Cross-Address Space Dynamic Linking

James Kempf
Peter B. Kessler

SMLI TR-92-2 September 1992

Abstract:

We describe an algorithm and implementation of dynamic linking that allows one user process to link a program in another address space without compromising the security of the other address space and without requiring the linking process to enter kernel mode. The same technique can also be used to load program code into an existing address space, e.g., for debugging or other purposes. The implementation makes extensive use of objects in the Spring object-oriented operating system. We have extracted the dynamic linking function from our operating system, and have made it available to user programs as a replaceable library service. In the process, we have taken advantage of features present in a modern, object-oriented operating system to simplify the dynamic linker.

Categories and Subject Descriptors:

C.2.4 Computer System Organization:
[**Computer-Communications Network**]: Distributed Systems-*Network operating systems*
D.4.9 Software:
[**Operating Systems**]: Systems programs and utilities-*Linkers; Loadersl*

Additional Keywords and Phrases: Spring operating system; Object-oriented programming

Sun Microsystems Laboratories, Inc.
A Sun Microsystems, Inc. Business

M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email address:
james.kempf@sun.com
peter.kessler@sun.com

Cross-Address Space Dynamic Linking

James Kempf and Peter B. Kessler

1. Introduction

Program linking and loading have traditionally been at the boundary between the programming environment and operating system. Linking constructs an executable image from compiled code, relocating code to fixed addresses and resolving any outstanding references between the separately compiled parts. Loading maps the executable image into an address space and starts a thread of execution running. If all the references in the executable image have been resolved prior to a request to load the code and begin execution, the image is said to be *statically linked*. If, however, some amount of linking is required as a part of the loading operation, then the program is said to be *dynamically linked*. For example, the runtime libraries that come with a programming environment are often provided as dynamically linked libraries.

Dynamic linking has several advantages. Dynamically linked program image files on disk are smaller than statically linked program images since library code is not copied into each program image file as happens during static linking. The time required to link the program becomes less for the same reason, speeding up the edit-compile-link loop. The sizes of program images in memory can also be smaller if the executable code of dynamically linked libraries can be shared between address spaces. Smaller program images facilitate better cache and virtual memory utilization.

Dynamic linking provides the opportunity to define alternative implementations of an application binary interface (ABI) for the program execution environment. Dynamically linkable libraries that conform to the ABI can be substituted as late as program startup time. Developers of services can deliver alternative implementations of libraries as “shrink-wrapped” components, reducing the impact of a library upgrade on client code. Similarly, applications can be delivered shrink-wrapped, to be linked against a library implementation for a particular target environment.

Dynamic linking does have certain costs. References to externally defined functions have to be resolved, though many resolutions can be deferred until the functions are called. Resolving references that cannot be deferred lengthens the time required to start a program. Careful implementation of position independent code can reduce this additional time to one relocation per externally referenced symbol per library, e.g., by transferring to the library through a transfer table. Function invocation in position independent code usually involves such an indirection through a jump table for global function calls. The indirection lengthens the amount of time required to call a function. However, the additional over-

head is not more than is required for statically typed object-oriented languages such as C++ [Stroustrup 91]. Dynamic linking often has a system cost: traditional implementation support for dynamic linking has been part of the operating system kernel, complicating system software [Organick 72].

In this paper, we describe an algorithm for dynamic linking and program loading implemented entirely outside the kernel. The algorithm features the ability to construct a program image from outside the address space in which the program will run. Additionally, the algorithm supports linking code into a running process from outside the address space of that process. Our implementation runs within the context of the experimental Spring object-oriented operating system.

We first review previous work in the area of dynamic linking. We then briefly describe the features of the Spring operating system that make cross-address space dynamic linking possible. The algorithm for cross-address space dynamic linking is discussed, and information on the implementation is presented. We conclude the paper with a brief discussion on how the Spring dynamic linking architecture could enable novel features in an object-oriented program development and execution environment.

2. Previous Work

Dynamic linking was originally part of the Multics operating system [Organick 72] and TENEX [Murphy 72]. These early implementations required the operating system to enter kernel mode to perform dynamic linking since it was part of the program loading process. Because dynamic linking was perceived to introduce excessive overhead into program loading and because the implementation complicated the memory management system in the kernel, dynamic linking was dropped from successors. As the set of base library functionality increased, dynamic linking reemerged to avoid the problems of increased program image size. Dynamic linking was reintroduced in UNIX* System V operating system kernel as part of the UNIX `exec()` program loader [Arnold 86]. The original UNIX System V implementation required libraries to be placed at specific addresses when the library was configured, rather than when the program began executing.

Dynamic linking was made available to user processes by SunOS 4.0 [Gingell 87]. In that implementation, the main module of a program dynamically links shared libraries with itself after the program loader has started an execution thread running within the address space. As a result, the dynamic linker does not run in kernel mode. A disadvantage is that the standard libraries are not available to the dynamic linker itself, so it runs in a restricted runtime environment where such services as memory allocation must be specially provided. Libraries are mapped into the address space of the process using a kernel memory map call, rather than being at fixed addresses. For libraries that are compiled into position independent code, code segments are sharable between processes. The Berkeley 4.4 [Seeley 90] implementation and the UNIX System V Release 4 implementation

* UNIX is a trademark of UNIX Systems Laboratories.

[SVR4 91] follow substantially the same strategy. The SunOS 4.0 dynamic linker provides an application program interface (API) for processes to link shared libraries during execution of user code and to look up symbols in the linked libraries and main program.

The Xerox Portable Common Runtime (PCR) [Weiser 89] similarly provides dynamic loading and linking outside of the kernel. Code can be loaded only into the address space in which the loading thread runs. Provision is made to execute code from the loaded module to perform language-specific linking. For example, this facility is used for link-time type checking of imported items and registration of exported implementations in the Cedar programming environment [Swinehart 86]. For the symbol formats it understands, PCR maintains a symbol table that can be queried programmatically. The linking code in each module can maintain language-specific symbol tables. The PCR dynamic linker does not support sharing of code between address spaces.

The public domain dynamic linker `dld` is designed to bring the linking semantics of languages such as Lisp to conventional languages such as C [Ho 91]. `dld` allows a program to link standard relocatable files as generated by the compiler. An API contains operations for obtaining function symbols from the library and determining if all references from a function are resolved. `dld` does not attempt to address system issues such as the packaging of modules as libraries or the sharing of code between address spaces.

Dynamic linking between address spaces of running processes is provided by OSF/1 [Allen 91], but only if the program doing the linking is running in kernel mode and the code is being linked into a target address space running in kernel mode. It is primarily designed for linking device drivers and other kernel modules into kernel mode servers. Dynamic linking during program loading in OSF/1 is implemented as part of the kernel program loader, and requires that shared libraries be installed at fixed addresses, as in the original UNIX System V implementation. As in SunOS 4.0, Berkeley 4.4, and UNIX System V Release 4, an API is provided for user-level processes to link code modules into themselves, and to look up symbols. The OSF/1 program loader can additionally be extended to handle multiple code formats by dynamically linking format handlers into the kernel.

3. The Spring Runtime Environment

Spring is an experimental distributed operating system designed around a microkernel architecture [Rozier 88] and a cache-coherent, network virtual memory system [Ramachandrand 91]. There are no user-visible kernel calls. The interfaces for services traditionally implemented as kernel calls are specified in an object-oriented interface definition language that supports multiple inheritance, similar to IDL [OMG 92]. An application can request the creation of an object, invoke methods from the interface of the object, and pass an object as a parameter to other invocations, without regard to the location of the object's implementation. Invocations of methods on an object are made via client-side stubs that perform remote procedure calls, if necessary, to the server-based implementation. The stubs are

compiled independently of any application and can be dynamically linked to any client that needs them. Since the stubs for different objects can come from different libraries, Spring relies heavily on dynamic linking.

The three fundamental interfaces used in dynamic linking are the domain interface, the name server interface, and the virtual memory interface. A domain in Spring is similar to a process in UNIX: it comprises an address space and a collection of execution threads. One of the operations available on a domain object returns an object that represents the address space of the domain. Note that operations *upon* domain and address space objects do not require execution threads to run *within* the domain or address space the object represents. The execution threads for these operations run in the address spaces for the domain manager or virtual memory manager. The virtual memory interface defines a memory object that can be mapped into one or more address spaces, possibly at a different address in each address space. Files, for example, are a subclass of memory object. The name service interface is used to look up files by name in a domain's name space. By convention, each domain starts execution with a standard collection of files and objects available within its name space.

The Spring architecture rules out implementing dynamic linking either as part of the kernel or by having the child bootstrap itself, as in SunOS 4.1. Implementing dynamic linking in the traditional manner, as part of the kernel, was explicitly ruled out by the desire to keep the kernel small. Spring's micro-kernel architecture requires that the domain interface, the virtual memory system (including the file system), and the name service—the system components upon which the dynamic linker depends—be implemented outside the kernel. Having the child process bootstrap itself by first linking a special dynamic linker, as is done in SunOS 4.1, would require that those interfaces be supported as part of the dynamic linker module itself. Since a major part of the Spring runtime library is devoted to implementing those interfaces, the net effect would be to require a substantial fraction of the Spring runtime library to be handled specially, instead of as a typical shared library. User mode, cross-address space dynamic linking thus fell out naturally as part of the overall Spring architecture.

Objects in Spring are somewhat like capabilities, in that possession of an object implies the right to invoke operations on it [Fabry 74]. The server may, however, insist that a client authenticate itself with a trusted, third-party authentication server before allowing invocations to succeed [Burrows 90]. Both domain and address space objects inherit from the authenticated object class. Thus, clients of an address space object or a domain object must authenticate themselves to the corresponding object manager before invocations can successfully complete. The result is that clients can perform operations on such fundamental system objects as domains and address spaces without compromising overall system security.

4. Cross-Address Space Dynamic Linking at Program Launch

The entire loading, linking, and launching sequence occurs within the parent process in user mode, using system services supplied by the name service, domain

manager and virtual memory manager interfaces. Loading is accomplished by mapping the code objects into the target address space. The algorithm for loading the pieces of a program into an address space, dynamically linking those pieces together, and starting the program running is illustrated in Figure 1 and described below. The numbers in Figure 1 are keyed to the following steps (returned results are indicated by the loop glyph around the return path):

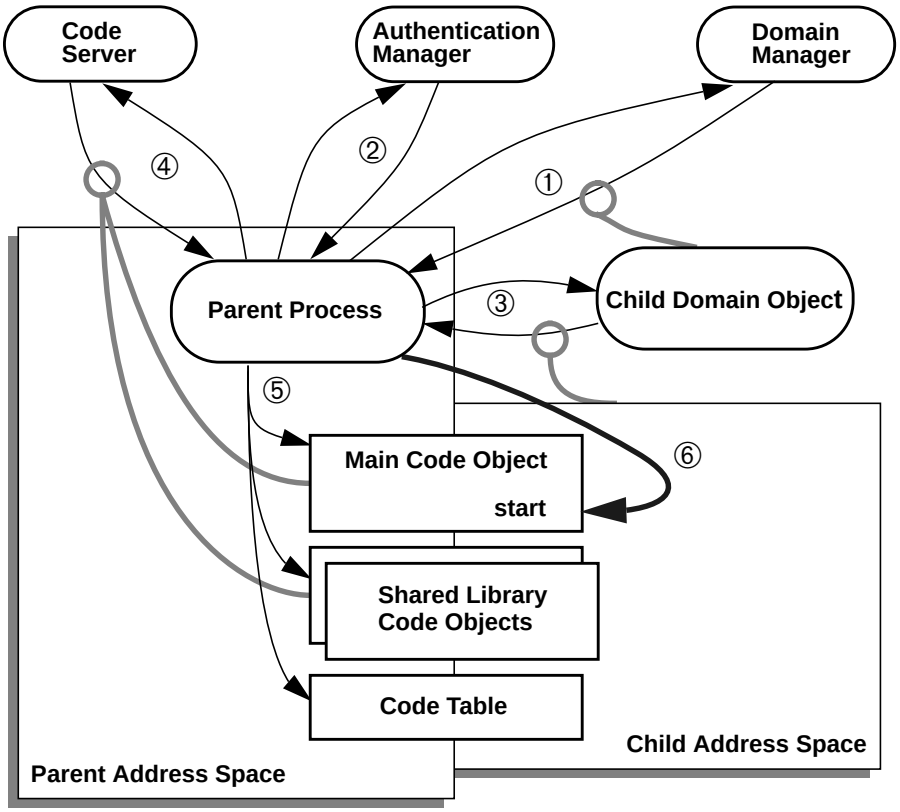

Figure 1. Cross-Address Space Dynamic Linking During Program Loading.

1. The parent process obtains a new child domain object from the domain manager. No threads are running in the child domain, and its address space contains no code at this point.
2. To operate upon the child domain, the parent process goes through an authentication exchange with a trusted, third-party authentication manager.
3. The parent process obtains the address space object for the child domain from the child domain object.
4. From the code server, the parent process obtains code objects for the main program of the child and any shared libraries needed by that program.
5. The code objects are mapped into the address space of the parent process. The code objects are also mapped into the address space of the child. The memory holding each code object is thus shared between the parent and

the child. The code objects need not be mapped to the same addresses in the two address spaces. The code objects are then linked together in the address space of the parent, resolving unresolved references relative to the address space of the child. The parent builds a code table in the shared memory giving the locations of dynamically loaded code in the address space of the child.

6. The parent process obtains an entry point object from the child domain object, passing in the entry point address in the child. The parent creates an execution thread which invokes an operation on the entry point object. This invocation causes the execution thread to enter the child domain, starting the child process running.

5. Cross-Address Space Linking Between Executing Programs

A similar procedure can be used to insert code into an address space in which threads are already executing. The algorithm, shown in Figure 2, allows a controlling process to link code into a target address space without requiring an execution thread to enter the target address space until after the loading and linking process is finished. This property is important for debuggers and other tools dealing with programs that are defective or otherwise unable to complete a remote procedure call. A program linking code into itself is a special case of linking code into an address space in which code is executing.

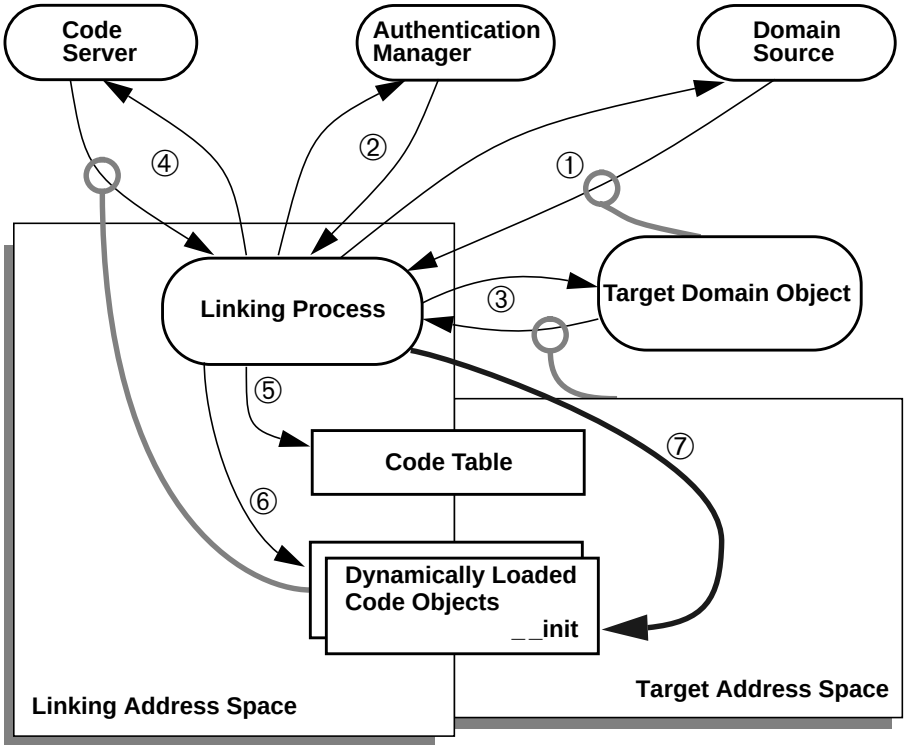

Figure 2. Cross-Address Space Dynamic Linking During Program Execution.

The steps by which cross address space linking occurs into an already executing program are detailed below. The numbers in Figure 2 are keyed to the following steps:

1. The linking process obtains the target domain object. The target domain may be the one in which the process itself is running, or it may be obtained in some other manner.
2. An authentication exchange may be required before the linking process can manipulate the target domain.
3. The linking process obtains the target address space from the target domain.
4. The linking process obtains the code to be linked into the target address space from the code server.
5. The linking process maps the code table from the target address space into its own address space. The code table contains the list of the base addresses and sizes of already linked code files. The linking process locks the code table to exclude any other processes from linking into this address space. The already linked code files are mapped from the target address space into the address space of the linking process.
6. The new code is mapped into the address space of the linking process. The new code is also mapped into the target address space, making the memory shared. The linking process resolves references relative to the code table from the target address space. It then adds an entry in the code table describing the newly linked code. After the code table is updated, the linking process unlocks the code table.
7. If the new code needs initialization, the linking process obtains an entry point object with the address of the initialization function in the target address space. The linking process then starts a thread to call the entry point object to initialize the new code.

6. The Implementation

The dynamic linker has been implemented as part of a standard Spring services library (equivalent of `libc` on UNIX). The implementation is in C++, and currently targets SunOS 4.1 code format. As a consequence of the cross-domain linking algorithm, the dynamic linker runs in the same runtime environment available to every Spring program. To bootstrap the dynamic linker, one program has the dynamic linker statically linked. From there, the dynamic linker is itself dynamically linked along with the other Spring libraries.

Like most of the rest of Spring, the dynamic linker is a replaceable, library level module requiring no special runtime environment. The implementation of the dynamic linker itself cleanly factors loading from linking, and separates policy from mechanism. Loading mechanisms dependent on the format of the underlying code file are separated from the linking level via a C++ abstract base class interface [Meyer 88] for code files, with specific code formats supported by a set

of concrete derived classes. The loader is extended to use a new code format by implementing a new set of derived classes, and bundling them into an application shared library. Although it currently only supports SunOS 4.1 format, the framework is designed to support multiple code formats, similar to OSF/1 [Allen 91]. Unlike OSF/1, however, the Spring dynamic linker does not require the kernel to dynamically link code to support the new format. Similarly, linking policy is separated from fundamental linking mechanisms by a set of abstract base classes. These base classes provide the interface for name resolution of code files and symbol lookup policy. The linker mechanism only deals with the base class interfaces. The concrete derived classes implementing the current default resolution policies mirror those of SunOS 4.1, but the these can be changed by implementing a new set of concrete classes.

7. Applications

Dynamic linking in Spring is used in much the same way as it is in other programming environments, namely for program launch and programmatic dynamic linking. Creation of shared libraries is not restricted to the standard system services library. Clients can create dynamically linked shared libraries if they choose. Spring libraries are compiled into position independent code, and linked into a SunOS 4.1 dynamic link format shared library (`.so` file). Since Spring libraries are position-independent, their code segments are sharable among Spring processes. The large number of client-side stubs in real Spring programs makes static linking unattractive. Dynamic linking amortizes the stub space overhead across all client processes, rendering the overhead acceptable.

Our dynamic linking architecture considerably facilitated implementation of UNIX binary compatibility. Because most programs on SunOS 4.1 are dynamically linked against the `libc` library, binary compatibility is provided by implementing a shared library containing the standard UNIX `libc` functions, but with the system calls implemented by calls on Spring services. UNIX binaries run on Spring are dynamically linked during program startup with this `libc` emulation shared library and any other shared libraries they require. Since no emulation of operating system traps is required, no kernel mode code needed to be written. A similar implementation strategy could be used for emulating other operating systems which support dynamic linking of any libraries containing system calls.

We are considering a number of innovative uses of cross-address space linking in other programming environment tools. Use of fast breakpoints [Kessler 90] would be simplified by using cross-address space dynamic linking to link in the breakpoint code. In fact, Spring has no special operating system trap for breakpoints. In a Spring debugger that uses fast breakpoints, the code for the fast breakpoints would be dynamically patched by the debugger into the address space of the program being debugged. Since the debugger requires no cooperation from the process being debugged, even programs that are not functional could be patched and examined. The same mechanism could be used to install code for tracing and profiling.

Another possible use of cross-address space dynamic linking would be to upgrade running programs without requiring them to be halted [Bloom 83]. In a network of long-running server processes, it may be impossible to bring down the entire network and reboot in order to upgrade a shared library module. Reasons for upgrading libraries include fixing defects, improving efficiency, changing the library interface, and adding supporting functional components. While not all of these can be handled by cross-domain linking alone, with the addition of some extra framework, the basis for a viable on-the-fly library upgrade service could be developed.

8. Summary

We have described an algorithm and implementation for cross-address space dynamic linking. We have extracted what is ordinarily a kernel service and made it a replaceable, user-level library. Our technique allows ordinary programs to load and link code into other address spaces, and to start threads running in those address spaces without entering kernel mode. We show how the security of address spaces during dynamic linking, normally the responsibility of the kernel, can be delegated to an external authentication service. The implementation cleanly separates policy from mechanism, using object-oriented mechanisms to provide interfaces for different code formats and different symbol and name resolution policies.

9. Acknowledgments

Thanks go to Graham Hamilton, technical leader of the Spring operating system development team, for his initial implementation of program loading and the Spring micro-kernel; Yousef Khalidi, for the Spring virtual memory implementation; and Mike Nelson for the Spring remote file system. Yousef and Mike were also the chief implementors of Spring UNIX emulation. Rob Gingell is the original developer of SunOS 4.1 dynamic linking code format and the SunOS 4.1 position-independent code, upon which the Spring dynamic linker is based.

10. References

[Allen 91] L. W. Allen, H. G. Singh, K. G. Wallace, and M. B. Weaver, “Program Loading in OSF / 1,” *Proceedings of the USENIX Winter 1991 Conference*, 1991, pp. 145-160.

[Arnold 86] J. Q. Arnold, “Shared Libraries on UNIX System V,” *Summer Conference Proceedings*, USENIX Association, 1986, pp. 395-404.

[Bloom 83] T. Bloom, “Dynamic Module Replacement in a Distributed Programming System,” MIT Laboratory for Computer Science Technical Report TR-303, Cambridge, MA, 1983.

[Burrows 90] M. Burrows, M. Abadi, and R. Needham, “A Logic of Authentication,” *ACM Transactions on Computer Systems*, 8(1), 1990, pp. 18-36.

[Fabry 74] R. S. Fabry, "Capability-Based Addressing," *Communications of the ACM*, Vol. 17, July 1974, pp. 403-412.

[Gingell 87] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks, "Shared Libraries in SunOS," *Proceedings of USENIX Summer Conference*, 1987, pp. 131-145.

[Ho 91] W. W. Ho, and R. Olsson, "An Approach to Genuine Dynamic Linking," *Software-Practice and Experience*, 21(4), 1991, pp. 375-390.

[Kessler 90] P. B. Kessler, "Fast Breakpoints: Design and Implementation," *Proceedings of the SIGPLAN '90 Programming Language Design and Implementation Conference*, SIGPLAN Notices, 25(6), 1990, pp. 78-84.

[Meyer 88] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, London, 1988.

[Murphy 72] D. L. Murphy, "Storage organization and management in TENIX," *Proceedings of the Fall Joint Computer Conference*, AFIPS, 1972.

[OMG 92] Object Management Group, "The Common Object Request Broker: Architecture and Specification," OMG Document Number 91.12.1, Object Management Group, 1991.

[Organick 72] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, 1972.

[Ramachandrand 91] U. Ramachandrand, and Y. Khalidi, "An Implementation of Distributed Shared Memory," *Software—Practice and Experience*, 21(5), 1991, pp. 443-464.

[Rozier 88] M. Rozier, *et al.*, "Chorus Distributed Operating Systems," *Computing Systems*, 1(4), 1988, pp. 305-367.

[Seeley 90] D. Seeley, "Shared Libraries as Objects", *Proceedings of the Summer Conference*, USENIX Association, 1990, pp. 25-37.

[Stroustrup 91] B. Stroustrup, *The C++ Programming Language*, Second Edition, Addison-Wesley, Reading, MA, 1991.

[SVR4 91] *System V Application Binary Interface*, Prentice-Hall, Inglewood Cliffs, NJ, 1991.

[Swinehart 86] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann, "A Structural View of the Cedar Programming Environment," *ACM Transactions on Programming Languages and Systems*, 8(4), 1986, pp. 419-490.

[Weiser 89] M. Weiser, A. Demers, and C. Hauser, "The Portable Common Runtime Approach to Interoperability," *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, Operating Systems Review, 23(5), December 1989, pp. 114-121.