Application-Specific Service Technologies for Commodity Operating Systems in Real-Time Environments

RICHARD WEST and GABRIEL PARMER, Boston University

In order to eliminate the costs of proprietary systems and special purpose hardware, many real-time and embedded computing platforms are being built on commodity operating systems and generic hardware. Unfortunately, many such systems are ill-suited to the low-latency and predictable timing requirements of real-time applications. This article, therefore, focuses on application-specific service technologies for low-cost commodity operating systems and hardware, so that real-time service guarantees can be met. We describe contrasting methods to deploy *first-class services* on commodity systems that are dispatched with low latency and execute asynchronously according to bounds on CPU, memory, and I/O device usage. Specifically, we present a "user-level sandboxing" (ULS) mechanism that relies on hardware protection to isolate application-specific services from the core kernel. This approach is compared with a hybrid language and runtime protection scheme, called *SafeX*, that allows untrusted services to be dynamically linked and loaded into a base kernel. SafeX and ULS have been implemented on commodity Linux systems. Experimental results have shown—that both approaches are capable of reducing service violations (and, hence, better qualities of service) for real-time tasks, compared to traditional user-level methods of service deployment in process-private address spaces. ULS imposes minimal additional overheads on service dispatch latency compared to SafeX, with the advantage that it does not require application-specific services to execute in the trusted kernel domain. As evidence of the potential capabilities of ULS, we show how a user-level networking stack can be implemented to avoid data copying via the kernel and allow packet processing without explicit process scheduling. This improves throughput and reduces jitter.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*

General Terms: Design, Experimentation

Additional Key Words and Phrases: System extensibility, predictability

ACM Reference Format:
West, R. and Parmer, G. 2011. Application-specific service technologies for commodity operating systems in real-time environments. ACM Trans. Embedd. Comput. Syst. 10, 3, Article 30 (April 2011), 21 pages.
DOI = 10.1145/1952522.1952523 http://doi.acm.org/10.1145/1952522.1952523

1. INTRODUCTION

Recent trends have seen the use of commercial off-the-shelf (COTS) systems and hardware being deployed in real-time and embedded computing environments. For example, systems such as Linux are now being deployed in real-time and embedded settings [Yodaiken and Barabanov 1997]. Not only does this lead to a reduction in development and operation costs but also it enables a common code base for systems software to be used in both special- and general-purpose computing. However, COTS systems (e.g.,

Authors' addresses: R. West and G. Parmer, Computer Science Department, Boston University, 111 Cummington Street, Boston, MA 02215. email: {richwest, gabep1}@cs.bu.edu.

DOI 10.1145/1952522.1952523 http://doi.acm.org/10.1145/1952522.1952523

Linux) are typically ill suited to the needs of specific applications, especially when they must operate with real-time requirements.

The protection afforded by commodity operating systems usually restricts applications to process-private address spaces via which system calls can be made to access more privileged services of the trusted kernel. While the process model has many virtues, it incurs significant overheads due to scheduling, context switching, and interprocess communication. Similarly, there is often a "semantic gap" between the requirements of code that executes at user-level and the interface via which requests are made for kernel-level services. For example, to implement a real-time monitoring and control application may require one or more processes to be executed periodically and/or may involve various tasks to respond to asynchronous events (e.g., hardware interrupts) with bounded latency. Using traditional system calls to establish handlers for kernel events (e.g., via signals) and to ensure processes execute within specific deadlines is cumbersome at best, but more typically does not guarantee the necessary responsiveness of real-time applications. For this reason, we have been developing various mechanisms to support application-specific services on commodity operating systems that can be activated with low latency and executed according to strict timing requirements without the need for scheduling and context-switching between process-private address spaces.

In our original work on support for application-specific service extensions of commodity operating systems, we developed *SafeX* [West and Gloudon 2002]. SafeX is a hybrid language and runtime approach, supporting kernel-level service extensions with quality-of-service (QoS) requirements. Extensions are written in a type-safe language and restricted on the range of memory addresses they may access. By dynamically linking them into a running kernel, they can be used to affect service management decisions, by monitoring and adapting resource usage on behalf of specific applications.

While SafeX enables applications to bridge the semantic gap between their needs and the provisions of the underlying system, it conflicts with one of the basic philosophies of good system design. For many years, system designers have considered the idea of a kernel to be the address space in which only the most trusted and fundamental services should reside. For this reason, we have taken lessons learned from SafeX to develop a new mechanism for deploying "first-class" application-specific services and handlers at the user level. The idea behind first-class user-level services is to grant them (where possible) the same privileges and capabilities of kernel services, with the exception that the kernel may revoke access rights to any services abusing their privileges.

With this vision in mind, this article compares our *user-level sandboxing* (ULS) scheme against SafeX for the purpose of implementing real-time and asynchronous services and handlers on commodity operating systems such as Linux. We show how user-level services may be dispatched with almost the same latency as kernel-level interrupt handlers [Wallach et al. 1997], while also being executed without scheduling and context-switching overheads associated with processes. In fact, both SafeX and ULS ensure that service extensions are invoked (when necessary) without being at the mercy of kernel-level scheduling policies that are inherently non-real-time, or which may result in unbounded delays. For example, in many traditional systems, user-level processes may register signal handlers to be invoked when specific kernel events occur, but these handlers only run when the corresponding process is scheduled and that may be after an arbitrary amount of time.

In contrast, our approaches enable an application process, P_i, to register a first-class handler that responds to for example, timer interrupts without P_i having to execute. Given our ability to bound the dispatch latency of application-specific services, we show empirically the improved service guarantees and reduced violation rates of both ULS and SafeX compared with alternative user-level methods of implementing application-specific services. Specifically, we compare user- and kernel-level implementations of a feedback-control service, for managing the allocation of CPU cycles to application processes according to their resource requirements over finite windows of real time. Such a service might be beneficial to multimedia applications requiring specific CPU shares at designated time intervals to encode/decode audio and video streams. Alternatively, a control application may wish to guarantee that the correct share of CPU time is available to process sensor readings in a timely fashion.

From experiments, we show that ULS and SafeX are low-cost mechanisms for the timely and predictable execution of application-configurable services and handlers on commodity operating systems. A series of adaptive CPU service management tests on Linux shows that ULS handlers and SafeX kernel extensions can reduce deadline miss rates by a factor of 4, compared to process/thread-based methods of service execution. ULS and SafeX bridge the gap between the agnostic services of general-purpose systems and the needs of individual applications, including those with real-time requirements. Unlike SafeX, ULS does not require the core kernel to be polluted with potentially unsafe code that may jeopardize the integrity of the system and, therefore, its ability to meet service guarantees. The potential capabilities of ULS are highlighted with the implementation of a user-space networking stack that improves throughput and reduces jitter when packet forwarding, without the need for special hardware support.

In the next section we provide a brief overview of our prior work on SafeX, followed in Section 3 by further details about the ULS method to deploy first-class services. The performance benefits of ULS and SafeX are evaluated empirically in Section 4. Related work is discussed in Section 5, while conclusions and future work are outlined in Section 6.

2. SAFEX SUPPORT FOR FIRST-CLASS SERVICES

SafeX and ULS represent two disparate methods for achieving some sense of isolation between first-class services and the rest of the system, while providing a predictable and efficient execution mechanism. When striving for performance and predictability, both methods utilize comparable strategies. Namely, both allow execution in the context of a “bottom half”[1] for low-latency response to events, and both mitigate the costs of context switches. Likewise, both approaches employ similar runtime checks to ensure CPU isolation and fairness. However, SafeX relies entirely on language-level software techniques to provide memory protection while ULS isolates its services *outside* the kernel. A brief summary of the SafeX approach now follows, with further details available in our earlier work [West and Gloudon 2002].

Language support. SafeX requires that service extensions be written in the Popcorn [Morrisett et al. 1999a] programming language. Popcorn is designed for syntactic similarity to C, and is compiled to TALx86, an extended version of the Intel IA-32 assembly language. TALx86 is an instance of a typed assembly language (TAL) [Morrisett et al. 1999a] that, by adding typing annotations and typing rules to traditional assembly language, guarantees the memory, control flow, and type safety of TAL programs. Popcorn is supported by a number of TALx86 tools that can verify internal type consistency of TALx86 source files and linked object code.

Memory protection. Ordinarily, extensions running within the kernel address space may access and modify any data in the system, potentially affecting system integrity and violating the memory protection enforced on user processes. The type safety of Popcorn prevents extension code from forging pointers to arbitrary addresses or casting

[1] A *bottom half* is functionality required in response to an interrupt that may be processed at a convenient time, rather than immediately when the interrupt occurs.

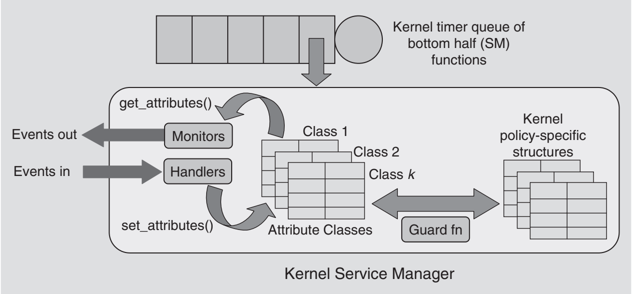


Fig. 1. The internals of a SafeX kernel service manager.

pointers to arbitrary types. Therefore, by controlling the pointers passed to extension code, the parts of the kernel address space that may be accessed by an extension can be finely controlled.

Another issue raised by passing pointers to extensions is the possibility that memory referenced by a pointer may be deallocated or reused by the core kernel. Extension code cannot be trusted to stop using pointers to such memory after reuse or deallocation. Consequently, some form of garbage collection must be used to safely manage memory referenced by extensions. The current safe extensions implementation does not do such garbage collection, but defers deallocation of memory objects until all extensions referencing them are unloaded from the kernel's address space.

CPU protection. Extension code may potentially execute for unbounded periods of time, taking control of the system. SafeX requires that applications reserve CPU time for extensions before they are executed. SafeX enforces time limits by aborting the execution of extension code that exceeds its reservation. In this way, SafeX can limit the total amount of CPU time given to and used by extensions. The CPU time used by an extension is charged to the associated application so that the total CPU time consumed on behalf of the application is considered in its scheduling. SafeX tracks extension execution time by decrementing a counter at each system timer interrupt.

SafeX service managers. SafeX allows service managers (see Figure 1) to be defined within a kernel, to manage service of a specific nature (e.g., to control scheduling and synchronization of threads on available CPUs, or to control memory allocation). Each service manager enqueues and invokes application-specific monitoring and handling functions that are able to observe actual service levels and, consequently, affect service changes. Monitor and handler functions operate on *attribute classes*. These are data structures that hold the names of various service attributes and their corresponding values (e.g., a CPU scheduling priority and its corresponding value). Service extensions get and set these attributes by name, as long as they have the necessary access rights.

Guard functions. Each service manager is equipped with a *guard function* that is automatically generated by the code generator in a SafeX daemon process running on the same host. A guard function is responsible for the mapping of attributes, contained in attribute classes, to kernel policy-specific structures. It ensures that attributes are within valid ranges and will not adversely affect the QoS guarantees to the corresponding application, or to other applications. Moreover, each SafeX daemon is capable of generating code for runtime safety checks of extensions, thereby guaranteeing they have bounded execution time.

SafeX interfaces. To affect changes to the service received by an application, the handlers need interfaces to adjust the parameters of the underlying mechanism providing the service. Though handlers execute within the kernel address space, they cannot be trusted to directly modify core kernel data. SafeX, therefore, provides service extensions with interfaces to manipulate kernel data structures and perform operations requiring special privileges (e.g., for synchronization purposes, so that interrupts are not inadvertently disabled). SafeX interface functions may be used only by services possessing the capabilities for these interfaces. Such capabilities are in fact pointers which are unforgeable due to the type safety of the extension language.

SafeX interfaces, like system calls, must validate arguments passed to them by application-specific services. They must also ensure that requested operations are safe, as some operations or decisions, while not violating system protection, may have a negative effect on system performance. SafeX interfaces are therefore responsible for limiting the possible global effects of operations requested by service extensions and require careful design, balancing the degree of application control over resource allocations with a concern for system stability.

3. ULS SUPPORT FOR FIRST-CLASS SERVICES

Overview: The basic idea of *user-level sandboxing* is to modify the address space of all processes, or logical protection domains, to contain one or more shared pages of virtual addresses. The virtual address range shared by all processes provides a sandboxed memory region into which application-specific services may be mapped. Under normal operation, these shared pages will be accessible only by the kernel. However, when the kernel wishes to pass control to a service extension, it changes the privilege level of the shared page (or pages) containing the service code and data, so that it can be executed with user-level capabilities. This prevents application-specific service code from violating the integrity of the kernel, with the benefit that such code can run in the context of *any* user-space process, even one that did not register the service with the system. There is the potential for corrupt or ill-written service extension code to modify the memory area of a running process. To guard against this, we require application-specific services registered with the system to either be written by a trusted programmer, or to have additional software safety checks (e.g., using type-safe languages [Jim et al. 2002; Morrisett et al. 1999a, b] or software-based fault isolation [Wahbe et al. 1993]).

In the absence of application-specific services being written by a trusted programmer (such as a kernel developer who wishes to isolate separate services), we only require software safety checks on untrusted code mapped to the sandbox. All other application and system-level code can be written in non-type-safe languages. This differs from the approach of the SPIN system [Bershad et al. 1995] and JavaOS, which require *all* software objects to be type-safe.

3.1. Hardware Support for Memory-Safe First-Class Services

Our approach assumes that hardware provides paging (i.e., MMU) capabilities. A series of caches, most notably one or more untagged translation look-aside buffers (TLBs), is desirable but not necessary. This minimum hardware requirement is met by many processors made today including those used in embedded systems (e.g., the Intel XScale).

On many processors, switching between protection domains mapped to different pages of virtual (or linear) addresses requires switching page tables stored in main memory, and then reloading TLBs with the necessary address translations. Such course-grained protection provided at the hardware-level is becoming more undesirable as the disparity between processor and memory speeds increases [Uhlig et al. 2002]. This is certainly the case for processors that are now clocking in the gigahertz range, while main memory is accessed in the 10^8 Hz range. In practice, it is clearly desirable to keep address translations for separate protection domains in cache memory

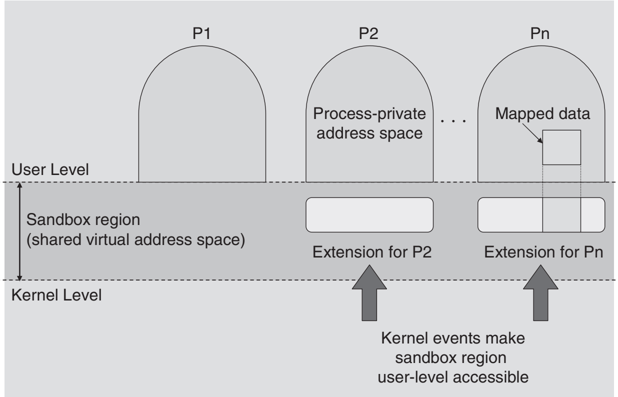

P1
P2
Pn
Process-private address space
. . .
Mapped data
User Level
Sandbox region (shared virtual address space)
Extension for P2
Extension for Pn
Kernel Level
Kernel events make sandbox region user-level accessible

Fig. 2. Each process address space has a shared virtual memory region, or sandbox, into which application-specific service extensions are mapped.

as often as possible. ULS avoids the need for expensive page table switches and TLB reloads by ensuring the sandbox is common to all address spaces.

3.2. Implementation Details

We have implemented ULS on a Linux x86-based system, with a few small changes (approximately 100 lines) to the core kernel. These changes are required to (1) create a shared sandbox region, (2) support protected mapping of a sandboxed service, (3) allow access to restricted sandboxed memory regions from conventional process address spaces, and (4) invoke application-specific services from within the kernel. The key modifications involve additional entries in the page tables (or, more precisely, directories) of processes, and the implementation of upcall code that toggles page protection bits.

For the most part, our approach is not restricted to Linux. However, where necessary, we describe the system-specific features required for user-level sandboxing to work. The user-level sandboxing implementation requires a few additional interface functions over those provided by the traditional system call interface. These interface functions are contained within kernel-loadable modules and invoked via `ioctls`, avoiding the need for new system calls.

Logical protection domains for application-specific services. Traditional operating systems provide logical protection domains for processes mapped into separate address spaces. With user-level sandboxing, as illustrated in Figure 2, each process address space is divided into two parts: a conventional process-private memory region and a shared virtual memory region. The shared region acts as a sandbox for mapped service extensions. The sandbox itself is divided into *public* and *protected* areas, as explained later, but this is not a general requirement of the approach. Kernel events delivered to sandbox code are handled in the context of the current process, thereby eliminating scheduling costs.

Sometimes it is important for a process to exchange data with services registered in the sandbox. As a result, we allow controlled access to a region of sandbox addresses by both code in a process-private region and also the sandbox.

Sandbox regions. The two areas of the sandbox (as shown in Figure 3) have the same virtual as well as physical addresses in all processes. These areas employ the page size extensions supported by the Pentium processor and each occupy one 4-MB

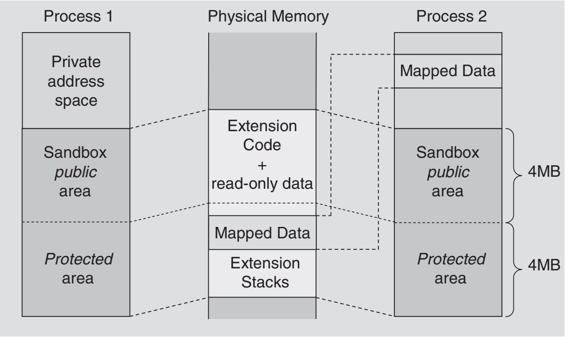


Fig. 3. Sandboxes common to all processes are mapped to the same physical address ranges. Pages of the sandbox can be mapped into process-private address spaces to exchange data.

page directory entry.[2] Although a number of MMU-enabled processors support multiple page-sizes, a sandbox should be designed to minimize the number of pages it uses while occupying the largest memory area necessary for extensions. This is to minimize the TLB footprint of extensions.

One sandbox region is permanently assigned read and execute permission at both user- and kernel-level and acts as a *public* area. The other (*protected*) region is permanently assigned read-write permission at the kernel level but, by default, is inaccessible at the user level. The protected region can be made accessible at the user level by toggling the user-supervisor flag of its page directory entry and invalidating the relevant TLB entry via the INVLPG instruction.

Sandbox/upcall threads. Sandboxed code can link with libraries that make system calls. Care must be taken that an application-specific service registered by one process does not affect the progress of another process, by issuing a blocking system call. For example, if process p_i registers an extension e_i that is invoked at the time process p_j is active, it may be possible for e_i to affect the progress of p_j by issuing "slow" system calls. Any sandbox code that issues a blocking system call is promoted to a new thread of execution, if it is not already associated with its own thread. Since sandbox threads execute in *any* process context, essentially they are inexpensive to schedule.

A sandbox-bound thread of execution is created via the `create_upcall()` interface function, invoked from within a user-space process. This interface function has similarities to the POSIX `pthread_create()` library routine, producing a new thread of control sharing the credentials and file descriptor tables of the caller. The thread produced by `create_upcall()`, however, does not possess a conventional hardware-based address space. Instead, sandbox threads execute using the page tables of the last active address space.

Mapping code into the sandbox. The existence of a shared sandbox requires the modification to the page tables and address spaces of all created processes (when they are first "forked"). As stated earlier, all processes will have page tables that can resolve virtual addresses of instructions and data in this memory area, thereby enabling sandbox code to execute in any process context.

A loader, utilizing functions from the GNU BFD (Binary File Descriptor) library, is used to map extensions into the sandbox. In the current implementation, an extension must be compiled into a target object (currently, ELF) format. The loader then maps the *.rodata* and *.text* sections of the object into the public superpage, with the *.bss* and *.data* sections being mapped into the protected region.

[2] The 32-bit x86 processor uses a two-level paging scheme, comprising page directories and tables.

Extension code is activated by upcalls from the trusted kernel. To ensure the protected region of a sandbox is user-level accessible, the kernel toggles the user-supervisor flag of the corresponding superpage before issuing the upcall. After toggling the privilege protection flag, the TLB entry for the superpage must be flushed and reloaded to eliminate stale flag settings. When the process whose page tables were used by a sandbox function is again scheduled, the user-supervisor flag must be reset before the process regains control of the CPU at the user level. This is necessary to prevent malicious or ill-written processes from accessing the protected sandbox area.

Additional support for user-level sandboxing. As sandbox extensions do not have conventional address spaces, they are unable to use certain system interfaces related to memory management without modification. Some of the affected interfaces include `brk()`, `mmap()`, and `shmget()`. These interfaces are used to fulfill a variety of needs: `brk()` affects the breakpoint at the end of the heap data area in a process, while `shmget()` allocates shared memory segments. Likewise, `mmap()` can allocate either process-private or shared virtual memory as well as providing memory-mapped file I/O.

In our current implementation, we allow C, Cyclone [Jim et al. 2002] and Cuckoo [West and Wong 2005] extensions to link with a slightly modified version of the dietlibc library, to manage sandbox memory, and to make system calls. Cyclone is syntactically similar to C but provides type-safety and, hence, memory protection for multiple extensions coexisting in the sandbox. Cuckoo is our own language that is similar to Cyclone but also provides memory-safety for multithreaded code. We envision type-safe languages being used for extensions written by untrusted users, to prevent them from accessing addresses of other sandbox extensions, or the private address space of an active process at the time the extension is invoked. In contrast, we allow extensions written in C to be produced by trusted users such as kernel developers, who are aware of the potential side effects of their code and do not intend to behave in a malicious manner, by deliberately corrupting the sandbox or process-private address spaces. It is important to note that "trusted" code implies that code is *nonmalicious*, not necessarily *error-free*. The isolation provided by ULS allows the system to survive ill-written service extensions.

Fast upcalls. Traditionally, signals and other such event notification schemes [Banga et al. 1999; Lemon 2001] have been used to invoke actions in user-level address spaces when there are specific kernel state changes. Unfortunately, these schemes incur costs associated with the traversal of the kernel-user boundary, process context-switching, and scheduling. Our upcall mechanism operates like a software trap (i.e., the mirror image of a typical system call), to efficiently vector events to user-level sandbox extensions. To make function invocations from kernel to user space, we utilize hardware support in the form of the SYSENTER and SYSEXIT instructions where available, and stack activations otherwise [Chiueh et al. 1999]. An upcall made while in the context of *any* process is termed a *pure upcall*. Finally, to avoid the problem of generating upcalls when no user-level process is running, all extensions utilize a private stack in the sandbox.

Though the expected behavior of application-specific services is to predominantly make pure upcalls in the context of any currently loaded address space, it is also possible for services to run in a threaded, schedulable context. For example, if a service blocks on a slow system call it can continue as a schedulable (albeit not necessarily real-time) thread. In this case, TLB flushing costs are still mitigated when switching to the service task.

Beyond memory safety. Issues of memory safety aside, it is also important to ensure both CPU and I/O protection. CPU protection is ensured in a manner similar to that in SafeX. Most importantly, the time spent executing a first-class service is bounded and charged to the process that registered that service. ULS addresses I/O protection by ensuring that the file-descriptors visible to a first-class service are those inherited from the registering process (which may not necessarily be the current process at the time the service is invoked).

4. EXPERIMENTAL EVALUATION

This section begins by assessing the effectiveness of a ULS implementation applied to a Linux kernel. With the exception of the experiments in Section 4.3, all other cases involved a patched Linux 2.4.9 kernel running on a series of Pentium 4-based systems. Unless otherwise stated, the Pentium 4-based systems were clocked at 1.4GHz. The nature of these experiments was partly to show that sandbox extensions can be executed with bounded overheads compared to user-level services mapped into private address spaces. Such bounded overheads can be achieved by relying only on page-based hardware as opposed to specialized features such as segmentation and *tagged* TLBs. In the following experimental results, the extensions have been written in C. Our work with Cyclone and our new language, called *Cuckoo* [West and Wong 2005], suggests that runtime overheads of type-safe languages can be kept fairly low, so performance results should be similar if we used type-safe extensions.

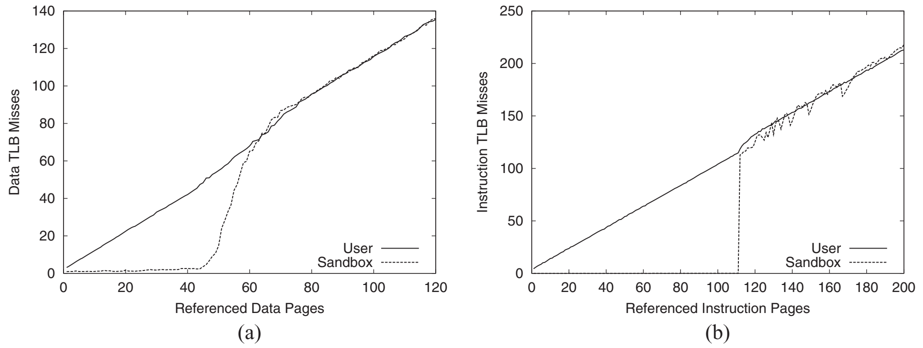
4.1. Interprotection Domain Communication

To investigate the effects of working set size on the effectiveness of sandbox-based extensions, a number of IPC ping-pong experiments similar to those conducted in the "small spaces" work [Uhlig et al. 2002] were carried out. These experiments also considered the effects of both instruction and data TLBs, found on the x86 architecture. The Pentium 4 processor has a 64 entry data TLB and an 128 entry instruction TLB for address translation. These experiments demonstrated the ability of ULS to make the best use of system caches as the size of applications running on the system vary in working set size. This property will affect the predictable behavior of caching for normal processes executing on the system.

Two threads exchanged 4-byte messages over connected pipes. One thread simulated an application thread in a traditional address space with a configurable instruction and data TLB working set. The second thread (having a small, fixed TLB footprint) acted as an extension running either in a separate full address space or in the sandbox. The "application" thread filled some number of TLB entries, sent a message to the "extension" thread, and read a reply message. To simulate various data TLB sizes, the application thread read 4 B of data from a series of memory addresses spaced 4160 B apart. To simulate instruction TLB sizes, the application thread performed a series of relative jumps to instructions spaced 4160 B apart. These spacings avoided cache interference effects. The TLB miss counts were obtained using the Pentium 4 CPU performance counters.

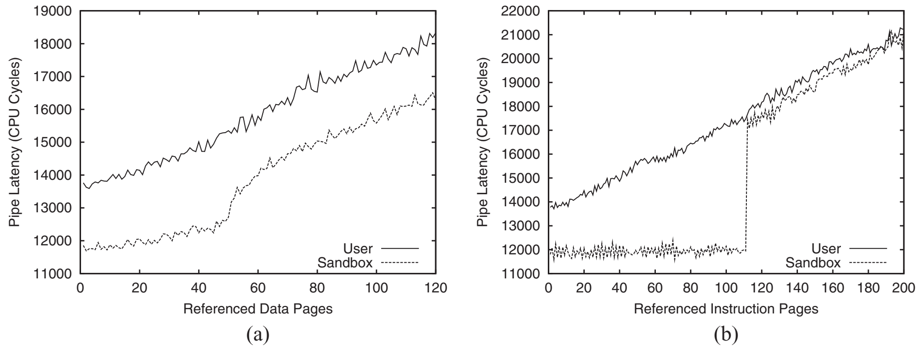
Figure 4(a) shows the data TLB working set of the application thread was maintained for up to approximately 45 entries when the extension thread was mapped into the sandbox. Thereafter, the combined data TLB demands of the OS, application, and extension no longer fit the 64 entries available on the Pentium 4 and each page access incurred a TLB miss. Note that, for the extension thread in a traditional address space, every data page access after the IPC ping-pong incurred a TLB miss regardless of the working set size, as all TLB entries were purged on every context switch.

As shown in Figure 4(b), the instruction TLB entries of the application thread were preserved when the extension was located in the sandbox. No instruction TLB misses occured until the working set approached 110 entries, which is close to the available 128 TLB entries. Thereafter, the number of instruction TLB misses were similar for both extension types. These results correspond to those in the "small spaces" work that used the segmentation features of the x86 to implement multiple logical protection



(a) (b)

Fig. 4. Effects of working set sizes in terms of (a) data, and (b) instruction pages on the number of TLB misses, for interprotection domain communication. The "User" case is for traditional interprocess communication, while the "Sandbox" case shows communication costs between a process and a sandboxed protection domain.



(a) (b)

Fig. 5. Latency of communication via a pipe between two protection domains, as a function of working set sizes in terms of (a) data, and (b) instruction pages.

Table I. Microbenchmarks Taken on a 1.4-GHz Pentium 4,512-MB RAM

Operation	Cost in CPU cycles
Upcall including TLB flush/reload	11,000
TLB flush and reload (includes call to `OpenSandbox()`)	8,500
Raw upcall	2,500
Signal delivery (current process)	6,000
Signal delivery (different process)	46,000

domains within a single address space. This shows that our user-level sandbox technique can achieve interprotection domain communication performance similar to approaches based on specialist hardware features such as segmentation.

Finally, Figure 5(a) shows the communication latency remained lower with the sandbox extension even when the data TLB miss rates were similar. Likewise, in Figure 5(b), the pipe latency is considerably lower for the sandboxed extension, until the instruction TLB is filled.

In the presence of the execution of application-specific services, we conclude that caches will perform predictably for user-level processes on the system. This is necessary to maintain execution isolation of processes from the service extensions.

4.2. Microbenchmarks

Table I presents a number of microbenchmarks that point to the efficiency of using our fast upcalls method for invoking sandbox code. In this table, the fast upcall costs are shown for the SYSEXIT/ENTER implementation. CPU clock cycles are measured using the processor's time stamp counter. The complete upcall cost includes the CPU cycles required to go from kernel space to a user-space upcall handler function. This includes the costs of flushing the sandbox data area TLB entry, placing arguments on the upcall stack, performing a SYSEXIT, and executing the user-level prologue of the upcall handler function. The TLB flush and reload time dominates the overall upcall cost, while the remaining "raw upcall" cost accounts for less than a quarter of the elapsed cycles. Note that, in these microbenchmarks, the TLB flush and reload cost includes the time to call our (unoptimized) `OpenSandbox()` function, which affects the flow of control and pushes arguments onto the kernel stack. Copying arguments and trampoline code to the (user-level) upcall stack consumes majority of the clock cycles associated with the raw upcall. The trampoline code is simply a SYSENTER instruction, which is referenced by the return address (also on the same stack) of the upcall handler. A few hundred cycles of the raw upcall can be attributed to the SYSEXIT instruction, while the rest are associated with saving information on the kernel stack for when we return via the corresponding SYSENTER.

The signal costs measure the overheads of delivering a signal to user space from the kernel within the same address space context as well as between different address spaces. The costs of delivering a signal *within* the same address space is lower than the cost of an upcall, but once an address space switch and scheduling operation are involved, the costs of delivering a signal from kernel to a user-space process are over four times the cost of a full upcall. Note that the measured cost of delivering a signal to a different process involves making that process the highest priority, so it is guaranteed to be scheduled next.

4.3. User-Level Sandboxing Versus SafeX

In this set of experiments, we compared the performance of kernel-level extensions against user-level approaches for monitoring and adapting system resource usage. The aim was to see whether it is possible to implement system-wide service extensions in a user-level sandbox, and still achieve a similar level of control over physical resources to that of kernel-based approaches, using our SafeX approach. This set of experiments used a standalone 550-Mhz Pentium III with 256 MB of RAM. In this case, a user-level sandbox was implemented on a patched Linux 2.4.20 kernel.

Four different methods of dynamically managing CPU usage were compared, for a set of processes each with specific resource requirements over finite windows of real time. The four methods implemented a CPU service manager within (1) a user-level process, (2) a sandboxed thread, (3) a pure upcall function in the sandbox, and (4) a kernel bottom-half handler.

Three processes, P_1, P_2, and P_3 had target CPU demands of 40 ms every period of 400 ms, 100 ms every period of 500 ms, and 60 ms every period of 200 ms, respectively. A process missed a deadline if it did not receive its CPU demand within its current period. For simplicity, the processes were all CPU-bound, had memory footprints less than 4 kB when stripped of symbols, and merely iterated over a number of integer computations. Note that, in similar experiments, we considered application processes that encode a number of video frames into groups of pictures, as part of a multimedia streaming system. The results of these experiments are not included because they showed similar performance patterns to those shown in this section. In any case, processes P_1, P_2, and P_3 had static real-time priorities initialized to $80 * (\text{target/period})$, where *target* and *period* denote the target CPU time required in a given request period, measured in milliseconds. Since Linux real-time priorities range from 1 (lowest) to 99 (highest), kernel daemons were assigned real-time priorities of 97 or higher, thereby ensuring the whole system continued to function responsively.

```
void monitor(){
   //  Get target and actual CPU usage
   //  from application's attribute class.
   actual_cpu = get_attribute("actual_cpu");
   target_cpu = get_attribute("target_cpu");

   // Generate an event carrying difference
   // in target and actual values.
   raise_event("Error", target_cpu-actual_cpu);
}

typedef struct {char *name; int value;} event;

void handler(event ev){
   // Get nth sampled error between target
   // and actual monitored values.
   e[n] = ev.value;

   // Update process' timeslice by PID fn
   // of target and actual CPU usage.
   // u[n] is the timeslice adjustment
   // at the nth sampling interval.
   // Kp, Kd and Ki are PID constants.
   u[n] = (Kp+Kd+Ki)*e[n]-Kd*e[n-1]+u[n-1];

   // Set timeslice adjustment field of
   // an application's attribute class.
   // A guard function will
   // subsequently verify the QoS safety
   // of the new timeslice value.
   set_attribute("timeslice-adjustment", u[n]);
}
```

Fig. 6. Pseudocode for monitor and handler extensions used in the experiments.

```
guard (attribute, value):
  if (attribute == "timeslice-adjustment")
    if (CPU utilization is QoS safe)
      timeslice=max(0,target_cpu+value);
    else block process;
```

Fig. 7. Guard function pseudocode.

The kernel-based service manager was invoked once every 10 ms from a Linux timer queue, to monitor the CPU allocations of the three CPU-bound processes. Similarly, the upcall-based service manager was invoked once every 10 ms by upcall events triggered from a timer bottom half. Corresponding handler functions in each case adjusted the timeslice of the three processes as necessary, using the same PID[3] controller described in prior experiments [West and Gloudon 2002]. Figure 6 shows the pseudocode for monitor and handler extensions created on behalf of all application processes. In the case of ULS-based monitor and handler functions, appropriate POSIX-compliant system calls were made to adjust priorities and timeslices of the affected application processes. A guard function, as shown in Figure 7 allowed a process's timeslice to increase as long

[3] Proportional plus integral plus derivative.

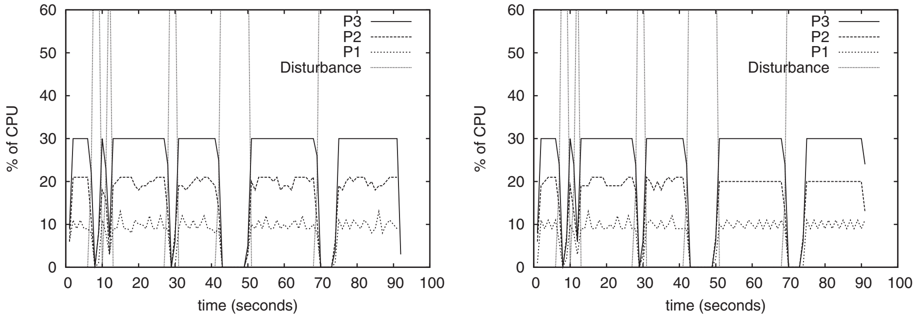


Fig. 8. CPU service management controlled by (a) a user-level real-time process, and (b) a sandboxed thread.

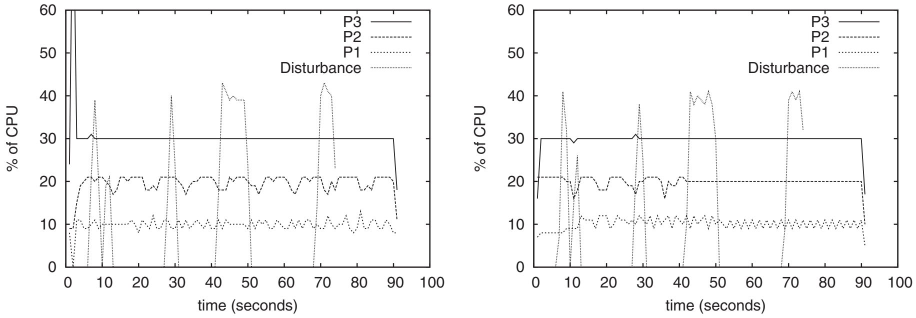


Fig. 9. CPU service management controlled by (a) a pure upcall function in the sandbox, and (b) a kernel bottom-half handler.

as its average CPU usage, measured over twice its period, was not above the target utilization. That is, if a process requires *target_cpu* units of CPU time every window of r time units, and the actual time spent executing on the CPU over a window of $2r$ is *actual_cpu* time units, then it is (QoS) safe to execute the process if $\frac{actual\_cpu}{2} \leq target\_cpu$.

Both the kernel- and pure upcall-based service managers check the identity of the running process when they are invoked via the kernel timer queue. Accounting information for the CPU usage of the current process is updated to the nearest clock tick (or jiffy). The kernel approach accounts for lost ticks but the sandboxed approach does not, making the latter method of tracking CPU usage slightly less accurate. In contrast, the process- and thread-based managers determine the CPU usage of the three processes via the `/proc` filesystem, when they are scheduled by the kernel. To ensure predictable service, the process- and thread-based managers are assigned real-time priorities of 96.

For all four service manager methods, a background disturbance process attempts to consume all available CPU cycles when it is active. Its execution pattern is based on a Markov Modulated Poisson Process, with average exponential interburst times of 10 s and average geometric burst lengths of 3 s. Each burst of the disturbance is triggered with an initial priority of 96, but when the corresponding service manager is active, the disturbance's priority is adjusted to maintain service to the other three processes. In all cases, the disturbance is scheduled using the POSIX.4 SCHED_FIFO policy. The aim is to maintain fine-grained control over CPU allocation for processes that could be part of a real-time application.

Figures 8 and 9 show the abilities of each service management method to maintain CPU allocations of the three processes at their target levels. Both the process- and thread-based approaches suffer from the need for scheduling by the kernel in order

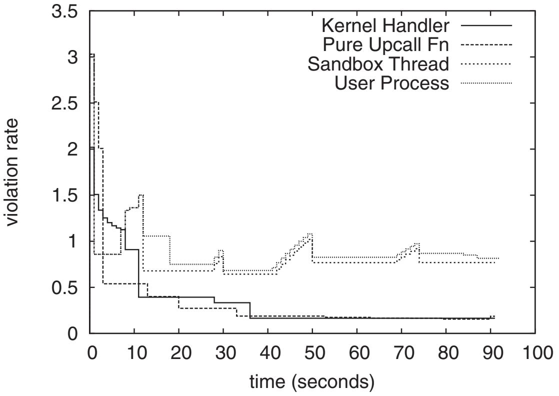


Fig. 10. Deadline violation rates.

to control resource allocation. When the disturbance uses SCHED_FIFO scheduling, it cannot be preempted by a service manager that is scheduled at the same initial priority. For brevity, we do not include results for the case when the disturbance is scheduled using a SCHED_RR policy, but the pure upcall- and kernel-based approaches still perform better. Moreover, having the disturbance scheduled using SCHED_FIFO indicates the vulnerability of process- and thread-based approaches to user-level service management. That is, they are dependent upon the parameters of other schedulable entities, and the scheduling policy enforced by the underlying kernel. This contrasts with the pure upcall- and kernel-based service managers, which do not entirely depend upon the underlying nature of the kernel's scheduling policy.

As can be seen from Figure 9, implementing an efficient service extension for dynamic management of CPU cycles is possible using user-level sandboxing. The upcall-based service manager successfully maintains the target CPU allocations to all three processes, without allowing the background disturbance to hog all the resources when it is active. While the kernel-based approach provides the finest granularity of control over resource allocation, implementing extensions in the kernel precludes the use of libraries, system calls, and the benefits of isolating application-specific code outside the kernel protection domain. With all the user-level approaches, including the pure upcall method, conventional system calls such as `sched_setscheduler()` are available to control CPU allocations. In general, the slight reduction in fine-grained control over resources is offset by the ease of programming at the user level.

The violation rate for tasks P_1, P_2, and P_3, measured in deadlines missed per second, is plotted in Figure 10 as a function of time. The ability to manage the CPU on a fine-grained basis is not satisfied by the thread-based methods, even threads running within a sandbox. However, sandboxed services invoked by pure upcalls are comparable in their ability to manage resources as predictably as SafeX-type methods that place application-specific services in the most trusted hardware protection domain. Both upcalls to sandboxed handlers and SafeX kernel extensions yield relatively low violation rates, close to 0.2 deadlines/s in the steady state, compared to around four times worse performance for sandboxed threads and user processes.

4.4. User-Level Networking in the Sandbox

As a further application of our ULS approach, we have implemented a network stack in the sandbox that avoids copying and processing within the kernel [Qi et al. 2004]. In effect, this allows custom stack configurations to be implemented, so that network data can be processed in an application-specific manner. For example, one could implement

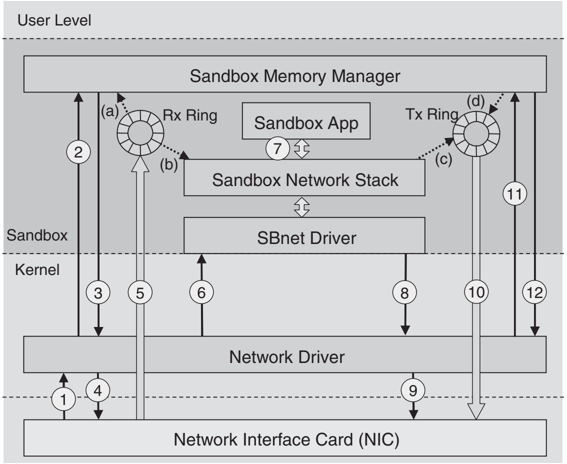


Fig. 11. User-level asynchronous networking in a sandbox.

a special-purpose routing protocol in user space using this technique. Such a routing protocol could be used to forward stream data between hosts according to various real-time constraints [Fry and West 2004; Parmer et al. 2004].

By implementing a network stack at the user level, we have control over the behaviors of various communication protocols. In effect, this is similar to U-Net [von Eicken et al. 1995], Ethernet Message Passing (EMP) [Shivam et al. 2001], and the Virtual Interface Architecture (VIA) [Dunning et al. 1998], which all provide abstractions for user-level network implementation. In contrast, our work allows user-level extensions to run efficiently enough to be invoked as handlers for networking events, without the need for special hardware support. Notwithstanding, the following experiments were intended to show the potential capabilities of user-level services and how they can perform predictably and efficiently even for real-time communication where throughput and jitter constraints are involved.

Figure 11 shows the control path, for the case when sandboxed network services are invoked when a packet arrives on the network interface card (NIC). This is the control path experienced by upcalls into the sandbox that are executed in the context of the active address space at the time of the upcall. We modified the kernel network driver so that packet processing and interaction with the NIC take place in the sandbox (using our own SBnet driver). Since device drivers in Linux are commonly implemented as loadable modules, our sandboxed stack implementation did not require changes to the core kernel, other than the minimal changes to create the sandbox itself. That said, the various stages of asynchronous computation involving the networking stack are as follows.

(1) When a packet is received by the NIC, an interrupt service routine is invoked in the network driver. This is a basic notification that a packet is ready and a minimal amount of processing is undertaken. No modifications to the interrupt handler in the default driver are made, so that it remains as efficient as possible.
(2) When the interrupt handler finishes, execution continues in the network driver but with interrupts enabled. Space is then allocated from a receiver ring buffer (label (a)) in the sandbox, by an upcall that directly invokes the sandbox memory manager.
(3) The return address of this allocation is passed to the network driver. A check is performed to verify that the memory location is within the sandbox region.

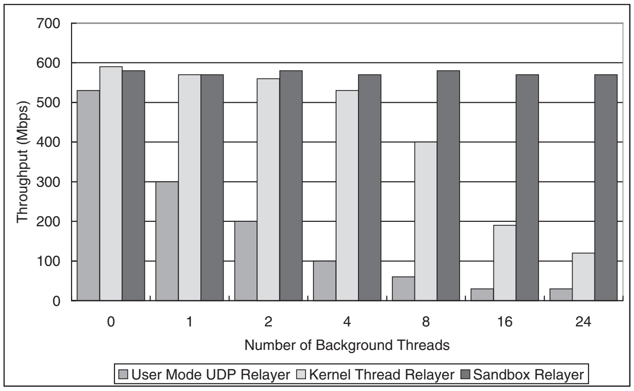


Fig. 12. Throughput comparison of an optimized sandbox stack versus alternative user- and kernel-level implementations.

(4) The network driver informs the NIC of the location into which it can transfer the packet, using direct memory access (DMA). Because this network driver is executed in the kernel domain, it has full I/O permissions for trusted communication with the NIC.
(5) The NIC copies the received packets into the allocated sandbox memory using DMA.
(6) After the new packet is resident in the sandbox, an upcall to the SBnet driver and, hence, the protocol stack occurs. Packets can be accessed from the ring buffer (label (b)) in the context of a bottom half, so execution is unaffected by host scheduling. Recall that when a pure upcall is triggered, the handler runs with user permissions.
(7) At this point in the configurable networking stack, application-specific handlers can execute. For example, we provide an application that performs transport-level forwarding of packets to another end host.
(8) After the network stack's processing is complete, the memory address of a packet awaiting transmission is placed in an outgoing buffer (label (c)). Control then returns to the kernel.
(9) Now full I/O permissions are restored, the NIC is notified of the packet it should transmit.
(10) The NIC uses DMA to retrieve and send the packet onto the network.
(11) After the DMA is complete, the network driver notifies the sandbox extension that it can free the memory formerly taken up by the packet (as in label (d)).
(12) Upon return from this pure upcall, network processing for this packet completes and control can return to the previously executing thread.

Though this entire control path seems complex, it is highly optimized and yields significant performance improvements over conventional user-space network protocol stacks confined to process-private address spaces. The following experiments showed the efficiency of our configurable, user-level networking stack. Here, we used three IBM x-series 305 servers connected via Tigon3 Gigabit Ethernet cards. Each machine has a 2.4-GHz Pentium 4 CPU and 1024-MB RAM. One machine acting as a source sends packets of data via an intermediate (or proxy) host, configured with a sandboxed network stack for packet forwarding/relaying, to a destination host.

Throughput. Figure 12 compares the throughput of a sandboxed networking stack versus alternative kernel- and user-level implementations, to forward data between two UDP socket end-points. The alternative user-level approach relays data via a

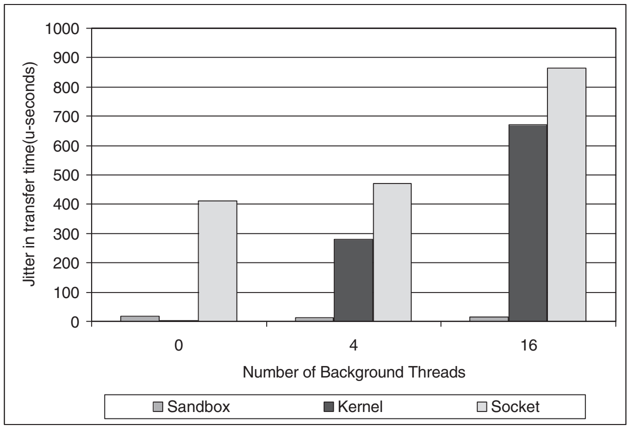


Fig. 13. Maximum jitter in transfer time.

process that simply reads from one socket and writes to another. In contrast, the kernel approach uses a kernel thread to connect two socket end-points. As can be seen, the kernel method yields the highest throughput when there are no background threads active on the end-host. However, since both the kernel- and user-level relaying agents execute in their own thread contexts, they are subject to scheduling overheads. This can be seen by the fact that only the sandboxed networking approach maintains the same level of throughput irrespective of the number of background threads.

Transfer time jitter. To conclude our experiments on user-level networking, we measured the jitter (i.e., variation in the transit time) of packets being forwarded over a period of time. The reduction, or elimination, of jitter is especially important in environments which require quality of service constraints to be met. If network performance is unpredictable (i.e., high jitter is present) then guaranteeing QoS constraints becomes increasingly difficult, if not impossible.

Running in the context of bottom halves gives the sandbox upcall code the ability to immediately process each incoming packet, which results in a very small amount of variation in the transfer time of those packets. In contrast, the kernel and process-based forwarding agents must suffer from scheduling delays. The amount of deviation from the average transfer time is a function of the size of the scheduler's run queue. Figures 13 and 14 show that a nearly constant amount of jitter is demonstrated by the sandboxed networking scheme, while the other two approaches show larger and more variable jitter as the number of background threads increases.

5. RELATED WORK

There have been a number of related research efforts that focus on OS structure, extensibility, safety, and service invocation. Extensible operating systems [Small and Seltzer 1996; Bershad et al. 1995; Chiueh et al. 1999; Jones 1993; Ghormley et al. 1997] aim to provide applications with greater control over the management of their resources. Additionally, microkernels [Accetta et al. 1986] and exokernels [Engler et al. 1995] offer a few basic abstractions, while moving the implementation of more complex services and policies into application-level components. By separating kernel- and user-level services, microkernels introduce significant amounts of interprocess communication overhead, although it has been argued that by leveraging hardware support many such costs can be made to disappear [Liedtke 1995]. In effect, our sandboxing technique provides a way to construct microkernel services without the inherent costs of heavyweight interaddress space communication. We support this without the need

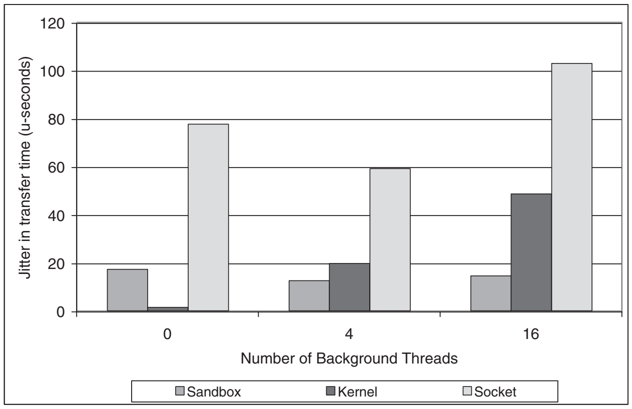


Fig. 14. Average jitter in transfer time.

for esoteric hardware features (e.g., segmentation as used by Palladium [Chiueh et al. 1999]) to implement fine-grained logical protection domains, so that the latency to invoke application-specific services is limited to the cost of an upcall and the flush of a TLB entry. Consequently, unbounded delays due to, for example, priority-based scheduling of process address spaces do not affect the timely execution of system service extensions.

It is worth noting that several other research groups have, in contrast to our work, leveraged Linux to build real-time and QoS-based systems, such as RTLinux [Yodaiken and Barabanov 1997], RED-Linux [Wang and Lin 1999], and QLinux [Sundaram et al. 2000]. RTLinux is a small, hard, real-time kernel that executes Linux as a separate thread. RED-Linux provides a general scheduling framework to support different real-time scheduling policies within the kernel, while QLinux provides QoS guarantees to multimedia applications using special thread, packet, and disk scheduling policies. However, none of these systems has focused on combined hardware and software support to separate application-specific services from the core kernel, and none has focused on efficient and predictable methods by which services can be invoked so that scheduling and context-switching overheads are largely eliminated.

This leads to another area of related research, which focuses on service invocation, kernel event notifications [Banga et al. 1999; Lemon 2001] and upcalls [Clark 1985; Gopalakrishnan and Parulkar 1998]. Much of this work is concerned with the way to trigger user-level services or handlers due to some condition or event in the kernel. With our ULS approach, we enable upcalls to be triggered no matter which address space is active at the time of a kernel event, thereby greatly reducing the overheads of service invocation. Almost all other approaches still involve the scheduling of process address spaces in which to handle events from the kernel.

Finally, while others have considered methods to instrument applications, to intercept requests for resources such as CPU cycles, memory, and bandwidth [Chang et al. 2000], the emphasis of our ULS and SafeX work is to develop safe and predictable execution domains in which application-specific services may be deployed. Our work enables COTS systems to be extended with resource management methods to improve and/or guarantee qualities of service [Rajkumar et al. 1998] to individual applications without the need for entire QoS architectures [Abdelzaher and Shin 1998; Rosu et al. 1998] to be constructed. As stated above, such execution domains do not suffer from scheduling and context-switching overheads as would be the case for services mapped into traditional process address spaces.

6. CONCLUSIONS AND FUTURE WORK

This article compares various methods to instrument commodity operating systems with services and handlers that are tailored to the needs of real-time applications. We compare methods to deploy service extensions at both the kernel and user levels, using our SafeX and user-level sandboxing (ULS) schemes, respectively. Both approaches enable applications to deploy services in a manner that does not require explicit scheduling and context switching between process-private address spaces, thereby ensuring bounded dispatch latencies and finer-grained resource management. SafeX relies on a combination of type-safe language and runtime support to enforce memory, CPU, and I/O-space protection of untrusted application-specific services within the address space of the trusted kernel. This enables users to deploy "first-class" services having the same capabilities as core kernel services, with the exception that the kernel may revoke access rights on any services abusing their privileges. Such an approach prevents, for example, an application-specific service from running for unbounded amounts of time and/or altering its resource usage beyond that allowed by the kernel. However, SafeX places restrictions on how extensions operate within the kernel by preventing them from disabling interrupts and accessing kernel symbols outside those within a defined API.

To alleviate the potential problems associated with application-specific services executing in the trusted kernel address space, our ULS approach allows first-class services to execute in a sandbox environment isolated by hardware-level (i.e., page-based) protection from the kernel. This imposes only minimal additional overheads over those associated with SafeX when dispatching application services. Specifically, ULS requires an upcall into a sandboxed memory region as well as a TLB flush of a single page entry. Such a cost is minimal and bounded compared to the overheads of otherwise scheduling and context switching between user-level process address spaces.

Experimental results show that ULS and SafeX incur similar performance penalties and benefits for an example service extension that adaptively manages CPU usage among competing processes in specific windows of real time. Both approaches yield lower service violations than alternative user-level methods of application-level resource monitoring and management. Given this observation, we feel ULS is preferred over SafeX as the first step toward supporting application-specific real-time services on commodity OSes. As evidence of the potential capabilities of ULS, we have implemented a user-space networking stack that avoids data copying via the kernel and allows packet processing to take place in the context of a kernel bottom half without explicit process scheduling, thereby increasing data throughput and reducing jitter. ULS requires no special hardware support other than page-based hardware protection and timer interrupt support to ensure predictable and low-latency execution of application-specific services.

Future work involves extending our ULS approach to multiprocessor platforms, and to provide safe and predictable resource management support for entire virtual machines rather than the more simplistic services and handlers currently supported.

REFERENCES

Abdelzaher, T. F. and Shin, K. G. 1998. End-host architecture for QoS-adaptive communication. In *Proceedings of the 4th Real-Time Technology and Applications Symposium*.

Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. 1986. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Conference*. 93–113.

Banga, G., Mogul, J. C., and Druschel, P. 1999. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the Annual Technical Conference*.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. 1995. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 267–284.

Chang, F., Itzkovitz, A., and Karamcheti, V. 2000. User-level resource-constrained sandboxing. In *Proceedings of the 4th Windows Systems Symposium*.

Chiueh, T., Venkitachalam, G., and Pradhan, P. 1999. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 140–153.

Clark, D. 1985. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 171–180.

Dunning, D., Regnier, G., McAlpine, G., Cameron, D., Shubert, B., Berry, F., Merritt, A. M., Gronke, E., and Dodd, C. 1998. The virtual interface architecture. *IEEE Micro 18,* 2, 66–76.

Engler, D. R., Kaashoek, M. F., and O'Toole, J. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 251–266.

Fry, G. and West, R. 2004. Adaptive routing of QoS-constrained media streams over scalable overlay topologies. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society Press, Los Alamitos, CA.

Ghormley, D. P., Rodrigues, S. H., Petrou, D., and Anderson, T. E. 1997. Interposition as an operating system extension mechanism. Tech. rep. CSD-96-920. University of California, Berkeley, Berkeley, CA.

Gopalakrishnan, G. and Parulkar, G. 1998. Efficient user space protocol implementations with QoS guarantees using real-time upcalls. *IEEE/ACM Trans. Netw. 6,* 4, 374–388.

Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y. 2002. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*.

Jones, M. B. 1993. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 80–93.

Lemon, J. 2001. Kqueue—a generic and scalable event notification facility. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 141–153.

Liedtke, J. 1995. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, USA.

Morrisett, G., Crary, K., Glew, N., Grossman, D., Smith, F., Walker, D., Weirich, S., and Zdancewic, S. 1999a. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*. ACM Press, New York, NY.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999a. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst. 21,* 3, 527–568.

Parmer, G., West, R., Qi, X., Fry, G., and Zhang, Y. 2004. An Internet-wide distributed system for data-stream processing. In *Proceedings of the 5th International Conference on Internet Computing*. CSREA Press, Las Vegas, NV.

Qi, X., Parmer, G., and West, R. 2004. An efficient end-host architecture for cluster communication services. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE Computer Society Press, Los Alamitos, CA.

Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D. 1998. Practical solutions for QoS-based resource allocation problems. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, Los Alamitos, CA.

Rosu, D., Schwan, K., and Yalamanchili, S. 1998. FARA—a framework for adaptive resource allocation in complex real-time systems. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*. IEEE Computer Society Press, Los Alamitos, CA.

Shivam, P., Wyckoff, P., and Panda, D. 2001. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In *Proceedings of the ACM/IEEE conference on Supercomputing*. ACM Press, New York, NY.

Small, C. and Seltzer, M. I. 1996. A comparison of OS extension technologies. In *Proceedings of the USENIX Annual Technical Conference*. 41–54.

Sundaram, V., Chandra, A., Goyal, P., and Shenoy, P. 2000. Application performance in the QLinux multimedia operating system. In *Proceedings of the 8th ACM Conference on Multimedia*. ACM Press, New York, NY.

Uhlig, V., Dannowski, U., Skoglund, E., Haeberlen, A., and Heiser, G. 2002. Performance of address-space multiplexing on the Pentium. Tech. rep. 2002-1. University of Karlsruhe, Karlsruhe, Germany.

VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. 1995. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 40–53.

WAHBE, R., LUCCO, S., ANDERSON, T., AND GRAHAM, S. 1993. Software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY.

WALLACH, D. A., ENGLER, D. R., AND KAASHOEK, M. F. 1997. ASHs: Application-specific handlers for high-performance messaging. *IEEE/ACM Trans. Netw. 5,* 4, 460–474.

WANG, Y.-C. AND LIN, K.-J. 1999. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Proceedings of the 20th IEEE Real-Time Systems Symposium* (RTSS).

WEST, R. AND GLOUDON, J. 2002. 'QoS safe' kernel extensions for real-time resource management. In *Proceedings of the the 14th EuroMicro International Conference on Real-Time Systems*. IEEE Computer Society Press, Los Alamitos, CA.

WEST, R. AND WONG, G. 2005. Cuckoo: A language for implementing memory- and thread-safe system services. In *Proceedings of the International Conference on Programming Languages and Compilers*. CSREA Press, Las Vegas, NV.

YODAIKEN, V. AND BARABANOV, M. 1997. A real-time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference* (USELINUX).

Received January 2006; accepted May 2006