An Overview of the Singularity Project[1]

Galen Hunt, James Larus, Martín Abadi, Mark Aiken, Paul Barham,
Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy,
Bjarne Steensgaard, David Tarditi, Ted Wobber, Brian Zill[2]

Microsoft Research
One Microsoft Way
Redmond, WA 98052

http://research.microsoft.com/os/singularity

Microsoft Research Technical Report
MSR-TR-2005-135

Abstract. Singularity is a research project in Microsoft Research that started with the question: what would a software platform look like if it was designed from scratch with the primary goal of dependability? Singularity is working to answer this question by building on advances in programming languages and tools to develop a new system architecture and operating system (named Singularity), with the aim of producing a more robust and dependable software platform. Singularity demonstrates the practicality of new technologies and architectural decisions, which should lead to the construction of more robust and dependable systems.

[1] This report is a snapshot of a project in motion. In particular, performance measurements are preliminary and subject to improvement. Please contact Galen Hunt (galenh@microsoft.com) or Jim Larus (larus@microsoft.com) or check at http://research.microsoft.com/os/singularity for the latest results and citation information.

[2] Martín Abadi is affiliated with the Computer Science Department, University of California at Santa Cruz. Paul Barham is at Microsoft Research Cambridge. Martín Abadi, Nick Murphy, and Ted Wobber are at Microsoft Research Silicon Valley.

We've also been fortunate to have a great group of interns who made many contributions to this project over the past couple years: Michael Carbin, Fernando Castor, Adam Chlipala, Jeremy Condit, Daniel Frampton, Chip Killian, Prince Mahajan, Bill McCloskey, Martin Murray, Martin Pohlack, Tom Roeder, Avi Shinnar, Mike Spear, Yaron Weinsberg, and Aydan Yumerefendi.

1 Introduction

Software runs on a platform that has evolved over the past 40 years and is increasingly showing its age. This platform is the vast collection of code—operating systems, programming languages, compilers, libraries, run-time systems, middleware, etc.—and hardware that enables a program to execute. On one hand, this platform is an enormous success in both financial and practical terms. The platform forms the foundation of the $179 billion dollar packaged software industry [3] and has enabled revolutionary innovations such as the Internet. On the other hand, the platform and software running on it are less robust, reliable, and secure than most users (and developers!) would wish.

Part of the problem is that our current platform has not evolved far beyond the computer architectures, operating systems, and programming languages of the 1960's and 1970's. The computing environment of that period was very different from today's milieu. Computers were extremely limited in speed and memory capacity; used only by a small group of technically literate and non-malicious users; and were rarely networked or connected to physical devices. None of these characteristics remains true, but modern computer architectures, operating systems, and programming languages have not evolved to accommodate a fundamental shift in computers and their use.

Singularity is a research project in Microsoft Research that started with the question: what would a software platform look like if it was designed from scratch with the primary goal of dependability, instead of the more common goal of performance?[3] Singularity is working to answer this question by building on advances in programming languages and programming tools to develop and build a new system architecture and operating system (named Singularity), with the aim of producing a more robust and dependable software platform. Although dependability is difficult to measure in a research prototype, Singularity shows the practicality of new technologies and architectural decisions, which should lead to more robust and dependable systems in the future.

With its exponential rate of progress, hardware evolution commonly appears to drive fundamental changes in systems and applications. Software, with its more glacial progress, rarely creates opportunities for fundamental improvements. However, software does evolve, and its change makes it possible—and necessary—to rethink old assumptions and practices. Advances in programming languages, run-time systems, and program analysis tools provide the building blocks to construct architectures and systems that are more dependable and robust than existing systems:

- Expressive, safe programming languages, such as Java and C#. Type safety ensures a value or object is always correctly interpreted and manipulated. Memory safety ensures a program references memory only within the bounds of valid, live objects.
- Optimizing compilers and high performance run-time systems generate safe code that runs at speeds comparable to unsafe code [20]. These compilers, unlike the more common just-in-time (JIT) compilers, perform global optimizations that mitigate safety-related overhead. Garbage collectors in these systems reclaim memory with overhead comparable to that of explicit deallocation.

---

[3] We use the term "dependability" rather than "reliability." IFIP WG10.4 on Dependable Computing and Fault Tolerance defines the terms as follows:

"The notion of dependability, defined as the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers, enables these various concerns to be subsumed within a single conceptual framework. Dependability thus includes as special cases such attributes as reliability, availability, safety, security." [29]

- Validation techniques ensure the end-to-end type safety of the compiler, compiled code, and run-time system. Typed intermediate and assembly language validate the proper operations of system components and ensure the language safety guarantees that underlie system correctness.
- Sound, specification-driven defect detection tools ensure the correctness of many aspects of the system. A sound tool finds all occurrences of an error—along with false positives—and consequently can reliably indicate when a particular defect has been eliminated. Specification-driven tools do not look for a hardwired collection of defects. They are extensible and can be adapted to check that many program or library-specific abstractions are used correctly.

Languages and tools based on these advances are in use detecting and preventing programming errors. Less well explored is how these mechanisms enable deep changes in system architecture, which in turn might advance the ultimate goal of preventing and mitigating software defects [28].

The rest of this paper describes the Singularity system in detail. Section 2 contains an overview of the system and its novel aspects. Section 3 describes the Singularity system architecture, focusing on the kernel, processes, and the language run-time system. Section 4 describes the programming language support for the system. Section 5 describes the I/O and security system. Section 6 provides some performance benchmarks. Section 7 surveys related work. Appendix A contains a list of the kernel ABI calls.

2 Singularity

Singularity is a new operating system being developed as a basis for more dependable system and application software [28]. Singularity exploits advances in programming languages and tools to create an environment in which software is more likely to be built correctly, program behavior is easier to verify, and run-time failures can be contained.

A key aspect of Singularity is an extension model based on *Software-Isolated Processes (SIPs)*, which encapsulate pieces of an application or a system and provide information hiding, failure isolation, and strong interfaces. SIPs are used throughout the operating system and application software. We believe that building a system on this abstraction will lead to more dependable software.

SIPs are the OS processes on Singularity. All code outside the kernel executes in a SIP. SIPs differ from conventional operating system processes in a number of ways:

- SIPs are closed object spaces, not address spaces. Two Singularity processes cannot simultaneously access an object. Communications between processes transfers exclusive ownership of data.
- SIPs are closed code spaces. A process cannot dynamically load or generate code.
- SIPs do not rely on memory management hardware for isolation. Multiple SIPs can reside in a physical or virtual address space.
- Communications between SIPs is through bidirectional, strongly typed, higher-order channels. A channel specifies its communications protocol as well as the values transferred, and both aspects are verified.
- SIPs are inexpensive to create and communication between SIPs incurs low overhead. Low cost makes it practical to use SIPs as a fine-grain isolation and extension mechanism.

- SIPs are created and terminated by the operating system, so that on termination, a SIP's resources can be efficiently reclaimed.
- SIPs executed independently, even to the extent of having different data layouts, run-time systems, and garbage collectors.

SIPs are not just used to encapsulate application extensions. Singularity uses a single mechanism for both protection and extensibility, instead of the conventional dual mechanisms of processes and dynamic code loading. As a consequence, Singularity needs only one error recovery model, one communication mechanism, one security policy, and one programming model, rather than the layers of partially redundant mechanisms and policies in current systems. A key experiment in Singularity is to construct an entire operating system using SIPs and demonstrate that the resulting system is more dependable than a conventional system.

The Singularity kernel consists almost entirely of safe code and the rest of the system, which executes in SIPs, consists of only verifiably safe code, including all device drivers, system processes, and applications. While all untrusted code must be verifiably safe, parts of the Singularity kernel and run-time system, called the *trusted base*, are not verifiably safe. Language safety protects this trusted base from untrusted code.

The integrity of the SIPs depends on language safety and on a system-wide invariant that a process does not hold a reference into another process's object space.

Ensuring code safety is obviously essential. In the short term, Singularity relies on compiler verification of source and intermediate code. In the future, typed assembly language (TAL) will allow Singularity to verify the safety of compiled code [36, 38]. TAL requires that a program executable supply a proof of its type safety (which can be produced automatically by a compiler for a safe language). Verifying that a proof is correct and applicable to the instructions in an executable is a straightforward task for a simple verifier of a few thousand lines of code. This end-to-end verification strategy eliminates a compiler—a large, complex program—from Singularity's trusted base. The verifier must be carefully designed, implemented, and checked, but these tasks are feasible because of its size and simplicity.

The *memory independence invariant* that prohibits cross-object space pointers serves several purposes. First, it enhances the data abstraction and failure isolation of a process by hiding implementation details and preventing dangling pointers into terminated processes. Second, it relaxes implementation constraints by allowing processes to have different run-time systems and their garbage collectors to run without coordination. Third, it clarifies resource accounting and reclamation by making unambiguous a process's ownership of a particular piece of memory. Finally, it simplifies the kernel interface by eliminating the need to manipulate multiple types of pointers and address spaces.

A major objection to this architecture is the difficulty of communicating through message passing, as compared with the flexibility of directly sharing data. Singularity is addressing this problem through an efficient messaging system, programming language extensions that concisely specify communication over channels, and verification tools [19].

2.1 Extensibility

Software creators rarely anticipate the full functionality demanded by users of their system or application. Rather than trying to satisfy everyone with a monolithic system, most non-trivial software provides mechanisms to load additional code. For example, Microsoft Windows supports over 100,000 third party device drivers, which enable it to control almost any hardware device. Similarly, countless browser add-ons and extensions augment a browser's interface and components for web pages. Even open source projects—although theoretically modifiable—

provide “plug-in” mechanisms, since extensions are easier to develop, distribute, and combine than new versions of software.

An extension usually consists of code that is dynamically loaded into its parent’s address space. With direct access to the parent’s internal interfaces and data structures, extensions can provide rich functionality. However, flexibility comes at a high cost. Extensions are a major cause of software reliability, security, and backward compatibility problems. Although extension code is often untrusted, unverified, faulty, or even malicious, it is loaded directly into a program’s address space with no hard interface, boundary, or distinction between host and extension. The outcome is often unpleasant. For example, Swift reports that faulty device drivers cause 85% of diagnosed Windows system crashes [49]. Moreover, because an extension lacks a hard interface, it can use unexposed aspects of its parent’s implementation, which can constrain evolution of a program and require extensive testing to avoid incompatibilities.

Dynamic code loading imposes a second, less obvious tax on performance and correctness. Software that can load code is an open environment in which it is impossible to make sound assumptions about the system’s states, invariants, or valid transitions. Consider 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 [47]. In general, the only feasible way to analyze a program running under such conditions is to start with the unsound assumption that the environment cannot change arbitrarily between any two operations.

One alternative is to prohibit code loading and isolate dynamically created code in its own environment. Previous attempts along these lines were not widely popular because the isolation mechanisms had performance and programmability problems that made them less appealing than the risks of running without isolation. The most common mechanism is a traditional OS process, but its high costs limit its usability. Memory management hardware provides hard boundaries and protects processor state, but it also makes inter-process control and data transfers expensive. On an x86 processor, switching between processes can cost hundreds to thousands of cycles, not including TLB and cache refill misses [25].

More recent systems, such as the Java virtual machine and Microsoft Common Language Runtime (CLR), are designed for extensibility and use language safety, not hardware, as the mechanism to isolate computations running in the same address space. Safe languages, by themselves, do not guarantee isolation. Shared data can provide a navigable path between computations’ object spaces, at which point reflection mechanisms can subvert data abstraction and information hiding. As a consequence, these systems incorporate complex security mechanisms and policies, such as Java’s fine grain access control or the CLR’s code access security, to limit access to system mechanisms and interfaces [40]. These mechanisms are difficult to use properly and impose considerable overhead.

Equally important, computations that share a run-time system and execute in the same process are not isolated upon failure. When a computation 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 [11].

Singularity uses SIPs to encapsulate. Every device driver, system process, application, and extension runs in its own SIP and communicates over channels that provide limited and appropriate functionality. If code in a SIP fails, it terminates, which allows the system to reclaim resources and notify communication partners. Since these partners did not share state with the extension, error recovery is entirely local and is facilitated by the explicit protocols on channels.

Another run-time source of new code is dynamic code generation, commonly encapsulated in a reflection interface. This feature allows a running program to examine existing code and data,

and to produce and install new methods. Reflection is commonly used to produce marshalling code for objects or parsers for XML schemas. Singularity's closed SIPs do not allow run-time code generation.

Instead, Singularity provides compile-time reflection (CTR), which provides similar functionality that executes when a file is compiled. Normal reflection, which has access to run-time values, is more general than CTR. However, in many cases, the class to be marshaled or the schemas to be parsed are known ahead of execution. In these cases, CTR produces code during compilation. In the other cases, Singularity will support a mechanism for generating code and running it in a separate SIP.

2.2 Application Abstraction

Operating systems currently do not treat programs or applications as a first-class abstraction. A modern application is a collection of files containing code, data, and metadata, which an untrusted agent installs by copying the pieces into a file system and registering them in namespaces. The system is largely unaware of relationships among the pieces and has little control over the installation process. A well-known consequence is that adding or removing an application can break unrelated software.

In Singularity, an application consists of a manifest and a collection of resources. The manifest describes the application in terms of its resources and their dependencies. Although many existing setup descriptions combine declarative and imperative aspects, Singularity manifests contain only declarative statements that describe the desired state of the application after installation or update.

The process of realizing this state is Singularity's responsibility. A manifest must provide enough information for the Singularity installer to deduce appropriate installation steps, detect conflicts with existing applications, and decide whether the installation succeeded. Singularity can prevent installations that impair the system.

Other aspects of Singularity also utilize information from a manifest. For example, Singularity's security model introduces applications as a security principal, which enables an application to be entered in a file's access control lists (ACL). Treating an application as a principal requires knowledge of the application's constituent pieces and dependencies and a strong identity, all of which come from the manifest.

2.3 Discussion

Among the key contributions of Singularity are:

- Construction of a system and application model called software-isolated processes, which uses verified safe code to implement a strong boundary between processes without hardware mechanisms. Since SIPs cost less to create and schedule, the system and applications can support more and finer isolation boundaries and a stronger isolation model.
- A consistent extension model for the system and applications that simplifies the security model, improves dependability and failure recovery, increases code optimization, and makes programming and testing tools more effective.
- A fast, verifiable communication mechanism between the processes on a system, which preserves process independence and isolation, yet enables process to communicate correctly and at low cost.
- Language and compiler support to build an entire system in safe code and to verify inter-process communications with explicit resource management.

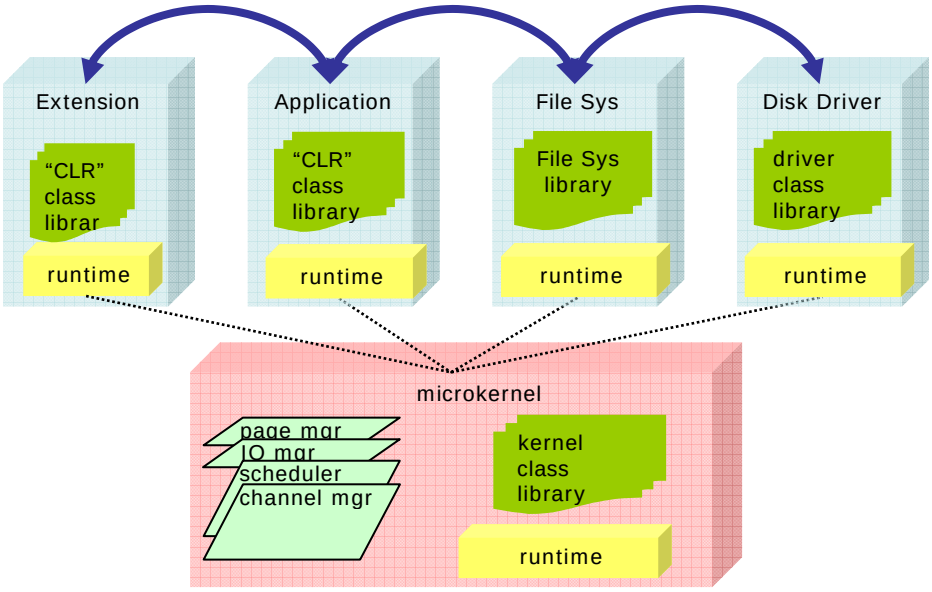

Figure 1 Singularity architecture.

- Elimination of the distinction between an operating system and a safe language run-time system, such as the Java JVM or Microsoft CLR.
- Pervasive use of specifications throughout a system to describe, configure, and verify components.

The rest of this paper is organized as follows. Section 3 describes the implementation of the Singularity system in detail. Section 4 discusses programming language and compiler support for Singularity. Section 5 describes the Singularity system services. Section 6 provides some performance measurements. Section 0 discusses related work. Section 8 concludes.

3 Singularity Architecture

Figure 1 depicts the architecture of the Singularity OS, which is built around three key abstractions: a kernel, software-isolated processes, and channels. The kernel provides the core functionality of the system, including memory management, process creation and termination, channel operations, scheduling, and I/O. Like other microkernels, most of the system's functionality and extensibility exist in processes outside of the kernel.

3.1 Trusted Base

Code in Singularity is either *verified* or *trusted*. Verified code's type and memory safety is checked by a compiler. Unverifiable code must be trusted by the system and is limited to the hardware abstraction layer (HAL), kernel, and parts of the run-time system. Most of the kernel is verifiably safe, but portions are written in assembler, C++, and unsafe C#.

All other code is written in a safe language, translated to safe Microsoft Intermediate Language (MSIL)[4], and then compiled to x86 by the Bartok compiler [20]. Currently, we trust that Bartok correctly verifies and generates safe code. This is obviously unsatisfactory in the long run and we plan to use typed assembly language to verify the output of the compiler and reduce this part of the trusted computing base to a small verifier [36]

[4]MSIL is the CPU-independent instruction set accepted by the Microsoft CLR. Singularity uses standard MSIL format. Features specific to Singularity are expressed through metadata extensions in the MSIL.

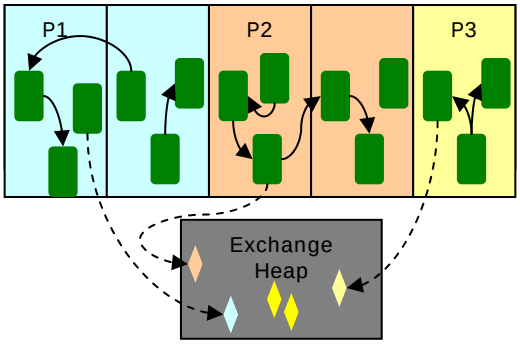

Figure 2. Exchange Heap.

The dividing line between the two types of code is blurred by the run-time system. This trusted, but unverifiable, code is effectively isolated from a computation, whose verified safety prevents it from interacting with the run-time system and its data structures, except through safe interfaces. Singularity's compiler is able to in-line some of these routines, thereby safely moving operations that would traditionally run in a kernel into a user process.

3.2 Kernel

The Singularity kernel is a privileged system component that controls access to hardware resources, allocates and reclaims memory, creates and schedules threads, provides intraprocess thread synchronization, and manages I/O. It is written in a mixture of safe and unsafe C# code and runs in its own garbage collected object space.

In addition to the usual mechanism of message-passing channels, processes communicate with the kernel through a strongly versioned application binary interface (ABI) that invokes static methods in kernel code. This interface follows the design of the rest of the system and isolates the kernel and process object spaces. All parameters to this ABI are values, not pointers, so the kernel and process's garbage collectors need not coordinate. The only exception is the location of the ABI methods. Our garbage collectors currently do not relocate code, but if they did, they would need to maintain the invariant that these methods remain at known addresses.

The ABI maintains the system-wide *state isolation invariant*: a process cannot alter the state of another process using the ABI. With only two exceptions, an ABI call affects only the state of its calling process. The two exceptions alter the state of a child process before or after it executes, but not during execution. The first is a call to create a child process, which specifies the code loaded for the child *before* it begins execution. The second is a call to stop a child process, which reclaims its resources *after* all threads cease execution. State isolation ensures that a Singularity process has sole control over its state.

3.2.1 Handle Table

The kernel exports synchronization constructs—mutexes, auto and manual reset events—to coordinate threads within a process. A thread manipulates these constructs through a strongly typed, opaque handle that points into the kernel's handle table. Strong typing prevents a process from changing or forging handles. In addition, slots in the handle table are reclaimed only when a process terminates, to prevent the process from freeing a mutex, retaining its handle, and using it to manipulate another process's object. Singularity does, however, reuse table entries within a process. In this case, retaining a handle can cause a more benign, but still painful, error within the process.

3.2.2 ABI Versioning

The kernel ABI is strongly versioned. By explicitly identifying ABI version information in every program, Singularity provides a clear path for system evolution and backward compatibility.

Code in a process is compiled against a compiled ABI interface assembly in a namespace that explicitly specifies the version. For example, `Microsoft.Singularity.V1.Threads` is a namespace that contains thread-related functionality for the first version of the ABI. Process source code names the specific namespace containing the desired version of the ABI. Process binary code contains explicit metadata references to the specific version of the ABI.

At install time, a program is installed only if this version of the ABI is supported on the target machine. If so, the ABI interface assembly is replaced by an implementation assembly, which provides a process-side implementation of the specified version of the ABI for the system's version of the kernel. The simplest implementation turns run-time calls on the kernel ABI into direct invocations of a static kernel method. However, a newer Singularity system can populate the namespace from an earlier version with a library of compatibility functions. Alternatively, compatibility code can run in the kernel, since the kernel can easily support multiple ABI implementations in their distinct namespaces.

Version 1 of the kernel ABI contains 126 entry points. (Appendix A lists the methods in the ABI.)

3.2.3 Scheduler

Singularity supports a compile-time replaceable scheduler. We have implemented four scheduler—the Rialto scheduler [32], a multi-resource laxity-based scheduler, a round-robin scheduler (implemented as a degenerate case of the Rialto scheduler), and a minimum latency round-robin scheduler.

The minimum latency round-robin scheduler is optimized for a large number of threads that communicate frequently. The scheduler maintains two lists of runable threads. The first, called the *unblocked* list, contains threads that have recently become runable. The second, called the *preempted* list, contains runable threads that have been pre-empted. When choosing the next thread to run, the scheduler removes threads from the unblocked list in FIFO order. When the unblocked list is empty, the scheduler removes the next thread from the preempted list. Whenever a scheduling timer interrupt occurs, all threads in the unblocked list are moved to the end of the preempted list, followed by the thread that was running when the timer fired. The first thread from the unblocked list is scheduled and the scheduling timer is reset.

The net effect of the two list scheduling policy is to favor threads that are awoken by a message, do a small amount of work, send one or more messages to other processes, and then block waiting for a message. This is a common behavior for threads running message handling loops. To avoid a costly reset of the scheduling timer, threads from the unblocked list inherit the scheduling quantum of the thread that unblocked them. Combined with the two-list policy, quantum inheritance is particular effective because Singularity can switch from one thread to another in as few as 394 cycles.

3.3 Processes

A Singularity system lives in a single virtual address space. Virtual memory hardware is used to protect pages, for example by mapping out the first 16K of address space to trap null pointer references. Within a Singularity system, the address space is logically partitioned into: a kernel object space, an object space for each process, and the Exchange Heap for channel data. A pervasive design decision is the *memory independence invariant*: cross-object space pointers only

point into the Exchange Heap. In particular, the kernel does not have pointers into a process's object space, nor does one process have a pointer to another process' objects. This invariant ensures that each process can be garbage collected and terminated without the cooperation of other processes.

The kernel creates a process by allocating memory sufficient to load an executable image from a file stored in Microsoft's portable executable (PE) format. Singularity then performs relocations and fixups, including linking kernel ABI functions. The kernel starts the new process by creating a thread running at the image's entry point, which is trusted thread startup code that calls the stack and page manager to initialize the process.

A process obtains additional address space by calling the kernel's page manager, which returns new, unshared pages. These pages need not be adjacent to the process's existing address space, since the garbage collectors do not require the address space be contiguous, though they may need contiguous regions for large objects or arrays. In addition to memory, which holds the process's code and heap data, a process has a stack per thread and can access the Exchange Heap.

3.3.1 Stack Management

Singularity uses linked stacks to reduce the memory overhead of a thread. These stacks grow on demand by adding non-contiguous segments of 4K or more. Singularity's compiler performs static interprocedural analysis to optimize placement of overflow tests [51]. Each of these compiler-inserted checks is trusted code that accesses system data structures, residing in the process's object space, to determine the amount of space remaining in the current stack segment. Before the running thread pushes a new stack frame which would potentially allow overflow of the current stack segment, the trusted code calls a kernel method, which disable interrupts and invokes the page manager to allocate a new stack segment. This code also initializes the first stack frame in the segment—between the running procedure and its callee—to call the segment unlink routine, which will deallocate the segment when the stack is popped. Since all processes run in ring 0 on an x86, the current stack segment must always leave enough room for the processor to save an interrupt or exception frame, before the handler switches to a dedicated interrupt stack.

3.3.2 Exchange Heap

The Exchange Heap, which underlies efficient communication in Singularity, holds data passed between processes (Figure 2). The Exchange Heap is not garbage collected, but instead uses reference counts to track usage of blocks of memory called *regions*. A process accesses a region through a structure called an *allocation*.

Allocations also reside in the Exchange Heap, which enables them to be passed between processes, but each is owned and accessible by a single process at a time. More than one allocation may share read-only access to an underlying region. Moreover, the allocations can have different base and bounds, which provide distinct views into the underlying data. For example, protocol processing code in a network stack can strip the encapsulated protocol headers off a packet without copying it. A region tracks the number of allocations that point to it, and it is deallocated when this reference count falls to zero. The Singularity compiler hides the extra level of indirection through an allocation record by strongly typing references into a region and automatically generating code to dereference through the record.

3.3.3 Threads

A process can create additional threads. Untrusted (but verified) code running in the process creates a thread object, initializes it with a supplied function, and stores the object in an unused slot in the run-time system's thread table. This code then invokes `ThreadHandle.Create`,

passing the thread table index. This kernel method creates a thread context to hold registers, allocates the initial stack frame, and updates its data structures. It returns to the process, where the runtime calls `ThreadHandle.Start`, to schedule the thread. When the thread starts, it is executing in the kernel and running code that calls the process's entry point, passing the thread's index in the run-time's thread table. The process startup code invokes the function in the thread object, which starts the thread's execution.

Throughout process and thread creation, the kernel is aware of only one address in a process: the thread startup code at the process's entry point, which, like the kernel ABI methods, cannot be relocated.

3.4 Garbage Collection

Garbage collection is an essential component of most safe languages, as it prevents memory deallocation errors that can subvert safety guarantees. In Singularity, the kernel and processes object spaces are garbage collected.

The large number of garbage collection algorithms and experience strongly suggest that no one garbage collector is appropriate for all system or application code [21]. Singularity's architecture decouples the algorithm, data structures, and execution of each process's garbage collector, so it can be selected to accommodate the behavior of code in the process and to run without global coordination. The four aspects of Singularity that make this possible are: each process is a closed environment with its own run-time support; pointers do not cross process or kernel boundaries, so collectors need not consider cross-space pointers; messages on channels are not objects, so agreement on memory layout is only necessary for messages and other data in the Exchange Heap; and the kernel controls memory page allocation, which provides a nexus for coordinating resource allocation.

Singularity's run-time systems currently support five types of collectors—generational semi-space, generational sliding compacting, an adaptive combination of the previous two collectors, mark-sweep, and concurrent mark-sweep. We currently use the latter for system code, as it has very short pause times during collection. With this collector, each thread has a segregated free list, which eliminates thread synchronization in the normal case. A garbage collection is triggered at an allocation threshold and executes in an independent collection thread that marks reachable objects. During a collection, the collector stops each thread to scan its stack, which introduces a pause time of less than 100 microseconds for typical stacks. The overhead of this collector is higher than non-concurrent collectors, so we use a simpler non-concurrent mark-sweep collector in applications.

Each SIP has its own collector that is solely responsible for collection of objects in the object space. From the garbage collector's perspective, when a thread of control enters or leaves an application (or the kernel) it is treated similarly to a call to or a call-back from native code in conventional garbage collected environments. Garbage collection for different object spaces can therefore be scheduled and run completely independently. If an application employs a stop-the-world collector, a thread is considered stopped with respect to the application object space, even if it is run in the kernel object space due to a kernel call. The thread, however, is stopped upon return to the application process space for the duration of the collection.

3.4.1 Stack Management

In a garbage collected environment, a thread's stack contains object references that are potential roots for a collector. Calls into the kernel are executed on a user thread's stack and may store kernel pointers in this stack. At first sight, this appears to violate the memory independence invariant by creating cross-process pointers, and, at least, entangles the user and kernel garbage collections.

To avoid these problems, Singularity delimits the boundary between each space's stack frames, so a garbage collector need not see references to the other space. At a cross-domain (process → kernel or kernel → process) call, Singularity saves callee-saved registers in a special structure on the stack, which also demarks a cross- domain call. These structures mark the boundary of stack regions that belong to each object space. Since calls in the kernel ABI do not pass object pointers, a garbage collector can skip over frames from the other space.

These delimiters also facilitate terminating processes cleanly. When a process is killed, its threads are stopped and the kernel throws an exception on each, which skips over and deallocates the process's stack frames.

3.5 Channels

Singularity processes communicate exclusively by sending messages over channels, which are a bidirectional, behaviorally typed connection between two processes. Messages are tagged collections of values or message blocks in the Exchange Heap that are transferred from a sending to a receiving process. A channel is typed by a contract, which specifies the format of messages and valid messages sequences along the channel (see Section 4.1).

A process creates a channel by invoking a contract's static `NewChannel` method, which returns the channel's two endpoints—asymmetrically typed as an exporter and importer—in its output parameters:

```
C1.Exp importCh;
C1.Imp exportCh;
C1.NewChannel(out importCh, out exportCh);
```

The process can pass either or both endpoints to other processes over existing channels. The process receiving an endpoint has a channel to the process holding the other, corresponding endpoint. For example, if an application process wants to communicate with a system service, the application creates two endpoints and sends a request containing one endpoint to the system's name server, which forwards the endpoint to the service, thereby establishing a channel between the process and the service.

A send on a channel is asynchronous. A receive synchronously blocks until a specific message arrives. Using language features, a thread can wait for the first of a set of messages along a channel or can wait for specific sets of messages from different channels. When data is sent over a channel, ownership passes from the sending process, which may not retain a reference to the message, to the receiving process. This ownership invariant is enforced by the language and run-time systems, and serves three purposes. The first is to prevent sharing between processes. The second is to facilitate static program analysis by eliminating pointer aliasing of messages. The third is to permit implementation flexibility by providing message-passing semantics that can be implemented by copying or pointer passing.

3.5.1 Channel Implementation

Channel endpoints and the values transferred across channels reside in the Exchange Heap. The endpoints cannot reside in a process's object space, since they are passed across channels. Similarly, data passed on a channel cannot reside in an object space since it would violate the memory independence invariant. A message's sender passes ownership by storing a pointer to the message in the receiver's endpoint, at a location determined by the current state of the message exchange protocol. This approach naturally allows a "zero copy" implementations of an I/O stack. For example, disk buffers and network packets can be transferred across multiple channels, through a protocol stack and into an application process, without copying.

3.6 Customized Run-time Systems

Singularity's architecture permits SIPs to host completely different run-time systems, which allows a runtime to be customized for each process. For example, SIPs running sequential code may not need the support for thread synchronization required by SIPs with multiple threads. SIPs without objects requiring finalization (or having finalizers that do not access data shared among threads) may not need a separate finalizer to observe the required language semantics for finalizers. SIPs with certain allocation strategies may be able to pre-allocate or stack-allocate memory for all used objects, obviating the need for a garbage collector in the SIPs runtime.

3.7 Discussion

Safe programming languages offer many advantages in building reliable, analyzable software that is immune to the low-level security exploits that plague C and C++ code. Because of these practical benefits, safe languages are increasing in popularity. Conventional operating systems offer no special support for safe programs, nor do they benefit from their properties. Singularity, by contrast, starts from a premise of language safety and builds a system architecture that supports and enhances the language guarantees.

Singularity integrates the language run-time system and operating system processes. In a safe system, all processes need this support, so a distinct virtual machine, such as the JVM or CLR, is redundant. However, the simple approach of providing a homogeneous run-time system, for example the CLR, across all processes imposes unnecessary penalties on services or programs whose behavior does not match the runtime system's properties. Language runtimes provide services—notably garbage collection—that can interact poorly with programs. For example, a generational garbage collector may introduce seconds-long pauses in program execution, which would disrupt a media player or operating system. On the other hand, a real-time collector suitable for the media player might penalize a computational task. Homogeneous environments also evolve into large, complex, and expensive systems since they must support the union of the requirements of every application that depends on them.

Singularity supports heterogeneous execution environments. Each process has its own run-time system, with its own memory layout, garbage collection algorithm, and libraries. Because of memory independence, a runtime can be tailored to meet the needs of a computation. In particular, a process's garbage collector can be selected for its algorithm and data structure layout, without awareness of or coordination with its counterparts in other processes.

Heterogeneous environments also provide a new mechanism to enforce policy. The contents of the environment in a process circumscribe its behavior. For example, device drivers run in a sparse environment that contains primarily driver-specific abstractions, such as `IoPorts`, tailored for this class of program. Abstractions unnecessary, or inappropriate, for drivers can be kept out of this environment. Another policy might be that untrusted applications can only run in an environment in which security automata validate and control program behavior [45].

Second, Singularity is built on and offers a new model for safely extending a system or application's functionality. In this model, extensions cannot access their parent's code or data structures, but instead are self-contained programs that run independently. This approach increases the complexity of writing an extension, as the parent program's developer must define a proper interface that does not rely on shared data structures and an extension's developer must program to this interface and possibly re-implement functionality available in the parent. Nevertheless, the widespread problems inherent in dynamic code loading argue for alternatives that increase the isolation between an extension and its parent. Singularity's mechanism works for applications as well as system code; does not depend on the semantics of an API, unlike domain-

specific approaches such as Nooks [49]; and provides simple semantic guarantees that can be understood by programmers and used by tools.

The principal arguments against Singularity's extension model center on the difficulty of writing message-passing code. We hope that better programming models and languages will make programs of this type easier to write, verify, and modify. Advances in this area would be generally beneficial, since message-passing communication is fundamental and unavoidable in distributed computing and web services. As message passing becomes increasingly familiar and techniques improve, objections to programming this way within a system are likely to become less common.

Finally, Singularity does not use memory management hardware on processors for protection, which suggests the possibility of reevaluating this hardware. In general, many programs only use some of the functionality of memory management hardware. Embedded systems (or adequately provisioned workstations and servers) rarely page because memory is inexpensive and abundant. Large (64-bit) address spaces reduce the need to use multiple address spaces to get around 32-bit limitations. And, Singularity shows how safe languages and conservative sharing policies can supplant process boundaries and protection rings, at lower cost. Current hardware, if not fully utilized, might be replaced by simpler mechanisms with fewer performance bottlenecks such as TLBs.

Singularity would benefit from memory protection for its trusted (unverified) base. For example, DMA currently is inherently unsafe and, because of different interfaces on each device, cannot be encapsulated or virtualized by a system. Memory protection for DMA transfers could protect the system against a misdirected DMA. Hardware support for segmented stacks could reduce the compiler complexity and run-time overhead of this mechanism.

4 Programming Language Support

Singularity is written in Sing#, which is an extension to the Spec# language developed in Microsoft Research. Spec# itself is an extension to Microsoft's C# language that provides constructs (pre- and post-conditions and object invariants) for specifying program behavior [7]. Specifications can be statically verified by the Boogie verifier or checked by compiler-inserted run-time tests. Sing# extends this language with support for channels and low-level constructs necessary for system code.

We developed and implemented programming language extensions for two reasons. First, few languages support message-passing communication. In most cases, message passing is relegated to libraries, which are a syntactically and semantically awkward way of grafting asynchronous operations onto a synchronous language such as C#. Sing# provides first-class support for message-passing communications, which makes this style of communication, and the SIP abstractions, more efficient to implement and more palatable to programmers. Second, integrating a feature into a language allows more aspects of a program to be verified. Singularity's constructs allow communication to be statically verified.

4.1 Channel Contracts

Channel contracts are central to Singularity's isolation architecture and are directly supported in Sing#. Here's a contract describing a simple interaction on a channel.

```
contract C1 {
  in message Request(int x) requires x>0;
  out message Reply(int y);
  out message Error();

  state Start: Request?
               -> (Reply! or Error!)
               -> Start;
}
```

Contract C1 declares three messages: `Request`, `Reply`, and `Error`. Each message specifies the types of arguments contained in the message. For example, `Request` and `Reply` both contain a single integer value, whereas `Error` does not carry any values. Additionally, each message may specify Spec# `requires` clauses restricting the arguments further.

Messages can also be tagged with a direction. The contract is always written from the exporter point of view. Thus, in the example, `Request` is a message that can be sent by the importer to the exporter, whereas `Reply` and `Error` are sent from the exporter to the importer. Without a qualifier, messages can travel in both directions.

After the message declarations, a contract specifies the allowable message interactions via a state machine driven by send and receive actions. The first state declared is considered the initial state of the interaction. The example contract `C1` only declares a single state called `Start`. After the state name, action `Request?` indicates that in the `Start` state, the export side of the channel is willing to receive (`?`) a `Request` message. Following that the construct (Reply! or Error!) specifies that the exporter sends (`!`) either a `Reply` or an `Error` message. The last part (`-> Start`) specifies that the interaction then continues to the `Start` state, thereby looping ad-infinitum.

A slightly more involved example is a portion of the contract for the network stack:

```
public contract TcpConnectionContract {
  in message Connect(uint dstIP, ushort dstPort);

  out message Ready();

  // Initial state
  state Start : Ready! -> ReadyState;

  state ReadyState : one {
      Connect? -> ConnectResult;
      BindLocalEndPoint? -> BindResult;
      Close? -> Closed;
  }

  state BindResult : one {
      OK! -> Bound;
      InvalidEndPoint! -> ReadyState;
  }

  in message Listen();

  state Bound : one {
      Listen? -> ListenResult;
      Connect? -> ConnectResult;
      Close? -> Closed;
  } ...
```

The protocol specification in a contract serves several purposes. It can help detect programming errors, either at run time or through a static analysis tool. Run-time monitoring

drives a contract's state machine in response to the messages exchanged over a channel and watches for erroneous transitions. This technique is simple to implement, but only detects errors in one program execution. Moreover, it cannot detect liveness errors such as deadlock. Static program analysis can provide a stronger guarantee that processes are correct and stuck-free in all program executions.

Singularity currently uses a combination of run-time monitoring and static verification. All messages on a channel are checked against the channel's contract, which detects correctness, but not liveness problems. We also have a static checker that verifies safety properties. To statically ensure deadlock freedom, we plan to verify the contracts with a more general static analysis based on conformance checking [42].

In addition, the Singularity compiler uses a contract to determine the maximum number of messages that can be outstanding on a channel, which enables a compiler to statically allocate buffers in channel endpoints. Statically allocated buffers improve communication performance.

4.2 Endpoints

Channels in Singularity manifest as a pair of endpoints representing the importing and exporting sides of the channel. Each endpoint has a type that specifies which contract the channel adheres to. Endpoint types are implicitly declared within each contract. A contract C1 is represented as a class, and the endpoint types are nested types within that class as follows:

- `C1.Imp` — Type of import endpoints of channels with contract C1.
- `C1.Exp` — Type of export endpoints of channels with contract C1.

4.3 Send/receive Methods

Each contract class contains methods for sending and receiving the messages declared in the contract. The example provides the following methods:

```
C1.Imp {
  void SendRequest(int x);
  void RecvReply(out int y) ;
  void RecvError();
}

C1.Exp {
  void RecvRequest(out int x)
  void SendReply(int y);
  void SendError();
}
```

The semantics of the Send methods are that they send the message asynchronously. The receive methods block until the given message arrives. If a different message arrives first, an error occurs. Such errors should never occur if the program passes the contract verification check. Unless a receiver knows exactly which message it requires next, these methods are not appropriate. Instead, Sing# provides a `switch receive` statement.

4.4 Switch-Receive Statement

Consider the following code, which waits for either the `Reply` or `Error` message on an imported endpoint of type C1.Imp.

```
void M( C1.Imp a) {
  switch receive {
    case a.Reply(x):
      Console.WriteLine(“Reply {0}”, x);
      break;

    case a.Error():
      Console.WriteLine(“Error”);
      break;
  }
}
```

The `switch receive` statement operates in two steps:

1. Block for a particular set of messages to arrive on a set of endpoints

2. Receive one of the set of messages and bind its arguments into local variables.

In the example above, the `switch receive` has two patterns, either receive `Reply` on endpoint `a`, or `Error` on the same endpoint. In the first case, the integer argument of the `Reply` message is automatically bound to the local variable `x`.

The `switch receive` construct is however more general, as patterns can involve multiple endpoints. The following example has two endpoints `a` and `b` that can receive `Reply` or `Error` messages:

```
void M (C1.Imp a, C1.Imp b) {
  switch receive {
    case a.Reply(x) && b.Reply(y):
      Console.WriteLine(“Both replies {0} and{1}”, x, y);
      break;

    case a.Error():
      Console.WriteLine(“Error reply on a”);
      break;

    case b.Error():
      Console.WriteLine(“Error reply on b”);
      break;

    case a.ChannelClosed():
      Console.WriteLine(“Channel a is closed”);
      break;
  }
}
```

The example illustrates shows how to wait for particular combinations of messages using the switch receive statement. The first branch is only taken if the `Reply` message is received on both endpoints `a` and `b`. The final case contains the pattern `ChannelClosed()`, which is a special pattern that fires when the channel is closed (by the other party) and no more messages remain to be received.

4.5 Ownership

In order to guarantee memory isolation of endpoints and other data transferred on channels, all blocks in the Exchange Heap are resources that need to be tracked at compile time. In particular, the static checks enforce that access to these resources occur only at program points where the resource is owned and that methods do not leak ownership of the resources. Tracked resources have a strict ownership model. Each resource is owned by at most one thread (or by a data structure within a thread) at any point in time. For example, if an endpoint is sent in a

message from thread T1 to thread T2, then ownership of the endpoint changes: from T1 to the message and then to T2, upon message's receipt.

To simplify the static tracking of resources, pointers to resources can be only held directly in local variables, messages, and data structures that are themselves tracked. These restrictions can sometimes be onerous, so Sing# provides a means to overcome them by storing tracked resources indirectly within data structures through an abstraction called a `TRef`.

4.6 TRefs

A `TRef` is a storage cell of type `TRef<T>` holding a tracked data structure of type `T`. TRef has the following signature:

```
class TRef<T> where T:ITracked {
  public TRef([Claims] T i_obj);
  public T Acquire();
  public void Release([Claims] T newObj);
}
```

When creating a `TRef<T>`, the constructor requires an object of type `T` as an argument. The caller must have ownership of the object at the construction site. After the construction, ownership has been passed to the newly allocated `TRef`. The `Acquire` method is used to obtain the contents of a `TRef`. If the `TRef` is full, it returns its contents and transfers ownership to the caller of `Acquire`. Afterwards, the `TRef` is said to be empty. `Release` transfers ownership of a `T` object from the caller to the `TRef`. Afterwards, the `TRef` is full. `TRefs` are thread-safe and `Acquire` operations block until the `TRef` is full.

`TRef`s represent a trade-off between static and dynamic checking. By using a `TRef`, incorrect multiple acquires are turned into deadlocks and the finalization mechanism of the garbage collector is responsible for reclaiming a resource.

4.7 The Exchange Heap

Since ownership of blocks of memory is transferred from one thread or process to another on message exchanges, Singularity needs a way to allocate and track blocks that can be exchanged in this fashion. The channel system requires that message arguments be either scalars or blocks in the Exchange Heap. There are two kinds of blocks in the Exchange Heap: individual blocks or vectors[5]. Their types are written, respectively, as follows:

```
using Microsoft.Singularity.Channels;
R* in ExHeap pr;
R[] in ExHeap pv;
```

The type of pointer `pr` specifies that it points to an `R` struct in the Exchange Heap. `ExHeap` is a type defined by the run-time system that provides allocation, deallocation, and other support for this heap. The type of `pv` is a vector of `R`'s in the Exchange Heap.

An invariant of the Exchange Heap is that it does not contain any pointers into any process GC heap. Thus the type of R must be an *exchangeable type*, i.e., a primitive value type (int, char, etc.), an enum, or a rep struct, where rep structs are simply structs in which all fields have exchangeable types.

[5] Endpoints are themselves represented as individual blocks in the Exchange heap.

4.8 Verification

Verifying that code executed in Singularity is type safe and satisfies the memory independence invariants is a three-stage process. The Sing# compiler checks type safety, ownership rules, and protocol conformance during compilation. The Singularity verifier checks these same properties on the generated MSIL code. Finally, the back-end compiler should—but does not as yet—produce a form of typed assembly language that enables these properties to be checked yet again by the operating system. One could argue that only the final stage is strictly necessary for safety. This is of course literally true, but in practice, programmers benefit from finding mistakes as early as possible and from having the errors explained completely, at a high level. Furthermore, the redundant verification guards against errors in the verification itself.

4.9 Compile-Time Reflection

The closed world of a SIP is incompatible with reflection facilities, an integral part of the Java and CLR environments, which can generate and invoke code at run time. As a consequence, Singularity does not support run-time reflection services.

Compile-time reflection (CTR) is a partial substitute for the CLR's full reflection capability. CTR is similar to techniques such as macros, binary code rewriting, aspects, meta-programming, and multi-stage languages. The basic idea is that programs may contain place-holder elements (classes, methods, fields, etc.) that are subsequently expanded by a generator.

The ability to produce boiler plate or other repetitious code from a template driven by inspection of existing program structures is a very powerful feature. For example, in Singularity, applications and device drivers declaratively describe their resource requirements, such as I/O ranges and service channels. The startup code for theses processes should be generated automatically from these descriptions.

Generators are written in Sing# as transforms. A transform contains a pattern matching program structure and a code template to build new code elements. Combining these two enables a transform to be analyzed and checked independent of the code to which it will be applied. For example, errors, such as generating a call on a non-existent method or calling with the wrong type of object, can be detected in a transform. In this respect, CTR is similar to multi-stage languages. Note that a CTR transform may be part of the trusted computing base, and so it can emit trusted code into an otherwise untrusted process.

```
transform DriverTransform
      where $IoRangeType: IoRange {

  class $DriverCategory: DriverCategoryDeclaration {
    [$IoRangeAttribute(*)]
    $IoRangeType $$ioranges;

    public readonly static $DriverCategory Values;

    generate static $DriverCategory() {
      Values = new $DriverCategory();
    }

    implement private $DriverCategory() {
      IoConfig config = IoConfig.GetConfig();
      Tracing.Log(Tracing.Debug, "Config: {0}", config.ToPrint());

      forall ($cindex = 0; $f in $$ioranges; $cindex++) {
        $f = ($f.$IoRangeType) config.DynamicRanges[$cindex];
      }
    }
  }
}
```

The transform above, named `DriverTransform`, generates the startup code for a device driver from a declarative declaration of the driver's resources needs. For example, the following declaration in the SB16 driver describes its `IoPorts` requirements:

```
internal class Sb16Resources: DriverCategoryDeclaration {

  [IoPortRange(0, Default = 0x0220,Length = 0x10)]
  internal readonly IoPortRange basePorts;

  [IoPortRange(1, Default = 0x0380,Length = 0x10)]
  internal readonly IoPortRange gamePorts;

  internal readonly static Sb16Resources Values;

  reflective private Sb16Resources();
}
```

`DriverTransform` matches this class, since it derives from `DriverCategoryDeclaration` and contains the specified elements, such as a `Values` field of the appropriate type and a placeholder for a private constructor. The keyword `reflective` denotes a placeholder whose definition will be generated by a transform using the `implement` modifier. Placeholders are forward references that enable code in a program to refer to code subsequently produced by a transform.

Pattern variables in the transform start with $ signs. In the example, `$DriverCategory` is bound to `SB16Resources`. A variable that matches more than one element starts with two $ signs. For example, `$$ioranges` represents a list of fields, each having a type `$IoRangeType` derived from `IoRange` (the types of the various fields need not be the same). In order to generate code for each element in collections (such as the collection of fields `$$ioranges`), templates may contain the `forall` keyword, which replicates the template for each binding in the collection. The resulting code produced by the transform above is equivalent to:

```
class SB16Resources {
   …
  static Sb16Resources() {
    Values = new Sb16Resources();
  }
  private SB16Resources() {
    IoConfig config = IoConfig.GetConfig();
    Tracing.Log(Tracing.Debug,
      "Config: {0}", config.ToPrint());

    basePorts = (IoPortRange)config.DynamicRanges[0];
    gamePorts = (IoPortRange)config.DynamicRanges[1];
  }
}
```

The example also illustrates that code generated by a transform can be type checked when the transform is compiled, rather than deferring this error checking until the transform is applied, as is the case with macros. In the example, the assignment to `Values` is verifiably safe, as the type of the constructed object (`$DriverCategory`) matches the type of the `Values` field.

5 Singularity System

Built on the kernel, SIPs, channels, and language model described above, Singularity supports a number of conventional operating system services.

5.1 I/O System

Singularity's I/O system consists of three layers: HAL, I/O manager, and drivers. The HAL is a small, trusted abstraction of PC hardware: `IoPorts`, `IoDma`, `IoIrq`, and `IoMemory` abstractions to access devices; interfaces to the timer, interrupt controller, real-time clock, and debug console; kernel debugging stub; event logger; interrupt and exception vector; BIOS resource discovery; and stack linking code. It is written in C#, C++, and assembler. The assembler and C++ portions of the HAL represent approximately 5% of the trusted code in the system (35 out of 561 files).

The Singularity kernel uses a manifest to create and bind device drivers. On startup, the kernel does a plug and play configuration of the system. The kernel use information acquired from the BIOS by the boot loader and from buses, such as the PCI bus, to enumerate devices, start the appropriate device drivers, and pass these drivers objects that encapsulate access to device hardware.

Each driver is written in safe code and runs in its own process. Drivers communicate with other parts of the system, including the network stack and file system, exclusively through channels. When a driver starts, the kernel provides it with four types of initialized objects that enable the driver to communicate with its device. All these objects provide a safe interface that checks each reference before directly accessing the hardware's memory mapped locations.

An `IoPort` provides an interface to a device's I/O port registers. It verifies that register references are in bounds and a driver does not write to read-only memory. An `IoDma` provides access to the built-in DMA controller for legacy hardware. `IoIrqs` notify a driver when a hardware interrupt arrives. `IoMemory` provides a bounds checked access to a fixed region of memory containing memory-mapped registers or pinned for use in DMA.

The only unsafe aspect of the driver-device interface is DMA. Existing DMA architectures provide no memory protection, so a misbehaving or malicious driver can program a DMA-capable device to overwrite any part of memory. Because of the diversity of DMA interfaces, we

have not found a good abstraction to encapsulating them. We anticipate that future hardware will provide memory protection for DMA transfers.

An interrupt from a device is serviced by the kernel, which masks the interrupt, and then signals the appropriate driver's `IoIrq`. Each driver process has a thread waiting on its Irq event, which starts processing the interrupt and re-enables the interrupt line through a kernel ABI. The scheduler runs immediately after the interrupt handler and signals all events in this queue.

5.2 Driver Configuration

The Singularity system makes extensive use of metadata to describe pieces of the system, explain how they fit together, and specify their behavior. The metadata in Singularity declaratively labels each Singularity component, system, or application with its dependencies, exports, and resources. Tools in Singularity use this metadata to verify and configure application and system code, both before and during system execution.

A Singularity system image is a compound artifact. It consists of a kernel, device drivers, applications, and sufficient metadata to describe these individual artifacts. It also contains a manifest that declares the policy for the system. The manifest also points to manifests describing individual component. Through these manifests, software, such as a boot loader or system verifier, can discover every component of a Singularity system.

A Singularity system image and its manifest are sufficient to enable off-line analysis of the system. Our goal is to enable an administrator to use only a description of the hardware devices and the system manifest to answer questions, such as: will the system boot on the particular hardware, which drivers and services will initialize, and which applications can run?

A Singularity system image contains metadata describing the device drivers. Through the metadata, Singularity maintains three invariants. First, Singularity will never install a device driver that cannot start successfully due to resources conflicts with another driver or portion of the system. Second, Singularity will never start a device driver that cannot run successfully due to either a conflicting or missing resource. Third, a device driver cannot access resources at runtime that were not declared in its metadata.

5.2.1 Specification

Where possible, Singularity uses C# custom attributes to interleave metadata into source code, so that only one source document must be maintained. Custom attributes may be attached to a program entity such as a class, method, or field declaration. A compiler passes attributes through to the resulting MSIL binary. Compilers, linkers, installation tools, and verification tools can read the metadata encoded in an attribute in a MSIL binary without executing code from the file.

As an example, the following code shows some attributes used to declare the dependencies and resource requirements of an S3Trio64 video device driver:

```
[DriverCategory]
[Signature("/pci/03/00/5333/8811")]
class S3TrioConfig : DriverCategoryDeclaration
{
  // Hardware resources from PCI config
  [IoMemoryRange(0, Default = 0xf8000000, Length = 0x400000)]
  IoMemoryRange frameBuffer;

  // Fixed hardware resources
  [IoFixedMemoryRange(Base = 0xb8000, Length = 0x8000)]
  IoMemoryRange textBuffer;

  [IoFixedMemoryRange(Base = 0xa0000, Length = 0x8000)]
  IoMemoryRange fontBuffer;

  [IoFixedPortRange(Base = 0x03c0, Length = 0x20)]
  IoPortRange control;

  [IoFixedPortRange(Base = 0x4ae8, Length = 0x02)]
  IoPortRange advanced;

  [IoFixedPortRange(Base = 0x9ae8, Length = 0x02)]
  IoPortRange gpstat;

  // Channels
  [ExtensionEndpoint(typeof(ExtensionContract.Exp))]
  TRef<ExtensionContract.Exp:Start> iosys;

  [ServiceEndpoint(typeof(VideoDeviceContract.Exp))]
  TRef<ServiceProviderContract.Exp:Start> video;
  ...
}
```

The `[DriverCategory]` and `[Signature]` attributes declare this module to be a device driver for a specific class of PCI video devices. `DriverCategory` denotes a category of applications that implement device drivers for specific hardware. Other categories include `ServiceCategory`, for applications implementing software services, and `WebAppCategory` for extensions to Singularity's Cassini web server.

The `[IoMemoryRange]` attribute declares that `frameBuffer` is derived the first entry in the device's PCI configuration space. This entry is discovered when the hardware is configured, and the hardware parameters, such as the size of the memory region, must be compatibile with the values in the attribute. The `[IoFixedMemoryRange]` and `[IoFixedPortRange]` attributes specify that a driver needs either a fixed range of address space for memory mapped access or a fixed ranges of I/O ports to access device registers.

The `[ExtensionEndpoint]` attribute specifies the channel contract and local endpoint used to communicate with the driver's parent process. In the case of device drivers, such as the S3Trio64, the I/O system is the parent process.

The `[ServiceEndpoint]` attributes declares a channel contract and local endpoint used to accept incoming bind requests from clients. Section 5.2.5 describes how the I/O system maps the other endpoint of the `ServiceProviderContract` into the system namespace

5.2.2 Compile Time

At compile time, the C# compiler transfers custom attributes into the MSIL binary. Using an MSIL metadata access library, Singularity tools can parse the instruction and metadata streams in the MSIL binaries.

At link time, the `mkmani` tool reads the custom attributes to create an application manifest. An application manifest is an XML file enumerating the application's components, exports, and dependencies.

The following XML contains part of the manifest information for the S3Trio64 device driver:

```
<manifest>
  <application identity="S3Trio64" />
  <assemblies>
    <assembly filename="S3Trio64.exe" />
    <assembly filename="Namespace.Contracts.dll" version="1.0.0.2299"
      />
    <assembly filename="Io.Contracts.dll" version="1.0.0.2299" />
    <assembly filename="Corlib.dll" version="1.0.0.2299" />
    <assembly filename="Corlibsg.dll"  version="1.0.0.2299" />
    <assembly filename="System.Compiler.Runtime.dll"
      version="1.0.0.2299" />
    <assembly filename="Microsoft.SingSharp.Runtime.dll"
      version="1.0.0.2299" />
    <assembly filename="ILHelpers.dll" version="1.0.0.2299" />
    <assembly filename="Singularity.V1.ill" version="1.0.0.2299" />
  </assemblies>
  <driverCategory>
    <device signature="/pci/03/00/5333/8811" />
    <ioMemoryRange index="0" baseAddress="0xf8000000"
      rangeLength="0x400000" />
    <ioMemoryRange baseAddress="0xb8000" rangeLength="0x8000"
      fixed="True" />
    <ioMemoryRange baseAddress="0xa0000" rangeLength="0x8000"
      fixed="True" />
    <ioPortRange baseAddress="0x3c0" rangeLength="0x20" fixed="True" />
    <ioPortRange baseAddress="0x4ae8" rangeLength="0x2" fixed="True" />
    <ioPortRange baseAddress="0x9ae8" rangeLength="0x2" fixed="True" />
    <extension startStateId="3" contractName="Microsoft.Singularity-
      .Extending.ExtensionContract" endpointEnd="Exp"
      assembly="Namespace.Contracts" />
    <serviceProvider startStateId="3" contractName="Microsoft-
      .Singularity.Io.VideoDeviceContract" endpointEnd="Exp"
      assembly="Io.Contracts" />
  </driverCategory>
  ...
</manifest>
```

5.2.3 Installation Time

As described in Section 2.2, an application is a first class abstraction in Singularity. To be run, a piece of code must be added to the system by the Singularity installer.

The installer starts with the metadata in the application's manifest. The installer verifies each of the application's assemblies exists and is type and memory safe. It also verifies all channel contracts are implemented correctly and all assembly dependencies and dependencies on the kernel ABI can be resolved correctly.

Once theses internal properties are resolved and verified, the installer next attempts to resolve and verify all external dependencies. For example, the install ensures that any hardware resources used by a device driver do not conflict with hardware resources required by any other driver. The installer also verifies the existence of every type of channel used by the application. If the application exports a channel, the installer verifies that an exported channel does not conflict with another application. When conflicts arise, policy in the system manifest resolves them. For example, the manifest might declare that only one device driver can provide the video console

contract. The installation of additional video drivers may be disallowed, or only a single video driver activated at boot time.

As described in Section 4.9, Compile Time Reflection (CTR) is used to generate trusted code to initialize in-process objects for referencing system resources. The CTR templates execute at install time using the attributed program elements in the assemblies named by the application manifest.

The installation process is completed by updating the system manifest metadata to incorporate the new application or device driver.

In the current implementation the entire installation process takes place offline with an installation becoming visible only at the next system boot. This purely off-line installation may be trivially augmented with on-line installation, but on-line installation has not yet been required by our usage scenarios.

5.2.4 Run Time

At run time, metadata drives the initialization of the kernel, device drivers, services, and applications. The boot loader reads a portion of the system manifest to determine which kernel, device drivers, and services should be loaded. The order in which these load and start executing is not specified anywhere; instead the system infers it from the specified dependencies.

As each application is started, the kernel verifies and resolves all metadata dependencies and builds a process configuration record in the kernel. Trusted code, emitted into the application using CTR, parses the configuration record, instantiates local objects for accessing external resources, and puts the local objects into a configuration object in the process' object space.

Returning to the example of the S3Trio64 device driver, the kernel records in the driver's configuration record the need for `IoMemoryRange` objects for `frameBuffer`, `textBuffer`, and `fontBuffer`. The kernel also records the `IoPortRange` objects for `control`, `advanced`, and `gpstat` I/O ports. The kernel creates a channel to connect the device driver to the I/O subsystem and a second channel to connect the driver to the namespace. The channel endpoints are added to the driver's configuration record.

When the device driver starts executing, trusted code in its runtime creates the appropriate `IoMemoryRange` and `IoPortRange` objects in the driver's object space. Because these objects' constructors are accessible only to the trusted runtime code, a device driver only can access I/O resources declared in its metadata and checked for conflicts by the kernel I/O subsystem.

Declaring channel endpoints in application metadata ensures three important properties. First, code for a Singularity process can be statically verified to ensure that it communicates only through fully declared channels, in strict conformance to the channel contracts. Second, applications do not contain global names. For example, the S3Trio64 video device driver is unaware of the `/dev/video` name in the system namespace. Instead, the driver uses a local name, `S3Trio64Config.video`, to refer to a channel with a given contract (`ServiceProviderContract`). The entire layout of the I/O namespace can change without affecting a single line of code in the video driver. Third, applications can be sandboxed, in conformance the principle of least possible privilege, to remove a source of error and security vulnerability in current systems. For example, although the S3Trio64 driver holds an endpoint connected to the system name space, the driver has no ability to create new names or to connect to any other system process.

5.2.5 Reflecting into the Namespace

To facilitate access to the metadata, it is reflected in the system namespace. For example, the I/O system creates a namespace tree describing the mapping of device drivers to the current hardware. `/hardware/locations` lists all buses and each location on a bus. A location is represented as a directory tree, which contains a symbolic link to the device instance that resides in this location.

Similarly, the `/hardware/registrations` tree lists every driver registered with the system. Within this tree, there is one symbolic link pointing to the driver registered for the corresponding hardware signature prefix.

The `/hardware/devices` tree contains an entry for each instance of a physical device in the system. The signature of a device (as determined by device enumeration) is reflected in the directory structure. Within this tree, each instance of the device is a separate subtree with symbolic links pointing to corresponding entries in the locations and drivers trees, to show how the device instance was found, is associated, and is activated.

The `/hardware/drivers` tree lists every registered driver, with a subtree for each instantiation of a driver. The names here are based on the namespace name of the driver class itself. For a particular driver, the tree consists of a symbolic link pointing to the executable image of the driver. It also contains a subtree for each instance of the driver. This subtree holds links to the corresponding device instance. Also contained in this space are the true bindings for all `ServiceProviderContract` endpoints created for each instance of the driver.

Finally, the `/dev` namespace is a public directory holding symbolic links to `ServiceProviderContract` endpoints in the `/hardware/drivers` subtree. In this manner, an application can be bound to a public name, without knowing the true name of the driver.

5.3 Name Server

Singularity provides a single, uniform name space for all services on a system. The name space encompasses transient system services, such as device drivers and network connections, and a persistent store in a file system. The name space is implemented by a distinguished (root) name server and services. The name server allows services to register and unregister themselves in a hierarchical namespace, so they can be discovered by clients. A service responds to requests and, by implementing the name server's contract, can extend the name space beyond its mount point.

The name space is hierarchical. Client programs can access a service by passing a pathname and fresh channel to the name server. Sample pathnames include: "/filesystems/ntfs" or "/tcp/128.0.0.1/80. Conceptually, the name space consists of directories and services. Directories are collections of directories and services that share a common pathname prefix. A service is the active entity that responds to requests on its registered channel.

The entire name space may not exist in a single name server. A service (including one identical to the root name server) may register to handle all requests below a point in the hierarchy. Register, deregister, and lookup messages for this subtree are forwarded to the helper name server. This functionality is similar to mount points in Unix file systems. However, the additional name servers need not operate in the same way as the root one. For example, a TCP service could export the huge dynamic space of IP addresses and create a connection on demand. Or, a helper name server could implement symbolic links.

A slightly simplified form of the name space contract for a client (a server contract includes registration) is:

```
public contract NamespaceContract : ServiceContract {
  in message Bind(char[] in path, ServiceContract.Exp:Start exp);
  out message AckBind();
  out message NakBind(ServiceContract.Exp:Start exp);

  in message Notify(char[] in pathSpec, NotifyContract.Imp:Start imp);
  out message AckNotify();
  out message NakNotify(NotifyContract.Imp:Start imp);

  in message Find(char[] in pathSpec);
  out message AckFind(FindResponse[] in results);
  out message NakFind();

  out message Success();

  override state Start: one {
    Success! -> Ready;
  }

  state Ready: one {
    Bind? ->  ( AckBind! or NakBind! ) -> Ready;
    Find? ->  ( AckFind! or NakFind! ) -> Ready;
    Notify? -> ( AckNotify! or NakNotify! ) -> Ready;
  }
}
```

The `Bind` message provides a path through the name space and a channel, which is passed to the service registered under that name. The `Notify` message passes in a channel, which receives notifications of changes in the directory denoted by the path. The `Find` message returns pathnames of items in the namespace that match a path specification. (The `Success` message is used in the standard protocol to initialize a channel.)

The following chronology illustrates how the name server is used. Processes **C**, **S**, and **NS** represent a client, a service, and a name service, respectively. *nsC* and *nsS* are channels to the name server held by the client and service.

1. (**S** to **NS** on *nsS*) Server registers with fresh channel *lookup.*
2. (**NS** to **S** on *nsS*) Register acknowledgement.
3. (**C** to **NS** on *nsC*) Bind with fresh channel *service.*
4. (**NS** to **S** on *lookup*) Bind with *service.*
5. (**S** to **NS** on *lookup*) Bind reply.
6. (**NS** to **C** on *nsC*) Bind reply.
7. **C** and **S** communicate using channel *service.*

5.4 File System

Though the Singularity name space is a convenient mechanism for naming and accessing services and objects, but it does not provide a means to persist data. Singularity also provides a file system service, which is a sub-tree of the name space. The file system registers itself as a name space service at its mount point (e.g., "/fs") and services requests under its domain. Because the file system acts as a name space server, file system pathnames are suffixes of name space paths (e.g., "/fs/foo/bar").

The Singularity file system supports the common abstractions and operations. It consists of directories and files. Directories can contain files and/or other directories, and they support

traditional operations like enumeration. Files are variable-length byte arrays that clients can read or write at arbitrary offsets.

Files and directories each have their own contract. The file contract permits read and write operations. The directory contract provides file and directory operations such as creation, deletion, and attribute querying. Due to the file system's role as a name space provider, operations such as directory enumeration and lookup do not need special messages in the file system contracts as they are covered by the name space. We are currently considering further ways to integrate file system and name space contracts, as the functionality of the two significantly overlap.

5.4.1 Implementation

Internally, the file system runs as a standard Singularity process. It is comprised of four types of workers: control worker, name space worker, directory worker, and file worker. The control worker, which registers itself separately in the Name space (e.g., "/Fsctrl"), services file system creation, initialization, and mount requests. Once a file system is mounted, its name space worker processes requests forwarded from the file system's parent name space provider, most importantly `Bind` requests. When it receives a `Bind` request, the name space worker passes the endpoint to either a directory or file worker, depending on the type of endpoint.

In turn, file and directory workers, once given an endpoint, service the actual file system operations passed across these endpoints. Endpoints received as part of Bind requests are conceptually bound to a specific file system file or directory. Thus, requests on these endpoints contain no path information, or even a file handle.

5.4.2 Boxwood

To durably store and retrieve data from stable storage (i.e., disks), Singularity uses a modified version of Boxwood as the underlying storage system [34]. Boxwood was originally designed as a distributed storage system that exported higher-level abstractions (e.g., B-trees) rather than simple block interfaces, to demonstrate that more abstract interfaces allow storage applications to be built more easily and with lower overhead. As such, building a file system-like interface on top of Boxwood is not difficult, since Boxwood eliminates the need for much data manipulation, concurrency, consistency, and recovery code. The file system structures in Singularity are nearly identical to those of BoxFS [34]. Files are stored and manipulated as B-trees whose data are file blocks and whose keys are block numbers. Directories are stored and manipulated as B-trees whose data are either files or other directories and whose keys are string names. Metadata about a file system entity is stored under a special key in its B-tree.

The only part of Boxwood that was significantly changed was the interface to the raw disks. On Windows, requests to a disk pass through system calls, but in Singularity, all interactions with a disk pass over channels. Consequently, the lowest layer of Boxwood was converted to use channels.

To avoid excessive and costly copying, Boxwood's C# byte arrays were replaced with pointers to data in the Exchange Heap. As a consequence, I/O aligned on file block boundaries entails no copying between the file system and disk.

5.5 Security

Singularity provides strong isolation between processes. We are constructing a security model on top of this foundation that seeks to maintain system integrity and control access to resources according to application and system policy [1].

5.5.1 Installation-time Mechanisms

Applications are central to Singularity's security model. As explained below, the principals of this model are made up from applications and their combinations. As explained in Section 5.3 applications are named in a hierarchical namespace.

Trust in application publishers can be reflected in the shape of the namespace. For example, system policy might dictate that only applications signed with the Microsoft publisher's certificate can inhabit a portion of the namespace dedicated to Microsoft. The shape of the namespace can also be used to differentiate groups of programs based on how trusted they are according to system policy.

Some of the security enforcement in Singularity can happen statically, at installation time. Because all access to resources in Singularity occurs through channels, the system installer may control the resources to which an application will have access by statically managing its channels. Each application's requirements are specified in a static manifest. The system installer resolves all unbound channels in an application manifest by providing an application configuration that instantiates these channels at run time. This static checking can sometimes be used in support of a least-privilege security environment. For example, if an application is responsible for local processing only, the installer need not provide a direct channel to the network.

5.5.2 Dynamic Access Control

Singularity applications are instantiated at run time in the form of one or more processes. Each process has an immutable identity due to the invocations from which it resulted. All channel pairs created by a process are initialized with this default identity. When a channel endpoint is passed to another process, the receiving process can discover the process identity at the peer endpoint. In this way, a process can obtain the principal associated with messages received over a channel. The principal can then be used in dynamic access control decisions for resources that might be shared.

The principals that are the subjects of access control decisions are compound entities formed from the identity of the requesting application and those of the applications in the invocation chain that led to the requesting application's execution. (In some cases, for server applications which use their own authority, the invocation chain will be suppressed.) We trace invocation history at process granularity. The identity that results from a process invocation need be neither weaker nor stronger than that of the invoker; it is just different. In contrast, some language-based security systems rely on an invocation stack rather than a history, with a finer granularity, and in such a way that each stack frame typically results in a reduction in authority.

In this environment, users are represented as roles of programs. The program that authenticates a user (e.g., by checking a password or a certificate) contributes to the resultant compound identity. Therefore, the identity of a user logged in through a remote protocol differs from one who was authenticated by a local smart-card handler.

Singularity's compound principals are compound identities are represented as text strings, such as:

```
/sys/login@/users/fred + /apps/ms/word
```

This string might represent the system password login program running as the user "`fred`" and then invoking Microsoft Word.

In lieu of access control lists, Singularity uses *access control expressions* (ACEs) in order to define patterns against which principals are matched. These expressions can be quite flexible. For example, one can specify that only Word can read files protected by a certain pattern, or that

"`fred`" running any Microsoft program can have access. Furthermore, the pattern language supports indirection to common subexpressions (in the naming hierarchy in our implementation). This facility is the equivalent to group expansion in conventional access control systems.

We expect to be able to define policy rules from which ACEs can be derived. The hope is to replace a large number of disparate ACEs with a much smaller set of rules. These rules will work best in structured environments like file systems. Since access to many resources (e.g., file system subtrees) can be adjudicated entirely by the system installer should make this easier.

5.5.3 Further Run-time Mechanisms

Channel contracts can be subtyped so as to specify which messages the holder of an endpoint can send. For example, a subtype of the `TcpConnectionContract` of Section 4.1 could describe only methods available to a principal that is allowed to listen but not connect. Thus, a subtype corresponds to a set of permissions. For many protocols, channel establishment will be restricted by an access control check that determines whether the requestor should be granted the permissions implied by the subtype of the channel.

Subject to run-time constraints, endpoints can be passed freely between processes. Along with them is passed the authority to send messages, as specified in the channel contract. Processes are free to act on such messages without further access checks. However, an access check may be performed, and in this case it will be based on the identity of the new holder of the channel. Thus, the holder cannot masquerade as the originator.

As described above, in the default case for process invocation, the identity of the new process is a compound principal of the form `invoker + invokee`. Other than process invocation, there are at least two additional scenarios in which a process may choose to lend some aspect of its identity to another process. In one case, a process may want to grant a partner a capability that allows the partner to act under a joint identity with respect to that capability. In a second case, the system policy may allow a new service to mediate access to an existing service, perhaps adding functionality. In this case, the mediator would need to act on behalf of the original client. In both of these cases, we support identity inheritance by specially *blessing* a channel endpoint: the capability (endpoint) in the first case, and the channel used to bind to the mediator in the second case. A blessed endpoint allows the recipient to inherit the identity of the partner in some limited context.

Processes holding multiple identities can become confused and may use such identities in an inappropriate fashion. Therefore, we hope to limit processes to a single identity in most cases, and to make dealing with multiple identities as easy as possible.

6 Performance

If Singularity's goal is more dependable systems, why does this report include performance measurements? The answer is simple: these numbers demonstrate that architecture that we proposed not only does not incur a performance penalty, but is often as fast as or faster than more conventional architecture. In other words, it is a practical basis on which to build a system.

On the other hand, this paper does not validate our goal of increased dependence. Measuring that aspect of a system is significantly more challenging than performance. We do not yet have results for Singularity.

This section contains measurements comparing the performance of Singularity against other systems. All systems ran on AMD Athlon 64 3000+ (1.8 GHz) on an NVIDIA nForce4 Ultra chipset, 1GB RAM, a Western Digital WD2500JD 250GB 7200RPM SATA disk (without command queuing), and the nForce4 Ultra native Gigabit NIC (without hardware TCP offload

acceleration). We used FreeBSD 5.3, Red Hat Fedora Core 4 (kernel version 2.6.11-1.1369_FC4), and Windows XP (SP2). Singularity ran with a concurrent mark-sweep collector in the kernel, a non-concurrent mark-sweep collector in processes (including drivers), and a minimal round-robin scheduler.

	Cost (CPU Cycles)			
	Singularity	**FreeBSD**	**Linux**	**Windows**
Read cycle counter	8	6	6	2
ABI call	87	878	437	627
Thread yield	394	911	906	753
2 thread wait-set ping pong	1,207	4,707	4,041	1,658
2 message ping pong	1,452	13,304	5,797	6,344
Create and start process	300,000	1,032,000	719,000	5,376,000

Table 1. Cost of basic operations.

6.1 Microbenchmarks

Table 1 reports the cost of primitive operations in Singularity and three other systems. On the Unix systems, the ABI call was `clock_gettres()`, on Windows, it was `SetFilePointer()`, and on Singularity, it was `ProcessService.GetCyclesPerSecond()`. All these calls operate on a readily available data structure in the respective kernels. The Unix thread tests ran on user-space scheduled pthreads. Kernel scheduled threads performed significantly worse. The "wait-set ping pong" test measured the cost of switching between two threads in the same process through a synchronization object. The "2 message ping pong" measured the cost of sending a 1-byte message from one process to another and then back to the original process. On Unix, we used sockets, on Windows, a named pipe, and on Singularity, a channel.

Singularity is a new system and its performance has not been heavily tuned. Basic thread operations in Singularity, such as yielding the processor or synchronizing two threads, are comparable or slightly faster than the other systems. Nevertheless, because of Singularity's SIP architecture, cross-process operations run significantly faster than in the mature systems. Calls from a process to the kernel are 5–10 times faster on Singularity, since the call does not cross a hardware protection boundary. A simple RPC-like interaction between two processes is 4–9 times faster. And, creating a process is 2–18 times faster than the other systems. These advantages should increase as we improve Singularity's thread implementation.

6.2 Disk I/O Benchmarks

To quantify the effect of Singularity's architecture on I/O, we measured the cost of random and sequential disk reads and writes on the various operating systems.

The sequential tests read or wrote 512MB of data from the same portion of the hard disk. The random read and write tests performed 1000 operations on the same sequences of blocks on the disk. The tests were single threaded and performed synchronous raw I/O. Each test was run seven times and the results averaged. All benchmarks ran on the same hardware. On Singularity, the benchmark communicated with the disk driver process over a channel, whereas FreeBSD, Linux, and XP use system calls to communicate with their drivers. FreeBSD and Linux drivers support ATA-7 and have a theoretical maximum throughput of 133MB/s, while Singularity and Windows drivers support ATA-5 and have a theoretical maximum throughput of 66MB/s.

Figure 3 shows the throughput of the systems in I/O operations per second. For random read operations, Singularity's performance was within 10% of the UNIX variants and marginally

better than Windows. For random write operations, Singularity has the highest performance for a majority of block sizes. It is interesting to note that all systems had higher throughput in the random write measurements than for random read. The disk drive buffers writes and re-orders them before transferring them.

For the sequential read operations, Windows XP performed significantly better than the other systems for block sizes less than 8 kilobytes. At 8KB, the difference between the systems becomes less pronounced, as the I/O constraint transitions from the number of I/O requests issued to the available I/O bandwidth from the disk. All systems perform within 6% of the best performer, FreeBSD, for block sizes above 8KB. We attribute the margin of 6% to the different ATA standards supported by the respective operating systems.

For the sequential write operations, each of the systems were the best performer for at least one of the block sizes less than 8KB. (FreeBSD failed to complete the test with a block size of 512 bytes—performance dropped to 50 operations per second and the test did not finish within a reasonable period of time.) At block sizes above 8KB, FreeBSD again achieved the highest performance, with a margin of 6% between the best and worst performers for each block size.

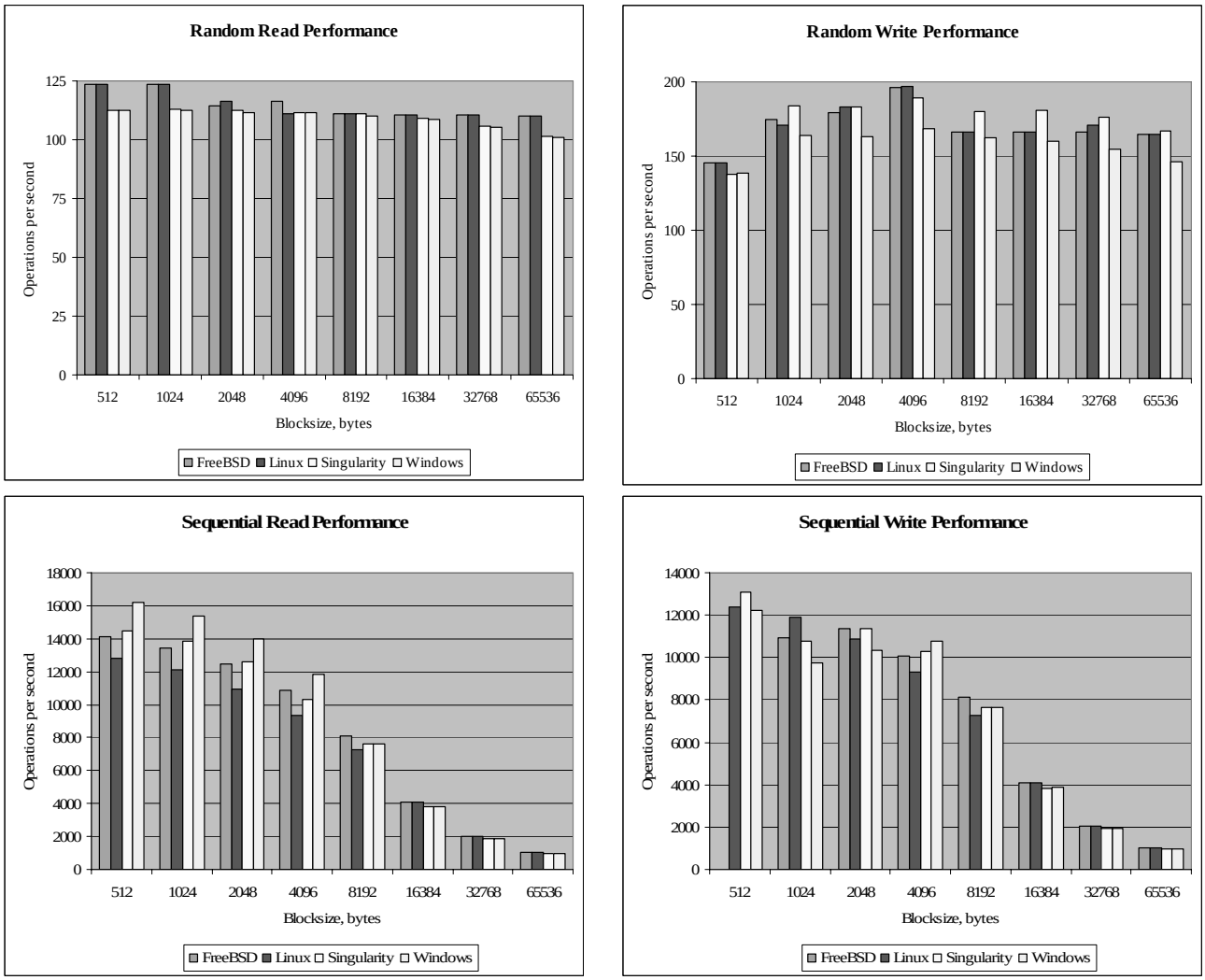

Figure 3 Raw disk benchmarks

The measurements in Figure 3 show that Singularity is competitive with contemporary operating systems in raw disk performance. Singularity's disk driver is not highly optimized, and it does not yet implement the latest version of the ATA specifications, but the overall system performance is still comparable. This demonstrates that running disk drivers in a SIP and communicating over channels do not incur significant performance penalties.

Singularity uses zero-copy channels in the previous disk benchmarks. To quantify the cost of channel operations, we modified the disk driver to execute the sequential read operations in the driver directly, without channels. Table 2 shows the results

	I/O Operations Per Second		
Block size (bytes)	**w/o Channels**	**w/ Channels**	**Degradation %**
512	17658	15831	10
1024	16652	15003	10
2048	15050	13669	9
4096	12491	11598	7
8192	7652	7650	0
16384	3825	3823	0
32768	1903	1902	0
65536	950	950	0

Table 2. Singularity performance with and without channels.

For block sizes below 8KB, the sequential read performance is limited by fixed costs per operation. These include issuing the read request, polling status bits on the device, and handling one interrupt per request. Channels impose an additional overhead of 2 send and receive operations per read request. A comparison of performance with and without channels estimates the cost of the channel operations at 6 µs. The micro benchmarks reported a cost of 1.1 µs (1,925 cycles) per send-receive operation—a discrepancy of 1.9 µs. However, there are two differences between the microbenchmark and the disk benchmark. Firstly, the disk driver has an elaborate select-receive pattern to accommodate the different messages the driver might receive, and this construct requires more cycles to finding a matching pattern. Secondly, the disk benchmark send-receive operations transfer ownership of a shared heap buffer from caller to callee, and this requires a small amount of bookkeeping work in the shared heap.

For block sizes of 8KB and above, sequential read performance is dominated by DMA transfer times. In the measurements, the channel cost is largely obscured by experimental noise.

6.3 SPECweb Benchmarks

To quantify the overhead of Singularity's extension mechanism in a more realistic scenario, we measured the performance of the SPECweb99 benchmark[6] running on Cassini, an open source web server written in C#. This test invoked a fair amount of software on Singularity. Cassini runs on a port of the Microsoft .NET classes, which use channels to communicate with Singularity's network stack and file system. Cassini's code is largely unmodified, except to use channels to communicate with web extensions, including the benchmark code, all of which ran in separate SIPs.

[6] Our use of this benchmark (as translated to C#) is non-conformant in several aspects—the Singularity TCP stack is not fully IPv4 compliant, Cassini is not fully standard compliant, benchmark warm-up time is reduced, length of execution is reduced, number of iterations are reduced, and server-side logging is absent. Nevertheless, Singularity fully implements all dynamic operations in the benchmark and the workload mix is standard.

Singularity achieves 91 ops/second with a weighted average throughput of 362 Kbits/second. By contrast, Microsoft Windows 2003 running the IIS web server, on identical hardware, achieves 761 ops/second with a weighted average throughput of 336 Kbits/second.

System instability under heavy load and file system performance bottlenecks limit the number of connections Singularity could serve at the benchmark's minimum acceptable rate, and consequently reduced Singularity's overall score. However, Singularity's average response time, with 23 connections, of 322 millisec./op is comparable to Window's time, with 25 connections, of 304 millisec./op. This suggests that Singularity's benchmark score is not limited by internal latency in the system or SIPs.

Singularity is currently constrained by its file system's limited throughput. The file system is based on the Boxwood abstractions [34]. Its performance problems are not limited to Singularity. We measured the file system's performance using a simple benchmark that read randomly chosen files from the SPECweb benchmark. On Windows, its throughput was 2.7MB/sec. On Singularity, its throughput was 2.6MB/sec. With a file system throughput of 2.6MB/sec, a system can support no more than 50–70 SPECweb connections.

By contrast, Singularity's network stack does not appear to be the bottleneck, as it can sustain a transmission throughput of 48Mbits/sec.

6.4 Executable Sizes

The memory overhead of a SIP limits the number and granularity of processes that can be created on a system. Table 3 reports the size of a file containing the minimal "hello world" program. On the Unix systems, the programs are statically linked to bring in their libraries. On Singularity, code is linked with its full run-time system (including GC), and measured after Bartok optimization to remove unused code and data. As can be seen, the amount of code and data in a Singularity process is roughly comparable to a Unix C program.

hello world	**File Size (bytes)**			
	Singularity	**Free BSD**	**Linux**	**Windows**
C		89,796	431,900	36,864
C++		534,856	984,520	69,632
Sing#	286,208			

Table 3. File sizes for "hello world" program.

Table 4 reports the amount of virtual address space used by these programs on various systems. The Singularity process is smaller than other systems' processes—with one exception—by up to an order of magnitude.

	Memory Usage (KB)			
	Singularity	**FreeBSD**	**Linux**	**Windows**
C		1,200	1,416	644
C (static)		232	664	544
C++		2,148	2,532	804
C++ (static)		704	1,216	572
Sing#/C#	408			4,116

Table 4. Dynamic memory usage for "hello world" program.

In Singularity, we hope to share read-only pages containing the run-time system among similar processes, to reduce memory utilization and accelerate process creation times. In this

example, approximately 280KB of the 408KB runtime originates in the executable and is not heap allocated. Of this, 137KB is code, 26KB is read-only data, and 72KB is read-write data (22KB VTables, 28KB immutable strings, and 15KB `System.Type` objects). With some changes—e.g., moving locks out of immutable objects—read-write items might be shared as well. In that case, 58% of the address space would be sharable.

7 Related Work

The large amount of related work can be divided to four major areas: OS architecture, system extensibility, language safety, and defect detection.

7.1 OS Architecture

Singularity is a microkernel operating system that differs in a number of respects from previous microkernel systems, such as Mach, L4, SPIN, Vino, and Exokernel [2, 8, 17, 25, 46]. Microkernel operating systems partition a monolithic kernel into components that run in separate processes. Previous systems, with exception of kernel extensions of SPIN, were written in an unsafe programming language and used processor memory management hardware and protection rings as an isolation mechanism. Singularity uses language safety and message-passing communication to isolate processes and prevent access to hardware resources.

Hardware-enforced processes have considerable overhead and so microkernel architectures evolved to allow kernel extensions, while attempting to protect system integrity. SPIN implemented extensions in a safe language and using programming language features to restrict access to kernel interfaces [9]. Vino used sandboxing to prevent unsafe extensions from accessing kernel code and data and lightweight transactions to control resource usage [46]. Both systems allowed extensions to directly manipulate kernel data, which left open the possibility of corruption through incorrect or malicious operations and inconsistent data after extension failure. Singularity's stronger extension model prevents data sharing between a parent and an extension. Singularity also uses a single, general extension mechanism throughout the system, from device drivers through applications, not a specialized mechanism for a kernel. Engler's Exokernel defined kernel extensions for packet filtering in a domain-specific language and generated code in the kernel for this safe, analyzable language [22]. This approach is attractive for well-defined domains like packet filtering, but is difficult to generalize.

Previous operating systems have been written in safe programming languages. Early examples were "open" systems [33], that ran in a single address space and supported threads (confusingly called "processes"). They were viewed as "single user" systems, and consequently paid little attention to security, isolation, or fault tolerance. Smalltalk-80 and Lisp Machine Lisp used dynamic typing and run-time validation to ensure language safety, but isolation depended on programmer discipline and could be subverted through introspective and system operations [23, 54]. Pilot and Cedar/Mesa were single-user, single-address space systems implemented in Mesa, a statically typed, safe language [43, 50].

Inferno is a single address space operating system that runs programs only written in a safe programming language (Limbo) [15]. Unlike Singularity, it supports only a single virtual machine image, depends on dynamic code loading, and provides no memory or failure isolation.

RMoX is an operating system partially written in occam [6]. Its architecture is similar to Singularity, with a system structured around message-passing between processes. However, RMoX's uses a kernel written in C, from the OSKit, and only its device drivers and system process are written in a safe language.

Several operating systems have been written in Java. JavaOS is a port of the Java virtual machine to bare hardware [44]. It replaces a host operating system with a microkernel written in

an unsafe language and Java code libraries. Unlike Singularity, it only supports a single process and object space.

The JX system is similar to Singularity in many respects. It is a microkernel system written almost entirely in a safe language (Java) [24]. Processes on JX do not share memory and communicate through synchronous RPC with deep copying of parameters. The processes run in a single hardware address space and rely on language safety for isolation. The primary differences between JX and Singularity are the communication and extension mechanisms. Singularity uses asynchronous message passing over strongly typed channels, which is more general (RPC is a special case) and permits verification of communication behavior and system-wide liveness properties. Singularity has 0-copy transfers on channels while preserving memory independence. JX uses Java's extension model of dynamic loading. Singularity SIPs are closed, which provides failure isolation and enables more accurate program analysis, thereby facilitating code optimization and defect detection.

Device drivers are both the most common extensions in operating systems and their largest source of defects [12, 37, 49]. Nooks provides a protected environment to run existing device drivers in the Linux kernel [48, 49]. 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, runs its newly written drivers in the more general SIP. A complementary line of research has developed tools to find defects in drivers. Software analysis tools, such as the Static Driver Verifier from Microsoft, perform domain-specific analysis to find errors in drivers [5]. A safe programming language would make these tools more accurate and enable them to make fewer unverified assumptions about conformance to language semantics.

7.2 Application Extensibility

There is considerable interest in developing better mechanisms to isolate extensions in application software. Software fault isolation (SFI) isolates untrusted code in its own domain by inserting run-time tests to validate memory references and indirect control transfers, a technique called sandboxing [52]. Sandboxing has high overhead and only offers memory safety, but not type safety. Nor does sandboxing provide any mechanisms to control data shared between the host and the extension. Finally, sandboxing finds errors late during execution rather than during compilation.

Minor changes to memory management hardware could provide finer grain protection boundaries within an address space. For example, Mondrian memory protection permits arbitrary access control at word boundary with reasonable overhead [55].

Java, among other goals, strongly encouraged dynamic code loading (e.g., Applets) and required a new security model to protect against untrusted extensions. The JVM combines verified type-safe code and fine-grain, run-time access control to provide an environment in which a system can constrain the execution of general untrusted extensions [35]. Singularity runs extensions in separate processes, which provide a stronger assurance of isolation and a more tractable security problem, which does not entail a large number of fine grain policy decisions.

Other projects have implemented OS-like functionality, such as process and scheduling mechanisms, in the Java runtime. When multiple applications run in a JVM process, these mechanisms control resource allocation and facilitate cleanup after failure. J-Kernel implemented protection domains in a JVM process, provided revocable capabilities to control object sharing, and developed a clean semantics for domain termination [26]. 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 [27]. The KaffeOS provides a process abstraction in a JVM along with

mechanisms to control resource utilization in a group of processes [4]. Java has incorporated many of these ideas into a new feature called isolates [41] similar to the existing concept of AppDomains in Microsoft's CLR. Singularity eliminates the duplication of resource management and isolation mechanisms between an operating system and language runtime by providing a consistent mechanism across all levels of the system. Singularity's SIPs are closed and non-extensible, which provides a greater degree of isolation and fault tolerance than Java or CLR-based approaches, which share a common run-time system.

7.3 Language Safety

Safe programming languages are not recent phenomena. Pascal and Ada are safe, statically verifiable imperative languages. Modula-3, Dylan, and Java are safe object-oriented languages. Safe languages have become more popular with faster processors, more refined type systems, and improved run-time systems. Nevertheless, they are not widely used for system implementation because their time and space overhead is higher than low-level languages such as C or C++ and they offer little control over data layout. In Java, some of this overhead is attributable to the language's open execution environment, in which reflection and dynamic class loading constrain a compiler's ability to globally analyze and optimize code. Singularity eliminates these features, so a globally optimizing compiler can produce object code competitive with conventional, unsafe languages [20].

Another line of research has led to type safe dialects of C (but not C++). CCured is a compiler and run-time system that extensively analyzes C code to determine where it is statically safe [39]. It inserts run-time tests for properties that cannot be statically verified. Cyclone is another safe C dialect [30]. It is less aggressive about inserting run-time tests than CCured, which may need to change the layout of structs to incorporate type information. Cyclone, however, can reject C programs as inherently unsafe. Vault is a more aggressive redesign of C, which introduces new safe language constructs and a specification language for explicit resource management and low-level data representations, It retain some binary compatibility with C and does not rely on garbage collection [14].

A system that depends on language safety cannot trust a compiler, but must verify the safety of code before it executes. If executables are delivered as typed intermediate languages, such as Java bytecodes or Microsoft's MSIL, verification is a relatively straightforward process. This is the approach that Singularity currently uses to ensure system and application code is type safe. It is also possible to perform a similar verification on assembly language, if a compiler augments it with type annotations [36, 38]. Low level, unverified, unsafe code is a potential weakness in any system, but is a particular problem in systems that do not rely on memory protection. Singularity contains unsafe code at the lower levels of the language runtime and operating system. Verifying the safety of this code would help ensure system reliability. One area of active research is producing a safe garbage collector [53].

7.4 Defect Detection Tools

Singularity is designed to facilitate the operation of static defect detection tools. Analyzing systems written in unsafe languages, such as C or C++, is difficult because these languages' weak guarantees do not provide a clear semantics for use in a tool and are difficult to analyze and enforce. Tools for these languages are either heuristic [10, 16, 18, 31] or make guarantees under the assumption that programs do not violate language semantics or use loopholes such as casting pointers to integers [5, 13]. Singularity is compiled into MSIL, which is a safe intermediate language with a clear, albeit informal, semantics that provides a firm basis for program analysis.

Another difficulty facing defect detection tools is the openness of the environment in which code executes. This openness arises from public interfaces that can be invoked in a variety of

contexts and from dynamic code modification arising from reflection and code loading. Singularity annotates its interfaces with specifications that describe their functional behavior and that can be verified statically or at run time. Currently, channels, the public interfaces to a process, contain a behavioral description of the protocol for the channel, which can be verified through a technique called conformance checking [42]. In addition, Singularity processes are closed, so a compiler or static analysis tool can see all of their code and can rely on it remaining unchanged at run time.

7.5 Security

Abadi et al. contains a discussion of related work in the security area [1].

8 Conclusion

Singularity is a micro-kernel operating system that uses advances in programming languages and compilers to build lightweight, software-isolated processes, which provide code with protection and failure isolation at lower overhead than conventional, hardware supported processes. Singularity provides an isolation boundary by running verifiably safe programs and by preventing object pointers from passing between processes' object spaces.

SIPs, in turn, enable a new solution to the problem of code extension in systems and applications. In Singularity's model, extensions are not loaded into their parent process, but instead run in their own process and communicate over strongly typed channels. This model fixes some of the major problems with extensions, since in Singularity, they cannot directly access their parents' data or interfaces, and, if they fail, they can be easily terminated by killing their parents.

Singularity is above all a laboratory for exploring interactions among system architecture, programming languages, compilers, specification, and verification. Advances in each of these areas enable and reinforce advances in the others domains, which limits the benefit and impact of studying an area in isolation. Singularity is small and well structured, so it is possible to make changes that span the arbitrary boundaries between these domains. At the same time, it is large and realistic enough to demonstrate the practical advantages of new techniques.

9 References

1. Abadi, M., Birrell, A. and Wobber, T. Access Control in a World of Software Diversity. in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, 2005.
2. Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. A New Kernel Foundation for UNIX Development. in *Summer USENIX Conference*, Atlanta, GA, 1986, 93-112.
3. Association, S.I.I. Packaged Software Industry Revenue and Growth, Software & Information Industry Association, 2004.
4. 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.
5. Ball, T. and Rajamani, S.K. The SLAM Project: Debugging System Software via Static Analysis. in *Proceedings of POPL 2002: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, OR, 2002, 1-3.
6. 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.
7. Barnett, M., Leino, K.R.M. and Schulte, W. The Spec# Programming System: An Overview. in *Proceedings of Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, Springer Verlag, Marseille, France, 2004.
8. Bershad, B.N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S. and Sirer, E.G. SPIN: An Extensible Microkernel for Application-specific Operating System Services. in *Proceedings of the 6th ACM SIGOPS European Workshop*, Wadern, Germany, 1994, 68-71.
9. 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.
10. Bush, W.R., Pincus, J.D. and Sielaff, D.J. A Static Analyzer for Finding Dynamic Programming Errors. *Software-Practice and Experience*, *30* (5), 2000, 775-802.

11. 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.
12. 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.
13. Das, M., Lerner, S. and Seigle, M. ESP: Path-Sensitive Program Verification in Polynomial Time. in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, Berlin, Germany, 2002, 57-69.
14. DeLine, R. and Fähndrich, M. Enforcing High-Level Protocols in Low-Level Software. in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*, Snowbird, UT, 2001, 59-69.
15. Dorward, S., Pike, R., Presotto, D.L., Ritchie, D.M., Trickey, H. and Winterbottom, P. The Inferno Operating System. *Bell Labs Technical Journal*, *2* (1), 1997, 5-18.
16. Engler, D., Chelf, B., Chou, A. and Hallem, S. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. in *Proceedings of the 4th Symposium on Operating Systems Design and International (OSDI 2000)*, Sand Diego, CA, 2000, 1-16.
17. 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.
18. Evans, D., Guttag, J., Horning, J. and Tan, Y.M. LCLint: A Tool for Using Specifications to Check Code. in *Proceedings of the ACM SIGSOFT Second Symposium on the Foundations of Software Engineering*, New Orleans, LO, 1994.
19. Fähndrich, M. and Larus, J.R. Language Support for Fast and Reliable Message Based Communication in Singularity OS. in *Submitted to EuroSys2006*, 2005.
20. Fitzgerald, R., Knoblock, T.B., Ruf, E., Steensgaard, B. and Tarditi, D. Marmot: an Optimizing Compiler for Java. *Software-Practice and Experience*, *30* (3), 2000, 199-232.
21. 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.
22. 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), 2002, 49-83.
23. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
24. 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.
25. Härtig, H., Hohmuth, M., Liedtke, J. and Schönberg, S. The Performance of m-kernel-based Systems. in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, 1997, 66-77.
26. 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.
27. 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.
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.
29. IFIP. IFIP WG10.4 on Dependable Computing and Fault Tolerance, 2005.
30. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J. and Wang, Y. Cyclone: A Safe Dialect of C. in *Proceedings of the USENIX 2002 Annual Conference*, Monterey, CA, 2002, 275-288.
31. Johnson, S.C. Lint, a C Program Checker *Computer Science Technical Report*, AT&T Bell Laboratories, 1978.
32. Jones, M.B., Leach, P.J., Draves, R.P. and III, J.S.B. Modular Real-time Resource Management in the Rialto Operating System. in *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, 1995, 12-17.
33. Lampson, B.W. and Sproull, R.F. An Open Operating System for a Single-user Machine. in *Proceedings of the Seventh ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, 1979, 98-105.
34. MacCormick, J., Murphy, N., Najork, M., Thekkath, C.A. and Zhou, L. Boxwood: Abstractions as the Foundation for Storage Infrastructure. in *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, 2004, 105-120.
35. McGraw, G. and Felten, E.M. *Java Security: Hostile Applets, Holes, & Antidote*. John Wiley and Sons, New York, 1996.
36. 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), 1999, 527-568.
37. Murphy, B. and Levidow, B. Windows 2000 Dependability. in *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, New York, NY, 2000.
38. Necula, G.C. Proof-Carrying Code. in *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, 1997.
39. Necula, G.C., McPeak, S. and Weimer, W. CCured: Type-safe Retrofitting of Legacy Code. in *Proceedings of POPL 2002: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, OR, 2002, 128-139.
40. 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.
41. Process, J.C. Application Isolation API Specification *Java Specification Request*, 2003, JSR-000121.

42. Rajamani, S.K. and Rehof, J. Conformance Checking for Models of Asynchronous Message Passing Software. in *Proceedings of the International Conference on Computer Aided Verification (CAV 02)*, Springer, Copenhagen, Denmark, 2002, 166-179.
43. Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., McJones, P.R., Murray, H.G. and Purcell, S.C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, *23* (2), 1980, 81-92.
44. Saulpaugh, T. and Mirho, C. *Inside the JavaOS Operating System*. Addison-Wesley, 1999.
45. Schneider, F.B. Enforceable Security Policies. *ACM Transactions on Information and System Security (TISSEC)*, *3* (1), 2000, 30-50.
46. 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.
47. 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.
48. Swift, M.M., Annamalai, M., Bershad, B.N. and Levy, H.M. Recovering Device Drivers. in *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, 2004, 1-16.
49. 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.
50. Swinehart, D.C., Zellweger, P.T., Beach, R.J. and Hagmann, R.B. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, *8* (4), 1986, 419-490.
51. 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.
52. 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.
53. Wang, D.C. and Appel, A.W. Type-preserving Garbage Collectors. in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, Berlin, Germany, 2002, 166-178.
54. Weinreb, D. and Moon, D. *Lisp Machine Manuel*. Symbolics, Inc, Cambridge, MA, 1981.
55. Witchel, E., Cates, J. and Asanovic', K. Mondrian Memory Protection. in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 2002, 304-316.

Appendix A

The Singularity V1 kernel ABI exposes 126 methods. In general usage, the ABI is typically used through a class library; in much the same way that `libc` wrap `syscalls` in Unix systems. MSIL verification ensures that methods marked with the `unsafe` attribute can only be accessed by trusted code in the run-time system.

```
namespace Microsoft.Singularity.V1.Processes {
     struct ProcessHandle {
       static unsafe bool Create(char *args, int *argLengths, int argCount, out
     ProcessHandle handle);
    static unsafe bool Create(char *args, int *argLengths, int argCount,
     ExtensionContract.Exp opt(ExHeap) * exp, out ProcessHandle handle);
    static unsafe bool Create(char *args, int *argLengths, int argCount, char
     *role, int rolelength, ExtensionContract.Exp opt(ExHeap) * exp, out
     ProcessHandle handle);
    static void Dispose(ProcessHandle handle);
    static bool Start(ProcessHandle handle);
    static void Join(ProcessHandle handle, out bool started);
    static bool Join(ProcessHandle handle, TimeSpan timeout, out bool started);
    static bool Join(ProcessHandle handle, DateTime stop, out bool started);
    static bool Suspend(ProcessHandle handle, bool recursive);
    static bool Resume(ProcessHandle handle, bool recursive);
    static void Stop(ProcessHandle handle, int exitcode);
    static void SuspendBarrier();
    static int GetProcessId(ProcessHandle handle);
    static int GetExitCode(ProcessHandle handle);
  }
}

namespace Microsoft.Singularity.V1.Services {
  struct DebugService {
    static unsafe void PrintBegin(out char * buffer, out int length);
    static unsafe void PrintComplete(char * buffer, int used);
    static unsafe void Print(char * buffer);
    static unsafe void Print(char * buffer, int length);
    static void Break();
    static bool IsDebuggerPresent();
  }

  struct DeviceService {
    static unsafe uint GetPnpSignature(char * output, uint maxout);
    static bool GetPciConfig(out ushort pciAddressPort, out ushort pciDataPort,
     out ushort identifier);
    static int GetIrqCount(byte line);
    static uint GetDynamicIoRangeCount();
    static bool GetDynamicIoPortRange(uint range, out ushort port, out ushort
     size, out bool readable, out bool writable);
    static unsafe bool GetDynamicIoMemoryRange(uint range, out byte * data, out
     uint size, out bool readable, out bool writable);
    static bool GetDynamicIoIrqRange(uint range, out byte line, out byte size);
    static bool GetDynamicIoDmaRange(uint range, out byte channel, out byte
     size);
    static uint GetFixedIoRangeCount();
    static bool GetFixedIoPortRange(uint range, out ushort port, out ushort
     size, out bool readable, out bool writable);
    static unsafe bool GetFixedIoMemoryRange(uint range, out byte * data, out
     uint size, out bool readable, out bool writable);
    static bool GetFixedIoIrqRange(uint range, out byte line, out byte size);
    static bool GetFixedIoDmaRange(uint range, out byte channel, out byte
     size);
  }
```

```
struct EndpointCore {
  static EndpointCore* opt(ExHeap)! Allocate(uint size, SystemType st);
  static void Free(EndpointCore* opt(ExHeap) endpoint);
  static void Connect(EndpointCore* opt(ExHeap)! imp, EndpointCore*
   opt(ExHeap)! exp);
  static void TransferBlockOwnership(Allocation* ptr, ref EndpointCore
   target);
  static void TransferContentOwnership(ref EndpointCore transferee, ref
   EndpointCore target);
  static uint GetPrincipal(EndpointCore* opt(ExHeap) endpoint, char
   *outprincipal, uint maxout);
}

struct ExchangeHeapService {
  static unsafe UIntPtr GetData(Allocation *allocation);
  static unsafe UIntPtr GetSize(Allocation *allocation);
  static unsafe UIntPtr GetType(Allocation *allocation);
  static unsafe Allocation * Allocate(UIntPtr size, SystemType type, uint
   alignment);
  static unsafe void Free(Allocation *allocation);
  static unsafe Allocation * Share(Allocation *allocation, UIntPtr
   startOffset, UIntPtr endOffset);
  static unsafe Allocation * Split(Allocation *allocation, UIntPtr offset);
}

struct PageTableService {
  static unsafe uint * GetPageTable();
  static UIntPtr GetPageCount();
  static uint GetProcessTag();
  static UIntPtr Allocate(UIntPtr bytes, UIntPtr reserve, UIntPtr alignment);
  static UIntPtr AllocateBelow(UIntPtr limit, UIntPtr bytes, UIntPtr
   alignment);
  static UIntPtr AllocateExtend(UIntPtr addr, UIntPtr bytes);
  static void Free(UIntPtr addr, UIntPtr bytes);
  static bool Query(UIntPtr queryAddr, out UIntPtr regionAddr, out UIntPtr
   regionSize);
}

struct ProcessService {
  static void Stop(int exitCode);
  static DateTime GetUpTime();
  static DateTime GetUtcTime();
  static long GetCycleCount();
  static long GetCyclesPerSecond();
  static ushort GetCurrentProcessId();
  static int GetStartupEndpointCount();
  static unsafe ExtensionContract.Exp opt(ExHeap) * GetStartupEndpoint(int
   arg);
  static int GetStartupArgCount();
  static unsafe int GetStartupArg(int arg, char * output, int maxout);
  static unsafe void GetTracingHeaders(out LogEntry *logBegin, out LogEntry
   *logLimit, out LogEntry **logHead, out byte *txtBegin, out byte *txtLimit,
   out byte **txtHead);
}

struct StackService {
  static void GetUnlinkStackRange(out ulong unlinkBegin, out ulong
   unlinkLimit);
  static void LinkStack0();
  static void LinkStack4();
  static void LinkStack8();
  static void LinkStack12();
```

```
    static void LinkStack16();
    static void LinkStack20();
    static void LinkStack24();
    static void LinkStack28();
    static void LinkStack32();
    static void LinkStack36();
    static void LinkStack40();
    static void LinkStack44();
    static void LinkStack48();
    static void LinkStack52();
    static void LinkStack56();
    static void LinkStack60();
    static void LinkStack64();
  }
}

namespace Microsoft.Singularity.V1.Threads {
  struct AutoResetEventHandle : SyncHandle
  {
    static bool Create(bool initialState, out AutoResetEventHandle handle);
    static void Dispose(AutoResetEventHandle handle);

    static bool Reset(AutoResetEventHandle handle);
    static bool Set(AutoResetEventHandle handle);
    static bool SetNoGC(AutoResetEventHandle handle);
  }

  struct InterruptHandle : SyncHandle {
    static bool Create(byte irq, out InterruptHandle handle);
    static bool Dispose(InterruptHandle handle);
    static bool Ack(InterruptHandle handle);
  }

  struct ManualResetEventHandle : SyncHandle {
    static bool Create(bool initialState, out ManualResetEventHandle handle);
    static void Dispose(ManualResetEventHandle handle);
    static bool Reset(ManualResetEventHandle handle);
    static bool Set(ManualResetEventHandle handle);
  }

  struct MutexHandle : SyncHandle {
    static bool Create(bool initiallyOwned, out MutexHandle handle);
    static void Dispose(MutexHandle handle);
    static void Release(MutexHandle handle);
  }

  struct SyncHandle {
    static bool WaitOne(SyncHandle handle);
    static bool WaitOne(SyncHandle handle, TimeSpan timeout);
    static bool WaitOne(SyncHandle handle, DateTime stop);
    static bool WaitOneNoGC(SyncHandle handle);
    static int WaitAny(SyncHandle * handles, int handleCount);
    static int WaitAny(SyncHandle * handles, int handleCount, TimeSpan
     timeout);
    static int WaitAny(SyncHandle * handles, int handleCount, DateTime stop);
  }

  struct ThreadHandle {
    static bool Create(int threadIndex, out ThreadHandle thread);
    static void Dispose(ThreadHandle thread);
    static void Start(ThreadHandle thread);
    static ThreadState GetThreadState(ThreadHandle thread);
    static TimeSpan GetExecutionTime(ThreadHandle thread);
```

```
    static bool Join(ThreadHandle thread);
    static bool Join(ThreadHandle thread, TimeSpan timeout);
    static bool Join(ThreadHandle thread, DateTime stop);
    static ThreadHandle CurrentThread();
    static UIntPtr GetThreadLocalValue();
    static void SetThreadLocalValue(UIntPtr value);
    static void Sleep(TimeSpan timeout);
    static void Sleep(DateTime stop);
    static void Yield();
    static void SpinWait(int iterations);
  }
}

namespace Microsoft.Singularity.V1.Types {
  struct SystemType {
    static SystemType RootSystemType();
    static SystemType Register(long lowerHash, long upperHash, SystemType
     parent);
    static bool IsSubtype(SystemType child, SystemType parent);
    static unsafe bool IsSubtype(Allocation* childData, SystemType parent);
    static bool IsNull(SystemType st);
  }
}
```