Sealing OS Processes to Improve Dependability and Safety

Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson,
James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber

Microsoft Research
One Microsoft Way
Redmond, WA 98052 USA

singqa@microsoft.com

ABSTRACT
In most modern operating systems, a process is a hardware-protected abstraction for isolating code and data. This protection, however, is selective. Many common mechanisms—dynamic code loading, run-time code generation, shared memory, and intrusive system APIs—make the barrier between processes very permeable. This paper argues that this traditional *open process architecture* exacerbates the dependability and security weaknesses of modern systems.

As a remedy, this paper proposes a *sealed process architecture*, which prohibits dynamic code loading, self-modifying code, shared memory, and limits the scope of the process API. This paper describes the implementation of the sealed process architecture in the Singularity operating system, discusses its merits and drawbacks, and evaluates its effectiveness. Some benefits of this sealed process architecture are: improved program analysis by tools, stronger security and safety guarantees, elimination of redundant overlaps between the OS and language runtimes, and improved software engineering.

Conventional wisdom says open processes are required for performance; our experience suggests otherwise. We present the first macrobenchmarks for a sealed-process operating system and applications. The benchmarks show that an experimental sealed-process system can achieve performance competitive with highly-tuned, commercial, open-process systems.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**] Coding Tools and Techniques; D.2.4 [**Software Engineering**] Software/Program Verification; D.4.1 [**Operating Systems**]: Process Management; D.4.5 [**Operating Systems**]: Reliability; D.4.6 [**Operating Systems**]: Organization and Design; D.4.7 [**Operating Systems**]: Security and Protection.

General Terms
Design, Reliability, Experimentation.

Keywords
Open process architecture, sealed process architecture, sealed kernel, software isolated process (SIP).

1. INTRODUCTION
Processes debuted, circa 1965, as a recognized operating system abstraction in Multics [48]. Multics pioneered many attributes of modern processes: OS-supported dynamic code loading, run-time code generation, cross-process shared memory, and an intrusive kernel API that permitted one process to modify directly the state of another process.

Today, this architecture—which we call the *open process architecture*—is nearly universal. Although aspects of this architecture, such as dynamic code loading and shared memory, were not in Multics' immediate successors (early versions of UNIX [35] or early PC operating systems), today's systems, such as FreeBSD, Linux, Solaris, and Windows, embrace all four attributes of the open process architecture.

The open process architecture is commonly used to extend an OS or application by dynamically loading new features and functionality directly into a kernel or running process. For example, Microsoft Windows supports over 100,000 third-party, in-kernel modules ranging in functionality from device drivers to anti-virus scanners. Dynamically loaded extensions are also widely used as web server extensions (e.g., ISAPI extensions for Microsoft's IIS or modules for Apache), stored procedures in databases, email virus scanners, web browser plug-ins, application plug-ins, shell extensions, etc. While the role of open processes in Windows is widely recognized, like any versatile technology they are widely use in other systems as well [10, 42].

1.1. Problems with Open Processes
Systems that support open processes almost always implement process isolation through hardware mechanisms such as memory management protection and differentiated

user and kernel instructions [37]. These mechanisms are not free, and their cost is a major motivation for open processes. In Aiken *et al.* [2], we measured hardware isolation costs ranging from 2.5% (in a compute-bound task with no paging) to 33% (in an IPC-bound task). This overhead arises from page table management and cache and TLB misses.

Developers avoid the performance overhead of hardware protection by using the open process architecture to closely couple software components. Software components located in a common process can communicate through shared data structures and use simple mechanisms, such as procedure calls, to transfer control. These mechanisms are particularly attractive because programming languages provide far richer data and control structures within a process than between processes.

Not surprisingly, the open process architecture has drawbacks. In particular, eliminating the isolation between a host process and an extension is a major source of software reliability, security, and compatibility problems. Although extensions are rarely trusted, verified, or fully correct, they are routinely loaded directly into a host kernel or process, with no hard interface or boundary between host and extension. The outcome is often unpleasant. For example, Swift reports that faulty device drivers cause 85% of diagnosed Windows system crashes [43]. Unpublished data from Microsoft Online Crash Analysis tools shows that in-process extensions are an important source of failure in many software products, including Windows, Word, Outlook, Exchange, and Internet Explorer. Results from a static analysis survey [9] suggest that in-process and in-kernel extensions are a major source of errors in open source code as well.

Open processes also weaken enforcement of security policies. Few, if any, operating systems provide a strong guarantee of application identity or include applications as principals in access control decisions. The complete code running in an open process cannot be known *a priori*, so an access control decision based on the identity of the program started in the process would be suspect. Instead, most systems control access to data based on the identity of an authenticated user [51].

In practice, the open process architecture also impairs software engineering. An extension gains unrestrained access to its host's memory. Extensions can, and frequently do, reach inside their host's implementation to access private data structures or functions. These undocumented and unwanted dependencies constrain the evolution of the host program and require software vendors to perform extensive compatibility testing to avoid breaking undisciplined extensions [12].

We can draw an instructive analogy between sealed processes and sealed classes in object-oriented languages. Experience with programming languages has shown that class designers and implementers need mechanisms to limit class extension and to force extensions to use declared interfaces, in order to enforce a software architecture and reduce implementation errors [7]. Similarly, sealed processes offer application developers explicit and enforceable interfaces for extensions.

Dynamic code loading also imposes less visible penalties on performance and correctness. A host program that can load code is an open environment in which it is difficult to make sound assumptions about states, invariants, or valid transitions. Consider, for example, the Java Virtual Machine (JVM). An interrupt, exception, or thread switch can invoke code that loads a new file, overwrites class and method bodies, and modifies global state to change code semantics [41]. This possibility either limits permissible compiler optimizations or requires extensive run-time support to recompile affected code.

In addition, static program analysis, which underlies both compiler optimizations and static defect detection, is complicated by open processes. A code extension must be analyzed in the context of its host environment, which can be complicated and expensive to specify and model [4]. The host itself cannot be fully or accurately analyzed without complete specification of extensions' behavior. To be practical, defect detection tools make assumptions about absent code that reduce the quality of their results.

1.2. Contributions

This paper defines a new *sealed process architecture* that addresses many shortcomings of the open process architecture. We describe the implementation and our early experience with a sealed process system. We present the first macrobenchmark results showing that a sealed-process system can have performance competitive with open-process systems even when the sealed-process system is written in a safe language with garbage collection. We also present an analysis of the costs of trade-offs of moving a major software component, the register allocator of a compiler, to a child process.

With the sealed process architecture, the OS ensures that code in a process cannot be altered once the process starts executing. The system prohibits dynamic code loading, self-modifying code, cross-process sharing of memory, and provides a process-limited kernel API.

Sealed processes offer many advantages. They increase the ability of program analysis tools to improve performance and reliability. They enable stronger security mechanisms. They can eliminate the need to duplicate OS-style access control in execution environments such as the Sun's JVM and Microsoft's Common Language Runtime (CLR). Finally, they encourage better software engineering.

We have implemented the sealed process architecture in our Singularity OS. Our experience demonstrates that the sealed process architecture and the process-based extension model it encourages are both feasible and desirable. Singularity uses sealed processes as its single extension mechanism for both the OS and applications: extension code always executes in a process distinct from its host's process. We have built a system in which all of the major subsystems and drivers reside in processes outside the kernel. We have also used the same process architecture to extend non-trivial applications, with a small programming effort and minimal effect on performance.

The rest of the paper is organized as follows. Section 2 describes the sealed process architecture and discusses its qualitative merits. Section 3 describes the implementation of sealed processes in Singularity. Sections 4 and 5 contain qualitative and quantitative evaluations of the sealed process architecture in Singularity. Section 6 describes related work and Section 7 contains conclusions and a discussion of future work.

2. SEALED PROCESS ARCHITECTURE

The sealed process architecture imposes two restrictions: the code in a process cannot be altered once the process starts executing and the state of a process cannot be directly manipulated by another process.

In this context, the definition of "code" is somewhat subjective. For a run-of-the-mill binary program, "code" would mean the binary instructions for the program. For an interpreted program, "code" would mean both the binary instructions of the interpreter as well as the code to be interpreted, whatever its form. Any case, the core restriction is that once the program has begun execution its code cannot be augmented.

A *sealed kernel* is an OS kernel that conforms to the same two restrictions: the code in the kernel cannot be altered once the kernel starts executing and the state of the kernel cannot be directly manipulated by any process. Technically, an operating system with an open kernel can provide sealed processes. However, this paper assumes that sealed processes will be implemented on top of a sealed kernel.

2.1. Sealed Process Invariants

A sealed architecture system implements maintains four invariants:

1. **The fixed code invariant:** Code within a process does not change once the process starts execution.
2. **The state isolation invariant:** Data within a process cannot be directly accessed by other processes.
3. **The explicit communication invariant**: All communication between processes occurs through explicit mechanisms, with explicit identification of the sender and explicit receiver admission control over incoming communication.
4. **The closed API invariant**: The system's kernel API respects the fixed code, state isolation, and explicit communication invariants.

The fixed code invariant does not limit the code in a process to a single executable file, but it does require that all code be identified before execution starts. A process cannot dynamically load code and should not generate code into its address space.

Because the code loaded in a process is known *a priori*, its identity can be verified against certificates provided by the code's publisher. This identity makes it is possible to assign access rights to the process [51]. For example, a process might have a security principal consisting of its authenticated user, authenticated program, and a publisher. Making a program part of a security principal enables the system to limit access to the application's data files to the application itself or its trusted peers. This is a much simpler and more robust mechanism than protecting the files with very restrictive access rights and requiring the application to assume superuser privileges to access its data.

The state isolation invariant ensures that only code within a process can directly access and modify its data. Without this invariant, a program cannot control information sharing and a developer cannot expect a program to behave predictably.

This invariant disallows communication through shared memory. Processes can pass data through a shared address space, but ownership and access to the data must be exchanged, so one process is not able to modify the contents of the memory while another is reading it.

The explicit communication invariant ensures that all communication is visible to and controllable by the system. Implicit or anonymous communication channels, such as shared memory or direct manipulation of another process's data structures, are forbidden. Furthermore, both the kernel and the recipient process have the ability to prevent communications by denying a request to establish a communication channel. Finally, the explicit communication invariant allows a process to invest one or more communication rights into a child process at creation time.

For example, in Singularity, the inter-process communication mechanisms allow a process to create a

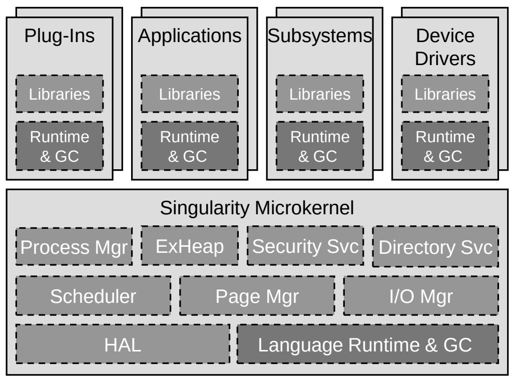

Figure 1. Singularity System Architecture.

communication channel and pass that channel (and its implied communication rights) to another process. Similarly, a process can receive a channel from one process and hand it off to a third process. The Singularity API for creating a child process allows a process to invest one or more communication rights into a child process at creation time.

In general, the explicit communication invariant ensures that a process can only communicate with the transitive closure of processes reachable through its existing communication graph (or extensions of the graph caused by creation of child processes). In practice, intermediate processes (including the kernel) and the OS communication mechanisms can further restrict the communication graph. For example, an intermediate process can use access control to decide whether to forward a communication.

The closed API invariant ensures that the API provided by the operating system to an unprivileged process does not include a mechanism to subvert the fixed code, state isolation, or explicit communication invariants. For example, the closed API invariant ensures that the base API does not include a mechanism to write to the memory of any other process.

Debuggers may need to violate some or all of these invariants in order to enable a developer to examine and control an executing process. Debuggers consequently must be privileged programs that are given access to functionality not available outside of a system's kernel. In practice, very few users write code or run debuggers, so production systems can enhance security by omitting this functionality or by limiting debuggers to read-only access.

3. SINGULARITY

Singularity is an experimental operating system under development in our lab as a basis for building more dependable applications and systems [28]. A key design criterion for Singularity was to increase isolation among software components. This motivation was based on widespread experience with Microsoft's software systems, which in general are very extensible and rely heavily on open process architectures. While we recognize both the practical and commercial benefits of open process architectures, we felt compelled to explore alternatives to improve dependability.

Singularity's sealed process architecture is constructed from a number of mechanisms: a sealed micro-kernel, light-weight *software isolated processes* (SIPs), a light-weight language runtime, light-weight inter-process communication channels, a process-limited API, light-weight threads, isolated process heaps, and verifiable code.

3.1. Sealed Kernel

Figure 1 depicts the key components of the Singularity operating system. The microkernel provides the core functionality of the system, including page-based memory management, process creation and termination, communication channels, scheduling, I/O, security, and a local directory service. The microkernel uses a hardware abstract layer (HAL) to communicate with low-level devices, such as interrupt controllers and timers. Most of Singularity's functionality resides in processes outside of the sealed kernel. In particular, all subsystems and device drivers run in separate processes.

Most of the kernel and language runtimes are verifiably type safe C# or Sing# [13], but small portions of trusted code are written in assembler, C++, or unsafe C#. All code outside the trusted computing base is written in a safe language (such as C#), translated to safe MSIL[1], and then verified and compiled to the native instruction set by the Bartok compiler [16] at install time. Currently, we trust that Bartok correctly generates safe code. In the long term we are moving to typed assembly language (TAL) to verify the safety of compiled code and to eliminate the compiler from the trusted computing base [30].

3.2. Software Isolated Processes

A Singularity process, called a software isolated process (SIP), consists of a set of memory pages, a set of threads, and a set of channel endpoints. Singularity's SIPs depend on language safety and the invariants of the sealed process architecture to provide low-cost process isolation. This isolation starts with verification that all untrusted code running in a SIP is type and memory safe. Language safety ensures that untrusted code cannot create or mutate pointers to access the memory pages of another SIP. The

[1]Microsoft Intermediate Language (MSIL) is the CPU-independent instruction set accepted by the Microsoft CLR. Singularity uses the MSIL format. Features specific to Singularity are expressed through metadata extensions in MSIL.

Singularity communication mechanisms and kernel API do not allow pointers to be passed from one SIP to another. Taken together, these mechanisms ensure the sealed process invariants, even for SIPs executing in the same address space.

A SIP starts with a single thread, enough memory to hold its code, an initial set of channel endpoints, and a small heap. It obtains additional memory by calling the kernel's page manager, which returns new, unshared pages. These pages need not be adjacent to the SIP's existing address space, since safe programming languages do not require contiguous address spaces.

Because user code is verified safe, several SIPs can share the same address space. Moreover, SIPS can safely execute at the same privileged level as the kernel. Eliminating these hardware protection barriers reduces the cost to create and switch contexts between SIPs.

Low cost, in turn, makes it practical to use SIPs as a fine-grain isolation and extension mechanism. With software isolation, system calls and inter-process communication execute significantly faster (30–500%) and communication-intensive programs run up to 33% faster than on hardware-protected operating systems. Aiken *et al.* [2] present an extensive comparison of hardware and software isolation in Singularity.

SIPs are created from a signed manifest [39]. The manifest describes the SIP's code, resources, and dependencies on the kernel and on other SIPs. All code within a SIP must be listed in the manifest. Singularity SIP manifests are entirely declarative. They describe the desired state of the application configuration after an installation, not the algorithm for installing the application. This frees the OS to employ consistent algorithms to update system configuration and to verify that an update has the desired effect.

Upon creation, SIPs receive an immutable security principal name based on their manifest. Because SIPs are sealed, security policies can place high confidence that a SIP will not be subverted by third party code. Wobber *et al.* [51] describe how the Singularity security architecture builds robust security policies on the foundation of sealed processes.

3.3. Light-Weight Language Runtime

Unlike previous systems that relied on language safety (e.g., Smalltalk, Cedar/Mesa, etc.), Singularity SIPs execute autonomously. Each SIP contains its own memory pages, language runtime, and garbage collector (GC). Moreover, even communicating SIPs need not share a common runtime or GC.

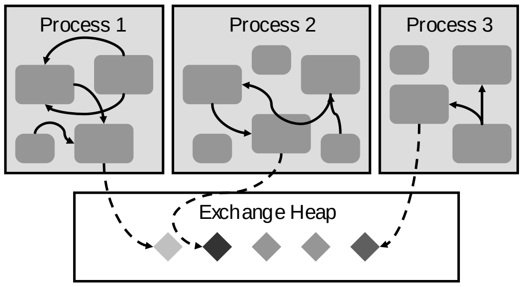

Figure 2. The Exchange Heap.

Because of the state isolation invariant, the runtime and garbage collector can employ data layout and GC algorithms appropriate for code in a particular SIP. Experience and the large number of published garbage collection algorithms strongly suggest that no one garbage collector is appropriate for all applications [17]. Singularity's sealed process architecture decouples the algorithm, data structures, and execution of each SIP's garbage collector. Each SIP can select a GC to accommodate its objectives. Moreover, the GC in a SIP can run without coordinating with any other SIP.

A light-weight, customizable runtime is an integral part of Singularity's implementation of the closed process architecture because it allows developers to use SIPs liberally without incurring large memory overheads. Because programs are compiled to native code at install time, Singularity's language runtime can be quite small. The language runtime includes a GC, exception handling mechanisms, and a limited amount of reflection to determine the type of objects at runtime. Above the language runtime sits the base class library. Because SIPs are sealed, Bartok can reduce the footprint of the runtime and base class library even further by removing unused code, a process called "tree shaking" [16].

3.4. Channels

Singularity SIPs communicate exclusively by sending messages over channels [14]. Channels enforce stronger semantics than the low-level IPC mechanisms of other systems. Channel communication is governed by statically verified channel contracts, which describe messages, message field types, and valid message interaction sequences as finite state machines.

Messages are tagged collections of values or message blocks in the Exchange Heap. Object references are excluded from messages by the type system. Messages are ownership is transferred from a sending SIP to a receiving SIP during communication.

Endpoints and message data reside in a special set of pages known as the *Exchange Heap*. The Exchange Heap is not garbage collected, but instead uses reference counts to track usage of blocks of memory (Figure 2). An allocation within the Exchange Heap is owned by one SIP at any given time with ownership enforced by static verification. When data or endpoints are sent over a channel, ownership passes from the sending SIP, which may not retain a reference, to the receiving SIP. This ownership invariant maintains the state isolation invariant and is enforced by the language using linear types and by the run-time systems [14].

Channel endpoints can be sent in messages. Thus, a communication network can evolve dynamically while conforming to the explicit communication invariant.

The operation of sending and receiving, as opposed to creating a message, entails no memory allocation. Sends are non-blocking and non-failing. Receives block synchronously until a message arrives or the send endpoint is closed.

3.5. Process-Limited API

In addition to the message-passing mechanism of channels, SIPs communicate with the kernel through a limited API that invokes static methods in kernel code. This interface isolates the kernel and SIP object spaces. All parameters to this API are values, not pointers, so the kernel's and SIP's garbage collectors need not coordinate.

The Singularity API maintains the closed API invariant. Only two API calls affect the state of another SIP. The call to create a child SIP specifies the child's manifest (identifying of all code allowed to run in the child) and gives an initial set of channel endpoints *before* the child SIP begins execution. The call to stop a child SIP stops its threads and then destroys its state.

3.6. Light-Weight Threads

Singularity employs light-weight threads to decouple execution in one SIP from another. A SIP with multiple channels must handle asynchronous message arrivals, through either an event-driven or threaded architecture. We chose threads for Singularity, though this design decision is not fundamental to the sealed process architecture. We are also investigating alternative thread synchronization, such as transactional memory [29].

A SIP can call kernel API functions to create and start threads on demand. Singularity uses linked stacks to reduce the memory overhead of a thread. These stacks grow on demand by adding non-contiguous segments of 4KB or more. The compiler performs static interprocedural analysis to optimize placement of overflow tests [47].

To reduce the overhead of API calls, Singularity does not switch stacks when a SIP calls the kernel. Instead, the Singularity runtime uses stack markers to track the ownership of stack frames so that the kernel's GC can traverse kernel frames and the SIP's GC can traverse SIP frames thus maintaining the state isolation invariant. These markers also facilitate terminating SIPs cleanly. When a SIP is killed, a kernel exception is thrown in each of its threads, which unwinds and frees the SIP's stack frames.

4. QUALITATIVE BENEFITS

The sealed process architecture increases the accuracy and precision of program analysis tools. The fixed code invariant allows a static program analysis to safely assume that it has knowledge of all code that will ever execute in a process. For example, in a sealed process, a whole-program optimizing compiler can perform aggressive inter-procedural optimization, such as eliminating from class libraries those methods, fields, and classes that are unused within a specific program.

This aggressive optimization cannot be performed *safely* in an open architecture. Imagine, for example, aggressively optimizing an open OS kernel that supports dynamically loaded device drivers. A compiler might remove an apparently unused field, to reduce the size of a data structure, only to later find that an unfortunate user loaded a driver that accesses this field.

The soundness of program correctness tools, such as Microsoft's Static Driver Verifier (SDV) [4], is limited by open process architectures. Tools of this kind are forced to make unsound, simplifying assumptions about worst case program behavior of missing code. For example, SDV incorporates a complex, conservative model of the Windows kernel's interface to device drivers. Constructing the model was a laborious process that was economically feasible only because the model can be reused across tens of thousands of device drivers. An equivalent expenditure is infeasible for most application or extension developers.

Sealed processes enable stronger security guarantees than open architectures. Code-based security systems, such as Microsoft Authenticode or Java [23], attempt to make security guarantees by validating the signature of a program residing *on disk*. However, these guarantees do not hold once the code is loaded into an open process and extended.

To compensate for the weakness of open processes, both the JVM and the CLR employ complex and expensive run-time code access security mechanisms. To estimate the cost of code access security (CAS) mechanisms in the CLR, we built a custom version of the Microsoft .NET 1.1 base class library (`mscorlib.dll`) without CAS. We removed all classes in the `System.Security.-Permissions` namespace from the CLR and all metadata tags and assertions related to it. The resultant `mscorlib.dll` file was 30% smaller than the original `mscorlib.dll`.

With a sealed process, the certification of disk contents can be extended to the executable contents of a process. A sealed architecture can guarantee that a program will execute in its certified form because of the state isolation and closed API invariants. No open process architecture can make such a claim. When coupled with hardware support for attestation [46] sealed processes can enable an execution model in which a process is a trustable entity.

Sealed processes encourage—but obviously cannot guarantee—that developers practice better software engineering by encouraging modularity and abstraction. In a sealed architecture, OS and application extensions, such as device drivers and plug-ins, can communicate only through well-defined interfaces. The process boundary between host and extension ensures that an extension interface cannot be subverted.

At least for an OS kernel, the benefits of a sealed architecture have long been at least partially recognized by the OS community. Successive generations of microkernels have explored the advantages of removing OS kernel extensions to separate processes [1, 11, 22, 24, 27] or to separate execution contexts [6, 38]. These microkernels found value in closing the kernel, but none extended this principle to applications as we have with sealed user processes.

4.1. Limitations

Although a sealed architecture offers benefits, it also imposes costs. The most prominent is that the set of programs that naturally conforms to a sealed architecture is limited by design. To encourage adoption, a system must provide mechanisms that are general enough to supplant prohibited techniques.

4.1.1 Communications

Communication that passes through memory shared between processes is notoriously prone to concurrency bugs. Sealed processes can prevent this class of errors, but they require message passing, which is also difficult to use correctly. Common message-passing errors include marshalling code that violates type-safety properties and communication protocol violations that lead to deadlocks and livelocks. These problems can be mitigated by programming language extensions that concisely specify communication protocols and by verification tools [14].

Sealed processes also increase the complexity of writing program extensions, as the host program's developer must define a proper interface that does not rely on sharing data structures. The extension's developer must program to this interface and possibly re-implement or import functionality available in the parent. Nevertheless, the widespread problems caused by dynamically loaded extensions strongly argue for increasing isolation between an extension and its parent. Singularity demonstrates that an out-of-process extension mechanism works for applications as well as system code. This general extension mechanism does not depend on the semantics of a specific interface, unlike domain-specific approaches such as Nooks [43]. Finally, the extension mechanism provides simple semantic guarantees that can be understood by developers and used by tools for defect detection.

4.1.2 Dynamic Code Generation

The inability to generate code into a running sealed process precludes common coding practices such as just-in-time compilation and the inline compilation of abstractions such as a regular expression. In a sealed architecture, these mechanisms must generate the code into a separate process, start this process, and communicate over a well-defined communication channel. We are investigating mechanisms, such as compile-time reflection, which can eliminate some uses of dynamic code generation [15]. Note that programming language interpreters can still be written in the sealed process architectures, since the interpreted program just invokes code that is part of the process.

4.1.3 Data Sharing

In an open process OS, shared libraries and DLLs are a common way to reduce the code footprint of the system. For example, a typical Windows Sever 2003 system running a mix of user applications shows 74 processes using a total of 5.4GB of virtual address space, but only 760MB of private data/code. Furthermore, in an open process system, extensions co-located in a single process share read-write, as well as read-only data.

A sealed process system can share read-only code and data pages among processes, which is similar to shared libraries in conventional systems. There is, however, a tradeoff between extensive compiler optimization, which customizes a library for a particular process and prevents code sharing, and sharing libraries among processes.

4.1.4 Scheduling Overhead

Since a sealed process can only be extended by creating a child process, it may require more processes than a conventional system to service a request. These additional processes introduce communication overheads, such as data copying, and exacerbate the scheduling problem faced by the kernel. In particular, a large number of (non-blocked) threads may preclude sophisticated scheduling algorithms. Multiplexing server processes among multiple clients also makes resource accounting more difficult.

In practice, we have found that a typical Singularity system contains a large number of threads (hundreds of threads are not uncommon even on small systems), but the number of non-blocked threads at any time is usually quite small. However, this may not be true for all sealed architectures.

5. QUANTITATIVE EVALUATION

The sealed process architecture represents a significant departure from the open process architecture in general use. In this section we evaluate the sealed process architecture by qualitatively comparing the performance of the current version of Singularity with existing open architecture systems, most notably Windows Server 2003 R2. Singularity is a research prototype. The evaluation in this section should be considered as more demonstrative than definitive.

All experiments were run using an AMD Athlon 64 X2 3800+ CPU(2 GHz, with second core disabled) with an NVIDIA nForce4 Ultra chipset, 1GB RAM, a Western Digital WD2500JD 250GB 7200RPM SATA disk (command queuing disabled), and the nForce4 Ultra native Gigabit NIC (TCP offload disabled). Versions of systems used were FreeBSD 5.3, Red Hat Fedora Core 4 (kernel 2.6.11-1.1369_FC4), and Windows Server 2003 R2.

5.1. SPECweb99

To quantify the overhead of the sealed process architecture, we measured the performance of Singularity and Windows running the SPECweb99 benchmark [40]. SPECweb99 measures the maximum number of simultaneous connections an HTTP server can support, while maintaining a minimum bandwidth of 320 Kbps on each connection. The benchmark consists of both static and dynamic content. Static content is selected using a Zipf distribution consisting of 35% files smaller than 1KB, 50% files larger than 1KB and smaller than 10KB, 14% files larger than 10KB and smaller than 100KB, and 1% files larger than 100KB and smaller than 1MB.

The Singularity implementation uses six SIPs: a NIC driver, the TCP/IP stack, the HTTP server, the SPECwebApp which processes both dynamic and static content requests, the file system, and a disk driver (Figure 3). The Singularity implementation of SPECweb99 is not fully conformant to the requirements for formal benchmarking. In particular, our HTTP server does not support logging and we used a slightly smaller execution time and both the Singularity and Windows tests use smaller ramp up time (30 seconds) and run time (180 seconds) than is required for formal SPECweb99 results.

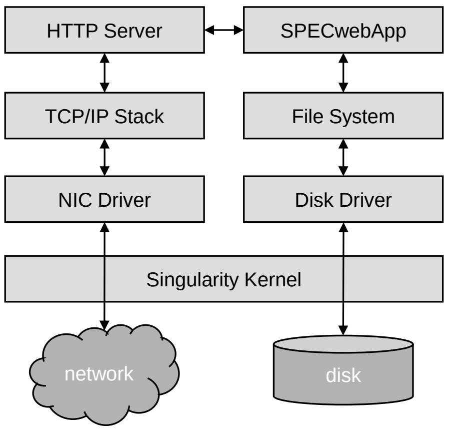

Figure 3. SPECweb99 in six SIPs on Singularity.

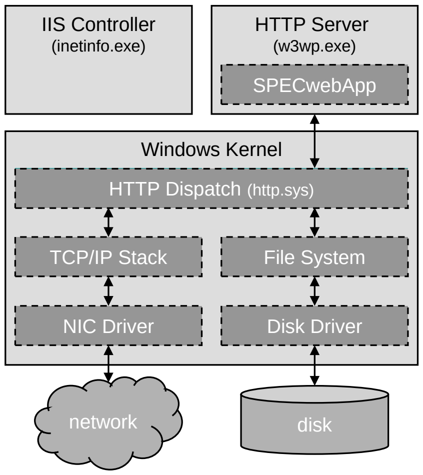

Figure 4. SPECweb99 on Windows

The Windows implementation of SPECweb99 runs on IIS 6.0 and takes advantage of all optimizations available in an open process architecture (Figure 4). In this implementation, the NIC driver, TCP/IP stack, disk driver, file system, and HTTP dispatcher code are all loaded into the Windows kernel using the device driver extension model. The HTTP dispatcher transfers content directly from the file system (or file system cache) to the TCP/IP stack without leaving the kernel. Dynamic content requests travel directly from the `http.sys` driver in the kernel to the IIS worker process, which contains the SPECwebApp dynamic content plug-in running as an ISAPI extension. The `inetinfo.exe` controller process is executed only on the first dynamic content request to start the worker process.

Table 1 shows the comparative performance of the Singularity and Windows running the SPECweb99 benchmark. In its current implementation, the Singularity system can sustain less than half (244/519) of the number of concurrent connections as the Windows system. The Singularity system is CPU bound whereas the Windows system is bound by disk I/O; it runs at roughly 60% CPU utilization. Average response time for the two systems is quite close; Singularity has an average response time with 5% of Windows.

Four factors accounts for much (48%) of the CPU utilization performance in the Singularity implementation (Table 2): the overhead of selecting which message has been received in a Sing# "switch receive" statement, the scheduler and raw send/receive operations on channels, the cost of allocating and duplicating and garbage collecting in-process byte[] arrays, and the cost of allocating cross-process ExHeap byte vectors. We believe that the "switch receive" statement cost can be significantly improved as it has a minimum overhead of approximately 1600 cycles over the cost of a similar non-switched receive operation.

Our experience as we have driven from prototype code to "commercial grade" code is that we can substantially reduce the costs of both byte[] arrays and ExHeap byte vectors. The current implementation of the NetStack is an awkward port of C# code from the CLR to Singularity. Data is copied from byte vectors at the top of the network stack into byte[] arrays. The network stack then processes data in the form of these byte[] arrays, and then they are copied into new byte vectors at the bottom of the network stack to send to the NIC driver.

A more natural style in Singularity would be to use byte vectors throughout. The results of such a conversion can be seen in the file system which spends just 1% of its total CPU time in garbage collection (Table 3). This contrasts with the HTTP server which spends over 11% of its CPU time in garbage collection. In the extreme case, the disk driver does no memory allocation as it exclusively uses byte vectors passed in from higher layers. The NIC drivers uses byte vectors on the transmit, but not the receive path. We see significant room for improvement in the TCP/IP stack, HTTP Server, and SPECwebApp as these are converted to zero-copy implementations using byte vectors.

5.2. Register Allocator

To further evaluate the practicality of sealed processes and process-based extension, we modified the Bartok compiler by moving its register allocator into a child SIP and using the modified compiler to compile the Singularity kernel.

Table 1. Performance on SPECweb99 benchmark.

System	Concurrent Connections (higher is better)	Average Response in ms. (lower is better)
Windows	519	332.1
Singularity	244	348.1

Table 2. CPU usage by Major Function for Singularity.

Overhead Function	% of Total CPU
Message Switch-Receive	16.61%
Message Receive operations	8.59%
Heap Byte[] operations	18.41%
ExHeap Byte[] operations	4.45%
Total	48.06%

Table 3. CPU usage by SIP for SPECweb99 on Singularity.

Process	% of Total CPU	% of Total CPU in GC	% in GC
NIC Driver	5.81%	0.35%	6.09%
TCP/IP Stack	44.51%	3.17%	7.12%
HTTP Server	15.44%	1.80%	11.65%
SPECwebApp	7.88%	0.67%	8.52%
File System	25.96%	0.30%	1.17%
Disk Driver	0.40%	0.00%	0.00%
Kernel	n/a	0.84%	n/a

The register allocator is a good example of a complex component that might be an extension to a large application.

The register allocator runs as one of roughly 50 phases in the Bartok compiler. Given a graph representing register lifetimes in a single function with unlimited virtual registers, the allocator implements a graph-coloring algorithm to assign virtual registers to physical registers. In the SIP, it updates the function graph to assign real registers and inserts register spill instructions as necessary. The register allocator uses various data structures that describe the physical hardware, calling conventions, etc.

The Bartok compiler is approximately 220K lines of C# code, a 5.7MB executable. The register allocator is approximately 10K lines of C# code, but when compiled with its necessary libraries, it is a 1.6MB executable.

The register allocator is a challenging extension. It exchanges numerous, complex data structures with its host. The interface between the compiler and allocator has 156 unique data structure classes. Over the 6,441 functions in the Singularity kernel, the median invocation of the allocator marshals approximately 2,400 unique objects. The smallest invocation marshaled over 1,400 unique objects and the largest invocation marshaled over 30,000 unique objects. The total data marshaled per invocation ranged from 50KB to 1.5MB, with a median of 85KB.

Our strategy with the Bartok implementation was to produce the smallest set of changes with "reasonable" performance. The changes consisted primarily of annotating classes that needed to be marshaled. Beyond tagging non-updated objects, we spent no effort to reduce communication across the interface between Bartok and the register allocator. For example, we did not alter the data structures within Bartok to more cleanly separate parent-only state from exchanged state.

Table 4 summarizes the execution time in cycles and overhead for running the Bartok register allocator as a child SIP. The version of Bartok with the separate child SIP executes approximately 11% slower then original version. While the overhead is not trivial, it is small enough that its cost might easily be justified for the improved system dependability. Furthermore, the cost might also be reduced by altering the data structures to better segregate the data structures required by the child SIP and therefore reduce marshalling costs. The overheads might be completely eliminated by using linear Sing# types that can be exchanges between SIPS without marshalling.

5.3. Program Complexity

The sealed architecture replaces access to shared memory and shared functions with explicit message passing. Hosts and extensions in sealed processes must incorporate code for communicating; in an open architecture they could directly access each other's state. Developers must now write contracts explicit communication code, and only use information obtained through the host's published interfaces. The resulting benefit is communication sufficiently explicit to be statically verified.

Table 5 summarizes the additional cost of explicit communication in the Cassini web server. The original Cassini web server was implemented on the CLR and exchanged state with plug-ins through a shared property bag structure for each HTTP request. On Singularity, Cassini uses two contracts (one for page requests and one for the HTTP properties). We added 263 lines of IPC code in the web server, 52 lines of channel contract, and 76 lines of extension stub, for a total increase of 391 lines (26%) over the original.

Table 6 summarizes the work required to turn the Bartok register allocator into an extension. We modified 3 lines of code in the Bartok compiler to invoke the register allocator as a child process and added about 400 lines of channel contracts and boilerplate code to turn the allocator into a separate program and to have Bartok create a child SIP. We added one line annotations (using C# custom attributes) to tag 107 classes as having objects that needs to be copied only once to the child SIP. We also applied a tool that automatically generates the class marshalling code from the compiled MSIL files for Bartok. Overall, the changes were minor and consisted primarily of annotating classes that needed to be marshaled. Beyond tagging non-updated objects, we spent no effort to reduce communication across the interface between Bartok and the register allocator. For example, we did not alter the data structures within Bartok to more cleanly separate parent-only state from exchanged state. Total code change for Bartok was less than 0.25%.

Table 4. Performance of Bartok Compiler with Register Allocator as an in-process module or as a child SIP.

Implementation	Cycles	Normalized
Original, In-Process	194.7 billion	1.000
Child SIP	215.9 billion	1.109
Overhead for SIP		**10.9%**

Table 5. Code added to Cassini for explicit communication.

Code Description	Lines	% of Orig.
Original web server	1486	100%
New host code	263	18%
New channel contract	52	3%
New extension code	76	5%
Total	**1877**	**126%**

Table 6. Code added to Bartok to move register allocator into a child process.

Code Description	Lines	% of Orig.
Original Bartok compiler	220,000	100.00%
Altered lines in Bartok	3	0.00%
Channel contract and child SIP	400	0.18%
Custom attribute tags	107	0.05%
Total	**220,510**	**100.23%**

While the 26% increase in code in Cassini is non-trivial, our experience suggests that this is the upper bound. In practice, the brunt of additional code is paid by the host developer and re-used across many extensions. For example, of 9445 lines of device driver code in Singularity, 1597 lines (17%) are related to interprocess communication with either client SIPs or the I/O subsystem.

5.4. Improved Static Analysis

Sealed processes offer improved opportunities for static analysis because all code that will run in a process is known before the process begins execution. Static analysis is available to any process architecture, but sound static analysis of a complete process is possible only when the code is fixed and known in advance.

Table 7, tree shaking can reduce program code size by as much as 75%. Most importantly, tree shaking of extensible programs, such as the web server, can reduce code size by as much as 72%. The latter is important because without sealed processes, the compiler could not remove code because it would not know which code might be required by future plug-ins. On Singularity, even though a plug-in includes its own libraries (as it runs in a separate SIP), the combined code size of the web server and the SPECweb99 plug-in is still 54% smaller than just the web server alone before tree shaking.

Other static analysis tools and techniques benefit from sealed processes. For example, Bartok checks that methods annotated `[NoAlloc]` do not invoke any code that might perform a heap allocation. This check is useful for verifying that portions of the kernel, such as interrupt handlers, do not allocate memory.

5.5. Costs of Primitive Operations

Table 8 reports the cost of primitive operations in Singularity and three other systems. For each system, we conducted an exhaustive search to find the cheapest API call. The FreeBSD and Linux "thread yield" tests use user-space scheduled pthreads, as kernel scheduled threads performed significantly worse. Windows and Singularity both used kernel scheduled threads. The "message ping pong" test measured the cost of sending a 1-byte message from one process to another. On FreeBSD and Linux, we used sockets, on Windows, a named pipe, and on Singularity a channel with a single message argument.

A basic thread operation, such as yielding the processor, is roughly three times faster on Singularity than the other systems. ABI calls and cross-process operations run significantly faster (5 to 10 times faster) than the mature systems because of Singularity's SIP architecture.

Singularity's process creation time is significantly lower than the other systems because SIPs do not need MMU page tables and because Singularity does not need to maintain extra data structures for dynamic code loading. Process creation time on Windows is significantly higher than other systems because of its extensive side-by-side compatibility support for dynamic load libraries.

Singularity encorporates first class language support to achieve zero-copy communication between SIPs [14]. Soundness of zero-copy semantics are verified by static analysis on the entire contents of sealed process. Table 9 shows the cost of sending a payload message from one process to another on Singularity, Linux, and Windows for comparison.

Table 7. Reduction in code size of OS and SpecWeb99 components via tree shaking, enabled by sealed processes.

Program	Whole	w/ Tree Shake	% Reduction
Kernel	2371 KB	1291 KB	46%
Web Server	2731 KB	765 KB	72%
SPECweb99 Plug-in	2144 KB	502 KB	77%
Ide Disk Driver	1846 KB	455 KB	75%

Table 8. Cost of basic operations.

System	Cost (in CPU Cycles)			
	API Call	Thread Yield	Message Ping/Pong	Create Process
Singularity	91	346	803	352,873
FreeBSD	878	911	13,304	1,032,254
Linux	437	906	5,797	719,447
Windows	627	753	6,344	5,375,735

Table 9. IPC costs.

Message Size (bytes)	CPU Cycles		
	Singularity	Linux	Windows
4	933	5,544	6,641
16	928	5,379	6,600
64	942	5,549	6,999
256	929	5,519	7,353
1,024	926	5,971	10,303
4,096	919	8,032	17,875
16,384	928	19,167	47,149
65,536	920	87,941	187,439

6. RELATED WORK

The large amount of related work can be divided to two major areas: OS architecture and specific mechanisms for extension isolation.

6.1. OS Architecture

Microkernel operating systems, such as Mach [1], L4 [24], SPIN [6], VINO [38], Taos/Topaz [45], and the Exokernel [11], partition the components of a monolithic operating system kernel into separate processes to increase the system's failure isolation and reduce development complexity. The sealed process architecture generalizes this sound engineering methodology (modularity) to the entire system. Singularity provides lightweight processes and inexpensive interprocess communication, which enable a partitioned application to communicate effectively.

Previous systems did not seal the kernel or processes. Hardware-enforced process isolation has considerable overhead, and so microkernels evolved to support kernel extensions, while adopting mechanisms to protect system integrity. SPIN was closest to Singularity, as its extensions were written in a safe language and relied on language features to restrict access to kernel interfaces [6]. Vino used sandboxing to prevent unsafe extensions from accessing kernel code and data and lightweight transactions to control resource usage [38]. However, both systems allowed extensions to directly manipulate kernel data, which left open the possibility of corruption through incorrect or malicious operations and of inconsistent data after extension failure. Exokernel defined kernel extensions for packet filtering in a domain-specific language and generated code in the kernel for this safe, analyzable language [19]. None of these systems generalized the mechanisms for system extensions to isolate application extensions.

Other operating systems were written in safe programming languages, but all used open process architectures. These range from single-user completely open systems [21, 50] to closed kernels with open processes [5, 44].

Several recent operating systems have included major components written in Java. JavaOS is a port of the Java virtual machine to bare hardware [36]. It replaces a host operating system with a microkernel written in an unsafe language and a JVM hosting Java code libraries. JavaOS supports a single open process shared between all applications.

The JX system [22] is very similar to JavaOS, but improves on it by moving each process (called a domain in JX) in the shared system address space into its own set of pages with its own GC heap. JX incorporates an open process architecture because domains can be altered through dynamic class loading. JX supports shared memory regions called memory portals, which require that each domain use the same class library and runtime system.

Software attestation solutions, such as Terra [20] and NGSCB [33] provide a strong identity based on the hash of the code in a process at a specific point in execution. However, neither Terra nor NGSCB prevents a process from loading new untrusted code or sharing its memory with other processes after attestation.

6.2. Extension Isolation

Independent of sealed processes, there has been many attempts to alleviate problems caused by open processes.

Device drivers are both the most common operating system extension and largest prime source of defects [9, 31, 43]. Nooks provides a protected environment in the Linux kernel to execute existing device drivers [43]. It uses memory management hardware to isolate a driver from kernel data structures and code. Calls across this protection boundary go through the Nooks runtime, which validates parameters and tracks memory usage. Singularity, without the pressure for backward compatibility, provides mechanisms (SIPs and channels) that are general programming constructs, suitable for application and system code, as well as for device drivers.

Software fault isolation (SFI) isolates code in its own domain by inserting run-time tests to validate memory references and indirect control transfers, a technique called sandboxing [49]. Sandboxing can incur high costs and only provides memory isolation between a host and an extension. It does not offer the full benefits of language safety for either the host or extension. Sandboxing also does not control data shared between the two, so they remain coupled in case of failure.

Sun's JVM and Microsoft's CLR are execution environments that use fine-grain isolation and security mechanisms to compensate for the weaknesses of open processes. Both are open environments that encourage dynamic code loading (e.g., Applets) and run-time code generation. Both require complex and expensive security mechanisms and policies, such as Java's fine grain access control or the CLR's code access security, to restrict the behavior of extensions [32]. These mechanisms are difficult to use properly and impose considerable overhead. Singularity runs extensions in separate sealed SIPs, which provide a stronger assurance of isolation and a more tractable security model imposed at process granularity.

In both the JVM and CLR, computations sharing a process are not isolated upon failure. A shared object can be left in an inconsistent or locked state when a thread fails [18]. When a program running in a JVM fails, the entire JVM process typically is restarted because it is difficult to isolate and discard corrupted data and find a clean point to restart the failed computation [8].

Other projects have implemented OS-like functionality to control resource allocation and sharing and facilitate cleanup after failure in open environments. J-Kernel implemented protection domains in a JVM process, provided revocable capabilities to control object sharing, and developed clean semantics for domain termination [25]. Luna refined the J-Kernel's run-time mechanisms with an extension to the Java type system that distinguishes shared data and permits control of sharing [26]. KaffeOS provides a process abstraction in a JVM along with mechanisms to control resource utilization in a group of processes [3]. Java has incorporated many of these ideas into a new feature called isolates [34] and Microsoft's CLR has had a similar concept called AppDomains since its inception.

By embracing the sealed process architecture, Singularity eliminates the duplication between an operating system and these run-time systems by providing a consistent extension mechanism across all levels of the system.

Singularity's SIPs are sealed and non-extensible, which provides a greater degree of isolation and fault tolerance than Java or CLR approaches.

7. CONCLUSIONS AND FUTURE WORK

In a quest to improve system dependability, we have defined a new sealed process architecture that offers a number of important advantages over the widely used open process architecture. At one level, this architecture compels explicit interfaces between a program and its extensions and prohibits shared data structures. This discipline can improve reliability in the presence of extension failures, a well-known source of unreliability in operating systems and applications. These restrictions also allow defect detection tools and compilers to make sound assumptions about program behavior. And, they encourage applications and developers to practice better software engineering. At another level, sealed processes enable an operating system to provide stronger security guarantees. They also eliminate the need to replicate OS access control mechanisms in language runtimes such as the JVM and CLR.

We implemented the sealed process architecture in the Singularity OS and demonstrated that a sealed process OS can offer performance competitive with commercial open process systems. Our results suggest that the restrictions of sealed processes are not overly burdensome and are at least partially compensated by improved detection of coding errors.

We believe the sealed architecture shows sufficient promise to merit further consideration by the research community. We see two important avenues of future research. The first is to implement more extensible programs on the architecture to further evaluate its validity and utility. For example, porting a large database server would be a valuable experiment. The second is to evaluate the feasibility of sealed processes in a hardware-protected operating system, such as Windows, Linux, or MINIX 3.

8. REFERENCES

1. Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. Mach: A New Kernel Foundation for UNIX Development. In *Summer USENIX Conference*, Atlanta, GA, 1986, 93-112.
2. Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G. and Larus, J. Deconstructing Process Isolation *2006 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC 2006)*, Microsoft Research, San Jose, CA, 2006.
3. Back, G., Hsieh, W.C. and Lepreau, J. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, San Diego, CA, 2000.
4. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K. and Ustuner, A. Thorough Static Analysis of Device Drivers In *Proceedings of the EuroSys 2006 Conference*, Leuven, Belgium, 2006.
5. Barnes, F., Jacobsen, C. and Vinter, B. RMoX: A Raw-Metal occam Experiment. In *Communicating Process Architectures*, IOS Press, Enschede, the Netherlands, 2003, 269-288.
6. Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M., Becker, D., Eggers, S. and Chambers, C. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, 1995, 267-284.
7. Biberstein, M., Gil, J. and Porat, S. Sealing, Encapsulation, and Mutability. In *Proceeedings of the 15th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science*, Springer-Verlag, Budapest, Hungary, 2001.
8. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G. and Fox, A. Microreboot—A Technique for Cheap Recovery. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, 2004, 31-44.
9. Chou, A., Yang, J., Chelf, B., Hallem, S. and Engler, D. An Empirical Study of Operating Systems Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Alberta, Canada, 2001, 73-88.
10. de Goyeneche, J.-M. and de Sousa, E.A.F. Loadable Kernel Modules. *IEEE Software*, *16* (1). 65-71.
11. Engler, D.R., Kaashoek, M.F. and O'Toole, J., Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, 1995, 251-266.
12. Erlingsson, Ú. and MacCormick, J. Ad hoc Extensibility and Access Control. *ACM Operating Systems Review*, *40* (3). 93-101.
13. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R. and Levi, S., Language Support for Fast and Reliable Message Based Communication in Singularity OS. In *Proceedings of the EuroSys 2006 Conference*, Leuven, Belgium, 2006, 177-190.
14. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R. and Levi, S. Language Support for Fast and Reliable Message Based Communication in Singularity OS. In *Proceedings of the EuroSys 2006 Conference*, ACM, Leuven, Belgium, 2006, 177-190.
15. Fähndrich, M., Carbin, M. and Larus, J., Reflective Program Generation with Patterns. In *5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, Portland, OR, 2006.
16. Fitzgerald, R., Knoblock, T.B., Ruf, E., Steensgaard, B. and Tarditi, D. Marmot: an Optimizing Compiler for Java. *Software-Practice and Experience*, *30* (3). 199-232.
17. Fitzgerald, R. and Tarditi, D. The Case for Profile-directed Selection of Garbage Collectors. In *Proceedings of the 2nd International Symposium on Memory Management (ISMM '00)*, Minneapolis, MN, 2000, 111-120.
18. Flatt, M. and Findler, R.B. Kill-safe Synchronization Abstractions. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI 04)*, Washington, DC, 2004, 47-58.

19. Ganger, G.R., Engler, D.R., Kaashoek, M.F., Briceño, H.M., Hunt, R. and Pinckney, T. Fast and Flexible Application-level Networking on Exokernel Systems. *ACM Transactions on Computer Systems*, *20* (1). 49-83.
20. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M. and Boneh, D. Terra: A Virtual-Machine Based Platform for Trusted Computing In *Proceedings for the 19th ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, 2003.
21. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
22. Golm, M., Felser, M., Wawersich, C. and Kleinoeder, J. The JX Operating System. In *Proceedings of the USENIX 2002 Annual Conference*, Monterey, CA, 2002, 45-58.
23. Gosling, J., Joy, B. and Steele, G. *The Java Language Specification*. Addison Wesley, 1996.
24. Härtig, H., Hohmuth, M., Liedtke, J. and Schönberg, S. The Performance of μ-kernel-based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint Malo, France, 1997, 66-77.
25. Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D. and Eicken, T.v. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998, 259-270.
26. Hawblitzel, C. and Eicken, T.v. Luna: A Flexible Java Protection System. In *Proceedings of the Fifth ACM Symposium on Operating System Design and Implementation (OSDI '02)*, Boston, MA, 2002, 391-402.
27. Herder, J.N., Bos, H., Gras, B., Homburg, P. and Tanenbaum, A.S. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *Operating System Review*, *40* (3). 80-89.
28. Hunt, G.C., Larus, J.R., Tarditi, D. and Wobber, T., Broad New OS Research: Challenges and Opportunities. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, 2005, 85-90.
29. Larus, J.R. and Rajwar, R. *Transactional Memory*. Morgan & Claypool, 2006.
30. Morrisett, G., Walker, D., Crary, K. and Glew, N. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, *21* (3). 527-568.
31. Murphy, B. and Levidow, B. Windows 2000 Dependability. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, New York, NY, 2000.
32. Paul, N. and Evans, D. NET Security: Lessons Learned and Missed from Java. In *20th Annual Computer Security Applications Conference (ACSAC)*, Tucson, AZ, 2004, 272-281.
33. Peinado, M., Chen, Y., England, P. and Manferdelli, J. NGSCB: A Trusted Open System. In *Proceedings of the 9th Australasian Conference on Information Security and Privacy (ACISP)*, Sydney, Australia, 2004.
34. Process, J.C. Application Isolation API Specification *Java Specification Request*, 2003, JSR-000121.
35. Ritchie, D. and Thompson, K. The UNIX Time-Sharing System. *Communications of the ACM*, *17* (7). 365-375.
36. Saulpaugh, T. and Mirho, C. *Inside the JavaOS Operating System*. Addison-Wesley, 1999.
37. Schroeder, M.D. and Saltzer, J.H. A Hardware Architecture for Implementing Protection Rings In *Proceedings of the Third ACM Symposium on Operating Systems Principles (SOSP)*, ACM, Palo Alto, CA, 1971.
38. Seltzer, M.I., Endo, Y., Small, C. and Smith, K.A. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI 96)*, Seattle, WA, 1996, 213-227.
39. Spear, M.F., Roeder, T., Levi, S. and Hunt, G. Solving the Starting Problem: Device Drivers as Self-Describing Artifacts. In *Proceedings of the EuroSys 2006 Conference*, Leuven, Belgium, 2006.
40. SPEC *SPECweb99 Release 1.02*. Standard Performance Evaluation Corporation Warrenton, VA, 2000.
41. Sreedhar, V.C., Burke, M. and Choi, J.-D. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI 00)*, Vancouver, BC, 2000, 196-207.
42. Stein, L. and MacEacbern, D. *Writing Apache Modules with Perl and C*. O'Reilly, 1999.
43. Swift, M.M., Bershad, B.N. and Levy, H.M. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, 2003, 207-222.
44. Thacker, C., Stewart, L.C. and Satterthwaite, E., Firefly: A multiprocessor workstation. . Technical Report SRC-023, DEC SRC, 1987.
45. Thacker, C.P. and Stewart, L.C. Firefly: a Multiprocessor Workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, 1987, 164-172.
46. Trusted Computing Group, Trusted Platform Module Specification Version 1.2 Revision 94. Technical Report 2006.
47. von Behren, R., Condit, J., Zhou, F., Necula, G.C. and Brewer, E. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, 2003, 268-281.
48. Vyssotsky, V.A., Corbató, F.J. and Graham, R.M. Structure of the Multics supervisor. In *AFIPS Conference Proceedings 27, 1965 Fall Joint Computing Conference (FJCC)*, Spartan Books, Washington, DC, 1965, 203-212.
49. Wahbe, R., Lucco, S., Anderson, T.E. and Graham, S.L. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, Asheville, NC, 1993, 203-216.
50. Weinreb, D. and Moon, D. *Lisp Machine Manuel*. Symbolics, Inc, Cambridge, MA, 1981.
51. Wobber, T., Abadi, M., Birrell, A., Simon, D.R. and Yumerefendi, A., Authorizing Applications in Singularity. In *Proceedings of the EuroSys2007 Conference*, Lisbon, Portugal, 2007.