Microsoft Research Technical Report MSR-TR-2006-43
This is a draft paper that is under submission. Please contact Galen Hunt (galenh@microsoft.com) or Jim Larus (larus@microsoft.com) for citation information.

Deconstructing Process Isolation

Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, James Larus

Microsoft Research
One Microsoft Way
Redmond, WA 98052

Abstract. Most operating systems enforce process isolation through hardware protection mechanisms such as memory segmentation, page mapping, and differentiated user and kernel instructions. Singularity is a new operating system that uses software mechanisms to enforce process isolation. A software isolated process (SIP) is a process whose boundaries are established by language safety rules and enforced by static type checking. With proper system support, SIPs can provide a low cost isolation mechanism that provides failure isolation and fast inter-process communication.

To compare the performance of Singularity's approach against more conventional systems, we implemented an optional hardware isolation mechanism. Protection domains are hardware-enforced address spaces, which can contain one or more SIPs. Domains can either run at the kernel's privilege levels and share an exchange heap or be fully isolated from the kernel and run at the normal application privilege level. These domains can construct Singularity configurations that are similar to micro-kernel and monolithic kernel systems. Hardware-based isolation incurs non-trivial performance costs (up to 25-33%) and complicates system implementation. Software isolation has less than 5% overhead on these benchmarks.

The lower cost of SIPs permits them to provide protection and failure isolation at a finer granularity than conventional processes. Singularity's ability to employ hardware isolation selectively, rather than at every process boundary, supports the construction of more resilient system architectures.

1 Introduction

Process isolation is fundamental function of most operating systems. Isolation protects system integrity by preventing a program from interfering with the system's or another application's code or data, and by preventing untrusted code from accessing protected resources. Isolation also contributes to system resilience by providing failure boundaries that permit part of a system to fail without compromising the whole.

Most operating systems use a CPU's memory management hardware to provide isolation at process granularity. Process isolation is built on two mechanisms. First, processes are only allowed access to certain pages of physical memory. Second, privilege levels prevent untrusted code from manipulating the system resources underlying processes, for example, the memory management unit (MMU) or interrupt controllers. These mechanisms can incur non-trivial performance costs. Mapping from virtual to physical addresses can incur overheads up to 10–30% due to exception handling, inline TLB lookup, TLB reloads, and maintenance of kernel data structures such as page tables [28]. In addition, virtual memory and privilege levels increase the cost of inter-process communication.

As a consequence, most operating systems provide processes that are too expensive to be used to provide fine-grain isolation, say between an application and a code extension. Using a conventional process to provide fine-grain isolation between components is only feasible if the components rarely interact. Closely coupled extensions, such as OS device drivers or application plug-ins, are difficult to isolate in separate processes. Typically, they are loaded into the same address space as their host, without any sort of isolation. Moreover, even when components are separated into processes, the high cost of inter-process communication encourages use of shared memory, which couples processes' failure behavior [17].

Extensions are a major source of problems in software reliability, security, and compatibility [13]. Although the extension code may be untrusted, unverified, faulty, or even malicious, it is loaded into its host's address space with no hard interface or boundary between the two—because the cost of traditional hardware protection mechanisms is too high. The outcome is often unpleasant, because a failure in the extension compromises or terminates the host. For example, Swift reports that faulty device drivers cause 85% of diagnosed Windows system crashes [38]. Moreover, without hard boundaries, extension can bypass public abstractions to access implementation internals, which constrains future program evolution and compels extensive compatibility testing.

To address these problems, we constructed a new operating system called Singularity [26] that uses two mechanisms to offer isolation with a wider range of cost-benefit tradeoffs. At one end of the range are *hardware isolated processes (HIPs)*, which are analogous to processes in other operating systems. Hardware isolation is built on a mechanism called a *protection domain*. At the other extreme are *software isolated processes (SIPs)*, which use programming language safety and system architecture to provide a less expensive isolation mechanism. Singularity also supports hybrid combinations of hardware and software isolation. One or more SIPs can reside in a protection domain, in configurations ranging from pure SIP to pure HIP. Figure 1 shows protection domains used to support combinations of software and hardware isolation with different cost/benefit trade-offs.

The design and implementation of a system based on SIPs is a major contribution of this work. A software isolated process is a collection of memory pages and a language safety mechanism that ensures that code in the process cannot access another process's pages. A SIP replaces hardware memory protection with static verification of program safety. Singularity uses language safety and a fast communication mechanism built on channels [14] to enforce a system-wide invariant that neither the kernel nor any other process contains a reference into a given process's object space. Singularity segregates objects on a per-process basis to facilitate reclaiming resources on process termination. This architecture provides fine-granularity isolation and high

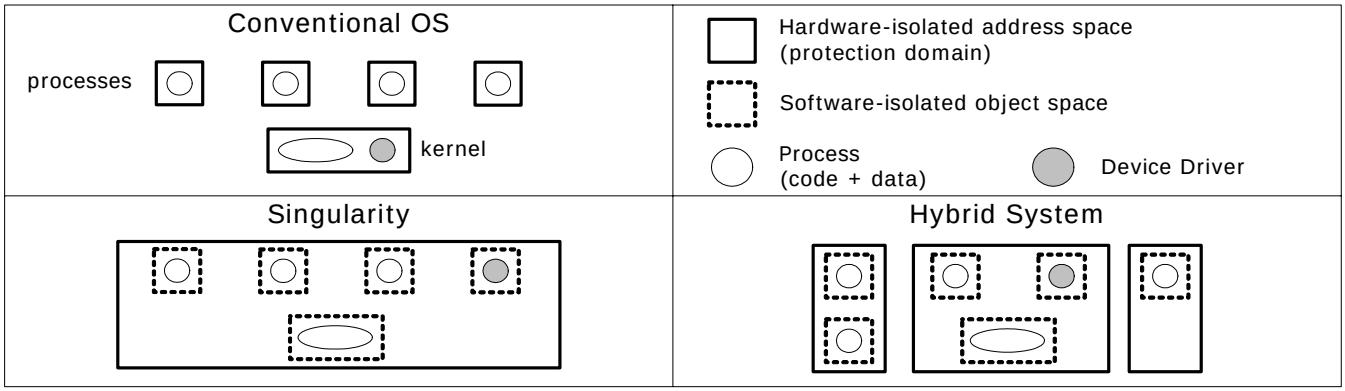

Figure 1. Alternative architectures enabled by software isolated processes.

performance. If a process fails, no other process's data is left in an inconsistent state, and failure notification is cleanly propagated through communication channels. Moreover, the system can easily reclaim the failed process's resources, including memory. And, without hardware isolation, system calls and inter-process communication run significantly faster (30–500%) and a communication-intensive benchmark runs approximately 20% faster.

A second major contribution of this work is a direct comparison of the costs of hardware and software isolation mechanisms. Previous studies implemented one or the other approach or used simulation to quantify overhead. Because Singularity implements both isolation mechanisms, we can directly compare costs on the same platform. We found that the two main contributors to the cost of memory management were establishing and maintaining the virtual to physical mapping and changing privilege levels between an applications and the kernel. Changing address spaces incurred minor costs for micro-benchmarks, but was a significant overhead for a macro-benchmark with a large amount of communications and context switching. In addition, hardware protection increased the complexity of interprocess communication mechanisms.

An additional contribution of this work is an initial exploration of Singularity's ability to use hardware isolation selectively, rather than at every process boundary. For example, system processes and device drivers (each of which run in their own process) can — but need not — reside in the same address space as the kernel. Using a single address space permits fast communication, but still provides memory and failure isolation. For example, a driver can fail without crashing the system. Similarly, application processes may share an address space with its extensions. Processes in the same address space again can communicate efficiently, but retain a strong assurance of isolation. These experiments show that the design space for process isolation is far richer than common practice.

The rest of the paper is organized as follows. Section 2 briefly describes software isolated processes (SIPs). Section 3 describes the implementation of Singularity's protection domains and virtual memory. Section 4 contains performance results. Section 5 concludes.

2 Software Isolated Processes

Software isolated processes (SIPs) use software verification, rather than hardware protection, to isolate portions of a system. They rely on verifying code's safe behavior to prevent it from accessing another process's (or the kernel's) instructions or data. A verifiably safe program can only access data that it allocated or was passed, and it cannot construct or corrupt a memory reference.[1]

This invariant is usually expressed as two simpler properties: type and memory safety. Type safety ensures that the only operations applied to a value are those defined for instances of its type. Memory safety ensures the validity of memory references by preventing null pointer references, references outside an array's bounds, or references to deallocated memory. These properties can be partially verified by a compiler, but in some circumstances require run-time tests. Java and C# are expressive, safe programming languages that can be compiled with a small performance penalty for safety [15]. The overhead of these tests in two large Singularity benchmarks is approximately 4.5% (Section 4.4). Other modern languages, such as Perl, Python, and Ruby, are also safe.

In Singularity, all untrusted code that runs in a SIP must be verifiably safe. The SIP may also contain trusted, unverified code that cannot be expressed in a safe language, for example, parts of the language runtime, a memory allocator, or an accessor for memory-mapped I/O. The correctness and safety of this type of code needs to be verified by other means.

A SIP is a closed object space that resides on a collection of memory pages exclusively owned by a process (Figure 2). An object space is the complete collection of objects reachable by a program at a point in its execution.

Singularity ensures that two processes' objects spaces are always disjoint. Since the code in the processes is safe, the only way in which one process could obtain a reference to another

[1] Object allocation creates new references and requires trusted, unsafe code. In Singularity, object allocation occurs within a trusted runtime, which logically is an extension of the kernel. Verifying the safety of a memory allocator or garbage collector is an interesting open research question [43].

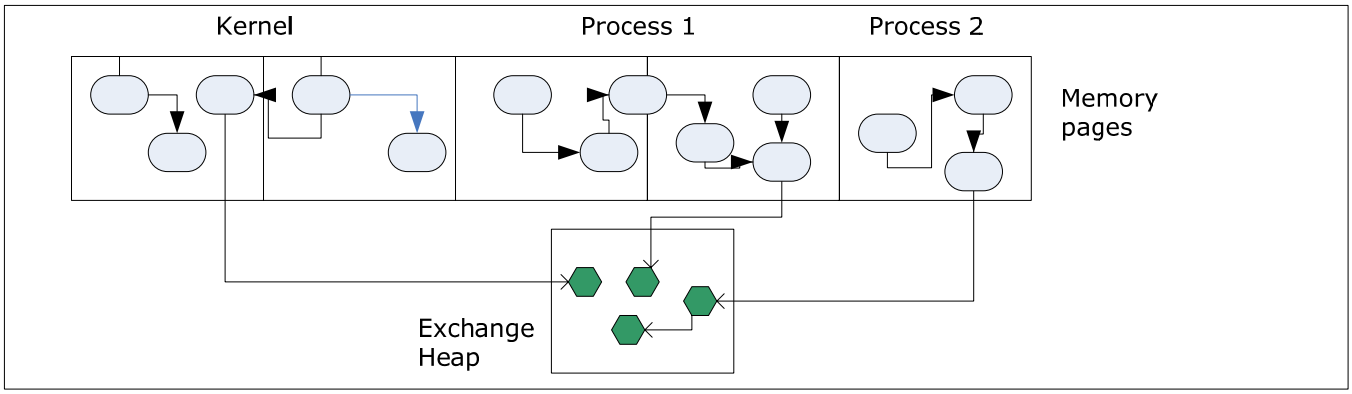

Figure 2. Software Isolated Processes (SIPs).

process's object is to be passed it. Singularity prevents cross-process references by not providing shared memory and by using the language type system to prevent a process from sending a object reference through the interprocess communication mechanism [14].

In Singularity, all interprocess communication occurs through message passing across channels, which allows two processes to exchange data through an area of memory called the exchange heap (Figure 2) [14]. Messages in this heap are structs, not objects (i.e., they do not contain methods) and cannot contain object references. Messages, however, can contain references to exchange heap data, so it is possible to send structured data between processes. However, an object reference cannot be embedded in a message or sent between processes. Moreover, the system maintains the invariant that there exists at most one pointer to an item in the exchange heap. When a process sends a message, it loses its reference to the message, which is transferred to the receiving process (analogous to sending a letter by postal mail). Therefore, processes cannot use this heap as shared memory and messages can be exchanged through pointer passing, not copying.

In Singularity, a process's objects reside on memory pages that belong exclusively to the process. A process acquires and releases memory from the kernel at page granularity. A SIP's pages typically are not contiguous, which presents no problem so long as each memory allocation in a SIP can be satisfied by a continuous range of memory.

Other safe language systems have taken a different approach and had processes allocate memory from one heap. Singularity's design has several advantages. First, it reduces coupling between processes by eliminating the shared memory allocator and garbage collector and by permitting each process to allocate and reclaim memory using whatever techniques are appropriate. The large number of existing garbage collection algorithms, and experience, suggest that no one garbage collector is appropriate for all systems or applications, so this flexibility helps achieve good system performance [16]. A shared service also constrains the object format and run-time system of every process that uses it. Moreover, these shared facilities are a common point of failure. Second, Singularity's design facilitates process termination, as the system can just reclaim memory pages in a conventional manner, rather than relying on garbage collection to reclaim individual objects.

Memory pages from different SIPs can reside in the same address space (physical or virtual), so that switching context need not flush the TLB or require address space identifiers. Moreover, since there are no cross-process references, SIPS can equally well reside in distinct address spaces, either to overcome a 32-bit address limitation or to enhance software isolation with hardware mechanisms (below).

Singularity also verifies that untrusted code does not contain privileged instructions, so all processes can run at Ring 0 (kernel level) on an x86 processor. As a consequence, system calls and interprocess communication on Singularity are up to an order of magnitude faster than other systems (Section 3.6). With lower overheads, SIPs can be used at a very fine granularity for failure containment.

2.1 Discussion

In Singularity, SIP isolation is built from language safety and system design. Language safety is a desirable end in itself, as it helps eliminate or mitigate software defects such as buffer-overrun attacks. For a safe language, the isolation provided by SIPs incurs no additional cost, beyond a deliberate loss of the ability to share data between processes. In a sense, SIPs have no run-time overhead and, in fact, as this paper shows, facilitate constructing a more efficient system.

Another potential cost is the loss of language freedom, since SIPs preclude unsafe languages such as C++. This has not been a problem in Singularity, since Sing#, our slightly extended dialect of C#, is a flexible and expressive language. While we implemented Singularity in Sing#, any language with a compiler that emits type-safe MSIL could run on the system (e.g., Visual Basic, F#, or Iron Python). We note in passing that software fault isolation [42] could encapsulate unsafe code in a SIP. However, SFI is less attractive since its run-time overhead is higher than a well-implemented safe language, and SFI does not improve defect-prone features of unsafe languages.

A serious concern with SIPs, however, is the correctness of the implementation of language safety. At present, we depend on our compiler (Bartok) to verify the type safety of a program, not introduce errors during optimization, and produce run-time checks. Bartok uses a typed intermediate language [8], which helps ensure that it maintains type safety through the compilation process from typed MSIL to untyped x86 code. Nevertheless, it is

a complex, highly optimizing compiler. We are working toward removing Bartok from the trusted computing base by using typed assembly language (TAL) [18, 30]. A TAL compiler produces a proof of the type safety of compiled x86 code along with the code itself. A properly structured proof of type safety can be verified by a small, simple checker whose correctness is far easier to ensure than a compiler's. We are confident that TAL can prevent unsafe code from executing on Singularity.

SIPS are also dependent on the type safety of trusted code, such as the runtime and, most notably, the garbage collector. We are actively investigating techniques for verifying the correctness of these parts of the system.

Another concern about SIPs is the possibility of hardware faults corrupting a pointer value or a computation, which could cause a program to violate type safety guarantees and subvert its SIP. Govindavajhala and Appel showed that memory errors could be used to subvert type safety in a Java virtual machine [21]. Their recommendation, which we share, is increased use of hardware error detection and correction techniques, such as error correcting codes (ECC) on memory and data paths, to ensure the assumption of correct execution that underlies all program safety. Advances in semiconductors are making transistors less reliable, so hardware error detection and correction is likely to become more common in future processors [31].

Section 3 describes how hardware mechanisms can be selectively used to provide a level of isolation beyond SIPs.

Singularity differs from other single address operating system, such as Pilot, Cedar, Smalltalk, Lisp Machines, Oberon, or Inferno [11, 19, 34, 39, 44], which encouraged sharing objects between processes and did not segregate a process's objects. These systems presented a programming model similar to threads in a process, rather than SIPs' process-like, segmented object spaces. They also relied on garbage collection to reclaim memory when a process terminates. Garbage collection has a higher per-object overhead than reclaiming pages and can be thwarted by pointers to "dead" objects. These systems also used a single garbage collector for all processes and did not have the flexibility to select a collection algorithm appropriate to a computation.

The JX system is similar to Singularity in many respects. It is a microkernel system written almost entirely in a safe language (Java) [20]. Processes on JX do not share memory and communicate through synchronous RPC with deep copying of parameters. The processes run in a single hardware address space and rely on language safety for isolation. The primary differences between JX and Singularity are the communication and extension mechanisms. Singularity processes exchange data, not objects, which eliminates the need for communicating processes to share a common object layout and method bodies. Moreover, Singularity uses asynchronous message passing over strongly typed channels, which is more flexible than RPC, but still permits verification of communication behavior and system-wide liveness properties [33]. Finally, Singularity does not allow dynamic code loading, but instead runs extensions in their own SIP.

Other systems implemented isolation mechanisms for Java. The J-Kernel implemented protection domains in a JVM process, provided revocable capabilities to control object sharing, and developed a clean semantics for domain termination [23]. Luna refined the J-Kernel's run-time mechanisms with an extension to the Java type system that distinguishes shared data and permits control of sharing [24]. KaffeOS provides a process abstraction in a JVM along with mechanisms to control resource utilization in a group of processes [4]. Java incorporated some of these ideas into isolates [32], which are similar to AppDomains in Microsoft's CLR. Singularity differs from these systems in several ways. It eliminates the duplication of resource management and isolation mechanisms between an operating system and language runtime system by integrating the two. It also segregates a process's objects, so that a terminated process's memory can be reclaimed without garbage collection. SIPs also strictly prevent sharing, which provides a greater amount of isolation and fault tolerance and permits each process to have a fully independent language runtime system, even to the extent of different runtimes and garbage collectors.

3 Hardware Protection Domains

For Singularity, hardware protection offers a *supplemental* layer of protection beyond the isolation provided by SIPs. For example, hardware can offer a defense in depth against untrusted code that might maliciously exploit compiler or system defects. Alternatively, hardware isolation could be routinely used where it incurs little cost, for example, as a barrier between two unrelated processes that do not communicate. Conversely, by using SIPs, an application, its trusted extensions, and its child processes could share an address space and communicate efficiently, while still maintaining a redundant level of protection and failure isolation. Other common scenarios that may sensibly run within a hardware boundary are: multiple instances of the same code, different code from the same source, or code from a trusted provider.

Singularity implements hardware isolation through a *protection domain* (*domain* for short), which is a hardware-enforced protection boundary that 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 has an additional dimension: whether it isolates its SIPs from the kernel. 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), thereby simplifying transitions between the processes and the kernel. Untrusted code in these SIPs, however, is verified to ensure it does not contain privileged instructions, so it cannot masquerade as the operating system and subvert system or domain boundaries. In addition, SIPs in a kernel domain use the kernel's exchange heap rather than a heap associated with the domain.

Since non-kernel domains do not share memory, communications between these domains entails data copying, but communications within a domain continues to use reference passing. The message-passing semantics of Singularity channels makes the two implementations indistinguishable to code (except for performance). However, SIPs in kernel domains continue to communicate efficiently with the system and between each other.

A protection domain could, in principle, host a single process containing unverifiable code written in an unsafe language such as C++. We have not explored this possibility. Rather, protection domains always contain a SIP, which continues

to provide isolation and a failure boundary in case the original process code loads extensions, libraries, or other code.

Singularity protection domains can be employed selectively to provide hardware isolation between processes or between the kernel and processes. The mapping of SIPs to protection domains is determined by system policy configuration and can be changed by an administrator. Since a process always resides in a SIP, processes can be safely co-located within a protection domain. Moreover, code in a process is unaffected by the presence or absence of a protection domain, except for performance. A Singularity system with a distinct protection domain for each process is analogous to a traditional hardware-isolated microkernel system. A Singularity system with a kernel domain around the file system, network stack, device drivers, and kernel is analogous to a conventional, monolithic operating system, but is more resilient to driver or subsystem failures because each component is contained in a SIP.

3.1 x86 Virtual Memory Implementation

Virtual memory is the primary mechanism that a traditional operating system uses to enforce isolation between processes. Singularity uses virtual memory in a similar way to implement protection domains. Each set of virtual-to-physical mappings is referred to as an *address space*. On Singularity, distinct address spaces map to non-overlapping regions of physical memory, so processes in different domains cannot reference each other's memory. This section briefly describes the subset of the x86 virtual memory hardware used by Singularity.

On modern x86 processors, the operating system specifies the virtual to physical mapping with a tree of mapping descriptors rooted at a processor control register. This control register, cr3, can only be accessed by privileged code. The structure of the tree is dictated by the processor and varies slightly according to the MMU features that are enabled. Singularity runs on x86 processors that support Physical Address Extension, which allows more than 4GB of physical memory to be addressed, even in 32-bit mode. With this feature enabled, the mapping descriptors are arranged in a three-tiered tree. The root node contains four, 64-bit entries, each spans 1GB of virtual memory. A second-tier node occupies one 4KB memory page and contains 512 64-bit entries, each of which spans 2MB of virtual memory. Entries in the root and second-tier nodes contain the physical address of their child node. A leaf node contains 512 64-bit entries, each of which is the physical address of an individual 4KB memory page. Every entry also contains a variety of control bits.

When virtual memory is enabled, an address manipulated by software is *virtual*; at execution time, it must be translated to a physical RAM address. When executing any instruction that references memory, the processor translates a virtual address by using its most significant 2 bits as an index into the root node of the memory-mapping descriptor tree. The selected entry points to a second-tier node, which is indexed by the 3rd through 12th most significant bits of the virtual address. The next 9 bits are used as an index into a leaf node. The least-significant 12 bits of the virtual address are an offset into the 4KB memory page indicated by the leaf node.

Because a complete mapping tree would occupy 8MB of physical RAM, a control bits in every entry indicates the entry's validity. When this bit is cleared, the entry indicates that no physical memory is mapped to its corresponding range of virtual memory. Accessing memory in an unmapped range raises a processor page fault exception.

Performing this translation process at every memory reference is impractical, so a Translation Lookaside Buffer (TLB) caches virtual-to-physical memory mappings. On modern x86 processors, the operating system must explicitly invalidate TLB entries when it updates memory mappings, as the TLB is not automatically updated or flushed. Singularity runs on processors that provide an instruction to invalidate an individual TLB entry, and it uses that instruction whenever possible. Reloading the memory-mapping control register (which specifies the root memory-mapping descriptor node) implicitly invalidates the entire TLB. TLB invalidation incurs significant overheads, as a subsequent memory access can cause three physical memory accesses to translate the address.

Singularity only uses virtual memory for protection and does not page memory to disk.

3.2 Virtual Memory Organization and Costs

As with many other systems, Singularity organizes its memory into two main regions: the kernel range and the process, or "user," range. In Singularity, the kernel uses the *lower* range of virtual memory (beginning at address zero and ending at a configurable boundary). All processes reside in virtual memory above the kernel boundary. The user memory range in a protection domain contains different physical pages than any other protection domain, which ensures hardware-enforced isolation. Every process's address space includes an identical mapping for the kernel range of memory, which ensures that kernel data structures are always accessible, regardless of which process is running. Pages in the kernel range of memory can be marked inaccessible to unprivileged code, creating a hardware-enforced boundary between the kernel and processes in a non-kernel domain.

Singularity uses a processor feature that marks the memory-mapping entries for the kernel range of memory as "Global," which indicates that they should not be flushed from the TLB when the TLB is implicitly invalidated during an address-space switch.

Compared to the basic Singularity configuration, which only uses physical memory, satisfying a kernel or process request to allocate a (virtually) contiguous range of memory is significantly more complex. First, the kernel finds a suitable, unused range of virtual memory. Next, the kernel must find an unused physical page of memory for each page within the range and manipulate the mapping structures to map the two addresses. On a configuration that does not use virtual memory, all addresses correspond directly to physical addresses. Satisfying a memory allocation request only involves identifying a suitable range of unused physical memory. The roughly five-fold increase in memory-management cost is quantified in Section 4.1.

Virtual memory also imposes additional costs on some context switches. The Singularity scheduler saves and restores the processor context when switching between runable threads. When scheduling a thread from a different protection domain, Singularity loads the processor control register that controls the address space, which invalidates all non-global TLB entries. As a

result, all references to the user memory range incur additional memory accesses to refill the TLB. A system running a large number of protection domains (and therefore address spaces) will incur overhead due to TLB misses. This effect is quantified in Section 4.3.

3.3 Privilege Modes and Costs

Singularity can behave like most systems and ensure the integrity of the process/kernel hardware boundary by running processes outside of a kernel domain at a lower privilege level ("Ring 3" on x86 processors). Since the instructions that manipulate the virtual memory mappings are invalid at lower privilege levels, this mechanism ensures that the operating system retains sole control of memory protection. In addition, the kernel range pages are marked as inaccessible to unprivileged code, which further ensures kernel integrity.

The primary cost of the privilege mechanism arises when a process invokes the kernel. When a Singularity process is running in a kernel protection domain, and therefore executing at elevated privilege, the kernel can be invoked with the same procedure-call mechanism that a process uses for internal calls. The only additional overhead is the bookkeeping necessary to mark a transition between two garbage-collection domains. This demarcation on the stack ensures that the kernel's garbage collector does not traverse process data structures (and vice versa).

A process running in a non-kernel protection domain (at lowered privilege) incurs additional costs. It must use an instruction that calls a routine in the kernel while simultaneously and safely elevating the processor's privilege level. Singularity uses the *sysenter* / *sysexit* mechanism offered by modern x86-class processors. This pair of instructions offers a streamlined mechanism for invoking privileged code from an unprivileged context. To initialize the mechanism, the operating system (running at elevated privilege) loads a processor control register with the address of a master-dispatch routine that handles all kernel-invocation requests. Additional control registers specify the code and stack segments and stack pointer to use during a kernel invocation. The sysenter instruction elevates the processor privilege level and jumps to the code location specified in the control registers, while loading new values for the code and stack segments and stack pointer. The calling convention for the kernel also includes a mechanism to save the process's code and stack pointers and to pass arguments. In addition, the *sysenter* mechanism always jumps to a fixed location in the kernel, where the master kernel entry point incurs additional overhead in dispatching to the appropriate kernel routine. The four-fold cost increase incurred by privilege transitions is quantified in Section 4.1.

3.4 Trusted Code and Hardware Isolation

In Singularity, a process contains two distinct categories of code: *untrusted* and *trusted* code. All application code is untrusted (and verified safe), while portions of the Singularity-provided runtime are trusted. Trusted code is permitted to access system data structures, such as the allocation records for the global exchange heap. This code is expected to preserve the integrity of these structures, which is distinct from language safety and is far more difficult to verify. Trusted code may not conform to the language type safety rules, although we strive to minimize the unsafe, trusted code, as it is more difficult to ensure the correctness of unsafe code.

In a SIP-only system, trusted code is inserted directly into an application. Trusted code can be generated by a compiler or linked as part of system-supplied libraries at install time. This code can share a SIP with untrusted code, as the integrity of the trusted code's data structures will not be compromised by *untrusted* (but safe) application code, which, because it obeys the type system, cannot access these structures.

Integrating trusted code into a SIP is a very flexible and useful technique, because in many cases, trusted code in a process can eliminate the need to invoke the kernel to carry out an operation. The kernel-invocation mechanism in a kernel domain does not require switching the processor privilege mode, but demarking the application's garbage-collection domain from the kernel increases the cost of the call to 80 clock cycles (versus 9 cycles for a normal procedure call and 365–616 cycles for a kernel invocation). Moving a simple operation from the kernel to trusted code in a process eliminates even this kernel-invocation overhead and allows the compiler to further reduce overhead by in-lining the trusted code and eliminating the procedure call. One important application of this approach in Singularity is I/O objects, which are trusted objects that the kernel allocates and passes to device drivers, to encapsulate and regulate their access to memory-mapped I/O control registers [37].

Protection domains restrict trusted — as well as untrusted — code, which can no longer directly access data structures that reside in other protection domains, including the kernel. Ordinary process code usually does not run at elevated privilege. Unfortunately, trusted runtime code cannot access system data structures when executing at lowered privilege. We identified two solutions. The first is to keep trusted code physically distinct, situate it on separate memory pages, and find a technique to elevate its privilege level upon invocation. We abandoned this approach because of its complexity. Instead, we used the more conventional approach of moving this code into the kernel. As discussed below, this solution significantly increases the cost of operations across protection boundaries, including inter-process communications.

3.5 Interprocess Communication

Singularity allows the efficient exchange of structured data (but not objects) through its communication channels. Message passing within a protection domain is inexpensive since the system exploits language-level invariants and a common exchange heap to avoid copying data between sender and receiver.

Our C# language dialect imposes a *linearity* constraint on data in the exchange heap: exactly one reference exists to each object [14]. This constraint facilitates static checking of application code to verify that it respects the protocol for sending a message, which requires the sender to relinquish all references at a send. This invariant enables Singularity processes in the same domain to exchange data without copying, since the sending process cannot read or modify the data after it is sent (and cannot distinguish copying from pointer passing). Entire trees of data can be safely moved solely by reference passing, due to the guarantee that no references may exist to data within the tree at the moment the tree is transmitted. In effect, data sent between SIPs is

transferred with the same mechanism (pointer-passing) used to transfer data between two procedures — albeit with a slightly more complex mechanism to coordinate the transfer between two asynchronously executing processes.

Communication channels are comprised of two endpoints, each owned by exactly one process (both can be owned by the same process). In the basic version of Singularity that uses SIPs, a channel endpoint is represented by a data structure in the exchange heap. The transmitting process writes directly into its peer's endpoint structure to signal the arrival of data (which may include references to blocks in their shared exchange heap). The combination of checked, safe code and carefully written, trusted code ensures that the sending process relinquishes its references to the transmitted data and that the integrity of the receiving process's endpoint structure is maintained.

In a Singularity system configured with protection domains, each non-kernel domain contains its own exchange heap, to ensure that the domain is a proper failure boundary. The distinct address spaces and physical separation of exchange heaps means that data can no longer be transferred across domains by reference passing. Rather, the kernel assists in validating and copying data from one domain to another.

When a channel spans a non-kernel protection domain, the target endpoint structure is not accessible to the transmitting process. Indeed, to protect key fields of the endpoint structure from corruption, the endpoint structure is divided into two pieces and the sensitive portions located in kernel memory. In addition, the allocation records for a communication heap are placed on separate pages from the heap's data blocks and are marked read-only for unprivileged code.

To accomplish a cross-protection-domain message send, the sending process first writes the data into a structure in its own exchange heap that serves as a proxy for the target endpoint. It then invokes the kernel to request that this data (and any exchange heap blocks to which it refers) be marshaled and copied to its peer. The kernel marshals in two steps: it first copies the data being transmitted into the kernel domain's exchange heap, taking care to validate any block pointers and to guard against concurrent deallocation. As the data is copied, it is deallocated in the sending process's exchange heap. When this process is complete, the sending process is unblocked and the receiving process is signaled.

The receiving process may or may not have been blocked waiting to receive data on the channel. If it was blocked, the marshalling process occurs when the process is next scheduled. If the process was not blocked, the marshalling process occurs when the receiving process next calls the kernel to poll the channel. At that time, the kernel suspends the receiving process while it allocates data blocks in the process's exchange heap and copies the data from the kernel domain into the target domain and the receiving endpoint structure.

Obviously, this multi-step process is considerably more costly than reference passing in the intra-protection-domain case. The cost of sending a one-byte message is roughly 10 times higher, and it increases with message size to 25 times for a 32KB message (Section 4.1). An obvious optimization performed by our implementation is to avoid the copy into the kernel domain, if the sender and the kernel share the same domain, and similarly in the case the receiver shares its domain with the kernel. For blocks of page size or larger, we can also perform memory remapping instead of copying bits. These optimizations are all transparent to the sender and receiver. We are also investigating techniques to eliminate the need to copy data in two stages.

3.6 Discussion

Hardware isolated processes (HIPs) are the norm for modern operating systems, such as Unix or Windows. HIPs rely on a processor's memory management unit (MMU) to implement a distinct virtual address space for each process. Each process has a per-process binding for each virtual address, and the MMU detects references to undefined addresses. Mapping and protection are implemented by a virtual memory system, which is a combination of hardware and software whose design varies among machines [27]. Hardware for HIPs does not come for free, though its costs are diffused and difficult to quantify:

- Virtual memory systems (with the exception of software-only systems such as SPUR [46]) rely on a hardware cache of address translations to avoid accessing page tables at every processor cache miss. Managing TLB entries has a cost, which Jacob and Mudge estimated at 5–10% on a simulated MIPS-like processor [28]. The virtual memory system also brings its data, and in some systems, code as well, into a processor's caches, which evicts user code and data. Jacob and Mudge estimate that, with small caches, these induced misses can increase the overhead to 10–20%. Furthermore, they found that virtual memory induced interrupts can increase the overhead to 10–30%. Other studies found similar or even higher overheads, though the actual costs are very dependent on system details and benchmarks [3, 5, 9, 25, 35, 40, 41]. In addition, TLB access is on the critical path of many processor designs [2, 29] and so might affect processor clock speed.

- Virtual memory increases the cost of calls into the kernel and process context switches [3]. Transferring control from a process to the kernel, or visa versa, crosses a protection boundary. The reported overhead of crossing this boundary is 82 cycles on a Pentium processor [22]. In addition, on architectures with an untagged TLB, such as the x86, a process context switch require flushing the TLB, which incurs refill costs.

- HIPs provide protection at the granularity of a memory page (typically 4KB). This size is much larger than a typical object (in SPECjvm98 benchmark applications, the average ranges from 22–29,949 bytes, with most well under 50 bytes[2]) or memory-mapped I/O Port, so a protection boundary encompasses large pieces of code or entire data segments, not individual objects. Alternatives, such as Mondrian memory protection [45], provide finer granularity access control, but are not yet available.

Singularity is a microkernel operating system that differs in several respects from other microkernel systems, such as Mach, L4, SPIN, Vino, and Exokernel [1, 6, 12, 22, 36]. Microkernel operating systems partition a monolithic kernel into components that run in separate processes. Previous systems, with exception of kernel extensions for SPIN, were written in an unsafe programming language and used processor memory management hardware and protection rings as an isolation mechanism. Singularity uses language safety and message-passing

[2] Emery Berger, personal communication, 3/21/06.

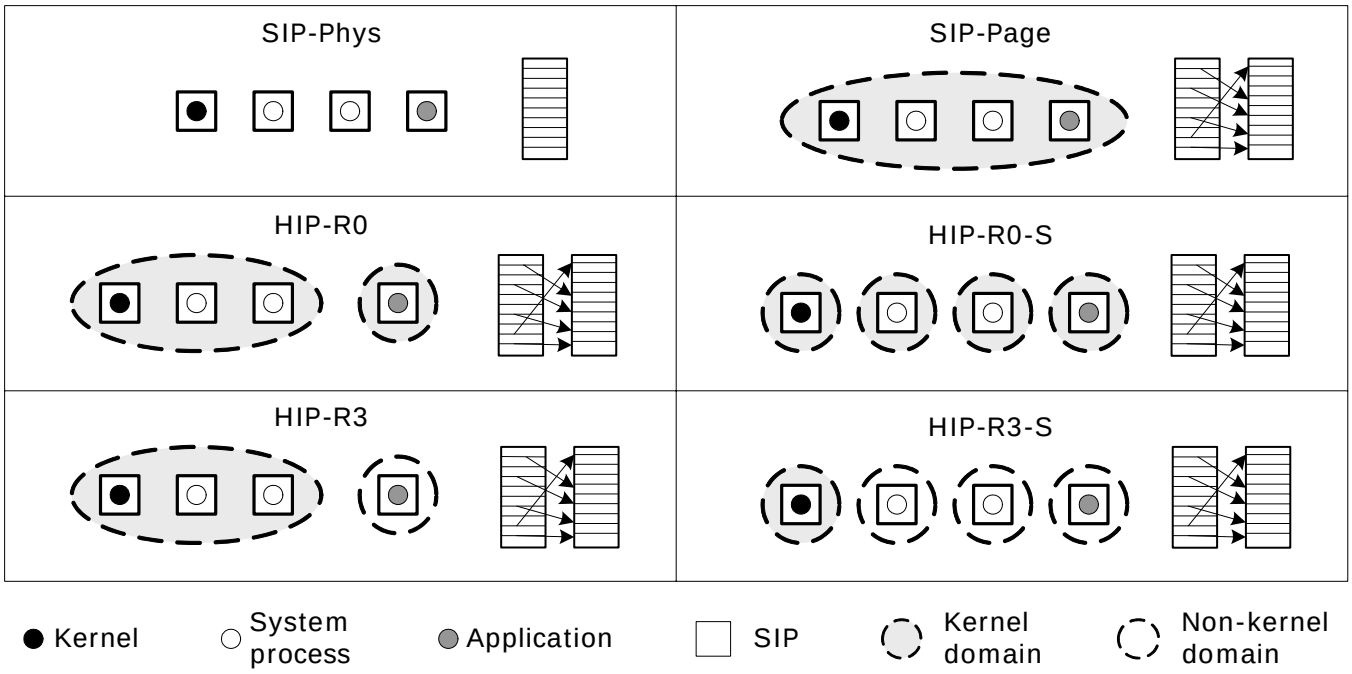

Figure 3. Measured Singularity configurations.

communication to isolate processes at a lower cost, thereby addressing an important difficulty in other microkernel systems.

Because hardware-enforced processes incur considerable overhead, microkernel systems evolved kernel extensions to lessen this cost while protecting system integrity. SPIN implemented extensions in a safe language and using programming language features to restrict access to kernel interfaces [7]. Vino used sandboxing to prevent unsafe extensions from accessing kernel code and data and lightweight transactions to control resource usage [36]. Both systems allowed extensions to directly manipulate kernel data, which left open the possibility of corruption through incorrect or malicious operations and inconsistent data after extension failure. Singularity's stronger extension model prevents data sharing between a parent and an extension. Singularity also uses a single, general extension mechanism throughout the system, from device drivers through applications, not a specialized mechanism for a kernel.

Nooks provides lightweight protection domains for Linux kernel extensions such as device drivers [38]. These domains use the MMU to restrict a driver to read-only access to the portion of kernel's address space that does not belong to a driver. Nooks also interposes instrumentation on all control and data transfers between the kernel and the driver. Drivers are written in unsafe language, so Nooks isolation mechanism is more expensive and domain-specific than SIPs.

van Doorn built a Java virtual machine that isolated classes in distinct, hardware-enforced address spaces [10]. To retain Java's semantics, it allows cross-domain pointers, which required considerable effort to maintain compatible mappings in different spaces. These pointers, and the shared language runtime and garbage collector, undermined much of the failure isolation.

4 Performance

We measured the performance of six Singularity configurations (Figure 3):

- **SIP-Phys.** The kernel and all processes execute in SIPs running in Ring 0 and accessing physical memory (the virtual memory hardware is disabled).
- **SIP-Page.** The kernel and all processes execute in SIPs running in Ring 0, with the MMU's virtual to physical mapping enabled. Performance differences relative to SIP-Phys measure the cost of manipulating the MMU and the additional cache misses to refill the TLB.
- **HIP-R0.** The kernel, device drivers, and system processes execute in one kernel protection domain. The application executes in a different kernel domain (which shares an exchange heap with the kernel). The kernel and all processes run in Ring 0. Note that code in a domain executes in a SIP, so it still incurs the overhead of enforcing language safety. Performance differences relative to SIP-Page measure the cost of changing the address space (but not privilege level) when transferring control between the application and the system.
- **HIP-R0-S.** This configuration is the same as HIP-R0, except that each process (including drivers and system processes) executes in its own kernel domain. All domains share an exchange heap. Performance differences relative to HIP-R0 measure the cost of changing the address space (but not privilege level) when transferring control between all processes and the kernel.
- **HIP-R3.** The kernel, drivers, and system processes execute in one protection domain. The application executes in a different, non-kernel protection domain (which does not share an exchange heap with the kernel). The kernel domain runs in Ring 0 and

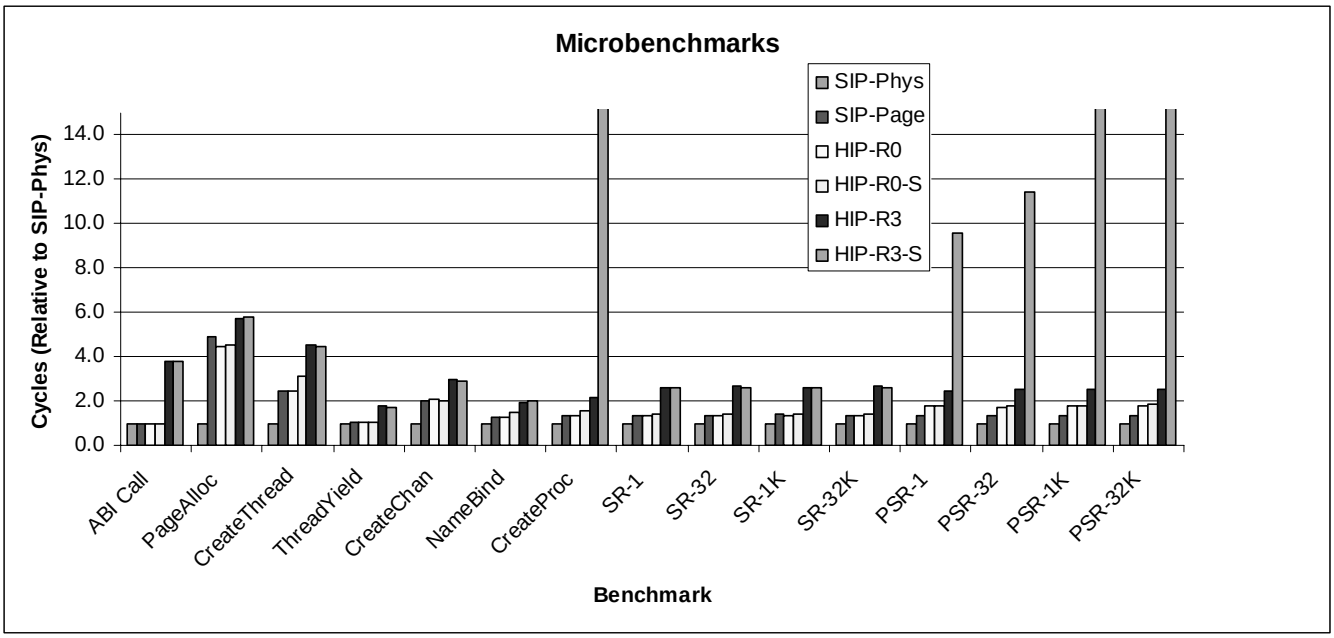

Figure 4. Microbenchmark performance.

application domain runs in Ring 3. This is comparable protection on a conventional operating system, except for the isolation provided by SIPs in the kernel domain. Performance differences relative to SIP-Phys measure the cost of conventional hardware isolation. Differences relative to HIP-R0 measure the cost of privilege levels.

- **HIP-R3-S.** This configuration is the same as HIP-R3, except that each process (including drivers and system processes) runs in its own unprivileged, non-kernel domain in Ring 3. The kernel runs in a kernel domain in Ring 0. Each domain has its own exchange heap. This configuration is comparable to the protection on a traditional, hardware-isolated micro-kernel operating system.

All configurations ran on AMD Athlon 64 3000+ (1.8 GHz) with an NVIDIA nForce4 Ultra chipset, 1GB RAM, a Western Digital WD2500JD 250GB 7200RPM SATA disk (without command queuing), and the nForce4 Ultra native Gigabit NIC (without hardware TCP offload acceleration). Singularity ran with a non-concurrent mark-sweep collector in both the kernel and processes (including drivers) and a simple round-robin scheduler.

4.1 Micro Benchmarks

The micro benchmarks (Figure 4) report the cost of low-level systems operations, relative to the SIP-Phys configuration. The "ABICall" benchmark measures the cost of a simple call on the Singularity kernel. The "PageAlloc" benchmark measures the cost of allocating and committing one page of memory. The "CreateThread" benchmark measures the cost of creating a thread in an existing process. The "ThreadYield" benchmark measures the cost of scheduling a thread in a process. The "CreateChannel" benchmark measures the cost of creating a channel. The "NameBind" benchmark measures the cost of binding a channel to a name in the system's name server. The "CreateProc" benchmark measures the cost of creating a process. The "SR" benchmarks measures the cost of sending and receiving a message (1, 32, 1K, and 32K bytes, respectively) between two threads in the same process. The "PSR" benchmarks measures the same scenario between threads in distinct processes.

Numbers are the average time (in cycles) to execute a test 20,000 times, except ABICall, PageAlloc, and CreateProc, which executed 10,000, 1,000, and 100 times, respectively.

The cost of these benchmarks increase dramatically when virtual memory and processor privilege levels are implemented (SIP-Page and HIP-R3, respectively). The performance effects of separate address spaces are minor. For example, the ABICall benchmark shows that privilege levels increase the cost of a process-kernel transition by a factor of 3.8 (80→304 cycles: SIP-Page→HIP-R3). Similarly, the PageAlloc benchmark shows that maintaining page tables increases the cost of allocating a page of memory by a factor of 4.9 (385→1876: SIP-Phys→SIP-Page). Processor privilege levels further increase this cost by 30% (2,198: HIP-R3).

For other benchmarks, page tables increase the cost of creating a thread by a factor of 2.4 (16,933→41,406 cycles: SIP-Phys→SIP-Page), creating a channel by a factor of 2 (4,007→7,940), binding a channel by 24% (48,251→39,067), and creating a process by 35% (388,162→522,727). Changing address spaces has relatively little effect on these benchmarks, except that name binding increases in cost by 18% (57,276→48,381: HIP-R0→HIP-R0-S) and process creation by 14% (593,418→520,473) when each process runs in its own domain. Implementing privilege levels has a larger effect. Creating a thread becomes 83% more expensive (41,616→76,188: HIP-R0→HIP-R3), creating a channel becomes 90% more expensive (8,192→11,784), name binding 81% more expensive (48,381→75,819), and process creation 60% more expensive (520,473→830,999).

The cost of communication is dependent on the system configuration and message size. Implementing virtual memory

increases the cost of a one-byte intra-process communication by approximately 33% (984→1,306 cycles: SIP-Phys→SIP-Page) and a one-byte inter-process communication by 36% (1,041→1,415 cycles: SIP-Phys→SIP-Page). Separate address spaces in kernel domains have relatively little effect on communication cost. However, implementing privilege levels increases the cost a one-byte intra-process communications by 94% (1,312→2,534: HIP-R0→HIP-R3) and a one-byte inter-process communication costs by 41% (1,826→2,580: HIP-R0→HIP-R3). The effect of copying data, as opposed to pointer passing, is illustrated by interprocess communication in HIP-R3-S, which is 3.9 – 9.6 times as expensive as HIP-R3, in which the communicating processes are in the same protection domain.

4.2 Other Systems

To validate these results, we compared simple operations on Singularity and several other operating systems. We used FreeBSD 5.3, Red Hat Fedora Core 4 (kernel version 2.6.11-1.1369_FC4), and Windows XP (SP2).

Table 1 reports the cost of the basic operations in Singularity and three other operating systems. On the Unix systems, the ABI call was `clock_getres()`, on Windows, it was `SetFilePointer()`, and on Singularity, it was `ProcessService.GetCyclesPerSecond()`. All these calls operate on a readily available data structure in the respective kernels. The Unix thread tests ran on user-space scheduled pthreads. Kernel scheduled threads performed significantly worse. The "PSR-1" measured the cost of sending a 1-byte message from one process to another and then back to the original process. On Unix, we used sockets, on Windows, a named pipe, and on Singularity, a channel.

	Cost (CPU Cycles)			
	ABI Call	Thread yield	PSR-1	Create Proc
Singularity SIP-Phys	80	365	1,041	388,162
Singularity HIP-R3	304	638	2,580	830,999
FreeBSD	878	911	13,304	1,032,254
Linux	437	906	5,797	719,447
Windows	627	753	6,344	5,375,735

Table 1. Cost of basic operations.

Basic thread operations in Singularity, such as yielding the processor or synchronizing two threads, are comparable or slightly faster than the other systems. Nevertheless, because of Singularity's SIP architecture, cross-process operations run significantly faster than in the mature systems. Calls from a process to the kernel are 5–10 times faster on Singularity, since the call does not cross a hardware protection boundary. A simple RPC-like interaction between two processes is 4–9 times faster. And, creating a process is 2–14 times faster than the other systems.

Singularity is a new system and its performance has not been heavily tuned. Nevertheless, this comparison shows that Singularity's implementation is competitive with commercial operating systems, and helps validate the Singularity measurements

4.3 Macro Benchmarks

We used two macro benchmarks to measure the overall performance of the various system configurations. The first benchmark ("Bartok") was an execution of the Bartok compiler compiling the Singularity kernel. The kernel is approximately 165,000 lines of code (1.6MB of MSIL) and compiles in approximately 40 seconds. The compiler is a single process that uses hundreds of megabytes of memory, so this benchmark consists of a single memory-intensive process with little interprocess communication. The other benchmark ("WebFiles") is a synthetic program that replays the file accesses that occur during the SPECWeb99 benchmark. It consists of a modified SPECWeb client that directly accesses the file system, without invoking a web server. This benchmark reads 50,000 files consuming 50.3MB of disk space. On Singularity, a file read involves several processes (the application, the file system, and the disk driver), so this benchmark measures a communication-intensive collection of processes.

Figure 5 reports the performance metrics for these two benchmarks under the different configurations. The bars labeled "Cycles" report their execution times, in processor clock cycles, relative to SIP-Phys (70 and 22.9 billion cycles for Bartok and WebFiles, respectively). "CPI" reports the average processor cycles per instruction, relative to SIP-Phys (1.29 and 1.30 CPI, respectively). "Insts Retired" reports the number of executed instructions, again relative to SIP-Phys (54.4 and 17.6 million instructions, respectively). "TLB_L2_Requests" reports the number of L2 cache misses that result from TLB refills, this time relative to SIP-Page (66.0 and 20.4 million, respectively).

The compute-intensive Bartok benchmark incurs an overhead of approximately 2.5% (in cycles) due to TLB misses in all system configurations, except the base configuration, SIP-Phys, which does not use the TLB. The non-SIP-Phys configurations incur a TLB miss approximately every 1080 instructions and execute approximately 2% more instructions than the base configuration. The cost of processing these misses is unaffected by the protection mechanisms, since most of this benchmark executes in a single process.

The picture is very different for the other benchmark. The performance of WebFiles is considerably reduced by hardware protection boundaries, because this benchmark heavily exercises context switching and communications. The SIP-Page configuration runs 6.3% slower than the baseline, as it executes 8% more instructions and has 20 million caches misses due to TLB refills. Adding protection domains (HIP-R0) decreases performance by another 12.6%., relative to SIP-Phys. The number of instructions does not change over SIP-Page, but cache misses due to TLB refills increase by a factor of almost 6 times. Adding privilege levels (HIP-R3) increases the execution time by another 14.0% relative to SIP-Phys, which is a combination of executing 11.9% more instructions and 16.8% more cache misses. Overall, HIP-R3 executes 33.0% more cycles than SIP-Phys and 25.1% more cycles than SIP-Page. Isolating every process in its own domain (the microkernel solution) further increases the hardware overhead. HIP-R3-S executes 37.7% more cycles than SIP-Phys and 29.6% more cycles than SIP-Page.

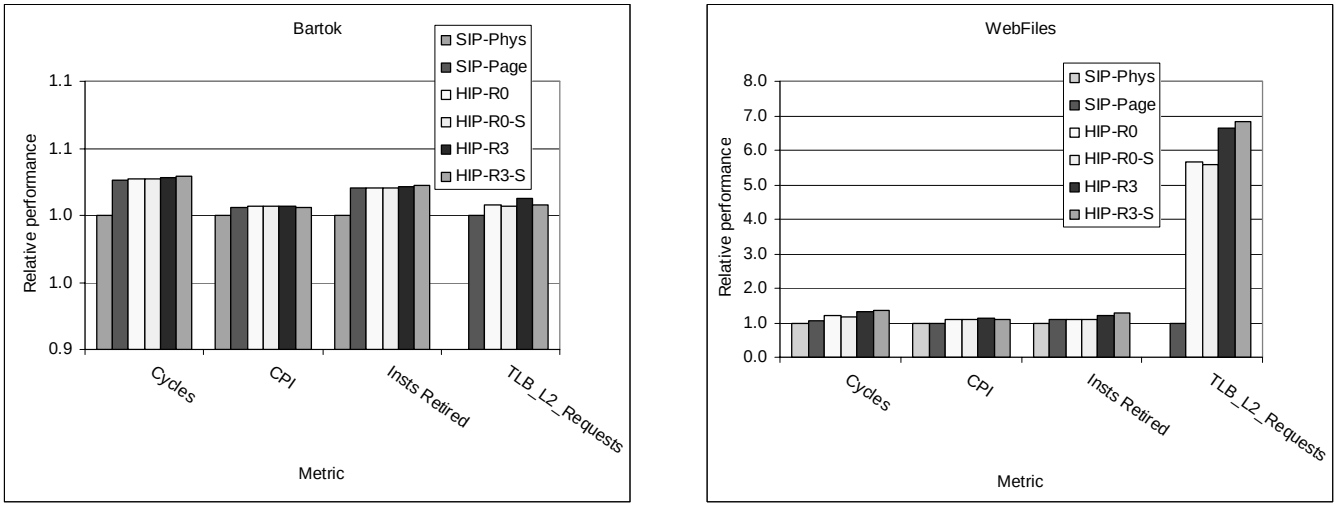

Figure 5. Macrobenchmark performance. Metrics are relative to SIP-Phys, except for TLB cache misses, which are relative to SIP-Page.

4.4 Safety Overhead

To measure the overhead cost of the run-time tests needed to ensure language safety for SIPs, we prevented the Bartok compiler from generating these tests and reran the two macro benchmarks. These tests were part of the code generated for array references, pointer dereferences, value unboxing, and type casts. Without these safety checks, the Bartok benchmark executed 4.5% fewer cycles and the WebFiles benchmark executed 4.7% fewer cycles. The run-time overhead of language safety is slightly higher than hardware isolation for the Bartok benchmark and far lower than hardware isolation for WebFiles. However, language safety offers important benefits not provided by hardware process protection, for example, detecting in-process errors such buffer overruns.

5 Conclusion

Virtual memory hardware is a powerful, multi-faceted mechanism, which originally permitted the implementation of demand paging in an era of small memories and eventually became the default implementation for process isolation. Although most operating systems implement isolation with this mechanism, its limitations, both in performance and protection granularity, make alternative protection mechanisms worth considering. This paper describes two new mechanisms and compares them directly against conventional systems. A software isolated process (SIP) is a process whose boundaries are established by language safety rules and enforced by static type checking. With proper system support, SIPs can provide a low cost isolation mechanism that provides failure isolation and fast inter-process communication. Protection domains are hardware-enforced address spaces, which can contain one or more SIPs. Domains can either run at the kernel's privilege levels and share an exchange heap or be fully isolated from the kernel and run at the normal application privilege level.

These two mechanisms can be flexibly combined in hybrid arrangements that balance the perceived security of hardware isolation against its measured costs. As software isolation mechanisms mature and developers become more familiar with them, we anticipate that perceived advantages of hardware isolation will lessen and developers will become more comfortable with software isolation. In the meantime, Singularity allows hybrid system architectures, such as encapsulating device drivers in SIPs in the kernel domain, which can increase system robustness and reliability by providing more memory and failure isolation.

Alternatively, this paper can be read as identifying an opportunity to revisit the design of MMUs, to find new mechanisms that provide isolation at lower cost.

Acknowledgements. Many thanks to Paul Barham, Tim Harris, and Rebecca Isaacs for their perceptive comments and assistance with this paper.

6 References

1. Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. A New Kernel Foundation for UNIX Development. in *Summer USENIX Conference*, Atlanta, GA, 1986, 93-112.

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*, Santa Clara, CA, 1991, 108-120.

4. Back, G., Hsieh, W.C. and Lepreau, J. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. in *Proceedings of the 4th USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, San Diego, CA, 2000.

5. Bala, K., Kaashoek, M.F. and Weihl, W.E. Software Prefetching and Caching for Translation Lookaside Buffers. in *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, 1994, 243-253.

6. Bershad, B.N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S. and Sirer, E.G. SPIN: An Extensible Microkernel for Application-specific Operating System Services. in *Proceedings of the 6th ACM SIGOPS European Workshop*, Wadern, Germany, 1994, 68-71.

7. Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M., Becker, D., Eggers, S. and Chambers, C. Extensibility, Safety and Performance in the SPIN Operating System. in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, 1995, 267-284.

8. Chen, J. and Tardit, 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)*, Long Beach, CA, 2005, 38-49.

9. Chen, J.B., Borg, A. and Jouppi, N.P. A Simulation Based Study of TLB Performance. in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, Queensland, Australia, 1992, 114-123.

10. Doorn, L.v. A Secure Java™ Virtual Machine. in *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, 2000, 19-34.

11. Dorward, S., Pike, R., Presotto, D.L., Ritchie, D.M., Trickey, H. and Winterbottom, P. The Inferno Operating System. *Bell Labs Technical Journal*, *2* (1), 1997, 5-18.

12. Engler, D.R., Kaashoek, M.F. and O'Toole, J., Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, 1995, 251-266.

13. Erlingsson, Ú. and MacCormick, J., Ad hoc Extensibility and Access Control. Report MSR-TR-2005-143, Microsoft Research, 2005.

14. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R. and Levi, S. Language Support for Fast and Reliable Message Based Communication in Singularity OS. in *To appear: EuroSys2006*, Leuven, Belgium, 2005.

15. Fitzgerald, R., Knoblock, T.B., Ruf, E., Steensgaard, B. and Tarditi, D. Marmot: an Optimizing Compiler for Java. *Software-Practice and Experience*, *30* (3), 2000, 199-232.

16. Fitzgerald, R. and Tarditi, D. The Case for Profile-directed Selection of Garbage Collectors. in *Proceedings of the 2nd International Symposium on Memory Management (ISMM '00)*, Minneapolis, MN, 2000, 111-120.

17. Flatt, M. and Findler, R.B. Kill-safe Synchronization Abstractions. in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI 04)*, Washington, DC, 2004, 47-58.

18. Glew, N. and Morrisett, G. Type-safe Linking and Modular Assembly Language. in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, TX, 1999, 250-261.

19. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

20. Golm, M., Felser, M., Wawersich, C. and Kleinoeder, J. The JX Operating System. in *Proceedings of the USENIX 2002 Annual Conference*, Monterey, CA, 2002, 45-58.

21. Govindavajhala, S. and Appel, A.W. Using Memory Errors to Attack a Virtual Machine. in *Proceedings of the 2003 Symposium on Security and Privacy*, Oakland, CA, 2003, 154-165.

22. Härtig, H., Hohmuth, M., Liedtke, J. and Schönberg, S. The Performance of µ-kernel-based Systems. in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint Malo, France, 1997, 66-77.

23. Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D. and Eicken, T.v. Implementing Multiple Protection Domains in Java. in *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998, 259-270.

24. Hawblitzel, C. and Eicken, T.v. Luna: A Flexible Java Protection System. in *Proceedings of the Fifth ACM Symposium on Operating System Design and Implementation (OSDI '02)*, Boston, MA, 2002, 391-402.

25. Huck, J. and Hays, J. Architectural Support for Translation Table Management in Large Address Space Machines. in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*, San Diego, CA, 1993, 29-50.

26. 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. Report MSR-TR-2005-135, Microsoft Research, 2005.

27. Jacob, B. and Mudge, T. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*, *18* (4), 1998, 60-75.

28. Jacob, B.L. and Mudge, T.N. A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations. in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, San Jose, CA, 1998, 295-306.

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

30. Morrisett, G., Walker, D., Crary, K. and Glew, N. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, *21* (3), 1999, 527-568.

31. Mukherjee, S.S., Weaver, C.T., Emer, J., Reinhardt, S.K. and Austin, T. Measuring Architectural Vulnerability Factors. *IEEE Micro*, *23* (6), 2003, 70-75.

32. Process, J.C. Application Isolation API Specification *Java Specification Request*, 2003, JSR-000121.

33. Rajamani, S.K. and Rehof, J. Conformance Checking for Models of Asynchronous Message Passing Software. in *Proceedings of the International Conference on Computer Aided Verification (CAV 02)*, Springer, Copenhagen, Denmark, 2002, 166-179.

34. Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., McJones, P.R., Murray, H.G. and Purcell, S.C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, *23* (2), 1980, 81-92.

35. Rosenblum, M., Bugnion, E., Herrod, S.A., Witchel, E. and Gupta, A. The Impact of Architectural Trends on Operating System Performance. in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, 1995, 285-298.

36. Seltzer, M.I., Endo, Y., Small, C. and Smith, K.A. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI 96)*, Seattle, WA, 1996, 213-227.

37. Spear, M.F., Roeder, T., Levi, S. and Hunt, G. Solving the Starting Problem: Device Drivers as Self-Describing Artifacts. in *Proceedings of the EuroSys 2006 Conference*, Leuven, Belgium, 2006, 45-58.

38. Swift, M.M., Bershad, B.N. and Levy, H.M. Improving the Reliability of Commodity Operating Systems. in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, 2003, 207-222.

39. Swinehart, D.C., Zellweger, P.T., Beach, R.J. and Hagmann, R.B. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, *8* (4), 1986, 419-490.

40. Talluri, M. and Hill, M.D. Surpassing the TLB Performance of Superpages with Less Operating System Support. in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 1994, 171-182.

41. Uhlig, R., Nagle, D., Stanley, T., Mudge, T., Sechrest, S. and Brown, R. Design Tradeoffs for Software-Managed TLBs. *ACM Transactions on Computer Systems*, *12* (3), 1994, 175-205.

42. Wahbe, R., Lucco, S., Anderson, T.E. and Graham, S.L. Efficient Software-Based Fault Isolation. in *Proceedings of the Fourteenth ACM*

Symposium on Operating System Principles, Asheville, NC, 1993, 203-216.

43. Wang, D.C. and Appel, A.W. Type-preserving Garbage Collectors. in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, Berlin, Germany, 2002, 166-178.

44. Weinreb, D. and Moon, D. *Lisp Machine Manuel*. Symbolics, Inc, Cambridge, MA, 1981.

45. Witchel, E., Cates, J. and Asanovic', K. Mondrian Memory Protection. in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 2002, 304-316.

46. Wood, D.A., Eggers, S.J., Gibson, G., Hill, M.D., Pendleton, J., Ritchie, S.A., Katz, R.H. and Patterson, D.A. An In-Cache Address Translation Mechanism. in *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture*, Tokyo, Japan, 1986, 158-166.