FLEX: A Tool for Building Efficient and Flexible Systems

John B. Carter, Bryan Ford, Mike Hibler, Ravindra Kuramkote, Jeffrey Law, Jay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson

University of Utah, Department of Computer Science

Abstract

Modern operating systems must support a wide variety of services for a diverse set of users. Designers of these systems face a tradeoff between functionality and performance. Systems like Mach provide a set of general abstractions and attempt to handle every situation, which can lead to poor performance for common cases. Other systems, such as Unix, provide a small set of abstractions that can be made very efficient, at the expense of functionality. We are implementing a flexible system building tool, FLEX, that allows us to support a powerful operating systems interface efficiently by constructing specialized module implementations at runtime. FLEX improves the performance of *existing* systems by optimizing interprocess communications paths and relocating servers and clients to reduce communications overhead. These facilities improve the performance of Unix system calls on Mach from 20-400%. Furthermore, FLEX can dynamically extend the kernel in a controlled fashion, which gives user programs access to privileged data and devices not envisioned by the original operating system implementor.

1 Overview

Operating systems consist of a collection of inter-related entities that provide a variety of services. Beyond this general description, the specifics of individual operating systems varies widely depending on the decisions made by each system's designers. We believe that there is no "optimal" organization, because the "optimal" design is dependent on a particular system's resources and the needs of its clients. No single operating system structure can perform optimally in a wide variety of environments. The most important aspect of an operating system from the user's point of view is its *application programming interface definition*. As long as the operating system efficiently supports its interface and protection requirements, it is irrelevant whether this is provided via functions found at known locations in the user address space (e.g., MS-DOS), via traps to the kernel (e.g., Unix), via remote procedure calls (RPCs) to server processes (e.g., much of Mach 3.0), or any combination of these options. To exploit the flexibility that this perspective provides, we are developing the FLEX system building tool.

FLEX is a coarse-grained system building service that allows systems to be dynamically constructed using the implementation most appropriate to a given situation. In this context, a "system" is an entity that interacts with the operating system, plus the operating system itself. FLEX provides functionality to an operating system similar to, but more powerful than, that provided to a programming system by a linker. FLEX works by reading *modules*, manipulating them as specified by module manipulation files, and writing the resulting executable either to a user's address space or to a file for later use. The active entity in FLEX is the OMOS server[12], which runs as a privileged process. It performs several functions, including: generating, relocating and binding object files, caching relocated object files for later use, managing address spaces, and interacting with the kernel to add or delete kernel routines on demand. FLEX can improve an operating system in two different ways: (i) by improving the performance of existing system implementations and (ii) by dynamically extending operating system functionality.

One of the goals of FLEX is to allow the implementations of *existing* systems to be improved, without requiring their component modules or interfaces to change. FLEX uses interface definition specifications to define how individual system components interact with one another. This specification allows FLEX to specialize instances of a system resource or communications channel to a particular situation. For example, when a client program first connects to a particular service, FLEX can determine where the server is physically located and provide the most efficient interprocess communications mechanism for this client-server interaction. If the server is on a remote machine, then RPC is appropriate, while if the client and server have been placed in the same address space, a capability of FLEX discussed below, then some form of procedure call is preferable. Similarly, if FLEX is able to determine that most of the interactions with a server are coming from a particular client, it can intervene to make those interactions less expensive (e.g., by mapping the client into the server address space in a controlled manner).

FLEX's other aspect is its ability to extend or modify the operating system dynamically in a (mostly) transparent way. FLEX can modify the operating system on the fly by binding routines to the kernel on a per-client basis. This ability allows users or operating systems designers to add or modify kernel functionality. The means by which this can be done safely is discussed in the following section. This feature of FLEX can be exploited in a number of ways. For example, this might be used to give an unprivileged user program direct control over its physical memory so that it can use a paging algorithm tuned to its particular access behavior or real-time needs. A long term goal of this aspect of FLEX is to allow operating systems researchers and vendors to distribute functional modifications or extensions to existing operating systems to sites without access to the operating system source code.

2 Design of FLEX

FLEX's power comes from OMOS's ability to manipulate modules in complex ways, its full access to interface definitions, its ability to derive specialized implementations of modules and their interactions, and its generalized constraint system for managing address spaces. A module is an entity containing a number of *symbols*, or linkage points, and optionally a fragment of executable code in the machine language of the target system. Most modules processed by FLEX are normal relocatable object files produced by a compiler, but there are also several kinds of interface definition modules. FLEX uses a Lisp-like module manipulation language developed as part of the Jigsaw framework [3] to specify how particular modules are to be combined. This language provides a complete implementation of module name manipulation, much more powerful than existing linkers. Augmented with detailed knowledge of communication mechanisms and corresponding argument semantics, this allows FLEX to transparently specialize inter-module communication.

Within a module, a symbol can either be a *declaration* or a *reference*. Declarations are most often associated with code fragments that *implement* a particular system service, while references are most often associated with code fragments that *invoke* a particular system service. FLEX is able to improve upon existing system implementations by transparently substituting a particular declaration of a symbol with a functionally equivalent implementation that provides the same interface and semantics. The details of the low-level binding mechanism are beyond the scope of this paper, but are described elsewhere [7]. For example, if a client process contains a reference to a system call that normally is implemented as an RPC to a user-level server process, which is how many Unix system calls are emulated in Mach 3.0 [1], it can replace the RPC interface with an LRPC-like interface if it recognizes that the server is always local. LRPC [2] exploits memory sharing and thread migration to reduce the overhead of an RPC, but it is only possible between two processes on the same machine. Implementing the full LRPC mechanism requires modifications to the Mach thread model to support thread migration[6]. Similarly, if the service is currently being provided by a server that has been loaded into the kernel address space, FLEX can replace the RPC with a trap that directly invokes the desired service. These examples demonstrate the potential that FLEX has for improving the quality of *existing* systems.

In addition to being able to specialize interfaces, FLEX is able to dynamically extend the operating system kernel or relocate services in a transparent way. The simplest example of how this can be accomplished arises when FLEX detects that a desired service is not currently being provided, as can occur if the server crashes or chooses to quit due to inactivity. In this case, FLEX can restart the appropriate server. An incremental improvement upon this is that FLEX can start a new instance of the server that is specialized to the client's needs and make the appropriate binding. FLEX goes beyond these simple mechanisms by allowing modules to be relocated between address spaces to provide more efficient inter-module interactions. For example, if FLEX observes or is informed that a particular user-level service is used frequently and the server is trusted, it can bind the service into the kernel and replace calls to that server with traps. This use of FLEX blurs the distinction between macrokernels and microkernels, which we believe should be the case. The microkernel organization improves code modularity and provides an easy to understand design philosophy, but when a rigid enforcement of the microkernel philosophy impacts performance without significant gains, it should be relaxed transparently to regain the performance benefits of a macrokernel system. Enabling optimization of more interactions, FLEX can transparently load clients and trusted servers in a single address space, while protecting the servers from client accesses. We view support for these "in-server clients" and "in-kernel servers" as the first step in an evolution towards a single address space system that can efficiently support existing programs written for existing systems [4].

Finally, we have experimented with the option of linking specially authorized routines into the kernel on a per-client basis. FLEX lets kernel level services and associated trap vectors be added, allowing users to access privileged data and devices in a controlled fashion, not envisioned by the original operating system implementor, and thus not supported by an existing system call. This gives FLEX'd programs the ability to read or modify kernel data structures (e.g., a controlled way to reintroduce the functionality of `/dev/kmem`), the ability to invoke functions not provided by the default kernel (e.g., direct control over its physical memory and paging algorithm), and the ability to override or modify existing kernel services (e.g., interposing a message logging layer in a particular process' IPC routines to support fault tolerance). Routines that interact with existing kernel routines, such as those that modify existing kernel data structures or device drivers, must be written with great care to avoid corrupting the system. Our current authentication mechanism is quite simple – only the OMOS server can install kernel-level routines, and it will install only authorized library routines that are provided by a superuser. However, these routines can be installed on behalf of any process. While simple, this mechanism lets us protect the integrity of the OS kernel without significantly reducing overall flexibility.

Figure 1 illustrates how FLEX can restructure the way that a particular user program interacts with the operating system. In this example, the user process is using four services: a database server, a file server, an authentication server, and a special extension to the kernel that lets the user program control its own physical memory. Because of the frequency of interactions, the user process has been loaded into the database server's address space. Thus, communications between the user process and database server occur via protected procedure calls (involving traps) or directly through normal procedure calls and shared memory, depending on the desired level of protection [4]. Additionally, because this user process needs real-time performance guarantees, it uses its own paging mechanism (implemented as an in-kernel server) to control its paging behavior. This mechanism uses normal kernel-level services to access disk and the VM hardware, but otherwise it is independent of the virtual memory subsystem after it acquires the desired amount of physical memory. Similarly, the fileserver is a user-level process being executed locally, so interactions with it use LRPC to reduce communications overhead. Finally, the system authentication server is running remotely, so all communications with it use standard RPC. FLEX's ability to restructure systems in this way can greatly enhance both performance and flexibility.

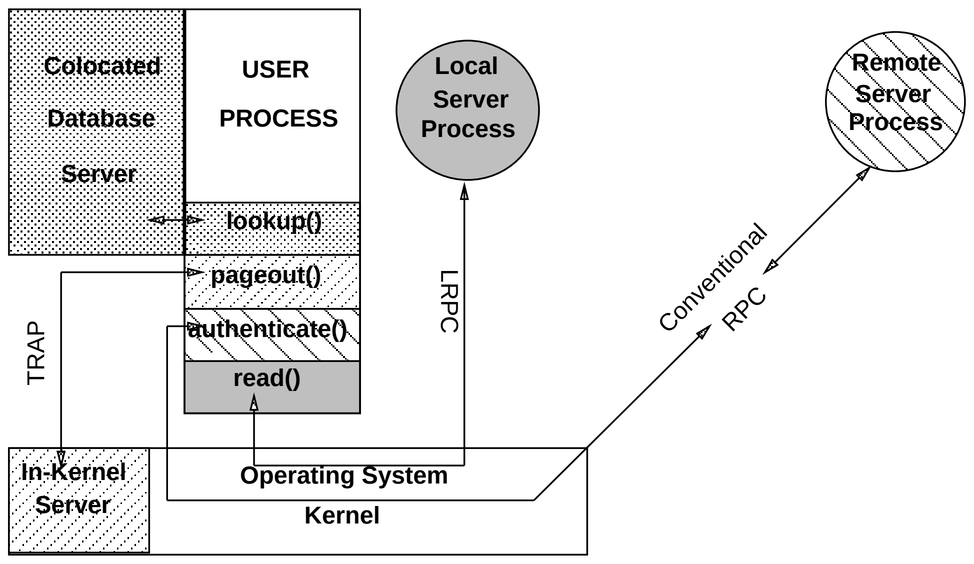

Figure 1 Example of How FLEX Can Restructure Programs

3 Results and Current Status

The core of FLEX is the OMOS server. OMOS has been integrated into the Mach 3.0 OSF/1 server's `exec` function, so it is invoked when FLEX'd programs are executed. As part of the preliminary implementation of FLEX, OMOS has been used to reduce the overhead of communications between mutually-trusting client and server processes by replacing RPC invocations with procedure calls. In addition, OMOS was augmented to support a protected region in a user's address space. This support allows OMOS to load routines representing extended operating system functionality into that region, reconcile unresolved references found in the added modules with entry points defined in the kernel, and provide special gating routines to the user process so that it can safely access the new functionality. Work is continuing on augmenting OMOS's interface definition support to handle many variations of argument semantics used by different communication mechanisms. In addition to its role in FLEX, OMOS has been able to improve the performance of a number of running systems by an average of 10% by optimizing the locality of instruction references performed by the systems, and thereby improving cache and paging behavior[13]. In addition, OMOS is providing a fast and portable shared library service for an 800,000 line CAD system[11], with substantial speedups due to its caching of relocated images.

We have performed three experiments that demonstrate the value of evolving an existing operating system, Mach 3.0 and the OSF/1 Unix server, into a flexibly structured system.

(i) We transparently merged a frequently accessed and trusted user-level server, the Unix server, into the kernel's protection domain, while fully preserving semantics [9]. We refer to this mechanism as in-kernel servers, or INKS. Client calls to the server are bound either to a simple system call interface, or to a full RPC mechanism, depending on the server's location. This optimization reduces the amount of work done copying arguments and executing the RPC control path to handle complex and infrequently encountered message types. We found that RPC performance improves by a factor of three, Unix system calls to the server improve by 20-200%, and the overall performance of large benchmarks improves by 4-13%.

(ii) While performing the INKS work, we discovered that many performance optimizations could be introduced if the Mach kernel supported a full migrating-threads model[6]. We have implemented such a mechanism, fully decoupling threads from tasks. Our prototype implementation of this mechanism has a faster call path than the most optimized message path in the existing Mach kernel.

(iii) In order to take full advantage of the potential of FLEX, protection domains must be decoupled from address spaces. We have prototyped this on Mach and the Unix server, using the HP PA-RISC's fine-grained protection mechanisms to provide protection between tasks in the same address space. This mechanism allows clients to be loaded into servers. We have successfully executed several unmodified user programs in the same address space as the OSF/1 server, while protecting servers from client accesses. Individual system calls demonstrate dramatic speedups, as much as a factor of four, due to the avoidance of microkernel calls to access system call arguments. Even for I/O intensive programs, such as `ls`, the use of in-server client technology reduced the total running time by 10%.

4 Related Work

There are a number of systems that provide various aspects of FLEX, although none supports its full generality of optimizations. Like FLEX, Lipto [5] provides architectural support for modules that is independent of protection domains, and thereby allows communications between modules to be optimized using techniques similar to those described above. However, it was not designed to work with existing system implementations, and currently does not include support for FLEX-like dynamic kernel extensions. Psyche [14] uses module interface information to construct systems composed of modules implemented by radically different operating systems and languages, so-called *multi-model* programming. It does not reconfigure modules for performance purposes. Synthesis [10] generates both specialized OS interfaces and actual operating systems functions, but does not reconfigure processes or move functions between modules. Finally, the VMS operating system [8] gave users the ability to extend the system dynamically by installing specially authorized trap handlers. VMS supported this capability by associating a per-process dispatcher routine with each protected shareable image that was installed and querying the appropriate dispatchers when an unrecognized trap was encountered. FLEX allows references to added routines to be installed directly in a single per-process trap table, because all such installations are performed by OMOS, which will improve the performance of this mechanism[4].

5 Conclusions

FLEX can improve an operating system in two different ways: (i) by improving the performance of existing system implementations and (ii) by dynamically extending operating system functionality. The core of FLEX is the OMOS server. FLEX's power comes from OMOS's ability to manipulate modules in complex ways, its full access to interface definitions, its ability to derive specialized implementations of modules and their interactions, and its generalized constraint system for managing address spaces. A major source of overhead in modern operating systems is the time spent executing general-purpose communications abstractions. FLEX allows systems to be dynamically constructed using the implementation most appropriate to a given situation. Preliminary results demonstrate the value of evolving an existing operating system into a flexibly structured system. By restructuring existing systems, FLEX has been able to improve the performance of Unix system calls on Mach 3.0 from 20-400%. In addition, FLEX allows programs to extend the functionality of the operating system, which gives user programs controlled access to privileged data and devices in ways not anticipated by the original operating system implementor. These results indicate that a dynamic system restructuring tool like FLEX can significantly improve the performance of existing systems and at the same time act as a cornerstone for future system development.

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 *Proceedings of the Summer Usenix Conference*, July 1986.

[2] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[3] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity, and Multiple Inheritance.* PhD thesis, University of Utah, March 1992.

[4] J.B. Carter, A. Cox, D. Johnson, and W. Zwaenepoel. Distributed operating systems based on a protected global virtual address space. In *Third Workshop on Workstation Operating Systems*, May 1992.

[5] P. Druschel, L.L. Peterson, and N.C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 512–520, June 1992.

[6] B. Ford, M. Hibler, and J. Lepreau. Notes on thread models in Mach 3.0. Technical Report UUCS-93-012, University of Utah Computer Science Department, April 1993.

[7] B. Ford, J. Lepreau, and D. Orr. Remote procedure call specialization with the OMOS object server. Technical report, University of Utah, June 1993.

[8] L.J. Kenah, R.E. Goldenberg, and S.F. Bate. *VAX/VMS Internals and Data Structures.* Digital Press, Bedford, Massachusetts, 1988.

[9] J. Lepreau, M. Hibler, B. Ford, and J. Law. In-kernel servers on Mach 3.0: Implementation and performance. In *Proceedings of the Third Usenix Mach Symposium*, pages 39–55, April 1993.

[10] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[11] D.B. Orr, J. Bonn, J. Lepreau, and R. Mecklenburg. Fast and flexible shared libraries. In *Proceedings of the Summer 1993 USENIX Conference*, Cincinnati, OH, Summer 1993. To appear.

[12] D.B. Orr and R. Mecklenburg. OMOS — an object server for program execution. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 200–209, September 1992.

[13] D.B. Orr, R.W. Mecklenburg, P.J. Hoogenboom, and J. Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, pages 232–241, January 1993.

[14] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. Multi-model parallel programming in Psyche. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pages 70–78, March 1990.