Practical software reuse for IBM System z I/O subsystems

A. M. Webb
R. Mansell
J. W. Knight
S. J. Greenspan
D. B. Emmes

The design and implementation of the z/VM® SCSI (Small Computer System Interface) I/O subsystem is described. z/VM is an operating system that provides multiple virtual IBM System z™ machines on a single IBM System z computer. The approach adopted herein allows the reuse of entire device drivers from AIX 5L™, a completely different operating system, essentially unchanged. AIX 5L is the IBM UNIX® operating system for the IBM System p™ platform. The design, and much of the implemented code that allows the incorporation of such "foreign" device drivers, is independent of both z/VM and AIX 5L and could potentially be used in other operating system environments.

Introduction

The potential benefits of software reuse are generally accepted in both industry and academia [1, 2]. Experience with technology suggests that there is rarely any benefit in "reinventing the wheel," and the risks of ignoring past technological approaches are well known [3]. Reuse can take many forms, including the reuse of computer programs, frameworks, and integrated development environments. Reuse may reduce costs, increase reliability, and accelerate the evolution of software. The growing number of object-oriented languages (such as C++ [4], Eiffel** [5], Java** [6], Smalltalk** [7], and C#** [8]) and object-oriented design and development environments (such as Rational Rose* [9]) attest to the importance of software reuse. All of these reflect examples of software that may be written with reuse in mind.

This paper concerns the reuse of software in a realm where neither the software nor its environment were actually architected or implemented with reuse in mind. The technology described offers a flexible and efficient methodology for the migration of operating system (OS) extensions, such as device drivers, from one operating environment to another. This methodology thus allows the sharing of a single implementation of appropriate operating system functions across multiple heterogeneous systems. It should be noted that this approach is not an architecture for the development of platform-independent OS functions; rather, it is a way to exploit existing OS functions in an environment other than that for which they were originally developed.

The efficacy of the technology is demonstrated in a practical manner by the successful integration, with minimal change, of device drivers from the AIX 5L* operating system in z/VM*, an IBM System z* operating system product. The success of this approach is significant because the two operating systems are very different in purpose and implementation. AIX 5L (hereafter referred to simply as AIX*) is the IBM proprietary UNIX** operating system for the IBM System p* and provides a rich application development and execution environment. The primary function of z/VM is to provide a virtual System z environment for other operating systems.

The methodology described in this paper offers significant benefits in terms of programmer productivity and allows developers to preserve the integration of reliability, availability, and serviceability (RAS) models. Overall programming error rates are reduced. The approach does present some interesting challenges with respect to performance and scalability. However, because much more code is potentially portable than was designed to be so, this methodology makes it possible to implement many capabilities that would otherwise be prohibitively expensive to introduce in a way that is responsive to market demands.

The technology is based on the concept of an architected execution container that preserves the

©**Copyright** 2007 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8646/07/$5.00 © 2007 IBM

reliability and robustness attributes expected of System z operating systems (for example, recovery from runtime errors) while implementing an execution environment that provides the data and programming interfaces expected by the migrated code. The AIX device drivers are incorporated into the host operating system, z/VM, as components in the execution container. As described below, the execution container provides a framework for combining multiple components in order to implement a particular function, in this case, access to Small Computer System Interface (SCSI) I/O.

The remainder of this paper is organized as follows. First, some background information is presented to briefly introduce the IBM System z, the AIX operating system, and their associated I/O models. Next, we discuss the design of the execution container and the way in which it may be tailored to meet particular migration needs. We also discuss the porting of AIX device drivers so that they run in the execution container, the integration of the execution container and device drivers into z/VM, and the deployment of the finished product. Included in this section is a discussion of the small changes made to the device drivers and why these changes are preferable to previously proposed alternative approaches for producing production-quality multiplatform device drivers. Also discussed are compiler and other environmental requirements and how these requirements were satisfied for the z/VM product. Finally, results are summarized and conclusions presented.

Background

A brief introduction to IBM System z

The IBM System z continues a succession of computing systems that started with the IBM System/360* introduced in 1964 [10–19]. Throughout this evolution of large-scale commercial computer systems, upward compatibility of hardware and software has been preserved. In an analogous manner, the System z I/O architecture [17–23] has evolved to add new functions while usually allowing existing applications and I/O to continue to function properly. The I/O architecture includes the notion of channels, which manage I/O devices under the control of channel programs stored in the computer main memory, and more recently an in-memory protocol called queued direct I/O (QDIO), which manages the I/O devices [24]. The effort to maintain hardware compatibility required a similar continuity of software, starting from the first OS/360 [25] and leading to current versions of z/OS* [26] and all IBM operating systems for System z, including z/VM (see the section on the z/VM CP environment below). Since the introduction of the IBM System/360, technology has evolved and hardware reliability has improved dramatically [27–29], thereby raising the expectations for software reliability.

AIX device driver environment

In AIX, the interface between a device driver (or other kernel extension) and the rest of the operating system kernel is similar to those in other UNIX systems [30–32]. Documentation on how to write device drivers for AIX for the IBM System p platforms is readily available [33–40]. When device drivers are introduced into a system, they register their services dynamically. The mechanism that allows the addition of new device drivers is in the AIX kernel itself, but much of the control, including the configuration of drivers and individual devices, is driven by application programs known as device methods [33, 39]. AIX uses the object data manager (ODM) as a repository for configuration information, and the device methods use defined interfaces to access the configuration information in the ODM [33, 39]. Neither the ODM nor the device methods were ported for integration in z/VM. Instead, the device methods were replaced by new code that offered equivalent function (see the section on device driver extensions below). Similarly, new code was written to meet the configuration information needs that were previously satisfied by the ODM. The new code was implemented as a “container component” (see the section on the anatomy of a container component below) and manages the required configuration data using a simple data model that can be readily mapped to a simple database or flat-file representation. (A flat file was used for the initial test scaffolding, which is discussed in the section on z/VM CP integration below.) The AIX SCSI support is implemented in multiple software layers. The lowest-level AIX device driver, the layer that interacts directly with System p I/O hardware, was not ported. Instead, the interfaces expected by the higher-level drivers are implemented by a new device driver for the System z QDIO interface.

z/VM CP environment

The current z/VM operating system is the latest in a succession of such systems that originated in the IBM Cambridge Scientific Center in the 1960s [41, 42]. The primary function of z/VM is to provide a fully virtualized System z environment in which System z operating systems (primarily Linux** for System z and z/OS, as well as z/VM itself) can execute. The control program (CP) is a fundamental component of the z/VM operating system that is responsible for the management of the real machine’s resources and the provisioning of those resources to the guest virtual machines. The primary purpose of the work described in this paper is to allow the CP to use standard SCSI devices instead of disks specific to System z.

In the past, IBM introduced the concept of Fixed Block Architecture (FBA) for storage in a range of mainframe I/O devices, and such devices worked with a fixed block size of 512 bytes. Because SCSI disks are also typically subdivided into 512-byte blocks, it was possible to create a relatively thin new layer of software between the pre-existing CP FBA functions and the interface to the AIX drivers in the execution container. This mapping between CP FBA functions and the AIX driver interfaces, as well as a new mechanism to allow the initial program load (IPL) of z/VM directly from a SCSI device [43], has made it possible for FBA-aware System z operating systems to operate entirely with SCSI devices.

For a substantial portion of its history, the CP component of z/VM executed without making use of the System z dynamic address translation (DAT) facilities. That is, the CP used "real addresses" rather than "virtual addresses." Even with the introduction of DAT facilities to CP, CP itself still allocated contiguous storage in chunks of no more than 4,096 bytes, which equals one page. As a consequence, new CP facilities for the allocation of larger contiguous ranges of memory were required in order to implement an execution environment for the AIX drivers. These facilities are based upon earlier work by C. J. Stephenson [44], which was adapted to the CP environment and extended to include several attributes required by the AIX driver, including storage allocations larger than 4,096 bytes and allocations on specific power-of-two boundaries.

Design

Overview

In an ideal world, reusable software will run in any execution or OS environment. Unfortunately, since little useful software can satisfy such a constraint, some modifications are inevitably required in order to use the software in another environment.

Moving software from one environment to another using a strategy of adaptation (also known as software *porting*) is often as expensive in terms of development and testing costs as development of new code, and does little to enable any future amortization of those costs. In order to obtain substantial benefits of reuse, both the programming and data models of the original system must somehow be preserved, and thus the required transformation must be performed at a level not directly apparent to the software itself. A number of ways exist to accomplish this goal.

One established approach is to create reusable software by developing it within a universally available operating environment [45], but this approach has performance challenges and may not achieve sufficient isolation from the native operating system environment. It also excludes reuse of software already developed without reuse in mind.

An alternative is to create an execution context that closely mimics that of the system from which the software is being migrated, which is not a trivial task. However, the cost of this approach can generally be amortized as other software is migrated. Provision of most required services in this way is relatively straightforward. However, assumptions with respect to late binding (i.e., binding performed at execution time) and the dynamic introduction of executables can be harder to resolve. Nevertheless, this approach offers an advantage over simple porting in that it is sufficient, from a testing perspective, to demonstrate that the mimicked services are functionally equivalent to the original system. The imported code that uses the mimicked services is not changed and thus does not require further testing.

The methodology presented in this paper allows internal operating system software, such as device drivers, to be migrated by providing services that mimic those of the original environment, and we present several mechanisms for addressing the issues that arise. The approach also provides a general structure within which such environments may be constructed with progressively lower costs—not only costs associated with the migration of subsequent functions, but also the lower costs of operating systems that can share much of the container infrastructure described below.

The execution container (hereafter referred to simply as the container) is based upon an architecture that specifies the way in which a program may operate within the environment of the container. The container architecture itself defines no specific interface with the operating system within which it is instantiated, leaving any obligations of communication and interaction to one or more of the executable components that are introduced during its operation. In this context, the term *component* is used to describe any object that conforms to the architectural requirements of the container. In other words, a component is aware of, and by implication capable of conforming to, the environment of the container. In this sense, the container is nothing more than a very simple programming framework.

To be useful, the container must be more than a framework, and the necessary additional functionality is provided by the *core system services component* (CSSC). This component provides a basic set of services that are discussed further in the section on environmental requirements below. The container alone serves no one specific purpose; rather, it provides an architected environment within which one or more loaded components provide functionality.

With the sole exception of the CSSC, all components are required to be introduced by means of the

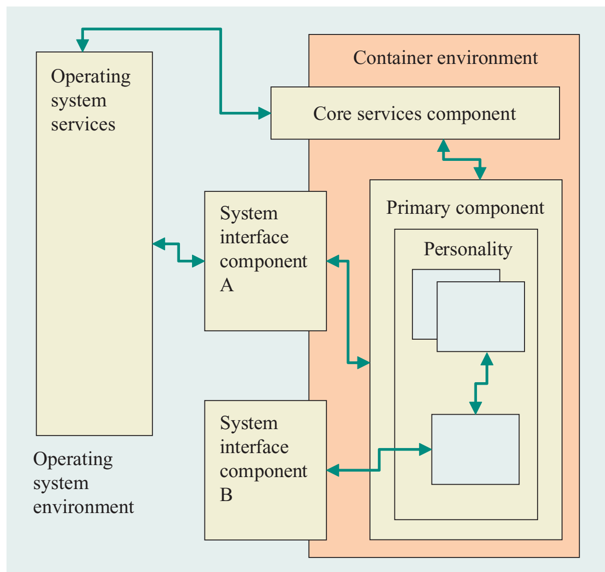

Figure 1

Container schematic showing the container environment and its interactions with the host operating system. All interactions with the host operating system are mediated by a system-interface component. The unlabeled boxes inside the personality component represent the AIX device drivers, which are isolated from the container environment by the AIX personality and the driver extensions (not shown).

`CSSC_loadComponent` service. Aside from taking care of the required tasks associated with loading and initializing a new component, being successfully "loaded" into the container via `CSSC_loadComponent` is a sufficient indication of architectural conformance. In other words, no further tests are made to ensure conformance before or after loading, beyond the provision of some mandated functions to be invoked when loading and unloading a component. A component is not obliged to communicate with, or offer services to, any other component, but in general a component is likely to do both.

Typically, a component has no dependencies upon any external capabilities except those offered by other components. This independence implies complete portability across any environments that support the container architecture. Portability in this context is compile-time portability, not binary compatibility. This platform independence is implied wherever the term *component* is used without other qualification.

We must also recognize a special class of component that does have an awareness of the host OS environment; these are known as *system interface components*. In discussions of container frameworks, a system interface component is any component that has some knowledge of or dependency upon the environment within which the container is being instantiated. This knowledge allows such a component to interact in some way with the environment. CSSC is an obvious example of such a component, because it is responsible for the integration of the container with its environment, including the task of container initialization. In general, other system interface components are required only in order to provide access to services provided by the container.

Another component that must be aware of a non-container environment is the *personality*, discussed in the section on personalities below. For the z/VM SCSI, the non-container environment is the AIX kernel environment. The personality component transforms the services available to container components to those expected by the ported device drivers. Together with the device driver extensions (described in the section on device driver integration below), it provides the AIX kernel services expected by the device drivers.

The container has been implemented using the C programming language. This choice of languages was natural, but not essential, because AIX drivers are themselves written in C. Operation of the container only requires object-code compatibility. Any language can be used when creating a component, provided that the simple interface obligations of the container can be met, together with the underlying complier assumptions for linkage.

Our approach is useful because it is relatively inexpensive to create a prototype in an environment other than that for which the work is ultimately intended. For example, the container for the z/VM SCSI was prototyped in an application environment on Linux, which allowed development and test of the platform-independent components using conventional development tools—something that is not easily achieved with operating system extensions. **Figure 1** illustrates the basic components of the programming environment being described.

Specification is not required with respect to the way in which the container communicates with the environment, and this offers some important benefits. The implementation of the container can be encapsulated within some other recognizable entity in the intended environment. For example, the container may be packaged so that it appears to be a conventional device driver in the Linux environment. In this way, a single device driver may be shared across a number of different and apparently incompatible environments. The container can be closely integrated with its environment, so that instead of requiring an intermediate software layer that maps the semantics of the environment to some public interface defined by the container, the environment itself can be adapted to behave as a component and communicate directly with the services provided by the

supported OS extensions. The issues of porting the OS extensions are easily separated from the issues relating to their exploitation on a given platform. It is a straightforward task to introduce extension-specific system interface components in support of specialized requirements. For example, workload management capabilities can be directly integrated with the container.

Finally, a system can have any number of containers concurrently active, subject to whatever host OS constraints apply. The use of multiple containers promotes isolation of code and data structures, and isolation of resources managed by the ported code. These characteristics have the potential to enhance RAS, security, workload balancing, and installation management and control.

Anatomy of a container component

Each component is a separately loadable object within the container-managed environment. As previously mentioned, it must conform to certain internal requirements of the container, and it may also have awareness of the environment outside of the container. The following summary describes material that is covered in more detail in the subsequent sections.

Each component has one or more associated interface header files that contain all of the information needed to load and access the component. Language preprocessing, in conjunction with specific programming models, allows one component to access the services of another component with no additional programming beyond the inclusion of the header file and a call to `CSSC_loadComponent` to load the desired component.

Versioning information contained in the interface header files ensures that any conflict (for example, between the versions of the interface used at compile time and the versions expected at runtime) is detected when the component is loaded. Depending upon the component, such a conflict may result in a load failure or the automatic provisioning of the appropriate interface by the component being referenced.

When a given component is first loaded, a component control area (CCA) is allocated and associated with the component. Control is given to a mandated function, as previously mentioned and discussed in more detail below, that has the opportunity to establish component-specific information for future use. The component control area contains the vector table discussed below as well as internal component control structures.

Each component interface is implemented using a vector table, and access to a given interface is obtained using either the `CSSC_loadComponent` service or the `CSSC_getInterface` service. Each vector contains the entry point of an associated function and the component control area that should be effective when that function is in control. Some powerful capabilities offered by this approach are enumerated in the following paragraphs.

First, by separating the definition of the interface from the executable component itself, it becomes possible for a single component to present multiple versions of the same interface to its callers. This separation facilitates both upward and downward compatibility as well as providing the possibility for an interface that can adapt to the needs of a specific caller. The use of vectors also enables multiple versions of a given component that can be concurrently supported.

Additionally, by storing both the function address and the control area address in each vector, it becomes possible for a component to export vectors owned by other components without incurring the normal computational overhead associated with cascaded function calls. This mechanism is effectively exploited by the system services component itself.

All components have a designated default interface that is accessed using the reference returned by `CSSC_loadComponent`. A component may also support additional interfaces, and access to these interfaces is obtained via the `CSSC_getInterface` facility.

The first request to load a component causes the *initialization* function for that component to be executed. Subsequent load requests increment the use count but do not cause further initialization. The process of unloading decrements the use count, and a component remains available until its responsibility count reaches zero. At that time, its *terminate* function is called, and it is considered to be unavailable within the container. A subsequent load will result in the initialize function being called. The initialization and termination vectors are two examples of a set of vectors that are an obligatory part of the container architecture. Other required vectors include *recovery*, *debug*, *recycle*, *validation*, *control*, *start*, and *stop*.

The *recovery* vector allows a component to define a procedure that should receive control as a result of a system-detected error during execution. This interface is part of an architected recovery hierarchy defined within the container. The *debug* vector allows a component to interact with an external debugging mechanism. This interface may be used to modify the behavior of the component. The *recycle* function informs a component that a client has terminated. This notification allows the component to “clean up” any resources, such as memory, held on behalf of that client. The *validation* vector allows a component to indicate its own functional status, that is, *true* if no errors are found or *false* if some unrecoverable problems are detected. The *control* vector is a function with unspecified semantics that allows additional component vectors to be defined that are not expressly

defined by the container architecture. Whether or not such functions exist depends on the component.

The *start* vector, if present, indicates that this component is capable of being a *primary container component*. A primary component is one to which control is given at some time after container initialization is complete. A hierarchy of such components may exist if a component that receives control using this interface is prepared to pass control to the next component in such a hierarchy. The exact nature of such a hierarchy is not programmatically apparent; rather, it is an emergent property of a set of components. For example, in the case of the container I/O application described in this paper, CSSC is the component to which the container initialization routine gives control. Once CSSC has completed its container-related initialization, it passes control to the first identified primary component in the load list (see the section on establishing purpose below). A primary component is not expected to return control until it has to terminate. Consequently, when the primary component previously called by CSSC returns control, the container will close. Typically, one primary component will establish interfaces for services to entities outside the container environment and then wait to service requests via that interface. The *stop* vector provides the means by which an external agent, such as a system operator, can indicate to a primary component that it should cease execution. This interface is intended to be the normal mechanism for shutting down the container.

Any function exported by a component for use by other components must conform to the programming model of the container. This obligation is satisfied by a collection of macros supplied for the purpose, such as the following:

```
void *
my_function(thread_ct * thread, void * arg)
{
  COMP_ENTRY(my_function);
  void * result;
  result = arg;
  COMP_EXIT(result);
}
```

In this example, the `COMP_ENTRY` and `COMP_EXIT` macros conceal the container-specific linkage obligations associated with the designated function. A variety of such macros are provided to address all of the basic requirements of structured programming within the container. As well as addressing issues of linkage, these macros provide a consistent and comprehensive RAS model within the container, which is integrated with the container environment by CSSC. The scaffolding macros are defined in such a way that inconsistent usage, or omission, is generally detected at compile time.

Core system services

CSSC comprises two separate components. A system interface component, CSPS (core services, platform specific), is responsible for providing an interface with the operating system in support of the various control and management services required. A platform-neutral component, CSPN (core services, platform neutral), is responsible for making various control and resource management services available to other components in the container. These services represent a minimal set of capabilities needed to allow consistent, platform-independent operation of other components loaded into the container.

For practical purposes, CSSC embodies the container from the perspective of other loaded components. Access to the CSSC public vector table allows a program to be a component in the container environment. The various parts that make up a working container are illustrated in Figure 1.

Environmental requirements

The AIX kernel environment, the z/VM CP environment, and other operating system kernel environments are all greatly restricted in terms of available system services when compared with the application environment. In this context, such limitations are not disadvantageous. Indeed, this characteristic makes device drivers relatively portable and makes the required infrastructure to support such portability reasonably straightforward.

These environmental services needed to support OS extensions can be classified as

- Executable content management (e.g., load/unload and exported reference identification).
- Memory management (e.g., allocation/deallocation and pinning/unpinning).
- Cross-memory addressing services.
- Synchronization (e.g., general locking services, wait, and notification).
- Scheduling (e.g., asynchronous scheduling, execution-mode switching, time-delayed scheduling, and time-limited suspension).
- RAS (e.g., trace, dump, and logging).

Some services are needed to gain access to system resources (e.g., services such as content management and memory management). Other services support the integration of container-managed events with the environment (e.g., synchronization and scheduling). The provision of other services might be considered optional (e.g., tracing and logging).

The nature of the required services is generally so fundamental that there is little likelihood of significant problems when implementing such services in any environment of interest. The container, as implemented, encapsulates these services in CSSC. Within CSSC, the CSPS subcomponent owns the actual interface between the container and existing operating system services. This design allows significant flexibility in terms of the way in which a particular function is implemented. The greatest challenge when implementing these services arises from the need to anticipate possible future requirements that might be placed upon a given service. For practical reasons, the services interface exported by CSSC is generally consistent with the analogous AIX kernel services. For example, the AIX xmalloc service is exported to the device drivers by the personality component. The AIX personality component does a simple remapping of the AIX xmalloc interface to the `CSSC_malloc` interface. The `CSSC_malloc` interface is in fact just the `CSPS_malloc` interface as exported by the CSPN component on both z/VM and Linux.

Intercomponent flows

We place two significant constraints upon the design of the container. In particular, the container cannot exploit the native mechanisms of the compiler for managing global data, and it cannot exploit any platform-unique services for the runtime binding between components.

When the initialization function of a component is called, the function is passed an area of memory that can be used as a global control area CCA for the component. The address of the CCA is maintained by CSSC and is returned to any caller that causes the component to be loaded. In the interests of linkage efficiency, this address is also that of the default public vector table for the component, as described below.

In the absence of runtime services for dynamic binding between executables, the container exploits an efficient compile-time mechanism to support runtime binding between components. Exploiting this mechanism involves certain conventions. First, each component must provide a public header file, referred to as an *interface header file*. This file provides all of the information needed to locate and exploit the owning component. The header file must be included wherever access to a component is required.

In addition to including the header file, a component must use `CSSC_loadComponent` to load the requisite component into the container environment at any time prior to first use. Once this has been accomplished, the reference returned by the loading process is stored in the control area of the local component in accordance with the container rules for component references. This reference provides access to the default *public vector table* of the component.

Finally, access to services in another component is achieved using the access macros supplied in the interface header file of the target component. For example,

```
xyz = (t *) CSSC_malloc(thread, size, alignment,
                        flags);
```

shows how a component might call the storage allocation service provided by CSSC. By convention, the first argument, `thread`, provides access to per-thread container control structures.

We employ a novel exploitation of the preprocessor to allow the access macros supplied in the interface header file to be used regardless of the caller's own control area organization. The code generated in support of a call to another component, for example `CSSC_malloc`, essentially resolves to

```
frameptr->cca = malloc.cca,
   malloc.fnc (thread, size, alignment, flags);
```

This greatly simplified example shows the required control area being loaded into the current frame, and the required function vector (`malloc`) being invoked. A *frame* is part of the container linkage mechanism. Each active function has a corresponding frame that allows the container to manage various aspects of serviceability and recoverability. In this case, the frame is used to communicate context information to the function being called. The control structures associated with the frame are C-language automatic variables declared and initialized by `COMP_ENTRY`. The `COMP_ENTRY` macro in the implementation of `CSSC_malloc` retrieves the CCA from the caller's frame and establishes appropriate local addressability.

By storing both the required control area address and the function address in each vector, the container allows a component to promote vectors obtained from other components. This approach improves overall performance by eliminating unnecessary intermediate execution paths when accessing non-local functions. For example, CSSC directly exports the memory services provided by CSPS. In other words, no intervening CSPN wrapper is necessary.

Integrating the container with its environment

The CSPS component of CSSC, or some platform-specific intermediary code acting as an agent for CSPS, receives initial control from the operating system environment. As a result, if CSPS is to be introduced by the operating system, it must be consistent with that environment. In the case of z/VM, CSPS and the entire container are actually directly integrated with CP. The part of

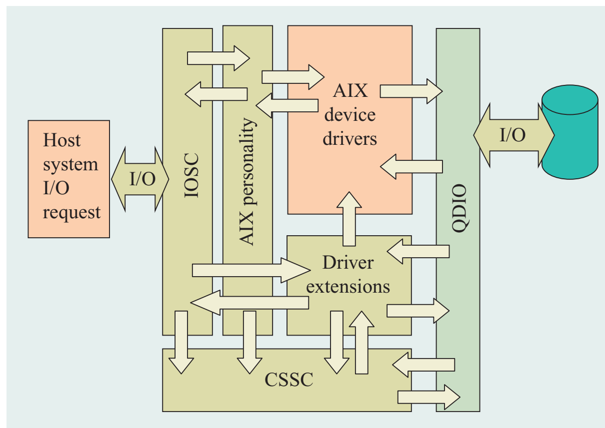

Figure 2

Driver component schematic.

CSPS responsible for initializing the environment of the container is called the *prolog*. This container prolog handles any differences between the native linkage of the system and the linkage required by the language environment chosen for CSPS. Various ways exist to accomplish this function, and the particular way that is chosen tends to reflect the language and compiler being used.

CSPS performs its own private initialization and then loads the CSPN component, simulating the behavior of `CSSC_loadComponent`. Once CSPN has been successfully loaded, CSPS calls the *start* meta-function supplied by CSPN. This event marks the end of the container initialization and the beginning of its application.

Establishing purpose

The CSSC *start* function loads a set of components that are identified in a special configuration file called the *load list*. The load list is loaded using the data management capabilities provided by CSSC. All of the components in the list are loaded, and one of them is identified as being the primary component. Nothing prevents a given operating system from having some or all of the components preloaded, and in fact z/VM does preloading. The primary component determines the specific purpose of the container, and is passed control by CSSC by invoking the start vector of the primary component. The container remains active until the primary component returns control to CSSC. If no primary component is provided, the container simply terminates.

In this paper, the primary component is the I/O services component (IOSC). The IOSC component has two principal responsibilities: It provides an interface to support I/O operations, and it routes such operations to whichever personality provides access to the resource in question.

Personalities

A *personality* is a specialized container component that is responsible for emulating a particular execution environment. Any number of concurrently active personalities may exist. The personality presents a standard interface to the other components and provides an environmentally unique interface to the migrated code that it is responsible for supporting.

The personality must map the service API presented by IOSC to the native API expected by the migrated code. This requirement applies not only to the I/O operations themselves, but also to the management tasks of initialization, configuration, and termination. Similarly, the personality must intercept all of the interactions of the migrated code with the environment of the migrated code in a manner that is indistinguishable from the native environment. Note that the requirement dictates that the environment is indistinguishable from, but not identical to, the native environment of the migrated code.

The AIX driver code is by definition unaware of the container or its architectural requirements. This shortcoming is addressed by integrating a set of functional extensions (identified as driver extensions in **Figure 2**) that are combined with the driver code during linkage editing. These extensions address all of the architectural obligations associated with being a component, such as providing the required vectors discussed above, and also relieve the personality of responsibility for driver-specific activities normally addressed by AIX facilities such as device methods [33, 39]. AIX device methods are application programs that are separate from the device drivers and that control device activation and configuration.

In addition, the extension implements the recovery and resource management interfaces that System z requires of an I/O handler and that are not part of the AIX environment. Specifically, these interfaces relate to recovery (handling unexpected programming errors or other failures at runtime), recycling (handling resource cleanup on behalf of failed clients), and I/O purge (forcing the completion of outstanding I/O requests on the basis of some specified filter).

When linked together, the device driver and its extension constitute a proper container component. Language preprocessing is used to remap device driver calls for external services to the appropriate container linkage. Function calls that are internal to the device driver are not modified and do not follow container linkage conventions.

Figure 2 presents a simplified view of the major components involved in I/O that uses the container, and indicates the principal dependencies among them. Note that IOSC communicates with the driver via the personality. The personality is responsible for presenting an implementation-independent view of the driver interface, as well as providing a convincing AIX execution environment for the drivers themselves. The figure shows that the driver communicates only with the personality and with the QDIO driver. The QDIO driver is an example of a device driver written specifically for the container environment. In essence, CSSC is the personality of QDIO. Conversely, the QDIO driver must implement the interfaces expected by the AIX device drivers, thus performing a role similar to that of the AIX personality. The QDIO driver must interact with operating system I/O services and thus requires a system interface component. Because the QDIO driver implements a driver for a System z architecture facility, a potential exists for amortizing its development cost across multiple System z operating system platforms.

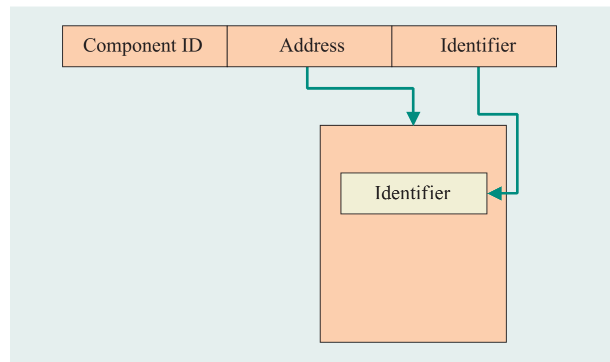

Figure 3

Simple handle implementation, showing the address that is used to locate the element to which the handle refers. Also shown are two identifiers, one in the handle and one in the referenced element. For the handle reference to the element to be valid, the two identifiers must match. (The upper set of rectangles represents the handle. The lower two rectangles represent the element referenced by the handle.)

Container reliability

In this section, we consider the ways in which the recovery environment that is assumed by the drivers is preserved, and also how that environment is integrated with the host operating system environment. Our intent is for the overall reliability, availability, and serviceability of the device drivers to be enhanced in the process of integrating them in the container.

Handles

The container employs handles to transparently address many of the reliability problems that arise from the use of pointers to access dynamically assigned, reusable resources owned by the system or another component. For example, when one component loads another, it obtains a license to access the services and resources of that component. In the event that that contract is terminated (e.g., because the component was unloaded), the system needs a means of detecting such *out-of-contract* usage. Handles provide such a mechanism.

The type `handle_ct` is declared by appropriate C typdefs to be a single-element array whose underlying structure anonymously reserves memory. By defining the handle as a single-element array in C, we ensure that the reference cannot be trivially modified. Anonymous mapping is done primarily to avoid any kind of implementation dependency by the user of the handle. This technique ensures that an individual handle supplier is not constrained in terms of implementation. This approach (and other related approaches in this paper) reduces unintentional programming errors. These approaches are not security mechanisms, and they can always be deliberately circumvented. Passing of the handle provides the system service with an area in the caller's space in which the service can place the information needed to manage a reference to a resource.

In our environment, a handle must at least allow the address of the item that is being referenced to be derived from the handle itself. Sufficient shared information must exist, stored in both the handle and the element to which it refers, to determine that the handle indeed represents the current instance of the resource. Note that the address information in the handle is meaningful only in the context of the component that supplied the handle. **Figure 3** illustrates a simple handle implementation.

This mechanism of using handles as references requires that while the element being addressed may be reassigned or reused, it should persist and retain a consistent type through time. The mechanism also requires that a given identifier should not be reused. Failure to satisfy these constraints significantly reduces the benefit that handles offer. The container addresses these issues using two mechanisms. First, the element representing the resource being referenced is allocated using cell pools, that is, pre-allocated areas of memory from which individual elements are allocated. The use of cell pools ensures that the internal structure of individual elements remains consistent over time. Cell pools offer additional benefits with respect to performance and memory management; most significantly, their use provides a consistent interpretation of a memory area over time, regardless of the current use status. When an element from a cell pool is freed, it retains all of the information it contained on last use, and is reinitialized only if it is eventually reused.

As a second mechanism, a numerical identifier is associated with each instance of a resource at the time the resource is allocated. This number is stored in the associated handle. Whenever the resource is referenced, the identifier in the handle is compared with the instance identifier in the element representing the resource, and if the numbers do not match, the reference is denied. The identifier is invalidated when the resource is deallocated. The size of the identifier field is determined by the resource owner and depends on the expected frequency of reallocation. This mechanism has some associated risk because it is possible that a handle may be used after the identifier has completely cycled and happens to be correct at the time of reference. Appropriate choice for the size of the identifier can make the probability of this occurrence very small.

These two mechanisms jointly allow for safe and verifiable access to the resources to which they are applied. The container further enhances the serviceability of instance identifier use by adopting a systematic mechanism for invalidating the instance identifier. Whenever a new instance identifier is being assigned to an element, it is incremented by 2. Thus, all active elements in a cell pool will have an even instance identifier. When the resource is deallocated, the low-order bit of the identifier in the cell is set to 1 in order to effectively make the instance identifier odd. This technique has the useful side effect of allowing the current allocation status of elements in a cell pool to be determined by simple examination of the instance identifiers. It also preserves debugging information about the prior use of an individual cell, and can give insight into the frequency with which resources are being reused. Any subsequent reference to the resource that uses the current handle will fail because the handle is now stale; that is, its instance identifier does not match the target data area.

To provide freedom of implementation to a handle owner, only part of the handle is required to be uniformly mapped by all users. This fixed portion contains the component identity of the handle owner. Each component is able to manage and interpret the remainder of the information in the handle in any way it sees fit.

Handles are used in the container to manage references to locks, latches, and several other critical data structures whose typical usage implies frequent acquisition and release. The use of pre-allocated storage for the resources of a given type and the use of a handle to reference the resource is effectively a form of "type-safe memory" [46–48].

Of course a penalty is paid in terms of the performance cost of validating the handle. Typically, the validation process first requires validating ownership (component identity), then validating type (its type matches the expected resource being processed), and finally validating the identifier. These costs are in addition to the fundamental cost introduced by the implied indirection.

The handle could be made more robust by including a small checksum, but a checksum was not implemented in the container. Damage to a handle that affected only the embedded memory reference was considered too unlikely to justify the additional cost.

Isolation

Some data elements and their memory areas are more critical than others in terms of sensitivity to inadvertent modification. For example, damage to the state elements of a software lock may have implications far beyond the success or failure of the execution flow that caused the damage.

Data isolation is particularly beneficial in the case of locks and latches, especially if they are to be shared by programs that are otherwise isolated. There are several reasons why the isolation provided by the use of handles is beneficial. First, locks and latches are examples of state that, while mutable with respect to the user, are internally constrained by the set of meaningful values that can be assumed. To increase reliability, a program should not be able to deliberately or inadvertently invalidate the allowable state model. Second, locks and latches are repeatedly acquired, used, and released; that is, frequent changes in ownership exist. The System z architecture does not allow areas within a page to be uniquely protected with respect to other areas in the same page, which means that a program solution is required.

In the case of the device drivers, the types used for locks and latches are transparently redefined so that a handle is stored in the local data structure instead of the lock or latch itself. The implied change to the semantics of the locking and latching calls themselves is easily provided by the transformation layer of the personality. The handle is then used to reference the lock or latch with all of the previously attributed benefits, and the actual memory element can be placed in memory not normally accessible to the originating program.

Trace

The container implements a comprehensive trace facility, which allows execution activity to be collected by thread, by function type, or by activity classification. The amount of data collected can also be controlled within these classifications.

All existing trace macros within the AIX drivers were remapped, using an AIX personality header file, to the container trace, and in this way the existing trace model of the driver was preserved.

Additionally, the trace produced by interaction with container services provides useful insight into driver activity and records those failed service requests whose

disposition is ignored by the drivers. The compile-time remapping of the device driver calls in order to use container linkages allows the collection of traces with the same granularity as other inter-component calls.

Finally, compiler hooks, and/or prolog and epilog macros, invoked by the compiler at the start and end of functions, can be used to incorporate flow tracing within the migrated driver stack without requiring any modifications to the driver code. This approach is particularly valuable because this same mechanism permits an almost full integration of the driver code with the container recovery model.

Detailed trace tools are essential during development but are less useful after deployment because of the runtime costs involved. We paid significant attention to minimizing the runtime penalty paid for making the trace/no trace determination.

Diagnostic aids

All major control structures managed by the container employ the notion of *eye catchers*. Originally used to improve serviceability by making core dump information more readily identifiable to analysts involved in problem determination and resolution, the eye catchers also make it easier to recognize bad pointers and the effects of memory overwrite errors. Each eye catcher consists of an eight-character acronym, the address of the data area within which the eye catcher is included, and the size of that data area. This information can be used by container-aware functions at runtime to validate data area references, and can be used by core-dump analysis tools to automate similar validations. Not all data structures have eye catchers; for example, the data structures directly accessed by the migrated drivers do not have eye catchers.

The nature of the vector interface between components makes it simple to incorporate additional diagnostic mechanisms without requiring any normal runtime penalty. Additionally, the container linkage provides a platform-independent basis for analyzing container activity. It is possible to determine the current status of all threads of control that are active within the container. The linkage also augments the normal compiler and driver linkage in order to support OS-specific retry and recovery models.

Device driver integration

Porting the device drivers

Preceding sections in this paper have discussed the general characteristics of code that executes in the architectural environment of the container. The current section examines issues that arise specifically from the AIX drivers.

The AIX device drivers are not aware of, nor do they conform to, the container architecture. However, the container allows this code to be migrated with minimum change and with a preservation of both programming and data models. This approach is in contrast to those in architectures such as UDI (Uniform Driver Interface) [49] that require new drivers to be developed that conform to a documented architecture. The container model offers the same benefits of single-driver development, but does so in a way that allows an existing driver to be used without the constraints and additional effort that a formal architecture imposes on initial driver development. In fact, migration of the AIX drivers to the container environment was relatively trouble-free. The most laborious task was the identification of required information from the AIX system header files referenced by the drivers for use in the container, discussed in the next section.

Very little modification of the drivers was necessary. Indeed, in our opinion, the changes described below amount to following good programming practices. In all cases these changes could have been introduced into the original driver code without affecting its function in its original environment. The following three situations accounted for most of the changes made.

First, when the personality requires explicit access to information in a data area also accessed by the drivers, that data area must be defined by the personality, and a `typedef` is used to map it into the expected type space of the driver. In some cases, the drivers referred to a structure explicitly, and in those cases it was necessary to modify the drivers so that an appropriate abstract type reference was used.

Second, references to system services were resolved during compilation of the drivers by using macros to rewrite the reference to conform to the container linkage model. However, in a few places the drivers made explicit use of embedded function pointers, making such rewriting impossible. In some cases, it was possible to leave these explicit uses of function pointers unmodified (where some other extant mechanism for communicating necessary container information existed). It was sometimes necessary to rewrite the code so that the function pointer usage was hidden by an appropriate macro.

Finally, instances of incorrectly typed assignments existed. The z/VM build environment, like some other product-build environments, does not accept such warning messages from the compilation. Rather than risk eliminating meaningful messages, we modified the code to resolve such problems.

AIX system header files

The device drivers referenced dozens of AIX header files, some of which, in turn, referenced dozens of other header

files. We did not want to import any more code than necessary in order to compile AIX device drivers. With respect to the first two device drivers, we commented out all references to AIX header files and incorporated the required material from AIX header files into a few new header files. We found that with this approach, the contents of many of the AIX header files were not required for the drivers being ported, and for some of the header files only a few lines were required.

Subsequent to our initial work, during the process of porting another SCSI disk device driver, we found that this consolidation of multiple AIX header files into a single header file was not appropriate because the appropriate separation of unrelated definitions and declarations was lost. For example, the second SCSI disk device required different implementations of similarly named data structures. The two SCSI disk device drivers support two different storage subsystems but use the same lower-level device driver, so this type of conflict is to be expected. Consequently, the original header file structure was reapplied to the subset of information required by the drivers.

Device driver extensions

A device driver extension provides the implementation of the services required of a component in the container environment, as discussed above. The device driver extension also provides the configuration and initialization services that are accomplished in the AIX environment by device methods [33, 39]. AIX device methods are application programs, separate from the device driver, that control device activation and configuration. These functions in the driver extensions were modeled on their AIX analogs but use a different interface for accessing configuration information. The driver extensions are also different from AIX device methods in that the equivalent function in driver extensions runs in the same environment as the rest of the device driver, while the AIX driver methods run as normal (albeit privileged) applications. In addition, the device driver extensions provide functions not implemented in the AIX kernel environment. For example, in the System z environment, a requirement exists to purge any outstanding I/O activity when physical storage resources required by the I/O must be used for some other purpose. Implementing the purge function required an additional minor change to the device drivers to ensure that requests being transferred between device drivers are not missed by the purge mechanism. Like the system interface components depicted in Figure 1, the device driver extensions are different from most container components because they are aware of both the container environment and the AIX kernel environment. In particular, they directly invoke some of the interfaces exported by their associated device drivers as well as invoking container interfaces exported by CSSC.

z/VM CP integration

The development of the SCSI for z/VM support did not follow the usual product development cycle for z/VM or for any other IBM-developed software. Before the development environment for the final product was available, the majority of the container was developed and tested on Linux running as a z/VM guest. The Linux implementation included the basic container functions, the AIX personality, and the device driver extensions, as well as additional device drivers required for the System z environment and test scaffolding to emulate the layers of hardware and software not available in the Linux environment. The container and all of its subcomponents executed as a normal user process. No actual SCSI I/O was performed in the Linux environment. Such I/O was simulated by the test scaffolding, which utilized the same container component infrastructure as product code. The platform-independent portions of the code developed on Linux were subsequently integrated into the z/VM CP component, and the first version of this support became generally available as part of z/VM 5.1 in September 2004.

CP environment

As discussed above, for many years the CP component of VM did not exploit virtual addressing; that is, all of the CP code and data areas were fixed in memory and referenced by real addresses. This fact, and the lack of facilities for dynamically allocating large areas of memory (greater than one page, or 4,096 bytes), meant that each "stack frame" of a C routine had to occupy less than one page. While this restriction may seem a stringent requirement at first glance, in reality it is not that different from the AIX or other UNIX kernel environments. Portions of the AIX kernel are pageable, but many parts are not. Routines that are not allowed to cause page faults must have fixed storage for their C-procedure call stacks. In AIX the stacks are contiguous ranges of virtual addresses, but they are limited in size. Thus, AIX kernel extensions and device drivers do not have routines with large stack frames. Rather than have large memory areas allocated as C "automatic" storage, the dynamic requirements of the drivers are met by explicit memory allocation calls at runtime or (quite commonly) by allocating a pool of space during driver initialization and suballocating the storage from the pool as required. In practice, meeting the requirement that the stack frame of each C routine be smaller than one page was not difficult, but it does require the use of a compiler that allows function prolog and epilog code to be supplied at compile time.

Two assembly language macros were provided for this purpose, corresponding to function prolog and epilog. Parameterized information passed to these macros at compile time allows the macros to generate an optimal code sequence, sensitive to the constraints, such as stack usage, of the particular environment.

CP SCSI exploitation

To minimize the changes required of other CP components, our initial SCSI support emulated an older I/O device, the FBA channel-attached disk. Existing interfaces allowed applications to access SCSI disks by the applications building the same channel programs that were used to access FBA disks. New code in CP interpreted these channel programs and translated the requests to calls to the SCSI interfaces. The processing of these channel programs by the central processor (instead of the channel processors) corresponds to processing overhead incurred to avoid large-scale changes to the z/VM I/O system. Ultimately, the interface exposed by the FBA device support is very nearly the same as that provided by the SCSI device drivers, so this additional computational overhead could easily be avoided with minor changes. We did not attempt to implement these changes in the initial release because they are pervasive, and even though they are not complex, they would require a substantial verification and testing effort that was avoided by emulating existing interfaces. In z/VM 5.2, the paging subsystem was changed to avoid the channel program emulation, resulting in significantly improved paging performance on SCSI devices, due largely to the increased concurrency allowed by direct access to the SCSI I/O interface.

Despite the increased CPU processing overhead discussed above, SCSI devices have been observed to produce a substantially increased paging rate compared with previous device interfaces [50]. The results reported in [50] are for a hypothetical configuration, specifically designed to force a high paging rate, and are not claimed to be representative of any production environment. However, the results do demonstrate that the SCSI device interface can sustain a higher I/O rate than previous interfaces while accessing the same external storage subsystem via the same host hardware interfaces.

Summary and conclusions

The approach described in this paper allowed a relatively small number of individuals to produce a new I/O subsystem for the z/VM product. The subsystem included tens of thousands of lines of code from a completely different operating system, all written in C, a language that had never been used in the CP component of z/VM. As successful as the approach has been in providing a new function for z/VM, with very low development and test costs, some drawbacks exist. Although providing an AIX execution environment (via the AIX personality) minimized the changes to the AIX device drivers, this environment incurs some CPU processing overhead, as does the improved software reliability provided by the mechanisms discussed in the section on container reliability. In addition to these possible performance problems, z/VM must maintain and support the AIX device drivers, components of which are unfamiliar to the z/VM team. Some additional education is also required for the service team to become familiar with the extensive use of vector tables and C preprocessor macros used to implement component linkages.

The design and development of a significant part of the subsystem (excluding the ported AIX device drivers) was accomplished by workers outside the traditional z/VM team. Much of this code was integrated into z/VM after it had undergone function and component testing on Linux. The schedule that started with the production of the first working executable and culminated with components that were ready for system test was very aggressive. However, the progress from system test to product general availability was relatively straightforward, and no significant serviceability problems have appeared with respect to the z/VM SCSI subsystem in customer installations. As discussed in the section on device driver integration, an AIX device driver that was not contemplated in the original design was added to z/VM subsequent to the initial development; it became generally available in December of 2005 with the release of z/VM 5.2. The addition of this device driver, which was new for z/VM, was accomplished largely by the traditional z/VM development team with substantially less effort than required for the first two device drivers, demonstrating the utility and flexibility of the container design and validating the basic design concepts.

A significant portion of the newly developed code is independent of the z/VM system. All of CSPN, IOSC, the AIX personality, the AIX device driver extensions, and the QDIO device driver have been designed and implemented with the intent that they be reusable in other System z operating system environments. Because of the isolation provided by the container, the ported AIX device drivers should also not require changes to be functional in a different environment. All of these components can simply be reused in order to provide the same function in another System z operating system. Of course, in addition to the z/VM-specific components, namely CSPS and the z/VM system interface components, other operating systems may require functions not implemented for z/VM. Nonetheless, in addition to reusing the AIX device drivers, the approach described will allow much of the container implementation itself to be reused.

In summary, the container design and implementation have proven to provide a robust mechanism for incorporating device drivers from the AIX operating system running on System p servers into the CP component of z/VM running on System z servers. This incorporation provides a significant saving in design and implementation effort. The advantages of this approach far outweigh the drawbacks mentioned above, and this approach to the reuse of kernel code can be generalized to other device drivers or kernel components and applied to other operating systems.

Acknowledgments

This paper has concentrated on the container design and the reuse of the AIX device drivers. Other aspects of the project have been discussed only briefly or omitted entirely. The hard work and dedication of many individuals were required for the timely delivery of z/VM 5.1 and z/VM 5.2, and any list of contributors would almost certainly be incomplete. However, with regard to the z/VM SCSI I/O subsystem, we would particularly like to acknowledge, in alphabetical order, the contributions of Roger E. Bonsteel, Charles J. Brazie, Juliet C. Candee, John L. Czukkermann, Eric R. Farman, Joseph M. Hust, John J. Majikes, James L. McGuinniss, Jr., Lisa H. Reese, Robert W. Schreiber, Steven G. Wilkins, John W. Yacynych, and Edward Zebrowski, Jr. We also thank the referees for their suggestions for improving the paper.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of The Open Group, Sun Microsystems, Inc., Xerox Corporation, Microsoft Corporation, Interactive Software Engineering, Inc., or Linus Torvalds in the United States, other countries, or both.

References

1. C. W. Kruger, "Software Reuse," *ACM Comput. Surv.* **24**, No. 2, 131–183 (1992).
2. B. W. Kernighan and P. L. Plaugher, *Software Tools*, Addison-Wesley, Reading, MA, 1976.
3. F. P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley, Reading, MA, 1995.
4. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 2000.
5. B. Meyer, *Object-Oriented Software Construction,* Prentice-Hall, New York, 1997.
6. G. Bracha, J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, Reading, MA, 2005.
7. P. H. Winston and S. Narasimhan, *On to Smalltalk*, Addison-Wesley, Reading, MA, 1998.
8. A. Hejlsberg, S. Wiltamuth, and P. Golde, *The C# Programming Language*, Addison-Wesley, Reading, MA, 2003.
9. G. Booch, *Object Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, CA, 1994.
10. G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM J. Res. & Dev.* **8**, No. 2, 87–101 (1964).
11. R. P. Case and A. Padegs, "Architecture of the IBM System/360," *Commun. ACM* **21**, No. 1, 73–96 (1978).
12. A. Padegs, "System/360 and Beyond," *IBM J. Res. & Dev.* **25**, No. 5, 377–390 (1981).
13. A. Padegs, "System/370* Extended Architecture: Design Considerations," *IBM J. Res. & Dev.* **27**, No. 3, 198–205 (1983).
14. D. Gifford and A. Spector, "Case Study: IBM's System/360–370 Architecture," *Commun. ACM* **30**, No. 4, 291–307 (1987).
15. K. E. Plambeck, "Concepts of Enterprise Systems Architecture/370*," *IBM Syst. J.* **28**, No. 1, 39–61 (1989).
16. K. E. Plambeck, W. Eckert, R. R. Rogers, and C. F. Webb, "Development and Attributes of z/Architecture*," *IBM J. Res. & Dev.* **46**, No. 4/5, 367–379 (2002).
17. N. S. Prasad, *IBM Mainframes Architecture and Design*, McGraw-Hill, New York, 1994.
18. IBM Corporation, *z/Architecture Principles of Operation* (SA22-7832-04); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
19. IBM Corporation, *Enterprise Systems Architecture/390* Principles of Operation* (SA22-7201-08); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
20. S. A. Calta, J. A. deVeer, E. Loizides, and R. N. Strangwayes, "Enterprise Systems Connection (ESCON*) Architecture*—System Overview," *IBM J. Res. & Dev.* **36**, No. 4, 535–551 (1992).
21. J. C. Elliott and M. W. Sachs, "The IBM Enterprise Systems Connection (ESCON) Architecture," *IBM J. Res. & Dev.* **36**, No. 4, 577–591 (1992).
22. T. A. Gregg, "S/390* CMOS Server I/O: The Continuing Evolution," *IBM J. Res. & Dev.* **41**, No. 4/5, 449–462 (1997).
23. D. J. Stigliani, Jr., T. E. Bubb, D. F. Casper, J. H. Chin, S. G. Glassen, J. M. Hoke, V. A. Minassian, J. H. Quick, and C. H. Whitehead, "IBM eServer* z900 I/O Subsystem," *IBM J. Res. & Dev.* **46**, No. 4/5, 421–445 (2002).
24. M. E. Baskey, M. Eder, D. A. Elko, B. H. Ratcliff, and D. W. Schmidt, "zSeries* Features for Optimized Sockets-Based Messaging: HiperSockets* and OSA-Express," *IBM J. Res. & Dev.* **46**, No. 4/5, 475–485 (2002).
25. G. H. Mealy, B. I. Witt, and W. A. Clark, "The Functional Structure of OS/360*" (Part I, Part II, Part III), *IBM Syst. J.* **5**, No. 1, 3–51 (1966).
26. C. E. Clark, "The Facilities and Evolution of MVS/ESA*," *IBM Syst. J.* **28**, No. 1, 124–150 (1989).
27. M. Mueller, L. C. Alves, W. Fischer, M. L. Fair, and I. Modi, "RAS Strategy for IBM S/390 G5 and G6," *IBM J. Res. & Dev.* **43**, No. 5/6, 875–888 (1999).
28. L. C. Alves, M. L. Fair, P. J. Meaney, C. L. Chen, W. J. Clarke, G. C. Wellwood, N. E. Weber, I. N. Modi, B. K. Tolan, and F. Freier, "RAS Design for the IBM eServer z900," *IBM J. Res. & Dev.* **46**, No. 4/5, 503–521 (2002).
29. M. L. Fair, C. R. Conklin, S. B. Swaney, P. J. Meaney, W. J. Clarke, L. C. Alves, I. N. Modi, F. Freier, W. Fischer, and N. E. Weber, "Reliability, Availability, and Serviceability (RAS) of the IBM eServer z990," *IBM J. Res. & Dev.* **48**, No. 3/4, 519–534 (2004).
30. M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
31. M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley, Reading, MA, 1996.
32. U. Vahalia, *UNIX Internals: The New Frontier*, Prentice-Hall, Upper Saddle River, NJ, 1996.
33. IBM Corporation, *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts* (SC23-4125-07); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.

34. IBM Corporation, *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs* (SC23-4128-08); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
35. IBM Corporation, *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions, Volume 1* (SC23-4159-06); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
36. IBM Corporation, *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions, Volume 2* (SC23-4160-05); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
37. IBM Corporation, *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems, Volume 1* (SC23-4163-05); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
38. IBM Corporation, *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems, Volume 2* (SC23-4164-05); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
39. IBM Corporation, *AIX 5L Version 5.3 System Management Concepts: Operating System and Devices* (SC23-4908-02); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
40. IBM Corporation, *AIX 5L Version 5.3 System Management Guide: Operating System and Devices* (SC23-4910-02); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
41. R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual Storage and Virtual Machine Concepts," *IBM Syst. J.* **11**, No. 2, 99–130 (1972).
42. R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM J. Res. & Dev.* **25**, No. 5, 483–490 (1981).
43. G. Banzhaf, F. W. Brice, G. R. Frazier, J. P. Kubala, T. B. Mathias, and V. Sameske, "SCSI Initial Program Loading for zSeries," *IBM J. Res. & Dev.* **48**, No. 3/4, 507–518 (2004).
44. C. J. Stephenson, "Fast Fits: New Methods for Dynamic Storage Allocation," *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, 1983, pp. 30–32.
45. T. Lindholm and F. Yellin, Sun MicroSystems, "*The Java Virtual Machine Specification—2nd Edition*," ISBN 0-201-43294-3, 1999.
46. M. Greenwald and D. Cheriton, "The Synergy Between Non-Blocking Synchronization and Operating System Structure," *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996, pp. 123–136.
47. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-Based Memory Management in Cyclone," *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002, pp. 282–293.
48. M. Hicks, G. Morrisett, D. Grossman, and T. Jim, "Experience with Safe Manual Memory-Management in Cyclone," *Proceedings of the 4th International Symposium on Memory Management*, 2004, pp. 73–84.
49. CerTek Software Designs, "Uniform Driver Interface: UDI Version 1.01 Specification"; see *http://www.projectudi.org/specs.html*.
50. IBM Corporation, "z/VM Performance Report: CP Disk I/O Performance"; see *http://www.vm.ibm.com/perf/reports/zvm/html/520dasd.html*.

Received May 2, 2006; accepted for publication May 24, 2006; Internet publication December 5, 2006

Alan M. Webb *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (alan_webb@us.ibm.com)*.

Mr. Webb is a Senior Software Engineer who joined IBM UK in 1977. He received an M.S. degree with distinction in software engineering from Oxford University in 1997. Mr. Webb has worked as a developer and architect in Hursley, England, and in Raleigh, North Carolina. In 1999, he joined the IBM Thomas J. Watson Research Center, where he has worked on various projects related to IBM server software technologies.

Ray Mansell *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (mansell@us.ibm.com)*.

Mr. Mansell joined the IBM UK Laboratories in 1974 after receiving a bachelor's degree in electronic engineering from the University of Bath. He joined the IBM Thomas J. Watson Research Center in 1990 and has worked on high-performance file systems, virtualization, and operating systems.

Joshua W. Knight *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (joshk@us.ibm.com)*.

Dr. Knight is a Research Staff Member. He received a B.S. degree in engineering physics from Cornell University in 1968 and a Ph.D. degree in applied physics from Stanford University in 1978. He joined IBM in 1981 and has since worked on hardware and software performance and on server software technologies.

Steven J. Greenspan *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (fletch@us.ibm.com)*.

Mr. Greenspan received his B.S. degree in computer science in 1982 from the City College of New York. That same year, he joined IBM in Poughkeepsie, New York, to work on development tools. Since 1987, he has worked on the design and development of System z operating systems.

David B. Emmes *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (emmes@us.ibm.com)*.

In 1974, Mr. Emmes received his B.A. degree in mathematics *summa cum laude* from Washington University in St. Louis. He received his M.S. degree in mathematics from M.I.T. in 1976, and his M.S. degree in computer and information science from Syracuse University in 1985. He joined IBM in 1978 and has designed and developed z/OS operating system support for real storage management, Sysplex communications, workload management, TCP/IP, and Java Virtual Machine performance enhancements. Since 2002, Mr. Emmes has been involved in the effort described in this paper, focusing on the QDIO driver and related hardware and firmware areas.