DOI:1145/1787234.1787253

Safe, modern programming languages let Microsoft rethink the architectural trade-offs in its experimental operating system.

BY JAMES LARUS AND GALEN HUNT

The Singularity System

THE SINGULARITY PROJECT at Microsoft Research began by asking what modern operating-system and application software would look like if it were designed with modern software-engineering practices and tools.[9] Answering is important, since almost every system today shares a common intellectual heritage with the time-sharing systems developed in the 1960s and 1970s. Computers and the computing environment have changed dramatically since then, but system software has evolved much more slowly, leaving a wide gap between system requirements and capabilities.

In the 1960s, computers were limited, expensive devices used only by small groups of highly trained experts. Their limited speed, memory capacity, and storage forced designers and programmers to be parsimonious with resources. Applications and systems were generally written in assembly language, not in high-level programming languages, as they are today. Extensive sharing of code and data was essential for efficient use of scarce memory. Moreover, computer users and uses were also very different; the small group of people with access to computers understood the technology and tolerated its shortcomings. Though computers were increasingly important in business, and thus operated in secure environments, they were not central to anyone's personal life. None of these characteristics is true today.

Construction of the Singularity operating system began in 2004 with three design principles:

Use safe high-level programming languages to the greatest extent possible. They prevent entire classes of critical errors (such as those enabling buffer overrun attacks) while facilitating development and use of accurate and efficient software-development tools;

Software failure should not lead to system failure. Despite advances in programming languages and tools, perfect software remains a vision for the future. However, robust system architecture can limit the consequences of a failure and give a system the ability to respond and recover without having to reboot; and

Systems should be self-describing at all levels of abstraction. Specification and verification are increasingly common for language features and library interfaces. However, as systems consist of many components, most are never formally described. Introducing specifications at the boundaries of components describes both their dependencies and their contributions to the system, enabling principled decisions about system architecture.

» key insights

- **New demands on computer systems require rethinking assumptions concerning language, operating system, and system architecture.**
- **Safe modern programming languages promise significant benefits for constructing high-performance systems.**
- **Systems must be self-describing at all levels of abstraction for building automatic tools that verify and validate their correctness and integrity.**

ILLUSTRATION BY STUDIO TONNE

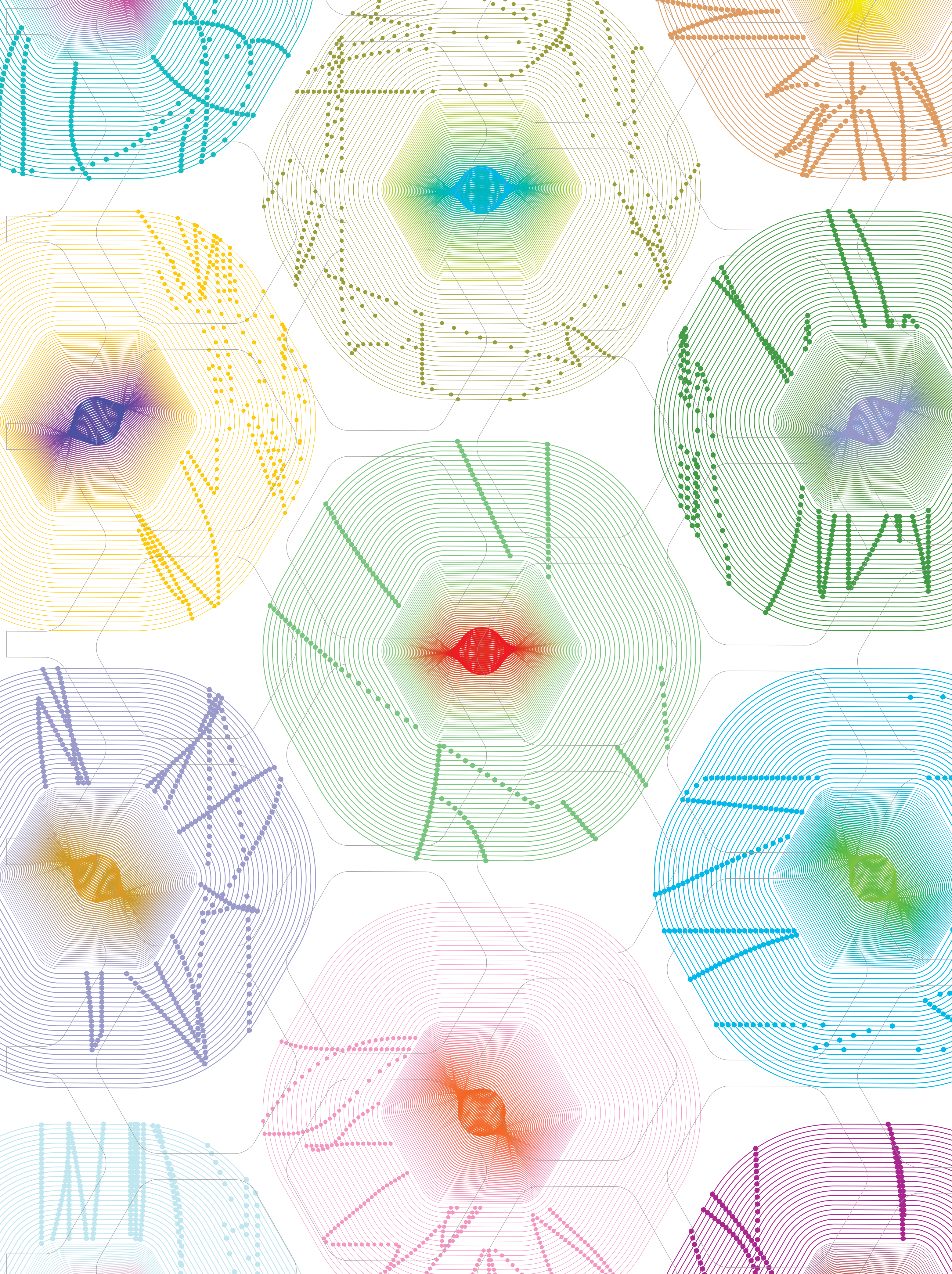
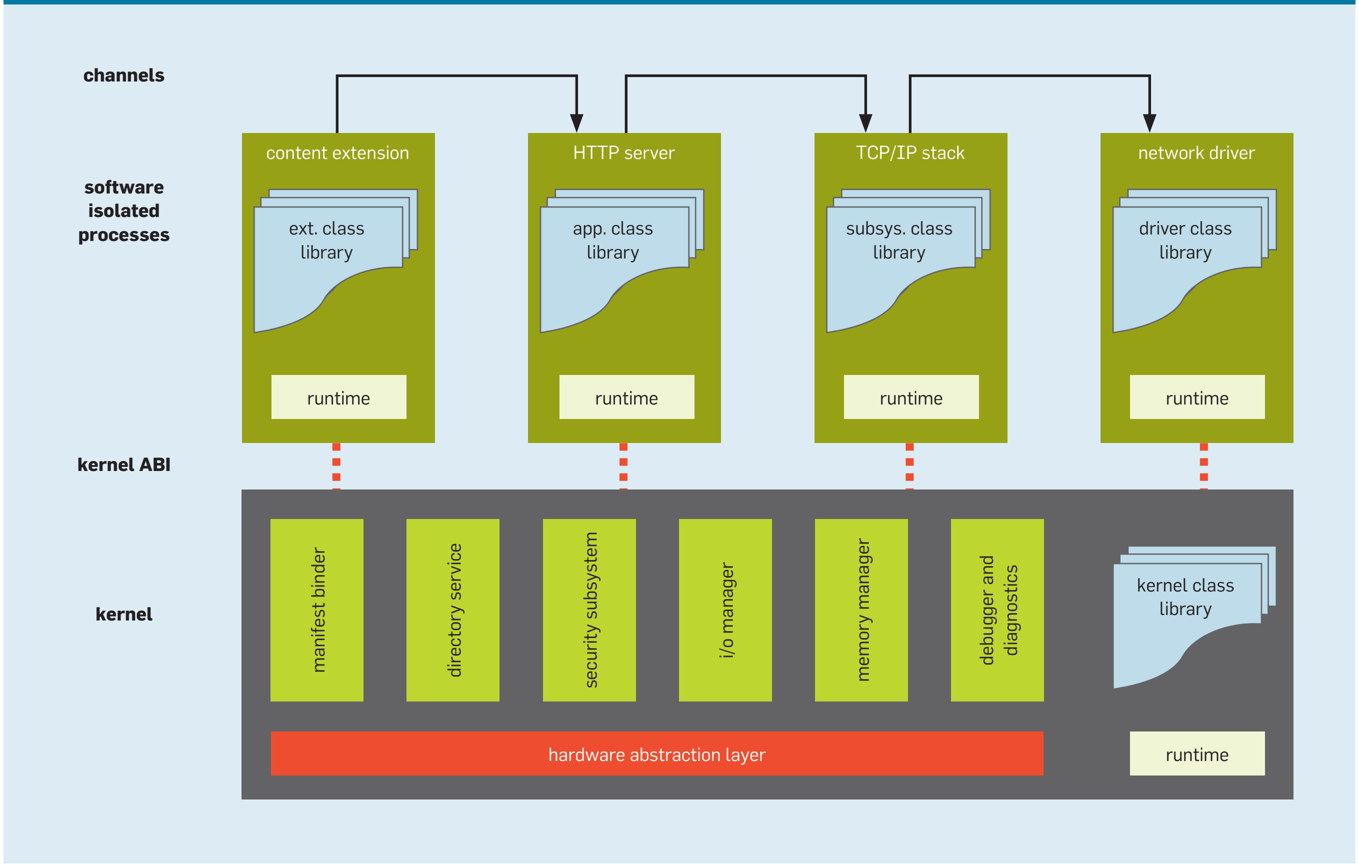


Figure 1. Structure of a Singularity system.



Singularity differs in significant ways from most previous operating systems, pointing the way to systems better able to respond to future computing requirements. Unlike Microsoft Windows and Unix systems, it follows a microkernel design philosophy in which much of a system's functionality, including its device drivers and major subsystems, resides in processes outside the kernel; Figure 1 outlines the architecture of a Singularity system. Unlike other microkernel systems, most Singularity code is written in safe high-level Sing#, a dialect of C#.[a] Moreover, also unlike other systems, all user code in processes—outside the OS-supplied runtime—must be written in a type- and memory-safe language (such as Sing#, C#, F#, or even Visual Basic).

Conceived as an extensible home server, Singularity has been used primarily as a research vehicle to investigate new OS abstractions. Although the Singularity kernel includes many features found only in production OS kernels (such as multiprocessor support, full-feature kernel debuggers, and support for hardware standards like ACPI), Singularity is not a replacement for Windows or Linux, as it has no GUI and only a sparse set of user applications.

Unlike in other systems, processes in Singularity are software-isolated processes, or SIPs, that rely on language safety, not hardware mechanisms, to isolate system software components from one another. SIPs provide isolation and failure containment at far less performance cost than hardware mechanisms, so they can be used in more places than conventional processes. Due to the lower cost of isolation, Singularity can require an extension ("plug-in") to reside in its own SIP that prevents the extension's failure from affecting its host SIP. (We describe later how hardware protection can be combined with SIPs in Singularity to provide multiple layers of protection.) Singularity also assumes more authority to decide which system components can be safely loaded and executed. Due to the additional information provided by manifests and specifications, Singularity is able to detect and avoid conflicts among components and prevent or isolate the use of unsafe code.

Safe Programming Languages

Modern programming languages (such as C# and Java) are type and memory safe. Safety ensures a program applies only operations appropriate to a particular type of object to instances of that object, a program does not create or modify memory references, and memory is reclaimed only when no longer in use. These properties, not present in C, C++, and other languages, help detect programming errors that could have serious consequences; for example, in a safe language, input that overwrites a string buffer causes a runtime exception, rather than silently failing and permitting an attacker to inject malicious code. In addition, safe languages rely on garbage collection to reclaim memory, relieving programmers of having to devise and enforce conventions concerning when an object is no longer in use and which component has the obligation to free the object.

a The hardware abstraction layer in Singularity consists of 21.5KLOC but only 1,700 lines of unsafe Sing# and 350 lines of assembly code. The counterpart hardware-abstraction layer in Windows includes 25KLOC of unsafe C and assembly.

Furthermore, because safe languages have a fully defined semantics, unlike languages like C, with one semantics if a program obeys the language rules and no guarantees if they don't, program-analysis tools are not put in the untenable position of assuming a buggy program plays strictly according to the language definition.

Safe languages are far more popular since the introduction of Java but are generally considered inappropriate for systems code, which is usually written in a low-level, glorified assembly language like C or its more sophisticated cousin C++. The common belief is that safe languages are inefficient, due in part to the size and complexity of their runtime systems and reliance on garbage collection.

Singularity's Bartok compiler provides language safety without the typical performance penalty by compiling C#'s Microsoft Intermediate Language representation to native (x86, x64, or ARM) code at installation time rather than at runtime. Bartok also links compiled code to a small runtime consisting of only a class library and a garbage collector, not a large runtime environment like the Common Language Runtime (the virtual machine component of Microsoft .NET) and the Java Virtual Machine. The Shared Source Common Language Infrastructure runtime and class library are more than five times larger than the Bartok runtime (64 thousand lines of code, or KLOC, vs. 350KLOC), roughly the same as the C runtime in the latest version of Windows (72KLOC). Moreover, Bartok is a highly optimizing compiler that generates high-quality code and reduces memory use through extensive tree shaking to discard unneeded class variables and method definitions.

Table 1 emphasizes this point by outlining the memory footprint for a small program written in C, C++, and C# running on several different operating systems. The program outputs "Hello World" using the standard I/O libraries and APIs for each system—printf for C and C++ and `Console.WriteLine` for C#. The C# code on Singularity is smaller than for all but one other system—the statistically linked code on Free BSD—in some cases half to one-third the size of C++ code. Table 2 outlines the reduction in memory footprint Bartok achieves for a variety of programs. Much of the code and data "shaken" out of these programs comes from the unused portions of general-purpose libraries.

Language safety is another foundation of Singularity's SIPs, which consist of memory pages holding the objects a process can access (see Figure 2). Singularity enforces the invariant that a reference manipulated by process P1 cannot point to a page belonging to process P2, where P1 ≠ P2. A process might try to violate this invariant in two ways:

Create a new reference or modify an existing reference to point to another process's page. Language safety guarantees that code running on Singularity cannot perform either of these operations; and

Pass a reference to another process's page. This operation is prevented by Sing#'s type system for inter-process communication.

Other systems, including Cedar/Mesa, Lisp Machines, and Java, were written in higher-level languages and depend on language safety to isolate different computations running in the same address space. While the SPIN operating system uses traditional page-based hardware protection between processes, it also depends on language safety to isolate OS extensions running in the kernel's address space.[4] Singularity's approach differs in that it isolates a process's objects by memory pages, rather than allocating them in a common address space. When a process terminates, Singularity quickly reclaims the process's memory pages, rather than turning to garbage collection to reclaim memory. Beyond the performance benefits of improved memory locality and a simplified garbage collector, the isolation invariant is far easier for the operating system to

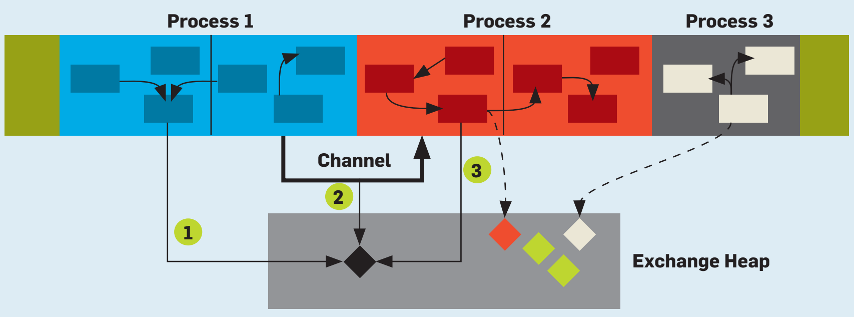
Table 1. Memory footprint for "Hello World" process (in kilobytes).

	Singularity	FreeBSD 5.3	Linux 2.6.11 (Red Hat FC4)	Windows XP (SP2)
C - static lib	—	232KB	664KB	544KB
C++ - static lib	—	704KB	1,216KB	572KB
C# - w/ GC	408KB	—	—	3,750KB

Table 2. Memory-footprint reduction due to tree shaking.

	Code (Total)	Code (Tree Shake)	% Reduction
Singularity Kernel	2,371 KB	1,291 KB	46%
Web Server	2,731 KB	765 KB	72%
SPECweb99 Plug-in	2,144 KB	502 KB	77%
IDE Disk Driver	1,846 KB	455 KB	75%

Figure 2. Singularity process objects reside on a dedicated collection of pages.



enforce at a process level, rather than word level.

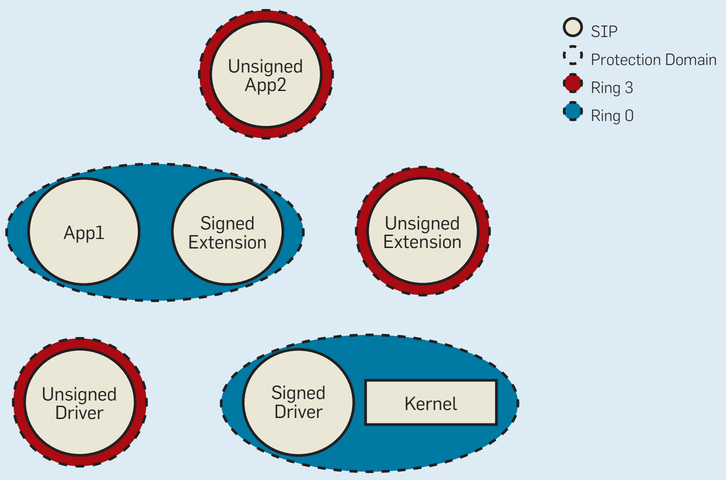
Singularity also provides flexible hardware-based process isolation as a secondary mechanism. A Singularity hardware-protection domain is an address space holding one or more SIPs. Domains can run in either user or kernel mode (ring 3 and ring 0 on an x86 processor). At runtime, the system-configuration manifest specifies which SIPs reside in which domains. Domains allow untrusted code to be isolated behind conventional hardware-protection mechanisms while more trusted code resides in the same address space, benefiting from faster communications and failure isolation (see Figure 3).

Domains also enable Singularity developers to run a series of experiments comparing the execution overheads of software and hardware isolation.[1] The basic cost of software isolation is the runtime checks for null pointers and array accesses (4.7% of CPU cycles). By contrast, hardware isolation similar to conventional operating systems (separate address spaces and protection domains) incurred a cost of up to 38% of CPU cycles (see Figure 4).

Table 3. Basic cost (in CPU cycles) of common operations between isolated processes on an AMD Athlon 64 3000+ system.

	Singularity	FreeBSD 5.3	Linux 2.6.11 (Red Hat FC4)	Windows XP (SP2)
Process create and start	353,000	1,030,000	719,000	5,380,000
Minimum kernel API call	91	878	437	627
Thread context switch	346	911	906	753
Message request/reply	803	13,300	5,800	6,340

Figure 3. Hybrid hardware-software isolation using SIPs and domains.



Modular System Architecture

Unlike many systems, Singularity assumes that software contains bugs and consequently is likely to fail occasionally. Singularity's architecture aims to contain the consequence of a failure within a fault-isolation boundary, thereby allowing the system to detect the failure and recover by restarting the failed component. Although less intellectually appealing than flawless operation, most complex artifacts share this paradigm and most programmers are comfortable with it; for example, a car does not stop running when a headlight burns out or a tire goes flat.

Tight coupling between components in monolithic software systems routinely means the failure of one component can bring down an application and, in the worst case, the system itself. The epitome of this problem is the common plug-in software architecture that allows extensions to be dynamically loaded into a host's address space. Plug-ins (such as device drivers, browser extensions, and spell checkers) share their host process's address space and have unconstrained access to its code and data structures. An extension's failure typically causes the host to fail as well. Considerable evidence shows that extensions are less reliable than host code; for example, Orgovan and Tricker reported[11] that approximately 85% of the Windows XP kernel crashes they studied is caused by device drivers, and Chou et al. reported that the Linux drivers they studied have up to seven times the bug density of other kernel code.[5]

Plug-in architectures also involve other disadvantages: First, code extensions can subvert modularity and engineering discipline. A plug-in can use any data structure or procedure it can discover. Most of a host's functionality may be private or inappropriate for plug-ins, but the host has no way to prevent its use, except, perhaps, by hiding names and documentation. Moreover, a plug-in that uses undocumented functionality can frustrate backward compatibility as a system evolves. Unless the system formally specifies the interface between a plug-in and its host, seemingly unrelated changes to the host can affect the plug-in and produce many failures despite extensive testing regimes.

The Singularity architecture avoids many of these problems. For example, SIPs are sealed processes that prohibit shared memory, in-process code generation, and dynamic code loading. A process that wishes to invoke an extension starts the extension code running in a separate SIP. If the extension fails, its process terminates, but the parent process continues and can recover from the error. Moreover, the extension is limited to the functionality explicitly provided by the parent process. This recovery is feasible in many cases because of three built-in Singularity design decisions:

SIPs are inexpensive.[10] The cost of creating a SIP and communicating between two SIPs is low in terms of CPU cycles, thus reducing the overhead of this isolation mechanism and allowing it to be used at finer granularity than a conventional process. The high cost of processes on other systems encourages monolithic software architectures and plug-ins to extend system behavior. On Singularity, programmers are able to encapsulate small extensions to existing applications or to the system itself in their own separate SIPs. Table 3 summarizes the cost in terms of CPU cycles of a variety of systems for creating a process and communicating with the kernel and another process. These operations are far less costly on Singularity;

SIPs do not share memory. Data structures shared between two processes provide a simple, high-bandwidth communication mechanism requiring little forethought on the part of the host. However, when a process fails, the shared structure couples the failure to the other process, supporting the conservative assumption that the first process left the shared structure in an inconsistent state.[7] Shared memory further opens each process to spontaneous corruption of shared state at any time by an errant or malicious peer. By forbidding shared memory, Singularity ensures that process state is altered by only one process at a time; and

Communication between SIPs passes through strongly typed channels.[6] A channel is a pair of bounded message queues between two SIPs. A message is a structure consisting of scalar types (such as integers, float, and strings), arrays of structures, and pointers to other structures sent in the same send operation. Messages are allocated in a special area of memory—the Exchange Heap—with programs accessing it through a special Sing# type system that permits at most one outstanding reference to a data structure. When a SIP sends a message across a channel, it relinquishes ownership of the message and can no longer access it (see Figure 5). This semantic prevents SIPs from sharing the memory in a message while allowing for efficient communications, as code cannot distinguish communication in which a message is copied from communication in which a pointer to the message is passed among the SIPs. The receiving SIP should still validate message parameters but need not worry about their asynchronous modifications.

Each channel is annotated with a specification, or "contract," of the content of each message and the allowable sequence of messages. For example, the following code is part of the contract for a channel to Singularity's TCP service, defining the legal messages that can arrive at the service when a socket is connected:

```
public contract TcpSocketCon-
tract {
...
state Connected : {
   Read? -> ReadResultPending;
   Write? -> WriteResultPending;
   GetLocalAddress?           ->
   IPAddress! -> Connected;
   GetLocalPort?  ->  Port!  ->
   Connected;
   DoneSending? -> ReceiveOnly;
   DoneReceiving? -> SendOnly;
   Close? -> Closed;
   Abort? -> Closed;
}
state ReadResultPending : {
   Data! -> Connected;
   NoMoreData! -> SendOnly;
   RemoteClose! -> Zombie;
...
}
```

If, for example, the service receives a `Read` message from a client, the contract transitions to the `ReadResultPending` state, where the service is expected to respond with a packet of data or a status or error indication. Singularity's compiler statically checks the code that sends and receives messages on a channel, ensuring it obeys the contract.

One objection to SIPs and channels is they make writing software more difficult than shared data structures and procedural APIs. Channel contracts clearly require forethought for designing and specifying an interface, which is a good thing. In practice,

Figure 4. Normalized execution time comparing the overhead cost of software and hardware process isolation mechanisms for a Web server running on Singularity. Our experiments ran on a 1.8GHz AMD Athlon 64 3000+ system, starting with a pure software-isolated version of Singularity, progressively adding hardware address-space protection.

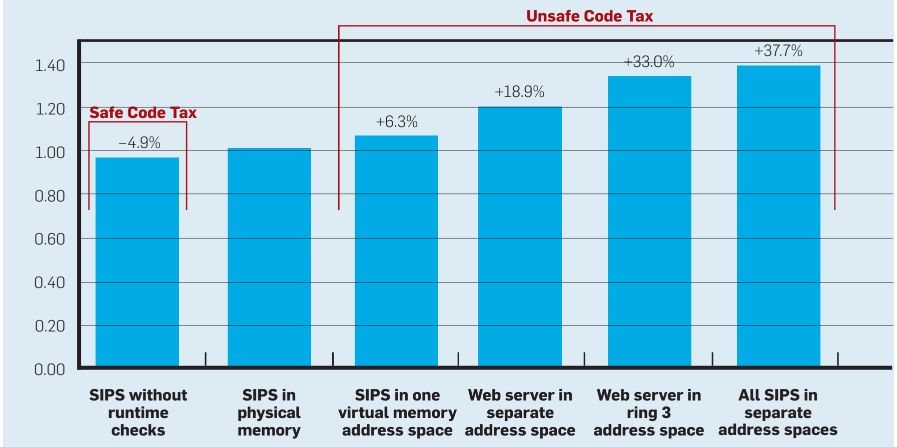
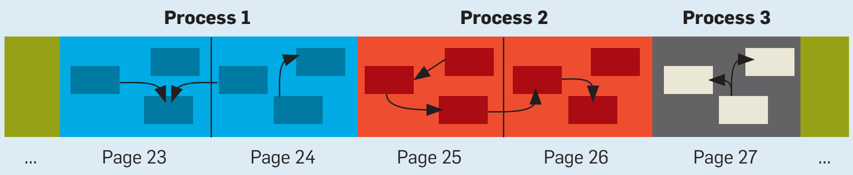


Figure 5. Message exchange across a channel; message ownership passes from Process 1 through a channel to Process 2.



programming language support for communications, explicit contracts, and compiler checking reduces the burden of this style of development. As an experiment, we removed one of the Bartok compiler's most complex components—its register allocator—and ran it in a separate SIP. It shared code for 156 classes with the rest of the compiler, running every time a function is compiled. Because its interface originated in a shared address space, it passes a large amount of data—50KB–1.5MB—at every invocation, much of which is the same across allocator invocations (such as the machine description). Nevertheless, we were able to run the allocator in a separate SIP by changing 508 lines of code (0.25% of the compiler), and the modified compiler ran only 11% slower while compiling the Singularity kernel. Designing the interface to the allocator appropriately in the first place could reduce the communications cost and overhead penalty. Still, the experiment shows the practicality of partitioning even a complex interface so it works across channels.

Self-Describing Systems

For the past 10 years, software-development tools based on formal methods have become increasingly sophisticated and available[8] for comparing a specification of the intended behavior of a system component against the component's actual code, pointing out discrepancies between the behaviors. Such tools, including SLAM[2] and Boogie,[3] generally check the behavior of procedure and method boundaries. While the proper use of these interfaces is central to writing correct software—and strongly supported by Singularity, including language support for the Boogie verification system—systems provide many other abstractions. The correctness of a system depends on them, as well as on low-level interfaces.

Singularity follows this paradigm of specification and checking at many different levels of system structure, including for purposes other than static-defect detection. Channel contracts, described earlier, capture the behavior of Singularity's primary communication mechanism. Another example of high-level specification is device-driver manifests.[12] A Singularity device driver specifies the underlying hardware resources (such as memory mapped I/O registers) it can access.

Unlike in other systems, processes in Singularity are software-isolated processes, or SIPs, that rely on language safety, not hardware mechanisms, to isolate system software components from one another.

Depending on hardware support, Singularity may corroborate only a subset of this information, but it uses the declared information in the following ways to ensure correct system configuration:

Look for conflicting claims. When a driver is loaded, Singularity looks for conflicting claims on hardware resources. If a new driver uses the same I/O registers as an existing driver, then Singularity avoids a conflict by refusing to load the new driver; and

Incorporate declared resources. If the system detects no conflicts, then Singularity incorporates its declared resources into the system manifest used to configure the boot process. When starting up, the Singularity kernel starts each device driver in its own SIP. It also creates in-process I/O objects for accessing the I/O registers and interrupt lines used by the driver. These pre-populated I/O objects simplify driver access to hardware while simultaneously providing low-cost access to hardware resources with language safety.

Singularity demonstrates that lightweight specifications are valuable if closely connected to the underlying system and offers a value greater than the additional burden they impose. Specifications may be closely tied to the actual code. Documentation grows stale in the absence of systematic tools to detect discrepancies between a description and the related code. On the other hand, specifications that drive tools remain closely linked to code and must meet only the lower bar of providing sufficient utility to justify learning a new language and unfamiliar tools.

Discussion

The Singularity project is first and foremost an experiment in building from scratch a nontrivial system (approximately 250KLOC) using a safe language. Much of what we have learned may be of value in other systems, and many ideas have been transferred into Microsoft products. Benefits include SIPs for encapsulating program components, configuration of system components by manifest, and a lightweight, compiled runtime system for safe code. Like any system, Singularity also has its rough spots, and future research should aim to help resolve three troubling issues: the garbage collector in the kernel; the inconsistencies between Sing#'s two type systems; and C#'s incomplete type system.

Despite early concern in the project and ongoing external skepticism, our experience shows that high-performance system software can be built in a garbage-collected language. Singularity performed much better on basic micro and macro benchmarks than we originally anticipated, and when failing to perform well, problems were seldom attributable solely to garbage collection. Our experience confirms wisdom in the Java and Common Language Runtime communities that garbage collection obviates the need for strict memory accounting but does not eliminate the need for carefully managing memory in high-performance code.

The design of an optimal garbage collector for an OS kernel is an open question. The assumptions underlying generational collectors do not agree with the lifetime of many kernel objects that persist as long as the system or process exists. Reference counting, despite trade-offs involving cost and the inability to reclaim cyclic structures, is common in conventional operating systems and deserves reexamination as a garbage-collection technique for safe kernels.

Sing#, the language of Singularity, supports two type systems: C# and data passed between processes. Data in a process is conventional C# objects, but data passed along channels lives in a distinct type system, limited to structs, not objects, and is governed by strict rules restricting references. This system allows static verification of channel contracts but exacts a price in programmer frustration and additional code for marshalling, unmarshalling, and operations on the structs. Increased interoperability or, better, a unified type system would simplify the code for creating and manipulating messages. In addition, the channel contracts we used were not expressive enough to describe asynchronous interactions between processes.

Finally, C#, like many modern languages, does not provide convenient mechanisms for manipulating bit-level formatted data and inlined arrays found in device-control registers and network packets. Not adding this functionality to Sing# early in the Singularity-development project was an omission that continues to incur a penalty.

Conclusion

Singularity is a small operating system we and a group of our colleagues at Microsoft Research built to demonstrate a nontrivial change in the standard practice of designing and constructing software. On today's fast computers, it is no longer necessary to design systems around the lowest common denominator of assembly language or C, seeking performance to the detriment of essential system attributes (such as modularity and reliability). Singularity shows that modern, safe programming languages enable new system architectures that not only improve robustness but perform better in many circumstances than traditional approaches.

The lessons of Singularity are applicable far beyond the ground-up design of new systems; for example, manifests could be used in more traditional operating systems to describe dependencies, cross-process communication, and hardware access. Likewise, replacing in-process plug-ins with components in separate processes would improve the resilience of any system. Gradually incorporating safe languages, software isolation, and increased specification into existing systems offers cost-effective incremental improvement.

Source code for the Singularity system is available for noncommercial use at http://www.codeplex.com/singularity.

Acknowledgments

Singularity was the work of large team of dedicated individuals: David Tarditi, Bjarne Steensgaard, Qunyan Mangus, Mark Plesko, and Juan Chen built the Bartok compiler and runtime. Manuel Fähndrich, Songtao Xia, Sriram Rajamani, Jakob Rehof, Herman Venter, Rebecca Isaacs, and Tim Harris worked on Sing# and tools. Orion Hodson, Chris Hawblitzel, Steven Levi, Nick Murphy, Mark Aiken, Derrick Coetzee, Ed Nightingale, Brian Zill, and Richard Black built portions of the operating system. Ted Wobber, Martin Abadi, Andrew Birrell, Ulfar Erlingsson, and Dan Simon developed the security architecture. In addition, more than 30 interns contributed heart, mind, and hands to the project. C

References

1. Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G., and Larus, J.R. Deconstructing process isolation. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (San Jose, CA, Oct.). ACM Press, New York, 2006, 1–10.
2. Ball, T. and Rajamani, S.K. The SLAM toolkit. In *Proceedings of the 13th Conference on Computer-Aided Verification* (Paris, July). Springer, 2001, 260–264.
3. Barnett, M., Change, B.-y.E., Deline, R., Jacobs, B., and Leino, K.R. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects* (Amsterdam, The Netherlands, Nov.). Springer, 2005, 364–387.
4. 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 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO, Dec.). ACM Press, New York, 1995, 267–284.
5. 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* (Chateau Lake Louise, Banff, Canada, Oct.). ACM Press, New York, 2001, 73–88.
6. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., and Levi, S. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the First ACM SIGOPS/EuroSys European Conference on Computer Systems* (Leuven, Belgium, Apr.). ACM Press, New York, 2006, 177–190.
7. Flatt, M. and Findler, R.B. Kill-safe synchronization abstractions. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Washington, D.C., June). ACM Press, New York, 2004, 47–58.
8. Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J.P., and Margaria, T. Software engineering and formal methods. *Commun. ACM 51*, 9 (Sept. 2008), 54–59.
9. Hunt, G. and Larus, J. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review 41*, 2 (Apr. 2007), 37–49.
10. Hunt, G., Aiken, M., Fähndrich, M., Hawblitzel, C., Hodson, O., Larus, J., Levi, S., Steensgaard, B., Tarditi, D., and Wobber, T. Sealing OS processes to improve dependability and safety. In *Proceedings of the Second ACM SIGOPS/EuroSys European Conference on Computer Systems* (Lisbon, Portugal, Mar.). ACM Press, New York, 2007, 341–354.
11. Orgovan, V. and Tricker, M. *An Introduction to Driver Quality.* Microsoft WinHEC 2004 presentation DDT301, New Orleans, LA, 2003.
12. Spear, M.F., Roeder, T., Levi, S., and Hunt, G. Solving the starting problem: Device drivers as self-describing artifacts. In *Proceedings of the EuroSys 2006 Conference* (Leuven, Belgium, Apr.). ACM Press, New York, 2006, 45–58.

James Larus (larus@microsoft.com) is director of Research and Strategy in the eXtreme Computing Group at Microsoft Research, Redmond, WA.

Galen Hunt (galenh@microsoft.com) is principal researcher in the Microsoft Research Operating Systems Group and leads the Menlo project and the Singularity project at Microsoft Research, Redmond, WA.