Singularity: Rethinking the Software Stack

Galen C. Hunt and James R. Larus
Microsoft Research Redmond

galenh@microsoft.com

ABSTRACT
Every operating system embodies a collection of design decisions. Many of the decisions behind today's most popular operating systems have remained unchanged, even as hardware and software have evolved. Operating systems form the foundation of almost every software stack, so inadequacies in present systems have a pervasive impact. This paper describes the efforts of the Singularity project to re-examine these design choices in light of advances in programming languages and verification tools. Singularity systems incorporate three key architectural features: software-isolated processes for protection of programs and system services, contract-based channels for communication, and manifest-based programs for verification of system properties. We describe this foundation in detail and sketch the ongoing research in experimental systems that build upon it.

Keywords
Operating systems, safe programming languages, program verification, program specification, sealed process architecture, sealed kernel, software-isolated processes (SIPs), hardware protection domains, manifest-based programs (MBPs), unsafe code tax.

1. INTRODUCTION
Every operating system embodies a collection of design decisions—some explicit, some implicit. These decisions include the choice of implementation language, the program protection model, the security model, the system abstractions, and many others.

Contemporary operating systems—Windows, Linux, Mac OS X, and BSD—share a large number of design decisions. This commonality is not entirely accidental, as these systems are all rooted in OS architectures and development tools of the late 1960's and early 1970's. Given the common operating environments, the same programming language, and similar user expectations, it is not surprising that designers of these systems made similar decisions. While some design decisions have withstood the test of time, others have aged less gracefully.

The Singularity project started in 2003 to re-examine the design decisions and increasingly obvious shortcomings of existing systems and software stacks. These shortcomings include: wide-spread security vulnerabilities; unexpected interactions among applications; failures caused by errant extensions, plug-ins, and drivers, and a perceived lack of robustness.

We believe that many of these problems are attributable to systems that have not evolved far beyond the computer architectures and programming languages of the 1960's and 1970's. The computing environment of that period was very different from today. Computers were extremely limited in speed and memory capacity. They were used only by a small group of benign technical literati and were rarely networked or connected to physical devices. None of these requirements still hold, but modern operating systems have not evolved to accommodate the enormous shift in how computers are used.

1.1 A Journey, not a Destination
In the Singularity project, we have built a new operating system, a new programming language, and new software verification tools. The Singularity operating system incorporates a new software architecture based on software isolation of processes. Our programming language, Sing# [8], is an extension of C# that provides verifiable, first-class support for OS communication primitives as well as strong support for systems programming and code factoring. The sound verification tools detect programmer errors early in the development cycle.

From the beginning, Singularity has been driven by the following question: what would a software platform look like if it was designed from scratch, with the primary goal of improved dependability and trustworthiness? To this end, we have championed three strategies. First, the pervasive use of safe programming languages eliminates many preventable defects, such as buffer overruns. Second, the use of sound program verification tools further guarantees that entire classes of programmer errors are removed from the system early in the development cycle. Third, an improved system architecture stops the propagation of runtime errors at well-defined boundaries, making it easier to achieve robust and correct system behavior. Although dependability is difficult to measure in a research prototype, our experience has convinced us of the practicality of new technologies and design decisions, which we believe will lead to more robust and dependable systems in the future.

Singularity is a laboratory for experimentation in new design ideas, not a design solution. While we like to think our current code base represents a significant step forward from prior work, we do not see it as an "ideal" system or an end in itself. A research prototype such as Singularity is intentionally a work in progress; it is a laboratory in which we continue to explore implementations and trade-offs.

In the remainder of this paper, we describe the common architectural foundation shared by all Singularity systems. Section 3 describes the implementation of the Singularity kernel which provides the base implementation of that foundation. Section 4 surveys our work over the last three years within the Singularity project to explore new opportunities in the OS and system design space. Finally, in Section 5, we summarize our work to date and discuss areas of future work.

2. ARCHITECTURAL FOUNDATION
The Singularity system consists of three key architectural features: software-isolated processes, contract-based channels, and manifest-based programs. Software-isolated processes provide an environment for program execution protected from external interference. Contract-based channels enable fast, verifiable message-based communication between processes. Manifest-

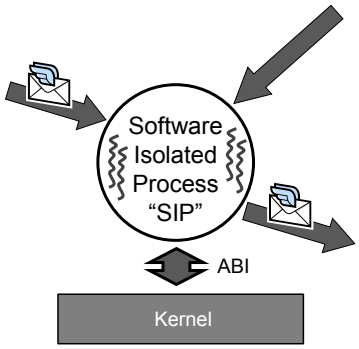

Figure 1. Architectural features of a Software-Isolated Process (SIP) including threads, channels, messages, and an ABI for the kernel.

based programs define the code that runs within software-isolated processes and specify its verifiable behavioral properties.

A guiding philosophy behind Singularity is one of simplicity over richness. The foundational features respectively provide basic support for execution, communication, and system verification and are a bedrock design upon which dependable, verifiable systems can be built.

2.1 Software-Isolated Processes

The first foundational feature common to Singularity systems is the *Software-Isolated Process* (*SIP*). Like processes in many operating systems, a SIP is a holder of processing resources and provides the context for program execution. However, unlike traditional processes, SIPs take advantage of the type and memory safety of modern programming languages to dramatically reduce the cost of isolating safe code.

Figure 1 illustrates the architectural features of a SIP. SIPs share many properties with processes in other operating systems. Execution of each user program occurs within the context of a SIP. Associated with a SIP is a set of memory pages containing code and data. A SIP contains one or more threads of execution. A SIP executes with a security identity and has associated OS security attributes. Finally, SIPs provide information hiding and failure isolation.

Some aspects of SIPs have been explored in previous operating systems, but with less rigor than in Singularity. SIPs do not share data, so all communications occurs through the exchange of *messages* over message-passing conduits called *channels*. Singularity adds the rigor of statically verifiable *contracts.* A contract specifies the messages and protocol for communication across all channels of a given type. SIPs access primitive functions, such as those that send and receive messages, through an Application Binary Interface (ABI) to the kernel. The Singularity ABI has a rigorous design that includes fully declarative versioning information. It provides secure local access to the most primitive aspects of computation—memory, execution, and communication—and excludes semantically ambiguous constructs such as `ioctl`.

SIPs differ from conventional operating system processes in other ways. They cannot share writable memory with other SIPs; the code within a SIP is sealed at execution time; and SIPs are isolated by software verification, not hardware protection.

	Cost (in CPU Cycles)			
	API Call	Thread Yield	Message Ping/Pong	Process Creation
Singularity	80	365	1,040	388,000
FreeBSD	878	911	13,300	1,030,000
Linux	437	906	5,800	719,000
Windows	627	753	6,340	5,380,000

Table 1. Basic process costs on AMD Athlon 64 3000+ (1.8 GHz) CPU with an NVIDIA nForce4 Ultra chipset.

In other words, SIPs are closed object spaces. The objects in one SIP may not be modified or directly accessed by any other SIP. Communication between SIPs occurs by transferring the exclusive ownership of data in messages. A linear type system and a special area of memory known as the *exchange heap* allows lightweight exchange of even very large amounts of data, but no sharing.

As a consequence, SIPs execute autonomously: each SIP has its own data layouts, run-time system, and garbage collector.

SIPs are sealed code spaces. As a sealed process, a SIP cannot dynamically load or generate code into itself. Sealed processes compel a common extension mechanism for both the OS and applications: extension code executes in a new process, distinct from its host's SIP. Sealed processes offer a number of advantages. They increase the ability of program analysis tools to detect defects statically. They enable stronger security mechanisms, such as identifying a process by its code content. They can also eliminate the need to duplicate OS-style access control in the runtime execution environments. A recent Singularity paper [14] provides a thorough analysis of the trade-offs and benefits of sealed processes.

SIPs rely on programming language type and memory safety for isolation, instead of memory management hardware. Through a combination of static verification and runtime checks, Singularity verifies that user code in a SIP cannot access memory regions outside the SIP. With process isolation provided by software rather than hardware, multiple SIPs can reside in a single physical or virtual address space. In the simplest Singularity systems, the kernel and all SIPs share a single kernel-mode address space. As will be discussed in Section 4.2, richer systems can be built by layering SIPs into multiple address spaces at both user and kernel protection levels. Aiken *et al.* [1] evaluate the trade-offs between software and hardware isolation.

Communication between SIPs incurs low overhead and SIPs are inexpensive to create and destroy, as compared to hardware protected processes (see Table 1). Unlike a hardware protected process, a SIP can be created without creating page tables or flushing TLBs. Context switches between SIPs also have very low overhead as TLBs and virtually addressed caches need not be flushed. On termination, a SIP's resources can be efficiently reclaimed and its memory pages recycled without involving garbage collection.

Low cost makes it practical to use SIPs as a fine-grain isolation and extension mechanism to replace the conventional mechanisms of hardware protected processes and unprotected dynamic code loading. As a consequence, Singularity needs only one error recovery model, one communication mechanism, one security architecture, and one programming model, rather than the layers of partially redundant mechanisms and policies found in current systems.

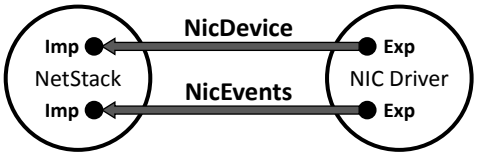

Figure 2. Channels between a network driver and stack.

A key experiment in the Singularity project is to construct an entire operating system using SIPs and demonstrate that the resulting system is more dependable than a conventional system. The results so far are promising. SIPs are cheap enough to fit a "natural" software development granularity of one developer or team per SIP and light-weight enough to provide fault-stop boundaries for aberrant behavior.

2.2 Contract-Based Channels

All communication between SIPs in Singularity flows through *contract-based channels*. A channel is a bi-directional message conduit with exactly two endpoints. A channel provides a lossless, in-order message queue. Semantically, each endpoint has a receive queue. Sending on an endpoint enqueues a message on the other endpoint's receive queue. A channel endpoint belongs to exactly one thread at a time. Only the endpoint's owning thread can dequeue messages from its receive queue or send messages to its peer.

Communication across a channel is described by a channel contract. The two ends of a channel are not symmetric in a contract. One endpoint is the importing end (Imp) and the other is the exporting end (Exp). In the Sing# language, the endpoints are distinguished by types `C.Imp` and `C.Exp`, respectively, where `C` is the channel contract governing the interaction.

Channel contracts are declared in the Sing# language. A contract consists of message declarations and a set of named protocol states. Message declarations state the number and types of arguments for each message and an optional message direction. Each state specifies the possible message sequences leading to other states in the state machine.

We will explain channel contracts through a condensed version of the contract for network device drivers shown in Listing 1. A channel contract is written from the perspective of the SIP exporting a service and starts in the first listed state. Message sequences consist of a message tag and a message direction sign (`!` for Exp to Imp), and (`?` for Imp to Exp). The message direction signs are not strictly necessary if message declarations already contain a direction (`in`, `out`), but the signs make the state machine more human-readable.

In our example, the first state is `START` and the network device driver starts the conversation by sending the client (probably the network stack) information about the device via message `DeviceInfo`. After that the conversation is in the `IO_CONFIGURE_BEGIN` state, where the client must send message `RegisterForEvents` to register another channel for receiving events and set various parameters in order to get the conversation into the `IO_CONFIGURED` state. If something goes wrong during the parameter setting, the driver can force the client to start the configuration again by sending message `InvalidParameters`. Once the conversation is in the `IO_CONFIGURED` state, the client can either start I/O (by sending `StartIO`), or reconfigure the driver (`ConfigureIO`). If I/O is started, the conversation is in

```
contract NicDevice {
  out message DeviceInfo(...);
  in  message RegisterForEvents(NicEvents.Exp:READY c);
  in  message SetParameters(...);
  out message InvalidParameters(...);
  out message Success();
  in  message StartIO();
  in  message ConfigureIO();
  in  message PacketForReceive(byte[] in ExHeap p);
  out message BadPacketSize(byte[] in ExHeap p, int m);
  in  message GetReceivedPacket();
  out message ReceivedPacket(Packet * in ExHeap p);
  out message NoPacket();

  state START: one {
    DeviceInfo! → IO_CONFIGURE_BEGIN;
  }
  state IO_CONFIGURE_BEGIN: one {
    RegisterForEvents? →
      SetParameters? → IO_CONFIGURE_ACK;
  }
  state IO_CONFIGURE_ACK: one {
    InvalidParameters! → IO_CONFIGURE_BEGIN;
    Success! → IO_CONFIGURED;
  }
  state IO_CONFIGURED: one {
    StartIO? → IO_RUNNING;
    ConfigureIO? → IO_CONFIGURE_BEGIN;
  }
  state IO_RUNNING: one {
    PacketForReceive? → (Success! or BadPacketSize!)
      → IO_RUNNING;
    GetReceivedPacket? → (ReceivedPacket! or NoPacket!)
      → IO_RUNNING;
    ...
  }
}
```

Listing 1. Contract to access a network device driver.

```
contract NicEvents {
  enum NicEventType {
    NoEvent, ReceiveEvent, TransmitEvent, LinkEvent
  }

  out message NicEvent(NicEventType e);
  in message AckEvent();

  state READY: one {
    NicEvent! → AckEvent? !READY;
  }
}
```

Listing 2. Contract for network device events.

state `IO_RUNNING`. In state `IO_RUNNING`, the client provides the driver with packet buffers to be used for incoming packets (message `PacketForReceive`). The driver may respond with `BadPacketSize`, returning the buffer and indicating the size expected. This way, the client can provide the driver with a number of buffers for incoming packets. The client can ask for packets with received data through message `GetReceived-Packet` and the driver either returns such a packet via `ReceivedPacket` or states that there are no more packets with data (`NoPacket`). Similar message sequences are present for transmitting packets, but we elide them in the example.

From the channel contract it appears that the client polls the driver to retrieve packets. However, we have not yet explained the argument of message `RegisterForEvents`. The argument of type `NicEvents.Exp:READY` describes an Exp channel endpoint of contract `NicEvents` in state `READY`. This endpoint argument establishes a second channel between the client and the network driver over which the driver notifies the client of important events (such as the beginning of a burst of packet arrivals). The client retrieves packets when it is ready through the `NicDevice` channel. Figure 2 shows the configuration of channels between the client and the network driver. The `NicEvents` contract appears in Listing 2.

Channels enable efficient and analyzable communication between SIPs. When combined with support for linear types, Singularity allows zero-copy exchange of large amounts of data between SIPs access channels [8]. In addition, the Sing# compiler can statically verify that send and receive operations on channels never are applied in the wrong protocol state. A separate contract verifier can read a program's byte code and statically verify which contracts are used within a program and that the code conforms to the state machine described in the contract's protocol.

Experience has demonstrated that channel contracts are valuable tool for preventing and detecting mistakes. Programmers on the team were initially skeptical of the value of contracts. The contract conformance verifier was not completed until almost a year after our first implementation of the network stack and web server. When the verifier came online, it immediately flagged an error in the interaction between network stack and web server. The error occurred where the programmer failed to anticipate a lack of data on an incoming socket, as expressed by a `NO_DATA` message. The bug existed in the web server for almost a year. Within seconds, the verifier flagged the error and identified the exact circumstances under which the bug would be triggered.

Channel contracts provide a clean separation of concerns between interacting components and help in understanding the system architecture at a high level. Static checking helps programmers avoid runtime "message not-understood errors." Furthermore, the runtime semantics for channels restricts failure to be observed on receives only, thus eliminating handling of error conditions at send operations where it is inconvenient.

2.3 Manifest-Based Programs

The third foundational architecture feature common to Singularity systems is the *Manifest-Based Program* (MBP). A MBP is a program defined by a static *manifest*. No code is allowed to run on a Singularity system without a manifest. In fact, to start execution, a user invokes a manifest, not an executable file as in other systems.

A manifest describes an MBP's code resources, its required system resources, its desired capabilities, and its dependencies on other programs. When an MBP is installed on a system, the manifest is used to identify and verify that the MBP's code meets all required safety properties, to ensure that all of the MBP's system dependencies can be met, and to prevent the installation of the MBP from interfering with the execution of any previously installed MBPs. Before execution, the manifest is used to discover the configuration parameters affecting the MBP and restrictions on those configuration parameters. When an MBP is invoked, the manifest guides the placement of code into a SIP for execution, the connection of channels between the new SIP and other SIPs, and the granting of access to system resources by the SIP.

A manifest is more than just a description of a program or an inventory of a SIP's code content; it is a machine-checkable, declarative expression of the MBP's expected behavior. The primary purpose of the manifest is to allow static and dynamic verification of properties of the MBP. For example, the manifest of a device driver provides sufficient information to allow install-time verification that the driver will not access hardware used by a previously installed device driver. Additional MBP properties which are verified by Singularity include type and memory safety, absence of privileged-mode instructions, conformance to channel contracts, usage of only declared channel contracts, and correctly-versioned ABI usage.

Code for an MBP can be included as an inline element of the manifest or be provided in separate files. Interpreted scripting languages, such as the Singularity shell language, can easily be included as inline elements of a manifest. On the other hand, large compiled applications may consist of numerous binaries, some unique to the MBP and some shared with other MBPs and stored in a repository separately from the MBP's manifest.

Singularity's common MBP manifest can be extended either inline or with metadata in other files to allow verification of sophisticated properties. For example, a basic manifest is insufficient to verify that a MBP is type safe or that it uses only a specific subset of channel contracts. Verification of the safety of compiled code requires additional metadata in MBP binary files.

To facilitate static verification of as many run-time properties as possible, code for Singularity MBPs is delivered to the system as compiled Microsoft Intermediate Language (MSIL) binaries. MSIL is the CPU-independent instruction set accepted by the Microsoft Common Language Runtime (CLR) [7]. Singularity uses the standard MSIL format with features specific to Singularity expressed through MSIL metadata extensions. With a few exceptions, the OS compiles MSIL into the native instruction set at install time. Replacing JIT compilation with install-time compilation is facilitated by the manifest, which declares all of the executable MSIL code for SIPs created from the MBP.

Every component in Singularity is described by a manifest, including the kernel, device drivers, and user applications. We have demonstrated that MBPs are especially helpful in creating "self-describing" device drivers with verifiable hardware access properties [25]. Singularity systems use manifest features to move command-line processing out of almost all applications and to centralize it in the shell program. Manifests guide install-time compilation of MBPs. We believe that MBPs will play a role in significantly reducing the costs of system administration, but we have not yet completed the experimental work to validate this hypothesis.

3. SINGULARITY KERNEL

Supporting the architectural foundation of Singularity is the Singularity kernel. The kernel provides the base abstractions of software-isolated processes, contract-based channels, and manifest-based programs. The kernel performs the crucial role of dividing systems resources among competing programs and abstracting the complexities of system hardware. To each SIP, the kernel provides a pure execution environment with threads, memory, and access to other MBPs via channels.

Like the previous Cedar [26] and Spin [4] projects, the Singularity project enjoys the safety and productivity benefits of writing a kernel in a type-safe, garbage-collected language. Counting lines of code, over 90% of the Singularity kernel is written in Sing#. While most of the kernel is type-safe Sing#, a significant portion of the kernel code is written in the unsafe variant of the language. The most significant unsafe code is the garbage collector, which accounts for 48% of the unsafe code in Singularity. Other major sources of unsafe Sing# code include the memory management and I/O access subsystems. Singularity includes small pockets of assembly language code in the same places it would be used in a kernel written in C or C++, for example, the thread context switch, interrupt vectors, etc. Approximately 6% of the Singularity kernel is written in C++, consisting primarily of the kernel debugger and low-level system initialization code.

The Singularity kernel is a microkernel; all device drivers, network protocols, file systems, and user replaceable services execute in SIPs outside the kernel. Functionality that remains in the kernel includes scheduling, mediating privileged access to hardware resources, managing system memory, managing threads and thread stacks, and creating and destroying SIPs and channels. The following subsections describe details of the kernel implementation.

3.1 ABI

SIPs access primitive kernel facilities, such as the ability to send and receive messages, through the Singularity Application Binary Interface (ABI). The Singularity ABI follows the principle of least privilege [23]. By default, a SIP can do nothing more than manipulate its own state, and stop and start child SIPs. The ABI ensures a minimal, secure, and isolated computation environment for each SIP.

SIPs gain access to higher-level system services, such the ability to access files or send and receive network packets, through channels, not ABI functions. In Singularity, channel endpoints are capabilities [19, 24]. For example, a SIP can only access a file if it receives an endpoint to the file system from another SIP. Channel endpoints are either present when a process starts, as specified by manifest-based configuration (Section 4.1.1), or arrive in messages over existing channels.

By design, the ABI distinguishes mechanisms for accessing primitive, process-local operations from higher-level services. This distinction supports the use of channel endpoints as capabilities and verification of channel properties. Dynamically, the ABI design constrains the entry of new endpoints—new capabilities—into a SIP to explicit channels, which have contracts that specify the transfer. Static analysis tools can use typing information to determine the capabilities accessed by code. For example, a static tool can determine that an application plug-in is incapable of launching a distributed denial-of-service attack, because it does not contain code that uses the network channel contract.

The kernel ABI is strongly versioned. The manifest for each MBP explicitly identifies the ABI version it requires. At the language level, program code is compiled against a specific ABI interface assembly in a namespace that explicitly names the version, such as `Microsoft.Singularity.V1.Threads`. A Singularity kernel can export multiple versions of its ABI, to provide a clear path for backward compatibility.

Table 2 provides a break-down of Singularity ABI functions by feature. Students of other microkernel designs may find the number of ABI functions, 192, to be shockingly large. The ABI design is much simpler than the number suggests, as its design favors explicit semantics over minimal function entry points. In particular, Singularity contains no semantically complex functions like UNIX's `ioctl` or Windows' `CreateFile`.

The ABI maintains a system-wide *state isolation invariant*: a process cannot directly alter the state of another process through an ABI function. Object references cannot be passed through the ABI to the kernel or to another SIP. Consequently, the kernel's and each process's garbage collector executes independently. ABI calls affect only the state of the calling process. For example, in-process synchronization objects—such as mutexes—cannot be accessed across SIP boundaries. State isolation ensures that a Singularity process has sole control over its state.

Feature	Functions
Channels	22
Child Processes	21
Configuration	25
Debugging & Diagnostics	31
Exchange Heap	8
Hardware Access	16
Linked Stacks	6
Paged Memory	17
Security Principals	3
Threads & Synchronization	43
	192

Table 2. Number of Singularity ABI functions by feature.

Crossing the ABI boundary between SIP and kernel code can be very efficient as SIPs rely on software isolation instead of hardware protection. The best case is a SIP running at kernel hardware privilege level (ring 0 on the x86 architecture) in the same address space as the kernel. ABI calls, however, are more expensive than functions calls as they must demark the transition between the SIP's garbage-collection space and the kernel's garbage-collection space.

3.1.1 Privileged Code

In any OS, system integrity is protected by guarding access to privileged instructions. For example, loading a new page table can subvert hardware protection of process memory, so an operating system such as Windows or Linux will guard access to code paths that load page tables. In an OS that relies on hardware protection, functions that contain privileged instructions must execute in the kernel, which runs at a different privilege level than user code.

SIPs allow greater flexibility in the placement of privileged instructions. Because type and memory safety assure the execution integrity of functions, Singularity can place privileged instructions, with safety checks, in trusted functions that run inside SIPs. For example, privileged instructions for accessing I/O hardware can be safely in-lined into device drivers at installation time. Other ABI functions can be in-lined into SIP code at installation time as well. Singularity takes advantage of this safe in-lining to optimize channel communication and the performance of language runtimes and garbage collectors in SIPs.

3.1.2 Handle Table

While the design forbids cross-ABI object references, it is necessary for SIP code to name abstract objects in the kernel, such as mutexes or threads. Abstract kernel objects are exposed to SIPs through strongly typed, opaque handles that reference slots in the kernel's handle table. Inside the kernel, handle table slots contain references to literal kernel objects. Strong typing prevents SIP code from changing or forging non-zero handles. In addition, slots in the handle table are reclaimed only when a SIP terminates, to prevent the SIP from freeing a mutex, retaining its handle, and using it to manipulate another SIP's object. Singularity reuses table entries within a SIP, as retaining a handle in this case can only affect the offending SIP.

3.2 Memory Management

In most Singularity systems, the kernel and all SIPs reside in a single address space protected with software isolation. The address space is logically partitioned into a kernel object space, an object space for each SIP, and the *exchange heap* for communication of channel data. A pervasive design decision is

the *memory independence invariant*: pointers must point to SIP's own memory or to memory owned by the SIP in the exchange heap. No SIP can have a pointer to another SIP's objects. This invariant ensures that each SIP can be garbage collected and terminated without the cooperation of other SIPs.

A SIP obtains memory via ABI calls to the kernel's page manager, which returns new, unshared pages. These pages need not be adjacent to a SIP's previously allocated memory pages because the garbage collectors do not require contiguous memory, though blocks of contiguous pages may be allocated for large objects or arrays. In addition to memory for the SIP's code and heap data, a SIP has a stack per thread and access to the exchange heap.

3.2.1 Exchange Heap

All data passed between SIPs must reside in the exchange heap. Figure 3 shows how process heaps and the exchange heap relate. SIPs can contain pointers into their own heap and into the exchange heap. The exchange heap only holds pointers into the exchange heap itself (not into a process or the kernel). Every block of memory in the exchange heap is owned (accessible) by at most one SIP at any time during the execution of the system. Note that it is possible for a SIP to hold dangling pointers into the exchange heap (pointers to blocks that the SIP no longer owns). Static verification ensures that the SIP will never access memory through a dangling pointer.

To enable static verification of this property, we enforce a stronger property, namely that a SIP has at most one pointer to a block at any point in its execution (a property called linearity). Ownership of a block can only be transferred to another SIP by sending it in a message across a channel. Singularity ensures that a SIP does not access a block after it has sent it in a message.

The fact that each block in the exchange heap is accessible by a single thread at any time also provides a useful mutual exclusion guarantee. Furthermore, block freeing is enforced statically. On abrupt process termination, blocks in the exchange heap are recovered through reference counting. Block ownership is recorded so that all relevant blocks can be released when a SIP terminates.

3.3 Threads

The Singularity kernel and SIPs are multi-threaded. All threads are kernel threads that are visible to the kernel's scheduler, which coordinates blocking operations. In most Singularity systems, the performance of kernel thread context switches is closer to the performance expected of user threads, because no protection mode transitions are necessary for SIPs running in the kernel's address space and at its hardware privilege level.

3.3.1 Linked Stacks

Singularity uses linked stacks to reduce thread memory overhead. 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 [28]. At a function call, if stack space is inadequate, the SIP code calls an ABI, which allocates a new stack segment and initializes the first stack frame in the segment—between the running procedure and its callee—to invoke the segment unlink routine, which will release the segment when the stack frame is popped. For SIPs running 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 per-processor interrupt stack.

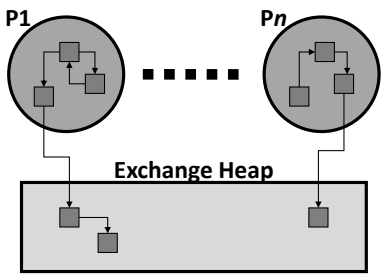

Figure 3. Pointers in process heaps and the exchange heap.

3.3.2 Scheduler

The standard Singularity scheduler is optimized for a large number of threads that communicate frequently. The scheduler maintains two lists of runnable threads. The first, called the *unblocked* list, contains threads that have recently become runnable. The second, called the *preempted* list, contains runnable threads that were 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 (also in FIFO order). 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. Then, the first thread from the unblocked list is scheduled and the scheduling timer is reset.

This two list scheduling policy favors threads that are awoken by a message, do a small amount of work, send one or more messages to other SIPs, 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 hardware timer, threads from the unblocked list inherit the scheduling quantum of the thread that unblocked them. Combined with the two-list policy, quantum inheritance allows Singularity to switch from user code on a thread in one SIP to user code on a thread in another SIP in as few as 394 cycles.

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, kernel and SIP object spaces are garbage collected.

The large number of existing garbage collection algorithms and experience strongly suggest that no one garbage collector is appropriate for all system or application code [10]. Singularity's architecture decouples the algorithm, data structures, and execution of each SIP's garbage collector, so it can be selected to accommodate the behavior of code in the SIP and to run without global coordination. The four aspects of Singularity that make this possible are: each SIP is a closed environment with its own run-time support; pointers do not cross SIP 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, which is reference counted; and the kernel controls memory page allocation, which provides a nexus for coordinating resource allocation.

Singularity's runtime 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 concurrent mark-sweep collector 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 typically use a simpler non-concurrent mark-sweep collector for applications.

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

In a garbage collected environment, a thread's stack contains object references that are potential roots for a collector. Calls into the kernel also execute 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-SIP 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-space (SIP $\rightarrow$ kernel or kernel $\rightarrow$ SIP) call, Singularity saves callee-saved registers in a special structure on the stack, which also demarks a cross-space 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 SIPs cleanly. When a SIP is killed, each of its threads is immediately stopped with a kernel exception, which skips over and deallocates the process's stack frames.

3.5 Channel Implementation

Channel endpoints and the values transferred across channels reside in the exchange heap. The endpoints cannot reside in a SIP's object space, since they may be passed across channels. Similarly, data passed on a channel cannot reside in a process, since passing it would violate the memory independence invariant. A message sender passes ownership by storing a pointer to the message in the receiving endpoint, at a location determined by the current state of the message exchange protocol. The sender then notifies the scheduler if the receiving thread is blocked awaiting to receive a message.

In order to achieve zero-allocation communication across channels, we enforce a finiteness property on the queue size of each channel. The rule we have adopted, which is enforced by Sing#, is that each cycle in the state transitions of a contract contains at least one receive and one send action. This simple rule guarantees that neither endpoint can send an unbounded amount of data without having to wait for a message. It allows static allocation of queue buffers in an endpoint. Although the rule seems restrictive, we have not yet seen a need to relax this rule in practice.

Pre-allocating endpoint queues and passing pointers to exchange heap memory naturally allow "zero copy" implementations of multi-SIP subsystems, such as the I/O stack. For example, disk buffers and network packets can be transferred across multiple channels, through a protocol stack and into an application SIP, without copying.

3.6 Principals and Access Control

In Singularity, applications are security principals. More precisely, principals are compound in the sense of Lampson *et al.* [16, 29]: they reflect the application identity of the current SIP, an optional role in which the application is running, and an optional chain of principals through which the application was invoked or given delegated authority. Users, in the traditional sense, are roles of applications (for example, the system login program running in the role of the logged in user). Application names are derived from MBP manifests which in turn carry the name and signature of the application publisher.

In the normal case, SIPs are associated with exactly one security principal. However, it might be necessary, in the future, to allow multiple principals per SIP to avoid excessive SIP creation in certain apps that exert the authority of multiple different principals. To support this usage pattern, we allow delegation of authority over a channel to an existing SIP.

All communication between SIPs occurs over channels. From the point of view of a SIP protecting resources (for example, files), each inbound channel speaks for a single security principal and that principal serves as the subject for access control decisions made with respect to that channel. Singularity access control is discretionary; it is up to the file system SIP, for example, to enforce controls on the objects it offers. The kernel support needed to track principal names and associate principals with SIPs and channels is fairly minimal. Access control decisions are made by matching textual principals against patterns (e.g., regular expressions), but this functionality is provided entirely through SIP-space libraries as discussed by Wobber *et al.* [30].

4. DESIGN SPACE EXPLORATION

In the previous sections we described the architectural foundation of Singularity: SIPs, channels, MBPs, and the Singularity kernel. Upon this foundation, we are exploring new points in the OS design space. Singularity's principled architecture, amenability to sound static analysis, and relatively small code base make it an excellent vehicle for exploring new options. In the following subsections, we describe four explorations in Singularity: compile-time reflection for generative programming, support for hardware protection domains to augment SIPs, hardware-agnostic heterogeneous multiprocessing, and typed assembly language.

4.1 Compile-Time Reflection

The Java and CLR runtime environments provide mechanisms to dynamic inspect existing code and metadata (types and members), as well as to generate new code and metadata at run time. These capabilities—commonly called reflection—enable highly dynamic applications as well as generative programming.

Unfortunately, the power of runtime reflection comes at a high price. Runtime reflection necessitates extensive run-time metadata, it inhibits code optimization because of the possibility of future code changes, it can be used to circumvent system security and information-hiding, and it complicates code generation because of the low-level nature of reflection APIs.

We have developed a new c*ompile-time reflection* (CTR) facility [9]. Compile-time reflection (CTR) is a partial substitute for the CLR's full reflection capability. The core feature of CTR is a high-level construct in Sing#, called a *transform,* which allows programmers to write inspection and generation code in a pattern matching and template style. The generated code and metadata can be statically verified to ensure it is well-formed, type-safe, and not violate system safety properties. At the same time, a programmer can avoid the complexities of reflection APIs.

Transforms can be provided by application developers or by the OS as part of its trusted computing base. Transforms can be applied either in the front-end compiler, as source is converted to MSIL (Microsoft Intermediate Language – a high-level, language independent program representation), or at install time, as MSIL is read, before being compiled to native instructions. In the sealed, type-safe world of a SIP, OS-provided transforms can enforce system policy and improve system performance, as they can safely introduce trusted code into an otherwise untrusted process.

4.1.1 Manifest-Based Configuration by CTR

An early use of CTR was to construct boiler-plate code for SIP startup from the declarative specifications of configuration parameters in a MBP manifest. The generated code retrieves startup arguments through a kernel ABI, casts the arguments to their appropriate declared type, and populates a startup object for the SIP. This CTR transform completely replaces traditional string-based processing of command-line arguments in programs with declarative manifest-based configuration. A similar transform is used to automate the configuration of device drivers from manifest information [25].

The transform in Listing 3, named `DriverTransform`, generates the startup code for a device driver from a declarative declaration of the driver's resources needs. For example, the declarations in the SB16 audio driver describe requirement to access banks of I/O registers through the `IoPortRange` class, see Listing 4.

`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, `$DriverConfig` is bound to `Sb16Config`. 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 Listing 5.

```
transform DriverTransform
        where $IoRangeType: IoRange {

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

    public readonly static $DriverConfig Values;

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

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

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

Listing 3. CTR transform for device driver configuration.

```
[DriverCategory]
[Signature("/pnp/PNPB003")]
internal class Sb16Config: DriverCategoryDeclaration {
  [IoPortRange(0, Default = 0x0220,Length = 0x10)]
  internal readonly IoPortRange basePorts;

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

  internal readonly static Sb16Config Values;

  reflective private Sb16Config();
}
```

Listing 4. Declarations of requirements in SB16 driver.

```
class SB16Config {
  …
  static Sb16Config() {
    Values = new Sb16Config();
  }

  private Sb16Config() {
    IoConfig config = IoConfig.GetConfig();
    Tracing.Log(Tracing.Debug,
         "Config: {0}", config.ToPrint());

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

Listing 5. Output of transform to SB16 code.

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 (`$DriverConfig`) matches the type of the `Values` field.

CTR transforms have proven to be an effective tool for generative programming. As we apply CTR to new domains in Singularity we continue to improve the generality of the transforms. For example, recent experiments with using CTR transforms to generate marshaling code have necessitated increased transform expressiveness.

4.2 Hardware Protection Domains

Most operating systems use CPU memory management unit (MMU) hardware isolate processes by using two mechanisms. First, processes are only allowed access to certain pages of physical memory. Second, privilege levels prevent untrusted code from executing privileged instructions that manipulate the system

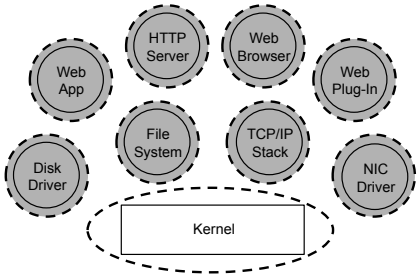

Figure 4a. Micro-kernel configuration (like MINIX 3). Dotted lines mark protection domains; dark domains are user-level, light are kernel-level.

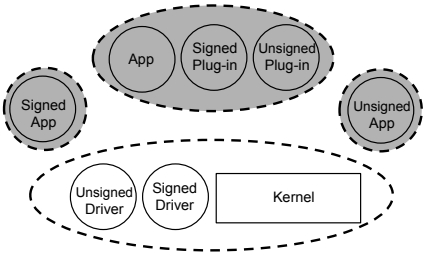

Figure 4b. Monolithic kernel and monolithic application configuration.

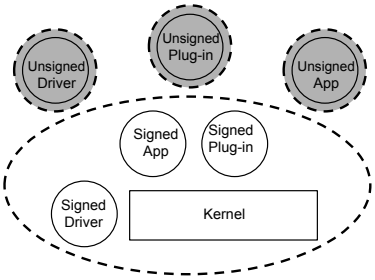

Figure 4c. Configuration with distinct policies for signed and unsigned code.

resources that implement processes, such as the MMU or interrupt controllers. These mechanisms have non-trivial performance costs that are largely hidden because there has been no widely used alternative approach for comparison.

To explore the design trade-offs of hardware protection versus software isolation, recent work in Singularity augments SIPs, which provide isolation through language safety and static verification, with *protection domains* [1], which can provide an additional level of hardware-based protection around SIPs. The lower run-time cost of SIPs makes their use feasible at a finer granularity than conventional processes, but hardware isolation remains valuable as a defense-in-depth against potential failures in software isolation mechanisms or to make available needed address space on 32-bit machines.

Protection domains are hardware-enforced protection boundaries, which can host one or more SIPs. Each protection domain consists of a distinct virtual address space. The processor's MMU enforces memory isolation in a conventional manner. Each domain has its own exchange heap, which is used for communications between SIPs within the domain. A protection domain that does not isolate its SIPs from the kernel is called a *kernel domain*. All SIPs in a kernel domain run at the processor's supervisor privilege level (ring 0 on the x86 architecture), and share the kernel's exchange heap, thereby simplifying transitions and communication between the processes and the kernel. Non-kernel domains run at user privilege level (ring 3 on the x86).

Communication within a protection domain continues to use Singularity's efficient reference-passing scheme. However, because each protection domain resides in a separate address space, communication across domains requires data copying or copy-on-write page mapping. The message-passing semantics of Singularity channels makes the implementations indistinguishable to application code (except for performance).

A protection domain could, in principle, host a single process containing unverifiable code written in an unsafe language such as C++. Although very useful for running legacy code, we have not yet explored this possibility. Currently, all code within a protection domain is also contained within a SIP, which continues to provide an isolation and failure containment boundary.

Because multiple SIPs can be hosted within a protection domain, domains can be employed selectively to provide hardware isolation between specific processes, or between the kernel and processes. The mapping of SIPs to protection domains is a run-time decision. A Singularity system with a distinct protection domain for each SIP is analogous to a fully hardware-isolated

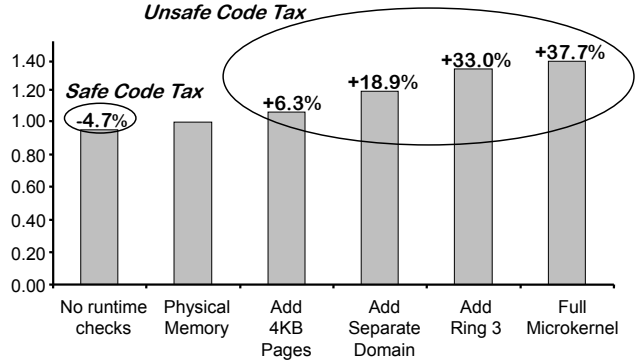

Figure 5. Normalized WebFiles execution time.

microkernel system, such as MINIX 3 [12] (see Figure 4a). A Singularity system with a kernel domain hosting the kernel, device drivers, and services is analogous to a conventional, monolithic operating system, but with more resilience to driver or service failures (see Figure 4b). Singularity also supports unique hybrid combinations of hardware and software isolation, such as selection of kernel domains based on signed code (see Figure 4c).

4.2.1 Quantifying the Unsafe Code Tax

Singularity offers a unique opportunity to quantify the costs of hardware and software isolation in an apples-to-apples comparison. Once the costs are understood, individual systems can choose to use hardware isolation when its benefits outweigh the costs.

Hardware protection does not come for free, though its costs are diffuse and difficult to quantify. Costs of hardware protection include maintenance of page tables, soft TLB misses, cross-processor TLB maintenance, hard paging exceptions, and the additional cache pressure caused by OS code and data supporting hardware protection. In addition, TLB access is on the critical path of many processor designs [2, 15] and so might affect both processor clock speed and pipeline depth. Hardware protection increases the cost of calls into the kernel and process context switches [3]. On processors with an untagged TLB, such as most current implementations of the x86 architecture, a process context switch requires flushing the TLB, which incurs refill costs.

Figure 5 graphs the normalized execution time for the WebFiles benchmark in six different configurations of hardware and software isolation. The WebFiles benchmark is an I/O intensive benchmarks based on SPECweb99. It consists of three SIPs: a

client which issues random file read requests across files with a Zipf distribution of file size, a file system, and a disk device driver. Times are all normalized against a default Singularity configuration where all three SIPs run in the same address space and privilege level as the kernel and paging hardware is disabled as far as allowed by the processor.

The WebFiles benchmark clearly demonstrates the *unsafe code tax,* the overheads paid by every program running in a system built for unsafe code. With the TLB turned on and a single system-wide address space with 4KB pages, WebFiles experiences an immediate 6.3% slowdown. Moving the client SIP to a separate protection domain (still in ring 0) increases the slowdown to 18.9%. Moving the client SIP to ring 3 increases the slowdown to 33%. Finally, moving each of the three SIPs to a separate ring 3 protection domain increases the slowdown to 37.7%. By comparison, the runtime overhead for safe code is under 5% (measured by disabling generation of array bound and other checks in the compiler).

The unsafe code tax experienced by WebFiles may be worst case. Not all applications are as IPC intensive as WebFiles and few operating systems are fully isolated, hardware-protected microkernels. However, almost all systems in use today experience the overheads of running user processes in ring 3. In a traditional hardware-protected OS, every single process pays the unsafe code tax whether it contains safe or unsafe code. SIPs provide the option of forcing only unsafe programs to pay the unsafe code tax.

4.3 Heterogeneous Multiprocessing

Thanks to physical constraints, it is easier to replicate processors on a die than to increase processor speed. With vendors already demonstrating prototype chips with 80 processing cores [27], we have begun experiments in support for so-called "many-core" systems in Singularity. These experiments build on the SMP support already provided by the Singularity kernel.

OS support for many-core systems goes beyond the simple thread safety and data locality issues required for scaling on traditional SMP systems. As work by Chakraborty *et al.* [5] suggests, code and metadata locality can become crucial performance bottlenecks. Chakraborty improved system performance by dynamically switching processors on user-kernel switches so that OS code ran on one set of processors and application code ran on another set. They assert that such *dynamic specialization* of processors achieves better instruction cache locality and also improves branch prediction as the processors tune themselves for either application or OS code characteristics. We expect such dynamic specialization to become even more beneficial as the number of cores per chip increases faster than cache per chip.

Singularity already offers further opportunities for dynamic specialization of processors beyond what Chakraborty could achieve with a monolithic OS. For example, because many traditional OS services—such as file systems and network stacks—are located in individual SIPs, Singularity can specialize many-core processors by dedicating them to specific SIPs. With a smaller code footprint per processor, there should be greater affinity between the SIP code and the i-cache and other dynamic performance optimization hardware in the processor [17].

Our hypothesis is that a smaller code footprint results in greater dynamic specialization of the processor. We have recently experimented with running only reduced subsets of the Singularity microkernel on processors dedicated to a specific SIP. In the smallest variant, none of the kernel runs on a dedicated SIP processor. Instead, all ABI calls are remoted from the dedicated processor to a processor running the full kernel using inter-processor interrupts.

4.3.1 Instruction Set Architectures

Processing cores are proliferating not only in CPUs, but in I/O cards and peripheral devices as well. Programmable I/O cards commonly found in PCs include graphics processors, network cards, RAID controllers, sound processors, and physics engines. These processing cores present unique challenges as they often have very different instruction set architectures and performance characteristics from a system's CPUs.

We currently see two approaches to programmable I/O processors within the OS community. In one camp are the "traditionalists" who argue that programmable I/O processors have come and gone before, so there is little long-term need to consider them in the OS. Programmable I/O devices should be treated as I/O devices with their processors hidden as implementation details behind OS I/O abstractions—this is the approach followed by Microsoft's TCP Chimney offload architecture [20], for example. In another camp are the "specialists" who argue that I/O processors, such as GPUs, should be treated as special, distinct processing units executing outside the standard OS computation abstractions—this is the approach followed by Microsoft's DirectX. To this camp, I/O processors will always require a unique tool set.

Within Singularity, we see an opportunity to pursue a new course for programmable I/O processors. We agree with the "specialists" that programmable I/O processors are here to stay due to the better performance-per-watt of specialized processors. However, unlike the "specialists," we are exploring the hypothesis that programmable I/O processors should become first-class entities to OS scheduling and compute abstractions. The core idea is quite simple: computation on programmable I/O processors should occur within SIPs. Because of our existing heterogeneous processing support, dedicated I/O processors need not run any more of the Singularity kernel than the minimal variant that remotes most ABI operations to the CPUs.

We believe the Singularity architecture offers five advantages that make this new design for programmable I/O processing promising. First, SIPs minimize the need for elaborate processor features on I/O cores. For example, I/O processors need not have memory management units for process protection. Second, contract-based channels explicitly define the communication between a SIP on an I/O processor and other SIPs. Third, Singularity's memory isolation invariant removes the need for shared memory (or cache coherency) between SIPs on coprocessors and CPUs. Fourth, the small, process-local ABI isolates operations that may be safely implemented locally—such as memory allocation—from services which must involve other SIPs. Finally, Singularity packages manifest-based programs in the abstract MSIL format, which can be converted to any I/O processor's instruction set. The same TCP/IP binary can be installed for both a system's x86 CPU and its ARM-based programmable network adapter.

We expect that the instruction-set neutrality of Singularity MBPs encoded in MSIL may ultimately be relevant even for many-core CPUs. As many-core systems proliferate, many in the engineering community anticipate hardware specialization of cores. For example, the pairing of large out-of-order cores with smaller in-

order-cores will provide systems with greater control over power consumption. Many-core systems enable processor specialization as each individual processor need not pay the full price of compatibility required for single core chips; a many-core chip may be considered backwards compatible as long as at least one of its cores is backwards compatible. Our hypothesis is that Singularity binaries can target "legacy-free" cores on many-core CPUs as easily as "legacy-free" cores on programmable I/O processors.

4.4 Typed Assembly Language

Since Singularity uses software verification to enforce isolation between SIPs, the correctness of its verifier is critical to Singularity's security. For example, to ensure that untrusted code in a SIP cannot access memory outside the SIP, the verifier must check that the code does not cast an arbitrary integer to a pointer. Currently, Singularity relies on the standard Microsoft Intermediate Language (MSIL) verifier to check basic type safety properties (e.g. no casts from integers to pointers or from integers to kernel handles). Singularity also has an ownership checker that verifies that MSIL code respects Singularity's rule that each block in the exchange heap is accessibly by only one thread.

Singularity uses the Bartok compiler [13] to translate an MBP's MSIL code to native machine language code (such as x86 code). If the compiler were free of bugs, then it would translate safe MSIL code into safe native code. Since Bartok is a large and highly optimizing compiler, it is likely to contain bugs, and some of these bugs might cause the compiler to translate safe MSIL code into unsafe native code.

We have begun integrating research on proof-carrying code [22] and typed assembly language [21] into Bartok and Singularity. Bartok has a typed intermediate language that maintains typing information as it compiles to native code. This information will allow Singularity to verify the safety of the native "typed assembly language" (TAL) code, rather than the MSIL code. Furthermore, a verifier for native code would allow Singularity to run safe native code generated by other compilers or written by hand.

As Bartok compiles MSIL code to native code, it translates the data layout from MSIL's abstract data format to a concrete data format. This concrete format specifies exactly where fields are in objects, where method pointers are in method tables, and where run-time type information resides. Generating types for the method tables and run-time type information is challenging; treating these naively using simple record types can lead to ill-typed native code (or worse, an unsound type system; see [18] for more information). On the other hand, using too sophisticated of a type system may render type checking difficult or even undecidable. Bartok uses Chen and Tarditi's LIL_C type system [6], which can type method table and run-time type information layouts, but still has a simple type checking algorithm.

The concrete data format also specifies garbage collection information, such as the locations of pointer fields in each object. If this information is wrong, the garbage collector may collect data prematurely, leaving unsafe dangling pointers. Furthermore, some of Singularity's garbage collectors impose additional requirements on a SIP's native code. For instance, the generational collector requires that code invoke a "write barrier" before writing to a field. Failure to invoke the write barrier may lead to dangling pointers. Finally, each Singularity garbage collector is currently written as unsafe Sing# code, and bugs in this code could undermine Singularity's security. We are addressing these issues by rewriting the garbage collectors as safe code, so that we can verify the SIP's untrusted code and the collector's code together.

Writing a garbage collector in a safe language is challenging because conventional safe languages do not support explicit memory deallocation. Therefore, we have developed a type system that supports both LIL_C's types and explicit proofs about the state of memory [11]. Using this type system, the garbage collector can statically prove that its deallocation operations are safe. The type system's support for both LIL_C and memory proofs will allow both the SIP's untrusted code and the garbage collector code to co-exist as a single, verifiable TAL program, ensuring the safety of the SIP's code, the collector, and their interaction. We have currently implemented only a simple copying collector written in a simple RISC typed assembly language, but we are working on porting more sophisticated collectors to an x86 version of our TAL.

5. CONCLUSIONS

We started the Singularity project over three years ago to explore how advances in languages, tools, and operating systems might produce more dependable software systems. Our goal remains to develop techniques and technologies that result in significant improvements in dependability. As a common laboratory for experimentation, we created the Singularity OS. Central to the OS are three fundamental architectural decisions that have significantly improved the ability to verify the expected behavior of software systems: software isolated processes (SIPs), contract-based channels, and manifest-based programs (MBPs).

5.1 Performance and Compatibility

The Singularity project deprecated two priorities important to most successful operating systems: high performance and compatibility with previous systems. Singularity has always favored design clarity over performance. At most decision points Singularity attempted to provide "good enough" performance, but no better. In some cases, such as communication micro-benchmarks, Singularity significantly out-performs existing systems. This performance is a pleasant side effect of architecture and design decisions motivated by a desire to build more dependable systems.

Singularity abandoned application and driver compatibility to explore new design options. This choice has been a double-edged sword. On the one hand, we have been free to explore new ideas without legacy constraints. On the other hand, we have been forced to rewrite or port every line of code in the Singularity system. We would not suggest this approach for every project, but we believe it was the correct choice for Singularity. The payoff from the research freedom has been worth the cost.

5.2 Architecture, Language, & Tool Synergies

One of the most important lessons we learned from Singularity is the benefits of a tight feedback cycle among programming languages, OS architectures, and verification tools. Advances in one area can have beneficial impact that enables an advance in another area; that advance then enables an advance in another area, and the cycle continues (see Figure 6).

For example, our decision to seal SIPs against code loading was inspired by discussions with a representative of the Microsoft Office product team who was concerned about the poor quality of

a specific plug-in. This change in OS architecture is practical to enforce because SIPs contain safe code—no code changes through pointers—and it enhanced the soundness and reach of the static verification techniques.

Similarly, the design of channels and contracts in Singularity was heavily influenced by recent advances by our colleagues in verification of communications in web services. Our colleagues in the OS community often note that, at an implementation level, two uni-directional channels are preferable to a single bi-directional channel. However, a bi-directional channel makes the communication between two processes much easier to analyze. Following the web services design not only improved verification, but it also improved message-passing performance by enabling zero-copy communication through pointer passing.

Choice of language and verification technologies affects OS architecture. Choice of OS architecture affects language and verification technologies. Singularity has only scratched the surface of benefits enabled by considering languages, tools, and architecture together, but it highlights the opportunities available.

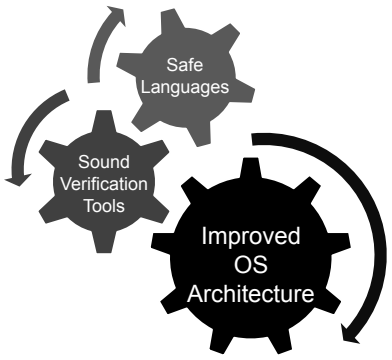

Figure 6. The synergy of improved OS architecture, safe languages, and sound verification tools.

6. ACKNOWLEDGEMENTS

Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, and Ted Wobber deserve special thanks for their contributions to Sections 4.2, 4.1, 4.4, and 3.6, respectively.

A project like Singularity is the work of many hands. The Singularity project has benefited from an incredibly talented collection of our colleagues at Microsoft Research including Sriram Rajamani from the Bangalore lab; Paul Barham, Richard Black, Austin Donnelly, Tim Harris, Rebecca Isaacs, and Dushyanth Narayanan from the Cambridge lab; Mark Aiken, Juan Chen, Trishul Chilimbi, John DeTreville, Manuel Fahndrich, Wolfgang Grieskamp, Chris Hawblitzel, Orion Hodson, Mike Jones, Steven Levi, Qunyan Mangus, Virendra Marathe, Kirk Olynyk, Mark Plesko, Jakob Rehof, Wolfram Schulte, Dan Simon, Bjarne Steensgaard, David Tarditi, Songtao Xia, Brian Zill, and Ben Zorn from the Redmond lab; and Martin Abadi, Andrew Birrell, Úlfar Erlingsson, Roy Levin, Nick Murphy, Chandu Thekkath, and Ted Wobber from the Silicon Valley lab.

To date 22 interns have contributed significant amounts of code and inspiration to Singularity. They are: Michael Carbin, Adam Chlipala, Jeremy Condit, Fernando Castor de Lima Filho, Marc Eaddy, Daniel Frampton, Haryadi Gunawi, Hiroo Ishikawa, Chip Killian, Prince Mahajan, Virendra Marathe, Bill McCloskey, Martin Murray, Martin Pohlack, Polyvios Pratikakis, Tom Roeder, Roussi Roussev, Avi Shinnar, Mike Spear, Cesar Spessot, Yaron Weinsberg, and Aydan Yumerefendi.

Finally, a number of individuals from Microsoft's engineering organizations have made direct contributions to Singularity—many in their spare time. These include Chris Brumme, Kent Cedola, Ken Church, Arlie Davis, Rob Earhart, Raymond Endres, Bruno Garagnon, Alexander Gounares, Jim Herbert, Ashok Kuppusamy, Adrian Marinescu, Ravi Pandya, Stathis Papaefstathiou, John Richardson, Evan Schrier, Valerie See, David Stahlkopf, Aaron Stern, Clemens Szyperski, Dean Tribble, and Sean Trowbridge.

7. REFERENCES

[1] Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G. and Larus, J., Deconstructing Process Isolation. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Correctness and Performance (MSPC 2006)*, San Jose, CA, October 2006.

[2] Allen, D.H., Dhong, S.H., Hofstee, H.P., Leenstra, J., Nowka, K.J., Stasiak, D.L. and Wendel, D.F. Custom Circuit Design as a Driver of Microprocessor Performance. *IBM Journal of Research and Development*, *44* (6) 2000.

[3] Anderson, T.E., Levy, H.M., Bershad, B.N. and Lazowska, E.D. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 108-120, Santa Clara, CA, 1991.

[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 Fifteenth ACM Symposium on Operating System Principles*, pp. 267-284, Copper Mountain Resort, CO, 1995.

[5] Chakraborty, K., Wells, P. and Sohi, G., Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pp. 283-302, San Jose, CA, October 2006.

[6] Chen, J. and Tarditi, D., A Simple Typed Intermediate Language for Object-oriented Languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pp. 38-49, Long Beach, CA, January 2005.

[7] ECMA International, ECMA-335 Common Language Infrastructure (CLI), 4th Edition. Technical Report 2006.

[8] Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R. and Levi, S., Language Support for Fast and Reliable Message Based Communication in Singularity OS. In *Proceedings of the EuroSys 2006 Conference*, pp. 177-190, Leuven, Belgium, April 2006.

[9] Fähndrich, M., Carbin, M. and Larus, J., Reflective Program Generation with Patterns. In *5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, Portland, OR, October 2006.

[10] 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)*, pp. 111-120, Minneapolis, MN, 2000.

[11] Hawblitzel, C., Huang, H., Wittie, L. and Chen, J., A Garbage-Collecting Typed Assembly Language. In *AGM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '07)*, Nice, France, January 2007.

[12] Herder, J.N., Bos, H., Gras, B., Homburg, P. and Tanenbaum, A.S. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *Operating System Review*, *40* (3). pp. 80-89. July 2006.

[13] Hunt, G., Larus, J., Abadi, M., Aiken, M., Barham, P., Fähndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T. and Zill, B., An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[14] Hunt, G., Larus, J., Abadi, M., Aiken, M., Barham, P., Fähndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T. and Zill, B., Sealing OS Processes to Improve Dependability and Safety. In *Proceedings of the EuroSys 2007 Conference*, pp. 341-354, Lisbon, Portugal, March 2007.

[15] Kongetira, P., Aingaran, K. and Olukotun, K. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, *25* (2). pp. 21-29. March 2005.

[16] Lampson, B., Abadi, M., Burrows, M. and Wobber, E.P. Authentication in distributed systems: Theory and Practice. *ACM Transactions on Computer Systems*, *10* (4). pp. 265-310. November 1992.

[17] Larus, J.R. and Parkes, M. Using Cohort-Scheduling to Enhance Server Performance. In *Proceedings of the USENIX 2002 Annual Conference*, pp. 103-114, Monterey, CA, 2002.

[18] League, C. A Type-Preserving Compiler Infrastructure, Yale University, New Haven, CT, December, 2002.

[19] Levy, H.M. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, 1984.

[20] Microsoft Corporation, Scalable Networking: Network Protocol Offload - Introducing TCP Chimney. Technical Report 2004.

[21] 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). pp. 527-568. May 1999.

[22] Necula, G.C. and Lee, P., Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, WA, October 1996.

[23] Saltzer, J.H. and Schroeder, M.D. The protection of information in computer systems. *Proceedings of the IEEE*, *63* (9). pp. 1268-1308. September 1975.

[24] Shapiro, J.S., Smith, J.M. and Farber, D.J. EROS: a Fast Capability System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pp. 170-185, Charleston, SC, 1999.

[25] Spear, M.F., Roeder, T., Hodson, O., Hunt, G.C. and Levi, S., Solving the Starting Problem: Device Drivers as Self-Describing Artifacts. In *Proceedings of the EuroSys 2006 Conference*, pp. 45-58, Leuven, Belgium, April 2006.

[26] 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). pp. 419-490. October 1986.

[27] Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Iyer, P., Singh, A., Jacob, T., Jain, S., Venkataraman, S., Hoskote, Y. and Borkar, N., An 80-Tile 1.28TFLPOPS Network-on-Chip in 65nm CMOS. In *2007 IEEE International Solid-State Circuits Conference*, San Francisco, CA, February 2007.

[28] 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)*, pp. 268-281, Bolton Landing, NY, 2003.

[29] Wobber, E.P., Abadi, M., Burrows, M. and Lampson, B. Authentication in the Taos Operating System. *ACM Transactions on Computer Systems*, *12* (1). pp. 3-32. February 1994.

[30] Wobber, T., Abadi, M., Birrell, A., Simon, D.R. and Yumerefendi, A., Authorizing Applications in Singularity. In *Proceedings of the EuroSys 2007 Conference*, pp. 355-368, Lisbon, Portugal, March 2007.