4 servers from OSKit components
3. Derive dynamic libraries from OSKit components

This paper is organized as follows: In the following three sections, I will discuss in order the modes of using the OSKit I just mentioned. Section 5 serves as a repository for this work's loose ends and unanswered questions; it also contains a summary and a preliminary conclusion.

2 Adapting the OSKit to L4

When an application is linked to the OSKit, it usually makes use of the OSKit's POSIX-compatibility library and one of its C libraries[2]. These libraries make use of the OSKit's kernel-support libraries including the device-driver libraries, and these in turn depend on an environment-specific device-driver–helper library. The OSKit comes with a default device-driver–helper library, `liboskit_dev`, that depends on a certain setup of the program's context, the "base environment." This environment consists of a certain assumed CPU state (protected mode, flat-segment memory model, kernel privileges), memory mapping (physical memory is mapped one-to-one into virtual memory), and a few data structures initialized by the startup code and accessible through access functions defined in `liboskit_kern`, including a memory pool suitable for memory allocation, an array of exception vectors, and support for console input and output.

Given that the aim was to run OSKit applications (including applications using device drivers) on top of L4/x86 and Fiasco, it was clear that certain parts of the OSKit had to

[1] The current version of the OSKit as of this writing is 0.97.

[2] The OSKit comes with two C libraries: a minimal C library, `liboskit_c`, and a more standard C library derived from FreeBSD's C library, `liboskit_freebsd_c`.

be replaced. It turned out that the OSKit is very well modularized; especially the interface of the device-driver–helper library is rich and versatile enough to allow for easy adaption to any operating-system environment.

L4 applications started from the DROPS system's resource manager Rmgr [6] see an environment that is quite similar to the standard OSKit base environment the default device-driver–helper library `liboskit_dev` assumes. That's why I decided against replacing the whole `liboskit_dev` with an L4-specific implementation. Instead, I modified only selected parts of this library together with certain parts of the base-environment support in `liboskit_kern`. These modified functions form a new library `liboskit10_support` that is meant to be one of the first libraries in the linking order, thereby overwriting the implementations in `liboskit_dev` and `liboskit_kern`.

In the following two subsections, I summarize the changes made to the OSKit. Section 2.3 shows a few sample applications using the OSKit, and Section 2.4 discusses some limitations of `liboskit10_support`.

2.1 Device-driver–helper library `liboskit_dev`

2.1.1 Hardware-interrupt handling

I rewrote the code to register interrupt handlers, to enable and disable interrupt delivery, and to enable and disable selected interrupts (interfaces `osenv_intr_*`, `osenv_irq_*`).

For every hardware interrupt for which drivers install a handler, this code starts a new L4 thread that attaches to the interrupt source using the appropriate Rmgr and L4 interfaces. The actual interrupt handlers are not run within these interrupt threads; instead, the interrupt threads interrupt the execution of the main thread using L4's `lthread_ex_regs` system call and put a pseudo interrupt frame on the main thread's stack so that the handler is executed within the context of the main thread. This ensures the interrupt semantics the OSKit assumes and makes synchronization very easy.

The extensive use of `lthread_ex_regs` has an important drawback, however: Applications now must cope with aborted and cancelled IPC operations[3]. This not only affects applications using the OSKit, but also their IPC partners. For example, this uncovered long-standing bugs in Rmgr and the Fiasco microkernel's Sigma0.

2.1.2 Memory-mapped input–output

Device drivers use the `osenv_mem_map_phys` function to map device pages into the virtual address space.

The L4 implementation of `osenv_mem_map_phys` keeps a map of available regions in the virtual address space. When a device page is requested, `osenv_mem_map_phys` allocates a region of virtual address space and requests a device page from the task's Sigma0 (or σ_0) server—that is, from the task's pager.

2.1.3 Sleeping and Timing

The device-driver–helper library must implement a sleep–wakeup mechanism (interfaces `osenv_sleep*`, `osenv_wakeup`). The default implementation, meant for stand-alone kernels, just idles in a close loop—this clearly is undesirable when running on top of a microkernel.

I replaced this implementation with one that uses an IPC system call to go to sleep. This system call is automatically terminated when an hardware interrupt occurs because the interrupt-handling thread does an `lthread_ex_regs` on the sleeping thread as described in Section 2.1.1.

The library also must implement a timer-interrupt facility and a non-blocking sleeping function (interfaces `osenv_timer_*`). The default implementation uses the PC's programmable interval timer at interrupt 0. While this implementation works under L4 as long as only one OSKit application is booted, it becomes a problem when using more that one application because only one of them can attach to interrupt 0. Also, the use of interrupt 0 is depreciated because a future implementation of L4 might use it internally.

My implementation uses a thread that emulates a timer interrupt using the microkernel's IPC timeouts.

2.1.4 Parts of `liboskit_dev` kept unmodified

The following summary shows some parts of `liboskit_dev` that work unmodified under L4. A few of them might need to be replaced with L4-specific implementations in the future, however.

Memory allocation (`osenv_mem_*`): These functions rely on a one-to-one virtual-to-physical page mapping (which Rmgr provides for its clients) and on an initialized memory pool named `malloc_lmm` from which memory can be allocated using the list-based memory management library coming with the OSKit (the `liboskit_kern` adaptions described in Section 2.2 provide `malloc_lmm`).

Hardware resource allocation (`osenv_{isadma,io}_*`): Currently, Rmgr clients can access all DMA channels and all of the input–output port address space, just like OSKit applications running stand-alone. That's why, I haven't modified these functions, but changing them might become necessary in the future.

Console output and panicking (`osenv_{*log,*panic}`): In the default implementation, these routines depend on console output routines in `liboskit_kern`, which is okay.

[3] "Cancelled" refers to IPC operations that are terminated before the message transfer started, and "aborted" means that the IPC operation is terminated during message transfer (that is, after sender and receiver have synchronized).

Device and driver registry (various interfaces): The liboskit_dev library contains default implementations for a driver registry and a device registry. This implementation can be overridden or integrated with an operating-system–specific registry, but currently there is no such registry in Dresden's L4-based systems.

2.2 Kernel library liboskit_kern

2.2.1 Base-environment initialization

The environment OSKit applications see when they run on the bare metal is similar in many respects to the environment Rmgr clients experience: The program is started according to the Multiboot specification [2], the CPU runs in protected mode, and virtual memory maps one-to-one to physical memory. To provide the "base environment"[4] liboskit_dev expects, only very little of the initialization code of liboskit_kern has to be modified:

CPU initialization: Of the x86's CPU data structures, only the IDT (interrupt descriptor table) is meaningful for L4 applications, and is used to translate exceptions into C signals. The GDT (global descriptor table) and TSS (task-state segment) don't need to be initialized and must not be loaded. (Also, the base_irq_* interface is not initialized and is not supported; drivers are supposed to use the osenv_irq_* interface.)

Memory-pool initialization: The L4-specific implementation only adds memory it can get from the task's Sigma0 server—that is, the task's pager—to the pool of allocable memory.

L4-specific initializations: I added a few L4-specific initializations to the initialization code. This code, for example, initializes librmgr, a library the device-driver–helper library uses.

2.2.2 Startup code

I provide two versions of the startup code:

One has been adapted from the OSKit's standard startup code and initializes all of the base environment before calling the program's main function. It also parses the Multiboot arguments passed to it from Rmgr and constructs and environment (accessible with getenv or via main's envp) and an argv array for main.

The other version doesn't do any of these initializations. It is meant for L4 servers that don't want the base environment but just some liboskit_kern functions.

[4]See the beginning of Section 2 for a definition of the term "base environment."

2.2.3 Exit code

The L4-specific implementation of _exit gives the user a chance to enter the kernel debugger before rebooting the computer.

2.3 Example applications

To conduct some early tests and help debug liboskit10_support, I took a few example programs from the OSKit distribution and built them without modifications using my library. I can report that the following example programs work:

multiboot: Dumps the Multiboot information and command line to the console

pingreply: Initializes a networking adapter, requests an IP address via BOOTP, and answers to PING requests

socket_com: Initializes a networking adapter, requests an IP address via BOOTP, makes a few FINGER queries to accounts at Utah university, and sets up a DAYTIME server (TCP port 13)

linux_fs_com: Initializes a disk adapter (SCSI or IDE), mounts a file system, and performs a few file-system operations on the disk

To conduct a more integrated test and try the POSIX facilities the OSKit provides, I built a version of Perl for the OSKit (which, by the way, was a stupid idea because the Perl distribution isn't really prepared for cross-compilation; but I succeeded). I haven't tried executing very sophisticated scripts yet, but I was able to run simple scripts accessing the file system and doing a FINGER request over the network.

In the future, I hope to extend the Perl port into a sophisticated shell and scripting facility for L4. Before that can happen, though, some of the OSKit components should move to their own servers in separate tasks, and native threading capabilities need to be added to Perl (Perl already comes with a thread interface) as well as an extension to access L4 system calls directly.

2.4 Limitations

The liboskit10_support library currently has a few flaws:

- It assumes that the application's pager is a Sigma0 server because physical memory must be mapped one-to-one into the virtual address space. Please note that Rmgr supports the Sigma0 protocol (i. e., Rmgr is a Sigma0 server), so Rmgr clients are supported.
- Currently, the memory-pool initialization routine just grabs all the physical memory it can get from the task's Sigma0 server (usually Rmgr). This is a problem if two or more OSKit application run at the same time. The

current workaround is to restrict the resources of OSKit applications by configuring Rmgr accordingly.

There is hope that all of these problems can easily be fixed.

Another shortcoming is related to the build process of OSKit applications: Overriding functions from the OSKit unfortunately turned out to be more difficult that it seemed at first; it is not as easy as specifying `liboskit10_support` as the first library on the linker's command line.

The problem arises when a symbol is first used from within one of the OSKit libraries *and* that library also contains an implementation for that symbol. In this case the implementation in that library is preferred rather than the definition in `liboskit10_support`.

I have worked around the problem using a compile-time kludge (please skip this paragraph if you're faint-hearted): I created header files for the modules I have overridden; these header files have the same name as the original OSKit header files. They contain some inline-assembly code that references a symbol only defined in my version of the module, thereby unconditionally sucking in the module in question. Also, they use the GNU C preprocessor's "`#include_next <...>`" to include the "real" header file.

3 Building L4 servers from OSKit components

3.1 Component Object Model

The OSKit uses the Component Object Model (COM) as defined by Microsoft as its component infrastructure. Most components such as device drivers, file systems, and an IP networking stack are packaged as self-contained COM components; dependencies on other components are modeled as COM references.

COM is a binary-interface standard that allows heterogeneous components to seamlessly interoperate. It is referred to as a binary-interface standard because is allows one component to reuse another component without the need of the second component's source code. [7]

Microsoft has defined standard ways for COM components to interact across address-space boundaries or even across machine boundaries, collectively called Distributed COM (DCOM). These techniques involve a fair bit of infrastructure, none of which the OSKit currently provides. Among other things, a component registry, a runtime library that accesses the component registry, and a COM-aware stub generator are missing. I'll discuss these three missing pieces in order.

A component registry (called Service Control Manager, SCM, on Microsoft systems) is a system-wide registry that, given an interface id, locates, starts up, and returns references to COM components implementing that interface (often called servers). A complementary runtime library provides clients with a convenient programming interface to connect to servers for a given interface.

The OSKit has some support for component registration, but it only supports finding components that are already available in the same address space.

On Microsoft systems, a program named MIDL generates server and client stubs from a COM interface description written in a COM-specific interface-description language. MIDL not only generates stub code to marshal and unmarshal data; it also generates code for a substitute component living in the client's address space that implements the same binary interface as the server and delegates all requests to the server (the "proxy"); therefore, the binary interface a client sees is independent of whether the component is located in the same or a different address space. Also, MIDL supports marshaling and unmarshaling COM references ("OBJREF;" when the automatically-generated stub receives such a reference, it actually creates a proxy object in its own address space and passes a reference to this proxy); that is, references to COM components can be passed between address spaces, allowing for delegation of communication and for connection brokerage.

For the OSKit, none of this functionality is easily available. The stub generator used for L4-based systems, Flick[5] [1], does not support DCOM directly. To support access to out-of-process COM servers, Flick would have to be extended to provide proxy components that make available the COM component's binary interface in the client's address space, and to marshal and unmarshal COM references into proxy components on the receiver's side. The latter is required because passing a pointer to a COM component is an idiom used often in many OSKit components.

I believe that in practice, DCOM-compliant communication between separate address spaces using OSKit components has little use on L4 systems, for two reasons: First, OSKit components don't use the L4-specific data types most applicable for exchanging large amounts of data between L4 tasks, flexpages and string references. Second, DCOM doesn't support marshaling these data types anyway.

Instead, I suggest a different method for making the OSKit's COM components interact across address spaces. I discuss this technique in the next subsection.

3.2 L4-specific components

Given L4's background as a base for real-time systems [4] and as a test-bed for performance-oriented design of microkernel-based systems [5], L4 components should be designed with an emphasis for performance. When components run in separate address spaces, the communication channel between the address spaces can easily become a performance bottleneck. That's why, L4 components should be designed interface-centric; that is, the components' IPC in-

[5] Flick, like the OSKit, has its origin at Utah University. Volkmar Uhlig has extended Flick to support stub generation for L4 IPC.

terface (or "wire protocol") should receive special consideration. This is true for all components, not only for OSKit-derived ones.

If the objective is to put two interacting OSKit COM components in separate L4 address spaces, I suggest the following technique:

1. Develop a wire protocol for the communication between the components from the COM interface.

2. Write an interface description in Flick IDL. Generate client and server stubs using Flick.

3. Develop a COM proxy for the client and a server stub for the component that use the Flick-generated stubs generated in the previous step.

References to components can easily be modeled as L4 thread ids for servers implementing the component. Depending on the generality intended for the glue code developed in step 3, it may be desirable to implement a small runtime library that automatically handles the creation of new server threads when a component reference is passed, but this probably is unnecessary for many applications.

The operating-systems group at Utah university apparently has used a similar design approach. They have developed an extensive runtime library that supports building components for their Fluke microkernel.[6]

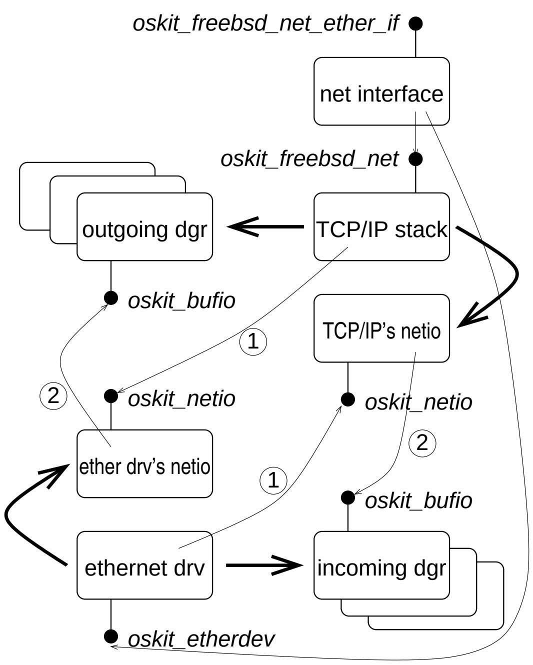
A technique similar to the one I described in this section can also be used to build from OSKit components new L4 components that do not have COM clients. In that case, clients can be programmed directly towards the Flick-generated stub code.

3.3 Example: An Ethernet-driver component

As a test of the feasibility of building L4 servers from OSKit components, I conducted a non-trivial experiment: I separated the OSKit's TCP/IP stack from the Ethernet drivers and put them into separate address spaces.

Figure 1 shows the scenario I started from: Both the TCP/IP stack and the Ethernet driver (or Ethernet drivers) live in the same address space. They exchange data by passing references to buffer objects of type *bufio;* stack and driver are a source of such buffers, and the *netio* objects they created act as a sink.

When driver and network stack live in separate address spaces, passing memory references to *bufio*s between them must be replaced by a different mechanism. The COM architecture makes this possible without modifying the source code of either component: Instead of using the other end's *netio* as a sink for *bufio*s, the components use a local proxy *netio* object with the same interface. This *netio* implementation consumes the *bufio* and sends the data via L4 IPC to a



Legend: Boxes represent components, black dots represent COM interfaces a component exports, fat arrows represent the "creates" relationship, thin arrows represent the "references" relationship.

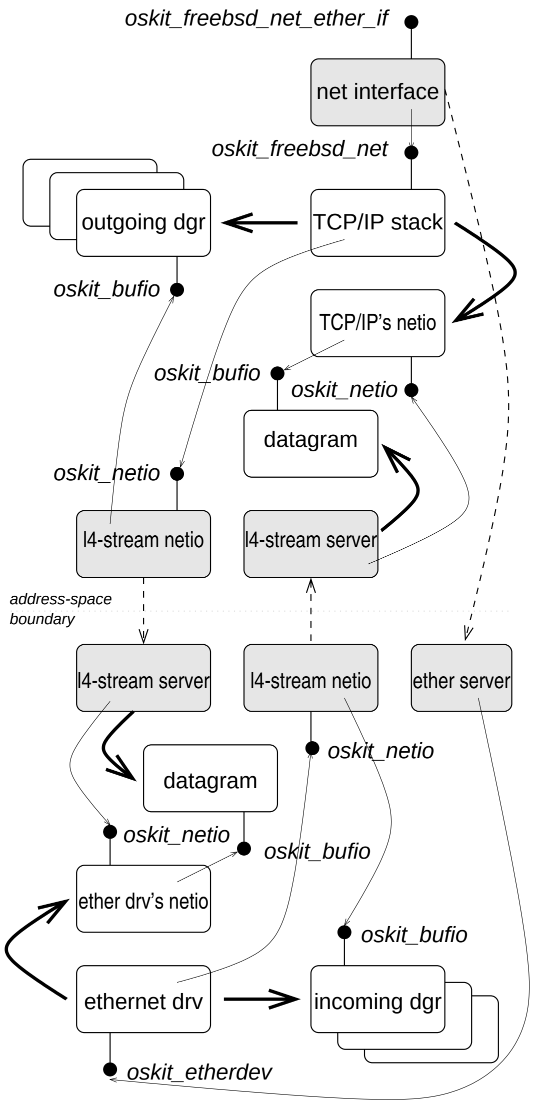
Figure 1 Default networking scenario in OSKit. Both the TCP/IP stack and the Ethernet driver create a *netio* object that serve as communication endpoints. References to *netio* objects are exchanged during initialization (1). Later, both ends exchange data using buffer objects (of type *bufio*). They create buffers and pass references to these buffers (2) to the other end's *netio* object which acts as a consumer for the buffers.

server waiting at the other end. In the other address space, this server constructs a new *bufio* object and passes it to the receiver's *netio* (Figure 2).

Figure 1 and Figure 2 also show an *freebsd_net_ether_if* object that initializes the networking code. To do that, it needs access to both the TCP/IP stack and the Ethernet driver because the networking card's hardware address needs to be passed to the TCP/IP stack to initialize the stack's Ethernet-address–to–IP-address translation table (the so-called ARP table, from "address resolution protocol"). To facilitate this object's access to the Ethernet driver in the separate-tasks scenario (Figure 2), a second server runs in the Ethernet driver's task.

My implementation of the L4-IPC–based *netio* object is very simple: It just copies the data using indirect-string IPC. This is OK for this proof-on-concept implementation, but in the future, a more sophisticated implementation using a flexpage-mapped shared-memory buffer might be more suitable because such an implementation would remove one memory copy for all data.

[6]Personal communication with Jay Lepreau, email message, November 1999



Legend: Boxes, dots, and nondashed arrows correspond to Figure 1. Dashed arrows denote L4 IPC. Shaded boxes represent handwritten glue code and Flick-generated stub code.

Figure 2 Networking with an Ethernet-driver component running in a separate L4 task. The datagrams produced by TCP/IP stack and Ethernet driver now are passed to proxy *netio*s that forward the data to a server running in the other address space. That server creates a new buffer and passed that to the receiver's *netio.*

I used the socket_com program I already described in Section 2.3 as a test application for the TCP/IP stack. The changes required to socket_com to make it use the Ethernet stack running in a separate task were trivial. I needed two days to implement and debug the glue code for the driver and the networking stack (shaded boxes in Figure 2).

4 Building shared libraries from OSKit components

The OSKit contains two sets of libraries: function libraries and COM-component libraries. Both types of libraries are attractive candidates for use as dynamically-loaded shared libraries.

As with out-of-process COM servers, discussed in Section 3.1, Microsoft has defined standard methods of invoking COM components located in shared libraries, also referred to as in-process servers. Again, these techniques rely on standard infrastructure—in this case, a runtime library that looks for the library implementing a particular interface and loads it into the client's address space—, and again, the OSKit doesn't provide that infrastructure. Fortunately, however, once a library has been loaded dynamically, accessing a COM component in it is as easy as looking up the address of the object's function table. That's why, in this section I will only take a short look on shared libraries in general and leave out COM specifics.

The OSKit does not yet support an executable interpreter like Unix's `ld.so` that transparently loads and links dynamic libraries during program startup. In other words, dynamically-linked programs are not supported. However, the OSKit does contain a runtime–link-editor library `liboskit_rtld` supporting the `dlopen(3)` family of functions for explicitly loading and linking shared libraries.

The `liboskit_rtld` library depends on `open`, `read`, and friends—functions that are parts of the OSKit's POSIX-emulation library which in turn depends a working file system abstraction. The OSKit comes with disk drivers, partition-table interpreters, disk file systems, and a library maintaining a file name space and making it possible to "mount" file systems. Using these facilities, it is possible load and link dynamic libraries at run time, as a small demonstration program included with the OSKit demonstrates—even though it currently is somewhat awkward to build such programs because the compiler tool chain doesn't support very well statically-linked programs wishing to link dynamic libraries at runtime. This demonstration program also works under L4 using the OSKit adaption presented in Section 2.

In the future, we should extend the OSKit with other file systems like the SawMill project's, NFS, TFTP, or a bridge to L[4]Linux's file systems. Also, we need to conduct tests with dynamic versions of the standard OSKit libraries. Finally, we may want to develop a library that makes it easy to dynamically link in-process COM servers.

5 Conclusion and future work

I have shown that the OSKit provides a powerful base for application and component development. There are still a few gaps to fill and white spots to map, but it can already be said that using the OSKit, it will be easier to write operating-system servers for L4 and to port more applications to the platform.

I want to stress again that I didn't touch some very sensitive spots: First, I have not made any performance measurements. Second, while my device-driver–helper library works, I'm not sure I always chose the best way of doing things; for example, delivering interrupts using `lthread_ex_regs` significantly reduced the complexity in the library itself, but it required modifications to a number of other tools to cope with the possibility of cancelled IPC, so all in all, maybe my design choice has not been a good one. Third, I haven't tried the OSKit's multi-threading support, and I have not used it much in a multi-threaded environment. Finally, it remains to be seen whether OSKit components can be successfully used for real-time applications.

Throughout the paper I mentioned a few weaknesses of the current OSKit port to L4, the most important problems being the ones in Section 2.4. These problems should be dealt with soon; in particular, the device-driver–helper library should be improved so that it supports a larger class of applications, not only Rmgr (or Sigma0) clients.

Other possible directions for making improvements could be:

- Provide a compatibility package for programs still using the old OSKit version 0.60. Many interfaces and type names have changed between 0.60 and 1.0.
- Alternatively, port Fiasco and all DROPS servers to OSKit 1.0.
- Improve the OSKit's multi-platform support. Utah provides a preliminary port for the StrongARM platform, but there is still a long way to go to reach the feature-richness and stability of the x86 support.
- Enhance Flick to support DCOM out of the box.

References

[1] Eric Eide, Jay Lepreau, and James L. Simister. Flexible and optimized IDL compilation for distributed applications. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, volume 1511 of *Lecture Notes in Computer Science*, pages 288–302. Springer, May 1998.

[2] Bryan Ford, Erich S. Boleyn, Kunihiro Ishiguro, and OKUJI Yoshinore. *The Multiboot Specification*. Free Software Foundation.

[3] The Flux Research Group. *The OSKit: The Flux Operating System Toolkit*. University of Utah, Department of Computer Science, July 1999.

[4] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[5] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997. Paper and slides available from URL: `http://os.inf.tu-dresden.de/L4/`.

[6] Michael Hohmuth. *RMGR, The L4 Resource Manager*. TU Dresden, 1998.

[7] Thuan L. Thai. *Learning DCOM*. O'Reilly & Associates, 1999.