An Efficient User-Level Shared Memory Mechanism for Application-Specific Extensions

Richard West, Jason Gloudon, Xin Qi and Gabriel Parmer

Computer Science Department
Boston University
Boston, MA 02215
{*richwest,jgloudon,xqi,gabep1*}*@cs.bu.edu*

Abstract

This paper focuses on an efficient user-level method for the deployment of application-specific extensions, using commodity operating systems and hardware. A *sandboxing* technique is described that supports multiple extensions within a shared virtual address space. Applications can register sandboxed code with the system, so that it may be executed in the context of *any* process. Such code may be used to implement generic routines and handlers for a class of applications, or system service extensions that complement the functionality of the core kernel. Using our approach, application-specific extensions can be written like conventional user-level code, utilizing libraries and system calls, with the advantage that they may be executed without the traditional costs of scheduling and context-switching between process-level protection domains. No special hardware support such as segmentation or tagged translation look-aside buffers (TLBs) is required. Instead, our "user-level sandboxing" mechanism requires only paged-based virtual memory support, given that sandboxed extensions are either written by a trusted source or are guaranteed to be memory-safe (e.g., using type-safe languages).

Using a fast method of upcalls, we show how our mechanism provides significant performance improvements over traditional methods of invoking user-level services. As an application of our approach, we have implemented a user-level network subsystem that avoids data copying via the kernel and, in many cases, yields far greater network throughput than kernel-level approaches.

1 Introduction

General purpose systems such as Linux are increasingly being used for a number of diverse applications, including those in desktop, server and embedded environments. Unfortunately, the services provided by general purpose systems are often ill-suited to the specific needs of many applications. For example, a real-time application is not well served by a scheduling policy that does not consider the timely and predictable execution of tasks. Similarly, a web server [1] may benefit from its own buffer cache algorithm that over-rides the default caching and paging policy [2]. This has motivated researchers to study various methods of system- and application-level extensibility [3, 4, 5, 6, 7, 8], thereby allowing services to be tailored for specific purposes [9, 10].

Unfortunately, designing a system that supports extensibility poses at least three conflicting challenges: (1) how to guarantee *efficient* execution of the extension code that modifies or adds functionality to the system and/or application, (2) how to ensure the *safety* of extensions that could otherwise violate the integrity of the system, or the application, and (3) how to provide support for extensibility without significant modification to the standard interfaces offered by the system. Arguably the most efficient approach is to allow extensions to be linked into a single address space shared with core system functionality. This enables extensions to be invoked by direct function calls, but requires special safety approaches to prevent untrusted application-specific code from jeopardizing system correctness.

Various approaches have been proposed to guarantee the safety of service extensions, including: *sandboxing* [11, 6], type-safe languages [12, 5], proof-carrying codes [13], and hardware-support [7, 14]. Hardware safety techniques beyond page-based protection schemes are not common across all architectures. By comparison, software-based safety approaches are not typically as efficient. That said, software fault isolation [11] is an effective method of isolating untrusted code in its own fault domain, by inserting safety checks that prevent jumps or stores to addresses outside a restricted range. Implementing such safety checks typically requires a number of dedicated machine registers. This is problematic on architectures such as the Intel x86 that have a limited number of general purpose registers.

Most software safety techniques assume the coexistence of untrusted and trusted code in a single address space. However, many researchers have argued for the separation of high-level abstractions and application-specific services from the basic services of the kernel. Micro-kernels adhere to this philosophy. Unfortunately, there are costs associated with the implementation of services outside the kernel. Overheads are typically incurred as a result of communication via the trusted kernel, as well as scheduling and switching between address spaces that isolate services. In fact these overheads have caused major problems for the efficient design of micro-kernels, unless special hardware features such as segmentation and tagged translation look-aside buffers (TLBs) are used [14].

While segmentation units and tagged TLBs are not common on all processors, page-level protection is prevalent. The challenge is to support safe and efficient service extensibility using only page-based hardware protection. Where possible, the separation of core system abstractions and high-level services, as in micro-kernels, is desired.

1.1 Motivation and Contributions

This paper is motivated by the desire to implement extensible services at user-level in a manner that is safe, efficient, and requires minimal changes to the underlying kernel. We show how it is possible to achieve these goals using a *user-level sandboxing* technique, that enables COTS systems to be extended for the specific needs of applications. In fact, our approach places no specific requirements on the underlying OS structure. As a consequence, it is possible for our technique to implement micro-kernel services, interposition agents [15], virtual machines [16, 17] and entire library OSes [18] in a sandboxed region above a kernel that is, say, monolithic.

By supporting service extensions at user-level, there are several advantages. Most notably:

- it is possible for such code to leverage libraries that would be unavailable within the kernel,
- there is no need for custom interfaces as extensions can instead leverage existing system call interfaces,
- it is possible to rapidly prototype code that would otherwise cause system failure, and
- extensions can be developed in a manner similar to regular application code without awareness of kernel internals.

We show how to efficiently extend the behavior of an existing system at user-level without the traditional costs of communicating between logical protection domains. While our approach is not as fast as pure software-based fault isolation in a *single* address space, it eliminates the costs of heavyweight context-switches between *multiple* process-level address spaces. Our approach provides an efficient way to pass control from a process-private address space to a sandboxed service extension. Using a fast method of upcalls [19], we show how our sandboxing technique for implementing logical protection domains provides significant performance improvements over traditional methods of invoking user-level services (e.g., using signals). Fundamentally, signal-handling mechanisms usually suffer from system scheduling overheads: *a signal generated by a kernel event remains pending until the corresponding address space is active.* If numerous other processes are running on a heavily-loaded system, it may be an arbitrarily-long time before a pending signal is serviced. In effect, traditional kernel event notification mechanisms suffer from scheduling delays that may be proportional to the number of processes in the system or, worse still, may depend on the CPU scheduling policy. In the latter case, a high priority process may starve the execution of another that has a pending signal.

In summary, our user-level sandboxing mechanism provides:

1. separation of application-specific services from the kernel address space, thereby avoiding unnecessary pollution of the most trusted protection domain, and
2. predictable delay bounds between the generation and delivery of kernel events that invoke user-level services. Such delay bounds are independent of system load, process activity and CPU scheduling policies.

This work is the basis for a more efficient system design with multiple trust levels, such that the kernel may mediate access rights to specific resources via upcalls to sandboxed services. We compare various techniques to access sandboxed service extensions. On a Pentium 4 processor, we can safely switch from the kernel to a sandboxed extension function in 11000 cycles, compared to 46000 cycles if we invoke a user-level extension in a private address space that is not currently active. Finally, we show the flexibility of our sandbox approach, by implementing a user-level CPU service manager, and a network protocol stack that avoids data copying via the kernel (similar to U-Net [20]). In the latter case, our stack implementation is able to achieve better network throughput than kernel-level methods, in many cases.

The following section describes our user-level sandbox technique in more detail. This is followed by Section 3 that evaluates the performance of our approach on a Linux x86 platform. Related work is then discussed in Section 4. Finally, conclusions and future work are described in Section 5.

2 User-Level Sandboxing

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 extensions 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 an extension, it changes the privilege level of the shared page (or pages) containing the extension code and data, so that it can be executed with user-level capabilities. This prevents the extension code from violating the integrity of the kernel. However, the extension code itself can run in the context of *any* user-space process, even one that did not register the extension with the system. There is potential for corrupt or ill-written extension code to modify the memory area of a running process. To guard against this, we require extension code is either written by a trusted source, or is guaranteed to be memory-safe (e.g., using type-safe languages such as Cyclone [12], or software-based fault isolation methods).

An astute reader might wonder why we cannot simply link extension code into the kernel address space, if we require it to be written either by a trusted source or in a memory-safe manner. If we allow trusted users (e.g., kernel developers) to link application-specific code with the kernel, it is still possible to introduce accidental bugs, whereas user-level sandboxing prevents this potential problem. Likewise, if a type-safe language is used to guarantee memory safety, managing access rights on I/O devices controlled by privileged instructions is still problematic: either the language prevents issuing I/O instructions or requires special features (e.g., a device-specific driver development library). With user-level sandboxing, an upcall could promote access rights of a sandbox extension to e.g., access a subset of I/O space. For example, the x86 allows this by controlling the I/O privilege levels on a task/process basis, thereby restricting the range of port addresses accessible by user-level code.

It should be noted that we do not require the entire system kernel to be written in a type-safe language. If a kernel is written in a language such as C or C++, then linking type-safe extensions with it is, in some ways, similar to linking sandbox extensions with unsafe user-level libraries. Arbitrary casts and pointer arithmetic in functions accessed via external interfaces can violate memory protection. We at least isolate extensions from the kernel to maintain system integrity, but care must be taken when allowing an arbitrary sandbox extension to execute on a given host. It is still possible that user-level processes can be adversely affected by a misbehaving sandbox routine.

2.1 Hardware Support for Memory-Safe Extensions

Our approach assumes that hardware support is limited to page-based virtual memory (i.e., processors with an MMU). More specialist hardware methods of implementing logical protection domains to accommodate extension code include the use of processors with tagged TLBs, or combined segmentation and paging units. Tagged TLBs provide a fast way to switch between protection domains mapped to separate address spaces, by storing the virtual-to-physical address translations of these address spaces in non-overlapping regions of a dedicated hardware cache. Alternatively, hardware lacking tagged TLBs but supporting segmented memory has been used to isolate these logical protection domains in different memory segments.

Tagged TLBs have the advantage that they do not need to be entirely flushed and reloaded when switching between address spaces (e.g., during a process context switch), unlike untagged TLBs that only cache address translations for an unspecified virtual memory region. Tagged TLBs are found in Sparc and MIPS processors, that use address space identifiers (ASIDs) to mark entries in the TLB. In contrast, processors such as the Intel x86 (at least the IA-32 variant) and the PowerPC have untagged TLBs but employ both segmentation and paging units. Specifically, the x86 processor uses segmentation hardware to convert between *logical* and *linear* addresses, and paging hardware to translate between linear and *physical* addresses; untagged instruction and data TLBs are used only to cache linear-to-physical translations. The advantage with an architecture such as the x86 is that protection domains can be mapped to separate memory segments restricted to specific ranges of linear addresses. Switching between protection domains in different linear address ranges simply involves changing the base and limit values of addresses for the active segment.

While segmentation and tagged TLBs offer various benefits, they are not common across all architectures. This has meant many systems such as Linux support only process-based protection domains at the page granularity. Switching between one process address space and another on a Linux x86 system requires changing page tables used for linear address translation. Unfortunately this results in a TLB flush and reload, which can be expensive for tasks with large working sets. As the disparity between processor and memory speeds increases [21], it is clearly desirable to keep address translations for separate protection domains in cache memory as often as possible. This is certainly the case for processors that are now clocking in the gigahertz range, while main memory is accessed in the 10^8Hz range. User-level sandboxing avoids the need for expensive page table switches and

TLB reloads by ensuring the sandbox is common to all address spaces.

2.2 Implementation Details

We have implemented user-level sandboxing 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 sandboxed extensions, (3) allow access to restricted sandboxed memory regions from conventional process address spaces, and (4) invoke extension functions from within the kernel. The key modifications involve additional entries in the page tables (or, more precisely, global directories) of processes, and the implementation of upcall code that toggles page protection bits.

Recently, we have just completed a binary-rewriting approach that eliminates the need to patch and recompile the kernel in order to enable user-level sandboxing. While this technique is out of the scope of this paper, it involves setting up jump instructions at a few key kernel memory locations, to invoke alternative code in a kernel-loadable module. The rewritten memory locations are identified using the kernel's system map, which is a table identifying the addresses of kernel symbols. In effect, our sandboxing approach can theoretically be applied to existing systems without significant disruption to the pre-existing kernel.

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 `ioctl`s, avoiding the need for new system calls.

Logical Protection Domains for Extension Code: Traditional operating systems provide logical protection domains for processes mapped into separate address spaces, as shown in Figure 1(a). With user-level sandboxing (Figure 1(b)), 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 extensions. Technically speaking, our sandbox implementation is further 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 extensions 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 code in the sandbox itself. An `ioctl` function, registered with the kernel, called `allocate_mapped_data()`, maps a region of the sandbox into a process-private address space, as in Figure 1(b). This mapped region is accessible by the calling process and sandbox extensions that can reference the corresponding range of addresses. This data-sharing scheme relies on hardware page protection, so `allocate_mapped_data()` allocates memory on $4KB$ page boundaries.

Sandbox Regions: In our current implementation, the sandbox consists of two $4MB$ regions of virtual memory that are identically mapped in every address space to the same physical memory (as shown in Figure 2). These regions employ the page size extensions supported by the Pentium processor and are each represented by one page directory entry in every process [1]. 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. As will become evident later, this design consideration shows benefits when the number of TLB entries for the sandbox is (on average) less than the working-set size of typical application and system processes.

One $4MB$ sandbox region is permanently assigned read and execute permission at both user- and kernel-level and acts as a *public* area. This region is marked as a global page using the global flag supported by Pentium Pro and more recent IA-32 [22] processors. This prevents the page directory entry for this page from being invalidated when a context switch occurs. The other region is permanently assigned read-write permission at kernel-level but, by default, it is inaccessible at user-level. We refer to this region as the *protected* area. The region can be made accessible to user-level by toggling the user/supervisor flags of its page directory entry and invalidating the relevant TLB entry via the INVLPG instruction. This is only allowed when an upcall occurs from the trusted kernel. In general, a sandbox need not have two classes of memory regions as in our case. We chose this way as a means to trade security for efficiency.

Sandbox/Upcall Threads: Sandboxed code can link with libraries that make system calls. Care must be taken that an extension 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

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

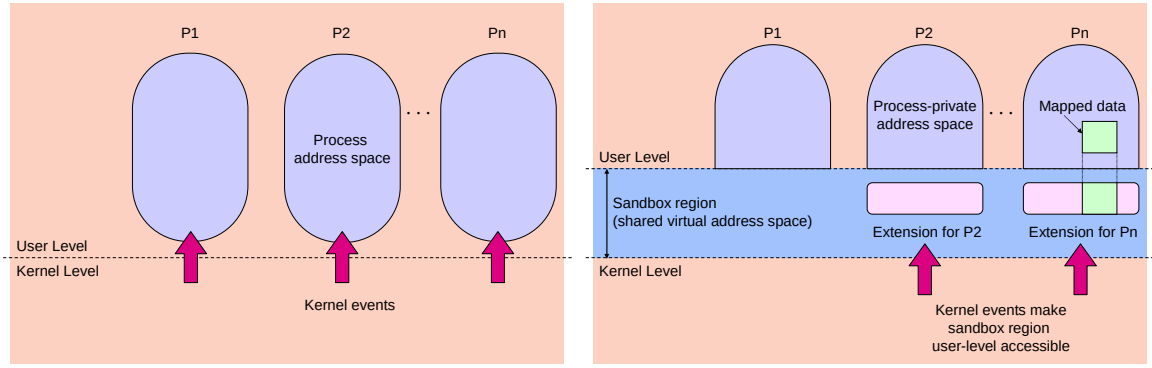

Figure 1: (a) Traditional view of logical protection domains, each mapped to a separate address space. (b) Each process address space has a shared virtual memory region, or sandbox, into which extensions are mapped.

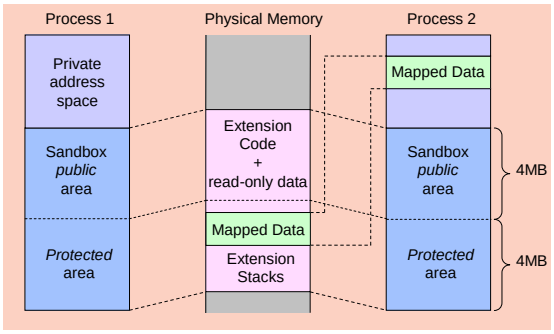

Figure 2: Sandboxes common to all processes are mapped to the same physical address ranges. Pages of the sandbox can be mapped into process-private address spaces to exchange data.

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 by a user-level process using the `create_upcall()` interface function. This interface function, like the POSIX `pthread_create()` produces 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 effect, our loader is similar to the `insmod` routine from the GNU modutils suite, but it loads an object into the sandbox rather than the kernel. In the current implementation, an extension must be compiled into a target object (currently, ELF) format, where the loader maps the $.rodata$ and $.text$ sections of the object into the public super-page. The $.bss$ and $.data$ sections of an object are loaded into the read-write (protected) region.

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 super-page before issuing the upcall. After toggling the privilege protection flag, the TLB entry for the super-page 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 user-level. This is necessary to keep malicious processes from gaining access to 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 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 and Cyclone extensions to link with a slightly modified version of the dietlibc library, to manage sandbox memory.

Fast Upcalls: Traditionally, signals and other such kernel event notification schemes [23, 24] 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.

Operating systems such as Linux that leverage hardware protection to separate user- and kernel-address spaces do not support conventional trap gates to user-level. General protection faults occur when attempting to trap to a 'ring of protection' that is less critical than the kernel. However, architectures such as the Intel IA-32 support instructions such as SYSENTER and SYSEXIT that can be used in conjunction with Model Specific Registers (MSRs) [22] to allow fast transitions between kernel and user-level address spaces. On the IA-32 architecture, where there are four rings of protection, these instructions enable efficient transitions between rings 0 (the kernel privilege level) and 3 (the user privilege level). Unnecessary memory references, to lookup and retrieve segment descriptors, followed by corresponding protection checks are avoided when using SYSEXIT/ENTER.

While these instructions are not a portable approach to implementing fast upcalls, it is possible to substitute them with 'activation records' placed on a kernel stack, that trick the hardware into thinking it is returning to user-level. For this reason, we allow our sandbox scheme to be configured with support for either SYSEXIT/ENTER, or traditional activation records. Using a return-from-interrupt instruction (e.g., `iret` on the x86, or something similar on other architectures), contents of a stack activation record can be popped into machine registers, to pass control to user-level. It is worth noting that Linux 2.6 and systems such as Windows XP use SYSEXIT/ENTER on the Intel x86 to implement system calls. Without being able to virtualize model specific registers we are thus unable to make use of these special instructions on such systems. Notwithstanding, stack activation records are not much more expensive, as shown in Section 3.4.

Finally, to avoid the problem of generating upcalls when no user-level process is running (e.g., a kernel thread is active), all extensions utilize a private stack in the sandbox.

Potential Protection Problems: In the general scheme, when an upcall event is issued from the kernel, the mechanism will modify relevant entries in the current process's page (or, equivalently, global directory) table. User-level access to the protected region of the sandbox is allowed *only while the upcall event is being processed*. Preemption and signal handling during the execution of code in the sandbox must be disabled. Allowing preemption may cause reentrancy problems (e.g., if another process runs and an upcall occurs again), while conventional signal handling can provide a 'trap door' into the sandbox for malicious users. For example, if a signal is delivered to the current process while executing sandbox code, the protected sandbox memory region is open to read-write access via the signal handler. Minor changes to the kernel simply delay delivery of signals until extensions have completed execution, and the sandbox protected area is reset to the supervisor privilege level.

Dispatching Upcalls: Each process that registers code with the sandbox is assigned a client identifier, that is used to ensure the correct extension is invoked when an upcall occurs. Since multiple upcalls may be pending, each client has a corresponding queue for related events. This is similar to the POSIX.4 specification for real-time signals. However, POSIX.4 signals can remain pending until the corresponding process is allocated the CPU, which depends on the scheduling order of other processes.

In the current implementation of our sandboxing scheme, we have developed a generic kernel routine, `start_upcall()`, that delivers upcall events to appropriate clients. An active sandbox extension loops through all events in its queue before returning (if it is a pure upcall function), or sleeping (if it is a thread). Specifically, each event is associated with a POSIX `siginfo_t` structure, that can carry different arguments to extensions. Eventually, we intend to implement a full-featured event notification scheme. However, we have already implemented a method to trigger events when various bottom half handlers are invoked at predetermined timer intervals in Linux. This method requires no changes to the core kernel, as it is wholly contained in a kernel module.

3 Experimental Evaluation

This section assesses the effectiveness of a user-level sandboxing implementation applied to a Linux kernel. With the exception of the experiments in Sections 3.5 and 3.6, all other cases involve a patched Linux 2.4.9 kernel running on a series of 1.4 GHz Pentium 4 based systems, connected by a Gigabit Ethernet network.

3.1 Interposition

Interposition agents [15] introduce user code between the operating system interface and applications, in order to modify or replace the services that the operating system provides. The Linux 2.4.x kernel series introduced extensions to `ptrace` to facilitate system call interception at user-level. The ptrace mechanism incurs a large overhead in the form of several context switches per system call. This significantly reduces the performance of applications under interposition that make frequent kernel service requests.

To show how the sandbox can reduce interposition overheads, the first experiment involves a series of agents that simply trace each system call made by an unmodified thttpd 2.20c web server under a range of HTTP request loads. The HTTP requests are generated from another host over a Gigabit Ethernet network using httperf. The same file is targeted in each request, which is made with a timeout of 1 second. The average rate of successful responses is recorded over 30000 requests, for three types of interposition agents:

- A kernel scheduled thread in a traditional process address space using ptrace ('Process traced')
- A kernel scheduled thread in the sandbox using ptrace ('Sandbox thread')
- An upcall handler based agent executing in the sandbox ('Sandbox upcall')

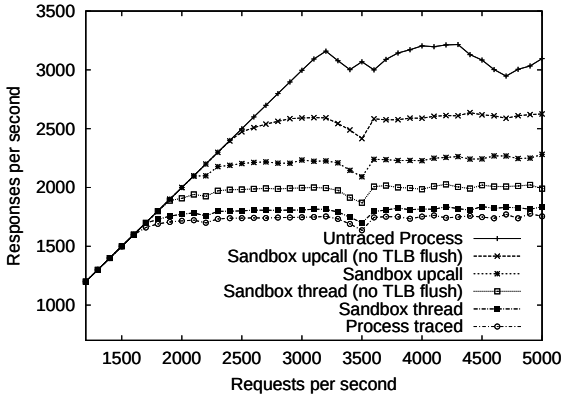

Figure 3: The performance of a thttpd web server while tracing its system calls using a variety of mechanisms.

Figure 3 shows the relative performance of these agents, compared to the situation where the web server runs untraced ('Untraced Process'). For comparison, measurements are taken for the sandbox-based agents when the sandbox is left open to user space, so that no TLB flushes are performed when context switching between the web server and the agent. This is the 'no TLB flush' case in Figure 3.

These results show that with the existing ptrace interface, the sandbox can be used to reduce interposition overheads, and that with the appropriate interface additional gains can be made. Part of the motivation here is based on the fact that projects such as User-Mode Linux use `ptrace` to intercept system calls and direct requests into a guest kernel running above a host OS. Using our technique, it is possible to implement an entire guest system such as User-Mode Linux in a sandbox, while avoiding many of the costs associated with switching between a tracing thread and the protection domain being traced.

3.2 Inter-Protection 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 [21] were carried out. These experiments also consider 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.

Two threads exchange four byte messages over connected pipes. One thread simulates an application thread in a traditional address space with a configurable instruction and data TLB working set. The second thread acts as an extension running either in a separate full address space or in the sandbox. The "application" thread fills some number of TLB entries, sends a message to the "extension" thread, and reads a reply message. The application thread then accesses its previously referenced pages. The extension thread, which has a small fixed TLB size, is executed either in the sandbox or in a traditional address space. To simulate various data TLB sizes, the application thread reads 4 bytes of data from a series of memory addresses spaced 4160 byte apart. To simulate instruction TLB sizes, the application thread performs a series of relative jumps to instructions spaced 4160 bytes apart. These spacings were chosen to avoid 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 is maintained for up to approximately 45 entries when the extension thread is mapped into the sandbox. At this point the combined data TLB demands of the operating system, the application and the extension no longer fit the 64 entries available on the Pentium 4 and each page access incurs a TLB miss. Note that for the extension thread in a traditional address space, every data page access after the IPC ping-pong incurs a TLB miss regardless of the working set size, as all TLB entries are purged on every context switch.

As shown in Figure 4(b), the instruction TLB entries

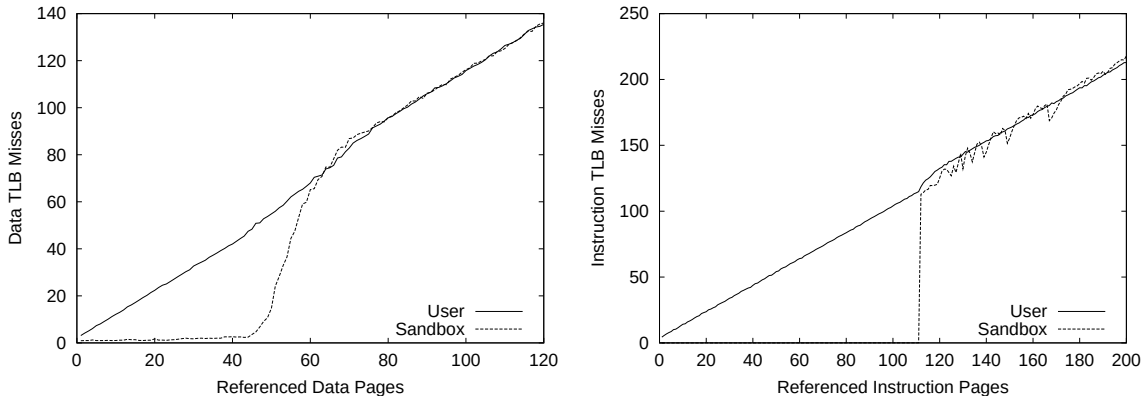

Figure 4: Effects of working set sizes in terms of (a) data, and (b) instruction pages on the number of TLB misses, for inter-protection domain communication. The 'User' case is for traditional inter-process communication, while the 'Sandbox' case shows communication costs between a process and a sandboxed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

of the application thread are preserved when the extension is located in the sandbox. No instruction TLB misses occur until the working set approaches 110 entries, which is close to the available 128 TLB entries. Thereafter, the number of instruction TLB misses are similar for both extension types. These results correspond to those in the "small spaces" work that uses the segmentation features of the x86 to implement multiple logical protection domains *within* a single address space. This shows that our user-level sandbox technique can achieve inter-protection domain communication performance similar to approaches based on specialist hardware features such as segmentation.

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

3.3 Web Server Performance Using Dynamic Content Requests

Further experiments were carried out to evaluate the performance of applications composed of traditional address space processes extended with sandbox-based code. An unmodified Apache 2.0.44 web server was configured to support FastCGI, an interface between web servers and external processes that generate dynamic content. Apache was configured to communicate over a local UNIX domain socket with a multi-threaded FastCGI process to satisfy HTTP requests coming from another host. On each request the FastCGI process reads a 36 Kilobyte XML file from the filesystem, transforming it into a 20 Kilobyte HTML response. Each request is generated by httperf with a 5 second timeout. Figure 6 shows the performance of the application, when FastCGI threads are running in a separate address space from the Apache server process (the 'User' case), and when FastCGI threads are executing within the sandbox (the 'Sandbox' case). As can be seen, the maximum response rate is improved when FastCGI threads are mapped to the sandbox. Similarly, the average request connection time remains lower for a larger request rate when FastCGI threads execute in the sandbox.

In summary, these experiments show that when the working sets of logical protection domains do not exceed the TLB limits, fast inter-protection domain communication is possible with our method. This is possible without the need for special hardware support, such as segmentation.

3.4 Microbenchmarks

Operation	Cost in CPU Cycles
Upcall including TLB flush/reload	11000
TLB flush and reload	8500
Raw upcall	2500
Signal delivery (current process)	6000
Signal delivery (different process)	46000

Table 1: Microbenchmarks taken on a 1.4GHz Pentium 4, 512 Megs RAM. Cycles given above are based on the time stamp counter.

Table 1 presents a number of microbenchmarks that point to the effectiveness of using our fast upcalls method, for invoking sandbox code. In this table, the fast upcall costs are shown for the SYSEXIT/ENTER implementation. 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

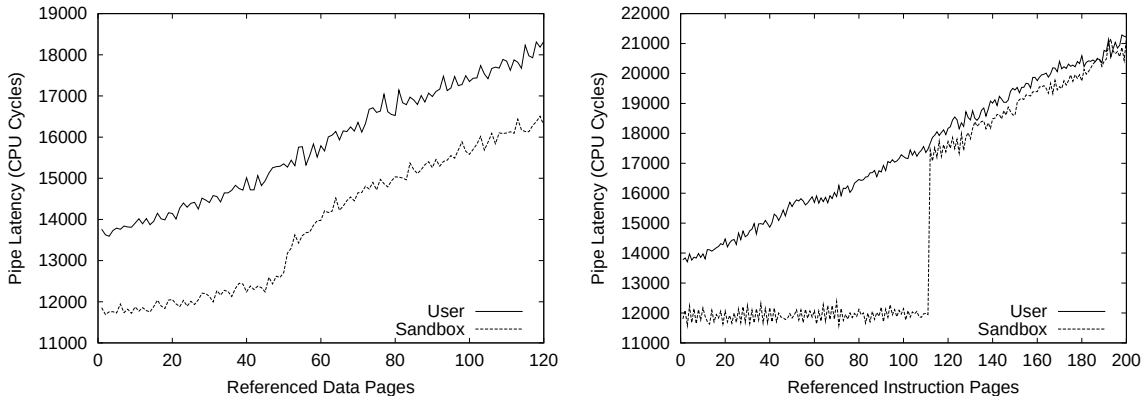

Figure 5: Latency of communication via a pipe between two protection domains, as a function of working set sizes in terms of (a) data, and (b) instruction pages.

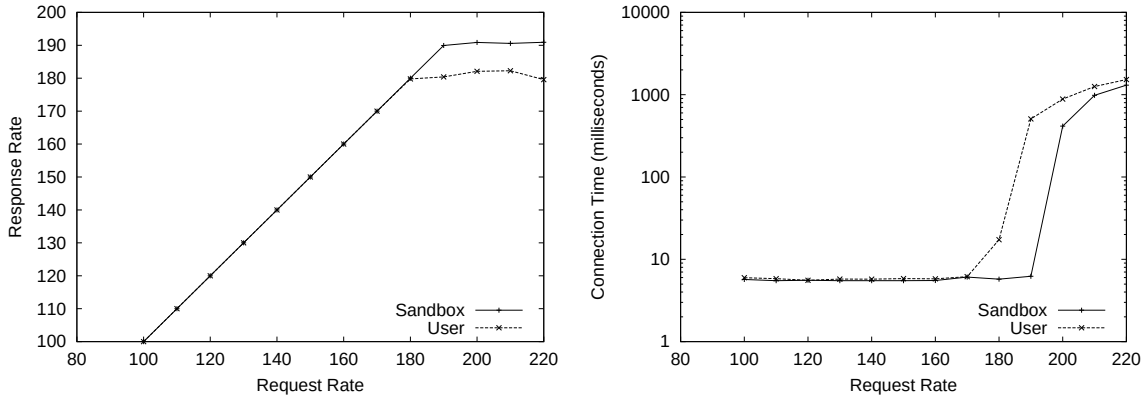

Figure 6: Performance of an unmodified Apache 2.0.44 web server handling dynamic content requests with the help of FastCGI processes mapped to both separate address spaces ('User' case) and sandboxed memory regions ('Sandbox' case).

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. 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, placed on the upcall stack before any arguments and 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 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 much lower than the cost of an upcall, as no hardware protection overheads are involved, 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 4 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.

While the previous table does not include the costs of using activation records, our tests suggest they incur less than 10% additional overheads compared to SYSEXIT/ENTER. This is still better than using signals to invoke user-level services, since activation records have fixed overheads that are not dependent on system load, process behavior, or CPU scheduling policies.

3.5 System Service Extensions in the Sandbox

In this set of experiments, we compare the performance of kernel-level extensions against user-level approaches

for monitoring and adapting system resource usage. The aim is 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. This set of experiments uses a standalone 550 Mhz Pentium III with 256 MB of RAM. In this case, a user-level sandbox is implemented on a patched Linux 2.4.20 kernel.

Four different methods of dynamically managing CPU usage are compared, for a set of processes each with specific resource requirements over finite windows of real-time. Further details about the exact setup of these experiments can be found in our earlier work [25]. The four methods implement 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 have target CPU demands of $40mS$ every period of $400mS$, $100mS$ every period of $500mS$, and $60mS$ every period of $200mS$, respectively. For simplicity, the processes are all CPU-bound, have memory footprints less than $4KB$ when stripped of symbols, and merely iterate over a number of integer computations. Note that in similar experiments, more realistic and complex application processes encode a number of video frames into groups of pictures. Results of these experiments are not included due to space constraints, and because they show similar performance patterns to those shown in this section. In any case, processes P_1, P_2 and P_3 have static real-time priorities initialized to $80 * (target/period)$, 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 are assigned real-time priorities of 97 or higher, thereby ensuring the whole system continues to function responsively.

The kernel-based service manager is invoked once every $10mS$ from a Linux timer queue, to monitor the CPU allocations of the three CPU-bound processes. Similarly, the upcall-based service manager is invoked once every $10mS$ by upcall events triggered from a timer bottom half. Corresponding handler functions in each case adjust the timeslice of the three process as necessary, using the same PID [2] controller described in prior experiments [25]. A guard function allows a process's timeslice to increase as long as its average CPU usage, measured over twice its period, is not above the target utilization.

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 inter-burst times of 10 seconds and average geometric burst lengths of 3 seconds. 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.

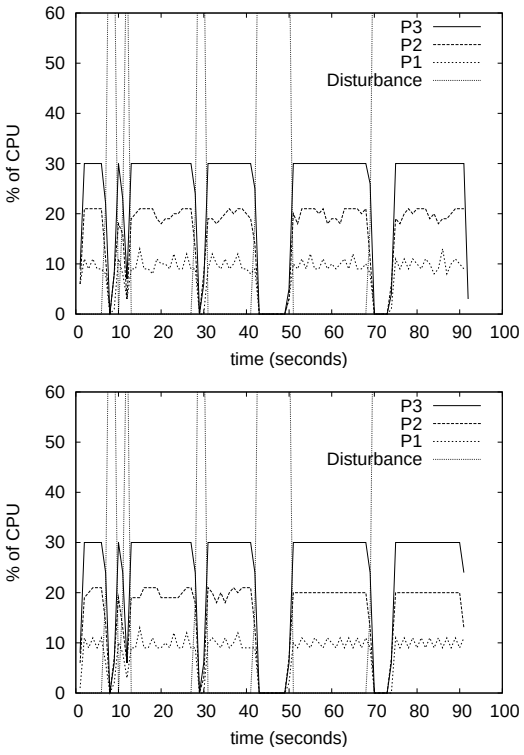

Figure 7: CPU service management controlled by (a) a user-level real-time process, and (b) a sandboxed thread.

Figures 7 and 8 show the abilities of each service management method to maintain CPU allocations of the three processes at their target levels. Both the process-

[2] Proportional plus integral plus derivative.

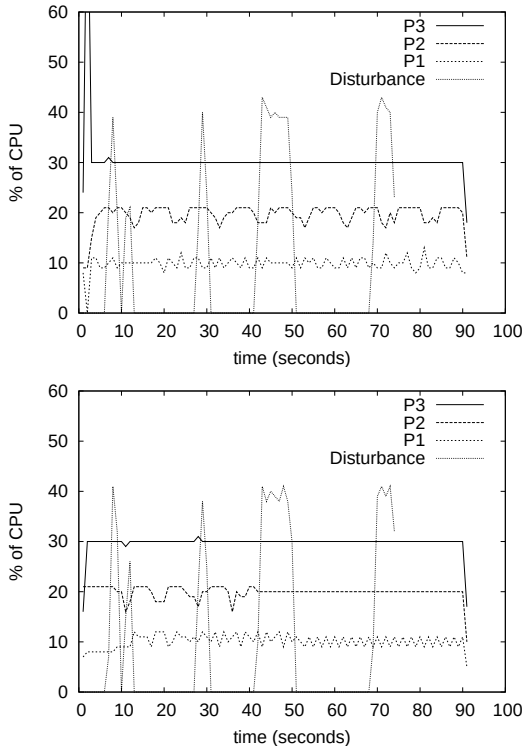

Figure 8: CPU service management controlled by (a) a pure upcall function in the sandbox, and (b) a kernel bottom-half handler.

and thread-based approaches suffer from the need for scheduling by the kernel in order 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, that do not entirely depend upon the underlying nature of the kernel's scheduling policy.

As can be seen from Figure 8, 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 user-level.

3.6 User-Level Networking in the Sandbox

As a further application of our approach we have implemented a network stack in the sandbox, that avoids copying and processing within the kernel. 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 a special-purpose routing protocol in user space using this technique.

In the following experiments, several IBM x-series 305 servers are connected via Tigon3 Gigabit Ethernet cards. Each machine has a 2.4GHz Pentium 4 CPU and 1024MB RAM. With slight modifications to the Ethernet driver in a Linux 2.4.20 kernel, we are able to DMA data directly into sandboxed memory. Details of this extension to our sandbox system can be found in an accompanying paper [26].

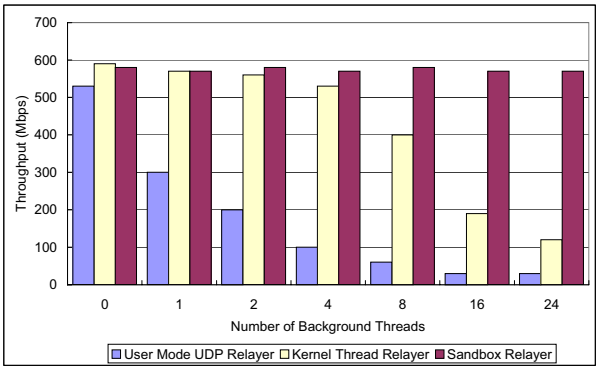

Figure 9: Throughput comparison of an optimized sandbox stack versus alternative user- and kernel-level implementations.

Figure 9 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 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.

Summary: The experiments discussed in this section primarily focus on performance gains of our approach, rather than the novelty of particular sandboxed services. However, user-level sandboxing is flexible enough to support a number of fairly powerful service extensions, whose capabilities can be controlled by upcalls from the trusted kernel. For example, sandbox services may be granted controlled access to I/O devices by upcalls that enable access to a subset of all available IO port addresses. Results show that e.g., jitter-sensitive applications may suffer from scheduling and switching overheads (and policies) associated with traditional user-level services isolated in separate protection domains. This is evident from our CPU service management experiments, similar to those already published as part of our safe kernel extension work [25].

4 Related Work

There have been a number of related research efforts that focus on OS structure, extensibility, safety, and service invocation. For example, micro-kernels such as Mach [27] 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, many micro-kernel implementations have either suffered from significant inter-protection domain communication costs or lack of portability [14].

Other OS approaches, such as the Exokernel [18], try to efficiently multiplex hardware resources among applications linked with library operating systems. In common with the Exokernel approach, user-level sandboxing enables services and extensions to be linked into process address spaces, along with library code they may use. However, our approach provides a method to execute service extensions in arbitrary address spaces, without scheduling and context-switching overheads.

We require service extensions for multiple different applications to be written by a trusted source, or to be guaranteed memory-safe by using techniques such as type-safe languages or software fault isolation. The SPIN operating system [5], for example, uses the type-safety of Modula-3 to enforce memory protection of untrusted extensions linked with core system abstractions. In our approach type-safe language support is useful to isolate extensions from one another, and to prevent them from unauthorized access to the active process-private address space. However, we use page-level protection to provide a clean separation between extensions and the kernel.

Other extensible systems use transaction schemes [6], process-private address spaces [4], and special hardware support to isolate or guard against the potential ill-effects of untrusted service extensions. For example, Palladium [7] leverages both segmentation and multiple rings of protection to support both user- and kernel-level extensions. Interestingly, Palladium reorganizes application and extension code so that extensions are always located in a less privileged ring of protection than the code that invokes their services. However, this method is primarily targeted at x86-based systems and relies on hardware support for protection. User-level sandboxing does not require segmentation and multiple rings of protection.

Another area of research related to ours has focused on service invocation, kernel event notifications [23, 24] and upcalls [19]. 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. For example, FreeBSD's 'kqueues' [24] and Banga et al's kernel event notifications [23] alleviate many of the costs associated with traditional methods of event delivery (e.g., using `poll()` and `select()`). However, they do not offer upcalls mechanisms akin to the mirror image of a system call, capable of triggering handlers without scheduling overheads. As part of our upcall mechanism, we intend to implement a general kernel event notification scheme based on these approaches, but with the ability to directly invoke a function in a sandbox.

Finally, observe that our work is not to be confused with user-level resource-constrained sandboxing [28], by Chang, Itzkovitz and Karamcheti. Their work focuses on the use of sandboxes to enforce quantitative restrictions on resource usage. They propose a method for instrumenting an application, to intercept requests for resources such as CPU cycles, memory and bandwidth. The emphasis of our work is to provide safe and efficient shared memory support for the execution of user-level extensions in arbitrary process address spaces. While we assume page-level protection is provided by hardware, a finer-granularity protection scheme such as Mondrian [29] would be desirable. That said, the hardware for such an approach is not yet readily available.

5 Conclusions and Future Work

This paper describes a safe and efficient method for user-level extensibility, employing a shared virtual memory area common to all address spaces. Our approach allows applications to register sandboxed code with the

system, that may be executed in the context of *any* process, thereby avoiding unnecessary scheduling and context switching overheads.

Upcalls from the trusted kernel are required to toggle page-level protection bits, to allow extension code to execute in a user-level sandbox. Since extensions are executed at user-level, they may utilize libraries and make system calls. Our approach differs from other implementations that require special hardware support, such as segmentation or tagged TLBs, to either implement multiple protection domains in a single address space, or to support fast switching between address spaces. Likewise, we do not require the entire system to be written in a type-safe language, to provide fine-grained protection domains. Instead, our user-level sandboxing mechanism requires only paged-based virtual memory support, given that sandboxed extensions are either written by a trusted source or are guaranteed to be memory-safe. The mechanism benefits most by the use of a small number of extended (or super-) pages on architectures where such pages are cached in untagged TLBs. Hardware support for a variety of super-pages is now appearing on different architectures (e.g., the UltraSPARC and IA-64). This will enable larger and more extensions to be mapped into a sandbox.

In contrast to work such as software fault isolation [11], which assumes untrusted code is isolated in its own fault domain in a *single* address space, we support the ability to place extension code in a memory region that is shared between *separate* address spaces. This makes our technique suitable for existing systems that support multiple process-private address spaces, rather than a single memory area, as in systems such as DOS. We envision our approach as laying the foundations for a method of implementing *first-class* user-level services that are tailored to the needs of specific applications. By first-class, we mean any service that has the same capabilities and privileges of traditional kernel services, with the exception that the kernel may always revoke access rights to any service abusing its capabilities. Our philosophy is in keeping with the micro-kernel design but unlike past micro-kernel implementations we can invoke user-level services with low overhead while not requiring esoteric hardware support.

Future work involves a thorough study of the costs of software fault isolation methods, and also type-safe languages such as Cyclone [12], to enforce memory-protection on multiple untrusted extensions in a user-level sandbox. We will also address issues related to CPU protection, to enforce that sandbox extensions do not execute for unbounded periods. Our earlier work on safe kernel extensions [25] suggests that CPU protection can be enforced by requiring extensions to reserve a CPU share before they are registered with the system. Each extension may have a corresponding timer that is decremented every clock tick during execution. A trap to the controlling kernel is issued when any extension executes beyond its CPU quota. At this point some form of abortive action may be taken.

The user-level sandboxing software is available upon request. The current implementation requires a minimal set of changes to the core Linux kernel. In recent work outside the scope of this paper, we have successfully applied binary-rewriting techniques to make the necessary minor changes to the kernel, without having to patch and recompile any source code. To date, the only requirement is that a user reboots his or her machine to reserve an area of physical memory for the user-level sandbox.

References

[1] P. Joubert, R. King, R. Neves, M. Russinovich, and J. M. Tracey, "High-performance memory-based web servers: Kernel and user-space performance," in *Proceedings of the USENIX Annual Technical Conference*, (Boston, Massachusetts), June 2001.

[2] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Exploiting gray-box knowledge of buffer-cache management," in *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[3] C. Small and M. I. Seltzer, "A comparison of OS extension technologies," in *USENIX Annual Technical Conference*, pp. 41–54, 1996.

[4] D. Ghormley, S. Rodrigues, D. Petrou, and T. Anderson, "SLIC: An extensibility system for commodity operating systems," in *Proceedings of the 1998 USENIX Annual Technical Conference*, 1998.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, and B. E. Chambers, "Extensibility, safety, and performance in the SPIN operating system," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 267–284, 1995.

[6] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith, "Dealing with disaster: Surviving misbehaved kernel extensions," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[7] T. Chiueh, G. Venkitachalam, and P. Pradhan, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," in *Proceeedings of the ACM Symposium on Operating Systems Principles*, pp. 140–153, 1999.

[8] M. Clarke and G. Coulson, "An architecture for dynamically extensible operating systems," in *Proceedings of the 4th International Conference on Configurable Distributed Systems*, May 1998.

[9] S. Chandra and A. Vahdat, "Application-specific network management for energy-aware streaming of popular multimedia format," in *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[10] M. E. Fiuczynski and B. N. Bershad, "An extensible protocol architecture for application-specific networking," in *Proceedings of the USENIX Annual Technical Conference*, (San Diego, CA), January 1996.

[11] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Software-based fault isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[12] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proceedings of the USENIX Annual Technical Conference*, (Monterey, California), June 2002.

[13] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pp. 229–243, 1996.

[14] J. Liedtke, "On μ-kernel construction," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, ACM, December 1995.

[15] M. B. Jones, "Interposition agents: Transparently interposing user code at the system interface," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 80–93, December 1993.

[16] R. P. Goldberg, "Survey of virtual machine research," *IEEE Computer Magazine*, vol. 7, no. 6, pp. 34–45, 1974.

[17] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMWare workstation's hosted virtual machine monitor," in *Proceedings of the USENIX Annual Technical Conference*, June 2001.

[18] D. R. Engler, M. F. Kaashoek, and J. O. Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[19] D. Clark, "The structuring of systems using upcalls," in *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 171–180, ACM, 1985.

[20] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A user-level network interface for parallel and distributed computing," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 40–53, 1995.

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

[22] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volumes 1, 2 and 3*.

[23] G. Banga, J. C. Mogul, and P. Druschel, "A scalable and explicit event delivery mechanism for UNIX," in *Proceedings of the USENIX Annual Technical Conference*, (Monterey, CA), June 1999.

[24] J. Lemon, "Kqueue - a generic and scalable event notification facility," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2001.

[25] R. West and J. Gloudon, "'QoS safe' kernel extensions for real-time resource management," in *the 14th EuroMicro International Conference on Real-Time Systems*, June 2002.

[26] X. Qi, G. Parmer, R. West, J. Gloudon, and L. Hernandez, "Efficient end-host architecture for high performance communication using user-level sandboxing," Tech. Rep. BUCS-TR-2004-009, Boston University, 2004. Submitted for publication.

[27] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development," in *Summer USENIX Conference*, pp. 93–112, 1986.

[28] F. Chang, A. Itzkovitz, and V. Karamcheti, "User-level resource-constrained sandboxing," in *Proceedings of the 4th USENIX Windows Systems Symposium*, 2000.

[29] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," in *ASPLOS-X*, ACM, 2002.