FlexSC: Flexible System Call Scheduling with Exception-Less System Calls

Livio Soares
University of Toronto

Michael Stumm
University of Toronto

Abstract

For the past 30+ years, system calls have been the *de facto* interface used by applications to request services from the operating system kernel. System calls have almost universally been implemented as a *synchronous* mechanism, where a special processor instruction is used to yield user-space execution to the kernel. In the first part of this paper, we evaluate the performance impact of traditional synchronous system calls on system intensive workloads. We show that synchronous system calls negatively affect performance in a significant way, primarily because of pipeline flushing and pollution of key processor structures (e.g., TLB, data and instruction caches, etc.).

We propose a new mechanism for applications to request services from the operating system kernel: *exception-less system calls*. They improve processor efficiency by enabling flexibility in the scheduling of operating system work, which in turn can lead to significantly increased temporal and spacial locality of execution in both user and kernel space, thus reducing pollution effects on processor structures. Exception-less system calls are particularly effective on multicore processors. They primarily target highly threaded server applications, such as Web servers and database servers.

We present FlexSC, an implementation of exception-less system calls in the Linux kernel, and an accompanying user-mode thread package (FlexSC-Threads), binary compatible with POSIX threads, that translates legacy synchronous system calls into exception-less ones *transparently* to applications. We show how FlexSC improves performance of Apache by up to 116%, MySQL by up to 40%, and BIND by up to 105% while requiring no modifications to the applications.

1 Introduction

System calls are the *de facto* interface to the operating system kernel. They are used to request services offered by, and implemented in the operating system kernel. While

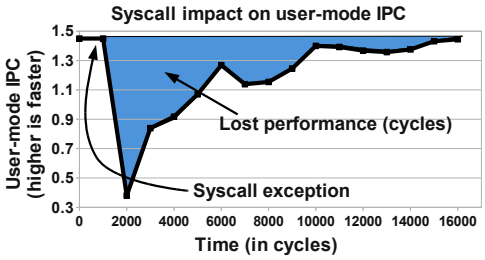


Figure 1: User-mode instructions per cycles (IPC) of Xalan (from SPEC CPU 2006) in response to a system call exception event, as measured on an Intel Core i7 processor.

different operating systems offer a variety of different services, the basic underlying system call mechanism has been common on all commercial multiprocessed operating systems for decades. System call invocation typically involves writing arguments to appropriate registers and then issuing a special machine instruction that raises a synchronous exception, immediately yielding user-mode execution to a kernel-mode exception handler. Two important properties of the traditional system call design are that: (1) a processor exception is used to communicate with the kernel, and (2) a synchronous execution model is enforced, as the application expects the completion of the system call before resuming user-mode execution. Both of these effects result in performance inefficiencies on modern processors.

The increasing number of available transistors on a chip (Moore's Law) has, over the years, led to increasingly sophisticated processor structures, such as superscalar and out-of-order execution units, multi-level caches, and branch predictors. These processor structures have, in turn, led to a large increase in the performance *potential* of software, but at the same time there is a widening gap between the performance of efficient software and the performance of inefficient software, primarily due to the increasing disparity of accessing different processor resources (e.g., registers vs. caches vs. memory). Server and system-intensive workloads, which are of particular

interest in our work, are known to perform well below the potential processor throughput [11, 12, 19]. Most studies attribute this inefficiency to the lack of locality. *We claim that part of this lack of locality, and resulting performance degradation, stems from the current synchronous system call interface.*

Synchronous implementation of system calls negatively impacts the performance of system intensive workloads, both in terms of the *direct* costs of mode switching and, more interestingly, in terms of the *indirect* pollution of important processor structures which affects both user-mode and kernel-mode performance. A motivating example that quantifies the impact of system call pollution on application performance can be seen in Figure 1. It depicts the user-mode instructions per cycles (kernel cycles and instructions are ignored) of one of the SPEC CPU 2006 benchmarks (Xalan) immediately before and after a `pwrite` system call. There is a significant drop in instructions per cycle (IPC) due to the system call, and it takes up to 14,000 cycles of execution before the IPC of this application returns to its previous level. As we will show, this performance degradation is mainly due to interference caused by the kernel on key processor structures.

To improve locality in the execution of system intensive workloads, we propose a new operating system mechanism: the **exception-less system call**. An exception-less system call is a mechanism for requesting kernel services that does not require the use of synchronous processor exceptions. In our implementation, system calls are issued by writing kernel requests to a reserved **syscall page**, using normal memory store operations. The actual execution of system calls is performed asynchronously by special in-kernel **syscall threads**, which post the results of system calls to the syscall page after their completion.

Decoupling the system call execution from its invocation creates the possibility for flexible system call scheduling, offering optimizations along two dimensions. The first optimization allows for the deferred batch execution of system calls resulting in increased temporal locality of execution. The second provides the ability to execute system calls on a separate core, in parallel to executing user-mode threads, resulting in spatial, per core locality. In both cases, system call threads become a simple, but powerful abstraction.

One interesting feature of the proposed decoupled system call model is the possibility of *dynamic* core specialization in multicore systems. Cores can become temporarily specialized for either user-mode or kernel-mode execution, depending on the current system load. We describe how the operating system kernel can dynamically adapt core specialization to the demands of the workload.

One important challenge of our proposed system is how to best use the exception-less system call interface. One option is to rewrite applications to directly interface with the exception-less system call mechanism. We believe the lessons learned by the systems community with event-driven servers indicate that directly using exception-less system calls would be a daunting software engineering task. For this reason, we propose a new M-on-N threading package (M user-mode threads executing on N kernel-visible threads, with $M >> N$). The main purpose of this threading package is to harvest independent system calls by switching threads, in user-mode, whenever a thread invokes a system call.

This research makes the following contributions:

- We quantify, at fine granularity, the impact of synchronous mode switches and system call execution on the micro-architectural processor structures, as well as on the overall performance of user-mode execution.
- We propose a new operating system mechanism, the exception-less system call, and describe an implementation, FlexSC[1], in the Linux kernel.
- We present a M-on-N threading system, compatible with PThreads, that transparently uses the new exception-less system call facility.
- We show how exception-less system calls coupled with our M-on-N threading system improves performance of important system-intensive highly threaded workloads: Apache by up to 116%, MySQL by to 40%, and BIND by up to 105%.

2 The (Real) Costs of System Calls

In this section, we analyze the performance costs associated with a traditional, synchronous system call. We analyze these costs in terms of mode switch time, the system call footprint, and the effect on user-mode and kernel-mode IPC. We used the Linux operating system kernel and an Intel Nehalem (Core i7) processor, along with its performance counters to obtain our measurements. However, we believe the lessons learned are applicable to most modern high-performance processors[2] and other operating system kernels.

2.1 Mode Switch Cost

Traditionally, the performance cost attributed to system calls is the *mode switch time*. The mode switch time consists of the time necessary to execute the appropriate system call instruction in user-mode, resuming execution in an elevated protection domain (kernel-mode), and the return of control back to user-mode. Modern processors implement the mode switch as a processor exception: flushing the user-mode pipeline, saving a few registers onto the

[1] Pronounced as "flex" (/'fleks/).

[2] Experiments performed on an older PowerPC 970 processor yielded similar insights than the ones presented here.

Syscall	Instructions	Cycles	IPC	i-cache	d-cache	L2	L3	d-TLB
`stat`	4972	13585	0.37	32	186	660	2559	21
`pread`	3739	12300	0.30	32	294	679	2160	20
`pwrite`	5689	31285	0.18	50	373	985	3160	44
`open+close`	6631	19162	0.34	47	240	900	3534	28
`mmap+munmap`	8977	19079	0.47	41	233	869	3913	7
`open+write+close`	9921	32815	0.30	78	481	1462	5105	49

Table 1: System call footprint of different processor structures. For the processors structures (caches and TLB), the numbers represent number of entries evicted; the cache line for the processor is of 64-bytes. i-cache and d-cache refer to the instruction and data sections of the L1 cache, respectively. The d-TLB represents the data portion of the TLB.

kernel stack, changing the protection domain, and redirecting execution to the registered exception handler. Subsequently, return from exception is necessary to resume execution in user-mode.

We measured the mode switch time by implementing a new system call, `gettsc` that obtains the time stamp counter of the processor and immediately returns to user-mode. We created a simple benchmark that invoked `gettsc` 1 billion times, recording the time-stamp before and after each call. The difference between each of the three time-stamps identifies the number of cycles necessary to enter and leave the operating system kernel, namely 79 cycles and 71 cycles, respectively. The total round-trip time for the `gettsc` system call is modest at 150 cycles, being less than the latency of a memory access that misses the processor caches (250 cycles on our machine).[3]

2.2 System Call Footprint

The mode switch time, however, is only part of the cost of a system call. During kernel-mode execution, processor structures including the L1 data and instruction caches, translation look-aside buffers (TLB), branch prediction tables, prefetch buffers, as well as larger unified caches (L2 and L3), are populated with kernel specific state. The replacement of user-mode processor state by kernel-mode processor state is referred to as the processor state *pollution* caused by a system call.

To quantify the pollution caused by system calls, we used the Core i7 hardware performance counters (HPC). We ran a high instruction per cycle (IPC) workload, Xalan, from the SPEC CPU 2006 benchmark suite that is known to invoke few system calls. We configured an HPC to trigger infrequently (once every 10 million user-mode instructions) so that the processor structures would be dominated with application state. We then set up the HPC exception handler to execute specific system calls, while measuring the replacement of application state in the processor structures caused by kernel execution (but not by the performance counter exception handler itself).

[3]For all experiments presented in this paper, user-mode applications execute in 64-bit mode and when using synchronous system calls, use the "syscall" x86_64 instruction, which is currently the default in Linux.

Table 1 shows the footprint on several processor structures for three different system calls and three system call combinations. The data shows that, even though the number of i-cache lines replaced is modest (between 2 and 5 KB), the number of d-cache lines replaced is significant. Given that the size of the d-cache on this processor is 32 KB, we see that the system calls listed pollute at least half of the d-cache, and almost all of the d-cache in the "open+write+close" case. The 64 entry first level d-TLB is also significantly polluted by most system calls. Finally, it is interesting to note that the system call impact on the L2 and L3 caches is larger than on the L1 caches, primarily because the L2 and L3 caches use more aggressive prefetching.

2.3 System Call Impact on User IPC

Ultimately, the most important measure of the real cost of system calls is the performance impact on the application. To quantify this, we executed an experiment similar to the one described in the previous subsection. However, instead of measuring kernel-mode events, we only measured user-mode instructions per cycle (IPC), ignoring all kernel execution. Ideally, user-mode IPC should not decrease as a result of invoking system calls, since the cycles and instructions executed as part of the system call are ignored in our measurements. In practice, however, user-mode IPC is affected by two sources of overhead:

Direct: The processor exception associated with the system call instruction that flushes the processor pipeline.

Indirect: System call pollution on the processor structures, as quantified in Table 1.

Figures 2 and 3 show the degradation in user-mode IPC when running Xalan (from SPEC CPU 2006) and SPECJBB, respectively, given different frequencies of `pwrite` calls. These benchmarks were chosen since they have been created to avoid significant use of system services, and should spend only 1-2% of time executing in kernel-mode. The graphs show that different workloads can have different sensitivities to system call pollution. Xalan has a baseline user-mode IPC of 1.46, but the IPC degrades by up to 65% when executing a `pwrite` every 1,000-2,000 instructions, yielding an IPC between 0.58 and 0.50.

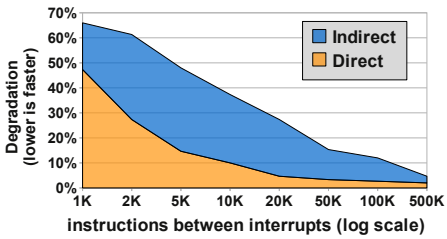


Figure 2: System call (`pwrite`) impact on user-mode IPC as a function of system call frequency for Xalan.

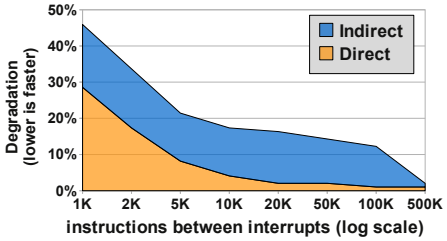


Figure 3: System call (`pwrite`) impact on user-mode IPC as a function of system call frequency for SPEC JBB.

SPEC-JBB has a slightly lower baseline of 0.97, but still observes a 45% degradation of user-mode IPC.

The figures also depict the breakdown of user-mode IPC degradation due to direct and indirect costs. The degradation due to the direct cost was measured by issuing a null system call, while the indirect portion is calculated subtracting the direct cost from the degradation measured when issuing a `pwrite` system call. For high frequency system call invocation (once every 2,000 instructions, or less), the direct cost of raising an exception and subsequent flushing of the processor pipeline is the largest source of user-mode IPC degradation. However, for medium frequencies of system call invocation (once per 2,000 to 100,000 instructions), the *indirect* cost of system calls is the dominant source of user-mode IPC degradation.

To understand the implication of these results on typical server workloads, it is necessary to quantify the system call frequency of these workloads. The average user-mode instruction count between consecutive system calls for three popular server workloads are shown in Table 2. For this frequency range in Figures 2 and 3 we observe user-mode IPC performance degradation between 20% and 60%. While the excecution of the server workloads listed in Table 2 is not identical to that of Xalan or SPEC-JBB, the data presented here indicates that server workloads suffer from significant performance degradation due to processor pollution of system calls.

Workload (server)	Instructions per Syscall
DNSbench (BIND)	2445
ApacheBench (Apache)	3368
Sysbench (MySQL)	12435

Table 2: The average number of instructions executed on different workloads before issuing a syscall.

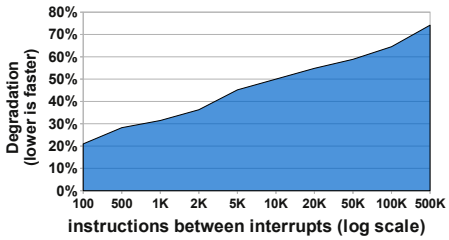


Figure 4: System call (`pwrite`), impact on kernel-mode IPCs for x as a function of system call frequency.

2.4 Mode Switching Cost on Kernel IPC

The lack of locality due to frequent mode switches also negatively affects kernel-mode IPC. Figure 4 shows the impact of different system call frequencies on the kernel-mode IPC. As expected, the performance trend is opposite to that of user-mode execution. The more frequent the system calls, the more kernel state is maintained in the processor.

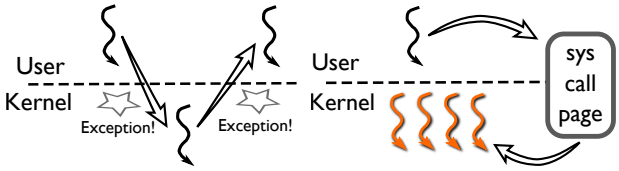
Note that the kernel-mode IPC listed in Table 1 for different system calls ranges from 0.18 to 0.47, with an average of 0.32. This is significantly lower than the 1.47 and 0.97 user-mode IPC for Xalan and SPEC-JBB, respectively; up to *8x* slower.

3 Exception-Less System Calls

To address (and partially eliminate) the performance impact of traditional, synchronous system calls on system intensive workloads, we propose a new operating system mechanism called **exception-less system call**. Exception-less system call is a mechanism for requesting kernel services that does not require the use of synchronous processor exceptions. The key benefit of exception-less system calls is the flexibility in scheduling system call execution, ultimately providing improved locality of execution of both user and kernel code. We explore two use cases:

System call batching: Delaying the execution of a series of system calls and executing them in batches minimizes the frequency of switching between user and kernel execution, eliminating some of the mode switch overhead and allowing for improved *temporal locality*. This improves both the direct and indirect costs of system calls.

Core specialization: In multicore systems, exception-less system calls allow a system call to be scheduled on a core different than the one on which the system call was invoked. Scheduling system calls on a separate processor core allows for improved *spatial locality* and with it lower



(a) Traditional, sync. system call (b) Exception-less system call

Figure 5: Illustration of synchronous and exception-less system call invocation. The left diagram shows the sequential nature of exception-based system calls, while the right diagram depicts exception-less user and kernel communication through shared memory.

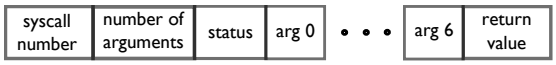


Figure 6: 64-byte syscall entry from the syscall page.

indirect costs. In an ideal scenario, no mode switches are necessary, eliminating the direct cost of system calls.

The design of exception-less system calls consists of two components: (1) an exception-less interface for user-space threads to register system calls, along with (2) an in-kernel threading system that allows the delayed (asynchronous) execution of system calls, without interrupting or blocking the thread in user-space.

3.1 Exception-Less Syscall Interface

The interface for exception-less system calls is simply a set of memory pages that is shared amongst user and kernel space. The shared memory page, henceforth referred to as **syscall page**, is organized to contain exception-less system call entries. Each entry contains space for the request status, system call number, arguments, and return value.

With traditional synchronous system calls, invocation occurs by populating predefined registers with system call information and issuing a specific machine instruction that immediately raises an exception. In contrast, to issue an exception-less system call, the user-space threads must find a free entry in the syscall page and populate the entry with the appropriate values using regular store instructions. The user-space thread can then continue executing without interruption. It is the responsibility of the user-space thread to later verify the completion of the system call by reading the status information in the entry. None of these operations, issuing a system call or verifying its completion, causes exceptions to be raised.

3.2 Syscall Pages

Syscall pages can be viewed as a table of syscall entries, each containing information specific to a single system call request, including the system call number, arguments, status (free/submitted/busy/done), and the result (Figure 6). In our 64-bit implementation, we have organized each entry to occupy 64 bytes. This size comes from the Linux ABI which allows any system call to have up to 6 arguments, and a return value, totalling 56 bytes. Although the remaining 3 fields (syscall number, status and number of arguments) could be packed in less than the remaining 8 bytes, we selected 64 bytes because 64 is a divisor of popular cache line sizes of today's processor.

To issue an exception-less system call, the user-space thread must find an entry in one of its syscall pages that contain a *free* status field. It then writes the syscall number and arguments to the entry. Lastly, the status field is changed to *submitted*[4], indicating to the kernel that the request is ready for execution. The thread must then check the status of the entry until it becomes *done*, consume the return value, and finally set the status of the entry to *free*.

3.3 Decoupling Execution from Invocation

Along with the exception-less interface, the operating system kernel must support delayed execution of system calls. Unlike exception-based system calls, the exception-less system call interface does not result in an explicit kernel notification, nor does it provide an execution stack. To support decoupled system call execution, we use a special type of kernel thread, which we call **syscall thread**. Syscall threads always execute in kernel mode, and their sole purpose is to pull requests from syscall pages and execute them on behalf of the user-space thread. Figure 5 illustrates the difference between traditional synchronous system calls, and our proposed split system call model.

The combination of the exception-less system call interface and independent syscall threads allows for great flexibility in the scheduling the execution of system calls. Syscall threads may wake up only after user-space is unable to make further progress, in order to achieve temporal locality of execution on the processor. Orthogonally, syscall threads can be scheduled on a different processor core than that of the user-space thread, allowing for spatial locality of execution. On modern multicore processors, cache to cache communication is relatively fast (in the order of 10s of cycles), so communicating the entries of syscall pages from a user-space core to a kernel core, or vice-versa, should only cause a small number of processor stalls.

3.4 Implementation – FlexSC

Our implementation of the exception-less system call mechanism is called **FlexSC** (Flexible System Call) and was prototyped as an extension to the Linux kernel. Although our implementation was influenced by a mono-

[4]User-space must update the status field last, with an appropriate memory barrier, to prevent the kernel from selecting incomplete syscall entries to execute.

lithic kernel architecture, we believe that most of our design could be effective with other kernel architectures, e.g., exception-less micro-kernel IPCs, and hypercalls in a paravirtualized environment.

We have implemented FlexSC for the x86_64 and PowerPC64 processor architectures. Porting FlexSC to other architectures is trivial; a single function is needed, which moves arguments from the syscall page to appropriate registers, according to the ABI of the processor architecture. Two new system calls were added to Linux as part of FlexSC, `flexsc_register` and `flexsc_wait`.

`flexsc_register()` This system call is used by processes that wish to use the FlexSC facility. Making this registration procedure explicit is not strictly necessary, as processes can be registered with FlexSC upon creation. We chose to make it explicit mainly for convenience of prototyping, giving us more control and flexibility in user-space. One legitimate reason for making registration explicit is to avoid the extra initialization overheads incurred for processes that do not use exception-less system calls.

Invocation of the `flexsc_register` system call must use the traditional, exception-based system call interface to avoid complex bootstrapping; however, since this system call needs to execute only once, it does not impact application performance. Registration involves two steps: mapping one or more syscall pages into user-space virtual memory space, and spawning one syscall thread per entry in the syscall pages.

`flexsc_wait()` The decoupled execution model of exception-less system calls creates a challenge in user-space execution, namely what to do when the user-space thread has nothing more to execute and is waiting on pending system calls. With the proposed execution model, the OS kernel loses the ability to determine when a user-space thread should be put to sleep. With synchronous system calls, this is simply achieved by putting the thread to sleep while it is executing a system call if the call blocks waiting for a resource.

The solution we adopted is to require that the user explicitly communicate to the kernel that it cannot progress until one of the issued system calls completes by invoking the `flexsc_wait` system call. We implemented `flexsc_wait` as an exception-based system call, since execution should be synchronously directed to the kernel. FlexSC will later wake up the user-space thread when at least one of posted system calls are complete.

3.5 Syscall Threads

Syscall threads is the mechanism used by FlexSC to allow for exception-less execution of system calls. The Linux system call execution model has influenced some implementation aspects of syscall threads in FlexSC: (1) the virtual address space in which system call execution occurs is the address space of the corresponding process, and (2) the current thread context can be used to block execution should a necessary resource not be available (for example, waiting for I/O).

To resolve the virtual address space requirement, syscall threads are created during `flexsc_register`. Syscall threads are thus "cloned" from the registering process, resulting in threads that share the original virtual address space. This allows the transfer of data from/to user-space with no modification to Linux's code.

FlexSC would ideally never allow a syscall thread to sleep. If a resource is not currently available, notification of the resource becoming available should be arranged, and execution of the next pending system call should begin. However, implementing this behavior in Linux would require significant changes and a departure from the basic Linux architecture. Instead, we adopted a strategy that allows FlexSC to maintain the Linux thread blocking architecture, as well as requiring only minor modifications (3 lines of code) to Linux context switching code, by creating multiple syscall threads for each process that registers with FlexSC.

In fact, FlexSC spawns as many syscall threads as there are entries available in the syscall pages mapped in the process. This provisions for the worst case where every pending system call blocks during execution. Spawning hundreds of syscall threads may seem expensive, but Linux in-kernel threads are typically much lighter weight than user threads: all that is needed is a `task_struct` and a small, 2-page, stack for execution. All the other structures (page table, file table, etc.) are shared with the user process. In total, only 10KB of memory is needed per syscall thread.

Despite spawning multiple threads, only *one* syscall thread is active per application and core at any given point in time. If system calls do not block all the work is executed by a single syscall thread, while the remaining ones sleep on a work-queue. When a syscall thread needs to block, for whatever reason, immediately before it is put to sleep, FlexSC notifies the work-queue. Another thread wakes-up and immediately starts executing the next system call. Later, when resources become free, current Linux code wakes up the waiting thread (in our case, a syscall thread), and resumes its execution, so it can post its result to the syscall page and return to wait in the FlexSC work-queue.

3.6 FlexSC Syscall Thread Scheduler

FlexSC implements a syscall thread scheduler that is responsible for determining when and on which core system calls will execute. This scheduler is critical to performance, as it influences the locality of user and kernel execution.

On a single-core environment, the FlexSC scheduler assumes the user-space will attempt to post as many exception-less system calls as possible, and subsequently call `flexsc_wait()`. The FlexSC scheduler then wakes up an available syscall thread that starts executing the first system call. If the system call does not block, the same syscall thread continues to execute the next submitted syscall entry. If the execution of a syscall thread blocks, the currently scheduled syscall thread notifies the scheduler to wake another thread to continue to execute more system calls. The scheduler does not wake up the user-space thread until all available system calls have been issued, and have either finished or are currently blocked with at least one system call having been completed. This is done to minimize the number of mode switches to user-space.

For multicore execution, the scheduler biases execution of syscall threads on a subset of available cores, dynamically specializing cores according to the workload requirements. In our current implementation, this is done by attempting to schedule syscall threads using a predetermined, static list of cores. Upon a scheduling decision, the first core on the list is selected. If a syscall thread of a process is currently running on that core, the next core on the list is selected as the target. If the selected core is not currently executing a syscall thread, an inter-processor interrupt is sent to the remote core, signalling that it must wake a syscall thread.

As previously described, there is never more than one syscall thread concurrently executing per core, for a given process. However in the multicore case, for the same process, there can be as many syscall threads as cores concurrently executing on the entire system. To avoid cache-line contention of syscall pages amongst cores, before a syscall thread begins executing calls from a syscall page, it *locks* the page until all its submitted calls have been issued. Since FlexSC processes typically map multiple syscall pages, each core on the system can schedule a syscall thread to work independently, executing calls from different syscall pages.

4 System Calls Galore – FlexSC-Threads

Exception-less system calls present a significant change to the semantics of the system call interface with potentially drastic implications for application code and programmers. Programming using exception-less system calls directly is more complex than using synchronous system calls, as they do not provide the same, easy-to-reason-about sequentiality. In fact, our experience is that programming using exception-less system calls is akin to event-driven programming, which has itself been criticized for being a complex programming model [21]. The main difference is that with exception-less system calls, not only are I/O related calls scheduled for future completion, *any* system calls can be requested, verified for completion, and handled, as if it were an asynchronous event.

To address the programming complexities, we propose the use of exception-less system calls in two different modes that might be used depending on the concurrency model adopted by the programmer. We argue that if used according to our recommendations, exception-less system calls should pose *no more* complexity than their synchronous counter-parts.

4.1 Event-driven Servers, a Case for Hybrid Execution

For event-driven systems, we advocate a *hybrid* approach where both synchronous and exception-less system calls coexist. System calls that are executed in performance critical paths of applications should use exception-less calls while all other calls should be synchronous. After all, there is no good justification to make a simple *getpid()* complex to program.

Event-driven servers already have their code structured so that performance critical paths of execution are split into three parts: request event, wait for completion and handle event. Adapting an event-driven server to use exception-less system calls, for the already considered events, should be straightforward. However, we have not yet attempted to evaluate the use of exception-less system calls in an event-driven program, and leave this as future work.

4.2 FlexSC-Threads

Multiprocessing has become the default for computation on servers. With the emergence and ubiquity of multicore processors, along with projection of future chip manufacturing technologies, it is unlikely that this trend will reverse in the medium future. For this reason, and because of its relative simplicity vis-a-vis event-based programming, we believe that the multithreading concurrency model will continue to be the norm.

In this section, we describe the design and implementation of **FlexSC-Threads**, a threading package that transforms legacy synchronous system calls into exception-less ones *transparently* to applications. It is intended for server-type applications with many user-mode threads, such as Apache or MySQL. FlexSC-Threads is compliant with POSIX Threads, and binary compatible with NPTL [8], the default Linux thread library. As a result, Linux multi-threaded programs work with FlexSC-Threads "out of the box" without modification or recompilation.

FlexSC-Threads uses a simple M-on-N threading model (M user-mode threads executing on N kernel-

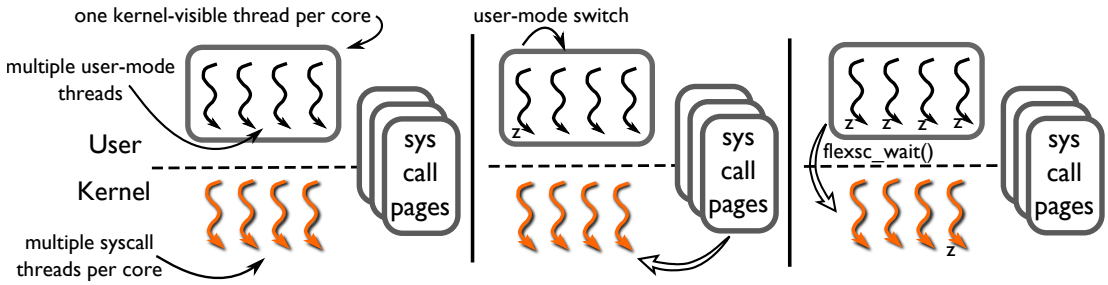


Figure 7: The left-most diagram depicts the components of FlexSC-Threads pertaining to a single core. Each core executes a pinned kernel-visible thread, which in turn can multiplex multiple user-mode threads. Multiple syscall pages, and consequently syscall threads, are also allocated (and pinned) per core. The middle diagram depicts a user-mode thread being preempted as a result of issuing a system call. The right-most diagram depicts the scenario where all user-mode threads are waiting for system call requests; in this case FlexSC-Threads library synchronously invokes `flexsc_wait()` to the kernel.

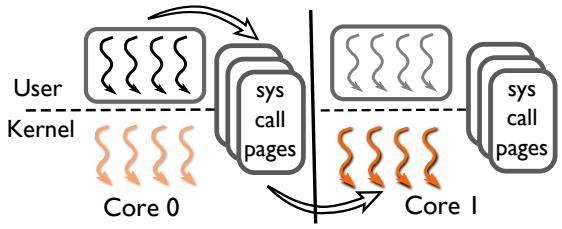


Figure 8: Multicore example. Opaque threads are active, while grayed-out threads are inactive. Syscall pages are accessible to both cores, as we run using shared-memory, leveraging the fast on-chip communication of multicores.

visible threads). We rely on the ability to perform user-mode thread switching solely in user-space to transparently transform legacy synchronous calls into exception-less ones. This is done as follows:

1. We redirect to our library each *libc* call that issues a legacy system call. Typically, applications do not directly embed code to issue system calls, but instead call wrappers in the dynamically loaded *libc*. We use the dynamic loading capabilities of Linux to redirect execution of such calls to our library.
2. FlexSC-Threads then post the corresponding exception-less system call to a syscall page and switch to another user-mode thread that is ready.
3. If we run out of ready user-mode threads, FlexSC checks the syscall page for any syscall entries that have been completed, waking up the appropriate user-mode thread so it can obtain the result of the completed system call.
4. As a last resort, `flexsc_wait()` is called, putting the kernel visible thread to sleep until one of the pending system calls has completed.

FlexSC-Threads implements multicore support by creating a *single* kernel visible thread *per* core available to the process, and pinning each kernel visible thread to a specific core. Multiple user-mode threads multiplex execution on the kernel visible thread. Since kernel-visitble threads only block when there is no more available work, there is no need to create more than one kernel visible thread per core. Figure 7 depicts the components of FlexSC-Threads and how they interact during execution.

As an optimization, we have designed FlexSC-Threads to register a private set of syscall pages *per* kernel visible thread (i.e., per core). Since syscall pages are private to each core, there is no need to synchronize their access with costly atomic instructions. The FlexSC-Threads user-mode scheduler implements a simple form of cooperative scheduling, with system calls acting as yield points. Consequently, syscall pages behave as lock-free single-producer (kernel-visible thread) and single-consumer (syscall thread) data structures.

From the kernel side, although syscall threads are pinned to specific cores, they do not only execute system call requests from syscall pages registered to that core. An example of this is shown in Figure 8, where user-mode threads execute on core 0, while syscall threads running on core 1 are satisfying system call requests.

It is important to note that FlexSC-Threads relies on a large number of independent user-mode threads to post concurrent exception-less system calls. Since threads are executing independently, there is no constraint on ordering or serialization of system call execution (thread-safety constraints should be enforced at the application level and is orthogonal to the system call execution model). FlexSC-Threads leverages the independent requests to efficiently schedule operating system work on single or multicore systems. For this reason, highly threaded workloads, such as internet/network servers, are ideal candidates for FlexSC-Threads.

5 Experimental Evaluation

We first present the results of a microbenchmark that shows the overhead of the basic exception-less system

Component	Specification
Cores	4
Cache line	64 B for all caches
Private L1 i-cache	32 KB, 3 cycle latency
Private L1 d-cache	32 KB, 4 cycle latency
Private L2 cache	512 KB, 11 cycle latency
Shared L3 cache	8 MB, 35-40 cycle latency
Memory	250 cycle latency (avg.)
TLB (L1)	64 (data) + 64 (instr.) entries
TLB (L2)	512 entries

Table 3: Characteristics of the 2.3GHz Core i7 processor.

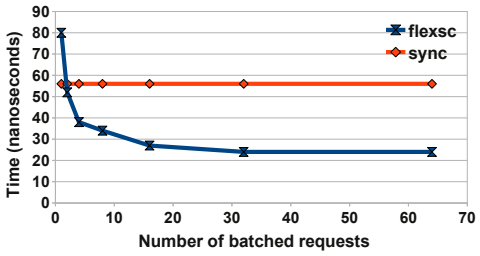


Figure 9: Exception-less system call cost on a single-core.

call mechanism, and then we show the performance of two popular server applications, Apache and MySQL, transparently using exception-less system calls through FlexSC-Threads. Finally, we analyze the sensitivity of the performance of FlexSC to the number of system call pages.

FlexSC was implemented in the Linux kernel, version 2.6.33. The baseline line measurements we present were collected using unmodified Linux (same version), and the default native POSIX threading library (NPTL). We identify the baseline configuration as "**sync**", and the system with exception-less system calls as "**flexsc**".

The experiments presented in this section were run on an Intel Nehalem (Core i7) processor with the characteristics shown in Table 3. The processor has 4 cores, each with 2 hyper-threads. We disabled the hyper-threads, as well as the "TurboBoost" feature, for all our experiments to more easily analyze the measurements obtained.

For the Apache and MySQL experiments, requests were generated by a remote client connected to our test machine through a 1 Gbps network, using a dedicated router. The client machine contained a dual core Core2 processor, running the same Linux installation as the test machine, and was not CPU or network constrained in any of the experiments.

All values reported in our evaluation represent the average of 5 separate runs.

5.1 Overhead

The overhead of executing an exception-less system call involves switching to a syscall thread, de-marshalling arguments from the appropriate syscall page entry, switching back to the user-thread, and retrieving the return value from the syscall page entry. To measure this overhead, we created a micro-benchmark that successively invokes a `getppid()` system call. Since the user and kernel footprints of this call is small, the time measured corresponds to the *direct* cost of issuing system calls.

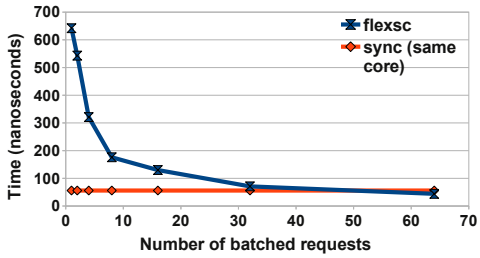


Figure 10: Exception-less system call cost, in the worst case, for remote core execution.

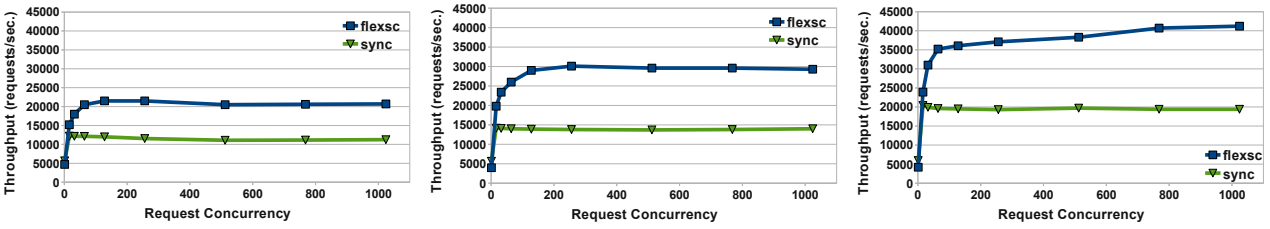
We varied the number of batched system calls, in the exception-less case, to verify if the direct costs are amortized when batching an increasing number of calls. The results obtained executing on a single core are shown in Figure 9. The baseline time, show as a horizontal line, is the time to execute an exception-based system call on a single core. Executing a single exception-less system call on a single core is 43% slower than a synchronous call. However, when batching 2 or more calls there is no overhead, and when batching 32 or more calls, the execution of each call is up to 130% faster than a synchronous call.

We also measured the time to execute system calls on a remote core (Figure 10). In addition to the single core operations, remote core execution entails sending an inter-processor interrupt (IPI) to wake up the remote syscall thread. In the remote core case, the time to issue a single exception-less system call can be more than 10 times slower than a synchronous system call on the same core. This measurement represents a worst case scenario when there is no currently executing syscall thread. Despite the high overhead, the overhead on remote core execution is recouped when batching 32 or more system calls.

5.2 Apache

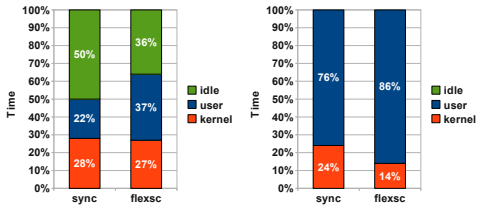
We used Apache version 2.2.15 to evaluate the performance of FlexSC-Threads. Since FlexSC-Threads is binary compatible with NPTL, we used the same Apache binary for both FlexSC and Linux/NPTL experiments. We configured Apache to use a different maximum number of spawned threads for each case. The performance of Apache running on NPTL degrades with too many threads, and we experimentally determined that 200 was optimal for our workload and hence used that configuration for the NPTL case. For the FlexSC-Threads case, we raised the maximum number of threads to 1000.

The workload we used was ApacheBench, a HTTP



(a) 1 Core (b) 2 Cores (c) 4 Cores

Figure 11: Comparison of Apache throughput of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores.



(a) Apache (b) MySQL

Figure 12: Breakdown of execution time of Apache and MySQL workloads on 4 cores.

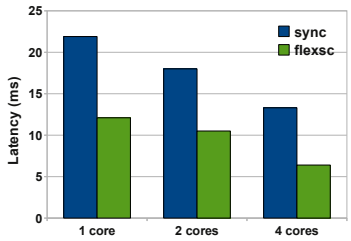


Figure 13: Comparison of Apache latency of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores, with 256 concurrent requests.

workload generator that is distributed with Apache. It is designed to stress-test the Web server determining the number of requests per second that can be serviced, with varying number of concurrent requests.

Figure 11 shows the results of Apache running on 1, 2 and 4 cores. For the single core experiments, FlexSC employs system call batching, and for the multicore experiments it additionally dynamically redirects system calls to maximize core locality. The results show that, except for a very low number of concurrent requests, FlexSC outperforms Linux/NPTL by a wide margin. With system call batching alone (1 core case), we observe a throughput improvement of up to 86%. The 2 and 4 core experiments show that FlexSC achieves up to 116% throughput improvement, showing the added benefit of dynamic core specialization.

Table 4 shows the effects of FlexSC on the microarchitectural state of the processor while running Apache. It displays various processor metrics, collected using hardware performance counters during execution with 512 concurrent requests. The most important metric listed is the instruction per cycles (IPC) of the user and kernel mode for the different setups, as it summarizes the efficiency of execution. The other values listed are normalized values using *misses per kilo-instructions* (MPKI). MPKI is a widely used normalization method that makes it easy to compare values obtained from different executions.

The most efficient execution of the four listed in the table is FlexSC on 1 core, yielding an IPC of 0.94 on both kernel and user execution, which is 95–108% higher than for NPTL. While the FlexSC execution of Apache on 4 cores is not as efficient as the single core case, with an average IPC of 0.75, there is still an 71% improvement, on average, over NPTL.

Most metrics we collected are significantly improved with FlexSC. Of particular importance are the performance critical structures that have a high MPKI value on NPTL such as d-cache, i-cache, and L2 cache. The better use of these microarchitectural structures effectively demonstrates the premise of this work, namely that exception-less system calls can improve processor efficiency. The only structure which observes more misses on FlexSC is the user-mode TLB. We are currently investigating the reason for this.

There is an interesting disparity between the throughput improvement (94%) and the IPC improvement (71%) in the 4 core case. The difference comes from the added benefit of localizing kernel execution with core specialization. Figure 12a shows the time breakdown of Apache executing on 4 cores. FlexSC execution yields significantly less idle time than the NPTL execution.[5] The reduced idle time is a consequence of lowering the contention on a specific kernel semaphore. Linux protects address spaces with a per address-space read-write semaphore (`mmap_sem`). Profiling shows that every Apache thread allocates and frees memory for serving requests, and both of these operations require the semaphore to be held with write permission. Further, the network code in Linux invokes `copy_user()`, which transfers data in and out of the user address-space. This function verifies that the user-space memory is indeed valid, and to do so acquires

[5]The execution of Apache on 1 or 2 core did not present idle time.

Apache	User							Kernel						
Setup	**IPC**	L3	L2	d-cache	i-cache	TLB	Branch	**IPC**	L3	L2	d-cache	i-cache	TLB	Branch
sync (1 core)	**0.48**	3.7	68.9	63.8	130.8	7.7	20.9	**0.45**	1.4	80.0	78.2	159.6	4.6	15.7
flexsc (1 core)	**0.94**	1.7	27.5	35.3	41.3	8.8	12.6	**0.94**	1.0	15.8	31.6	45.2	3.3	11.2
sync (4 cores)	**0.45**	3.9	64.6	67.9	127.6	9.6	20.2	**0.43**	4.4	49.5	73.8	124.9	4.4	15.2
flexsc (4 cores)	**0.74**	1.0	37.5	55.5	49.4	19.3	13.0	**0.76**	1.5	19.1	50.2	63.7	4.2	11.6

Table 4: Micro-architectural breakdown of Apache execution on uni- and quad-core setups. All values shown, except for IPC, are normalized using misses per kilo-instruction (MPKI): therefore, lower numbers yield more efficient execution and higher IPC.

the semaphore with read permissions. In the NPTL case, threads from all 4 cores compete on this semaphore, resulting in 50% idle time. With FlexSC, kernel code is dynamically scheduled to run predominantly on 2 out of the 4 cores, halving the contention to this resource, eliminating 38% of the original idle time.

Another important metric for servicing Web requests besides throughput is latency of individual requests. One might intuitively expect that latency of requests to be higher under FlexSC because of batching and asynchronous servicing of system calls, but the opposite is the case. Figure 13 shows the average latency of requests when processing 256 concurrent requests (other concurrency levels showed similar trends). The results show that Web requests on FlexSC are serviced within 50-60% of the time needed on NPTL, on average.

5.3 MySQL

In the previous section, we demonstrated the effectiveness of FlexSC running on a workload with a significant proportion of kernel time. In this section, we experiment with OLTP on MySQL, a workload for which the proportion of kernel execution is smaller (roughly 25%). Our evaluation used MySQL version 5.5.4 with an InnoDB backend engine, and as in the Apache evaluation, we used the same binary for running on NPTL and on FlexSC. We also used the same configuration parameters for both the NPTL and FlexSC experiments, after tuning them for the best NPTL performance.

To generate requests to MySQL, we used the *sysbench* system benchmark utility. Sysbench was created for benchmarking MySQL processor performance and contains an OLTP inspired workload generator. The benchmark allows executing concurrent requests by spawning multiple client threads, connecting to the server, and sequentially issuing SQL queries. To handle the concurrent clients, MySQL spawns a user-level thread per connection. At the end, sysbench reports the number of transactions per second executed by the database, as well as average latency information. For these experiments, we used a database with 5M rows, resulting in 1.2 GB of data. Since we were interested in stressing the CPU component of MySQL, we disabled synchronous transactions to disk. Given that the configured database was small enough to fit in memory, the workload presented no idle time due to disk I/O.

Figure 14 shows the throughput numbers obtained on 1, 2 and 4 cores when varying the number of concurrent client threads issuing requests to the MySQL server.[6] For this workload, system batching on one core provides modest improvements: up to 14% with 256 concurrent requests. On 2 and 4 cores, however, we see that FlexSC provides a consistent improvement with 16 or more concurrent clients, achieving up to 37%-40% higher throughput.

Table 5 contains the microarchitectural processor metrics collected for the execution of MySQL. Because MySQL invokes the kernel less frequently than Apache, kernel execution yields high miss rates, resulting in a low IPC of 0.33 on NPTL. In the single core case, FlexSC does not greatly alter the execution of user-space, but increases kernel IPC by 36%. FlexSC allows the kernel to reuse state in the processor structures, yielding lower misses across most metrics. In the case of 4 cores, FlexSC also improves the performance of user-space IPC by as much as 30%, compared to NPTL. Despite making less of an impact in the kernel IPC than in single core execution, there is still a 25% kernel IPC improvement over NPTL.

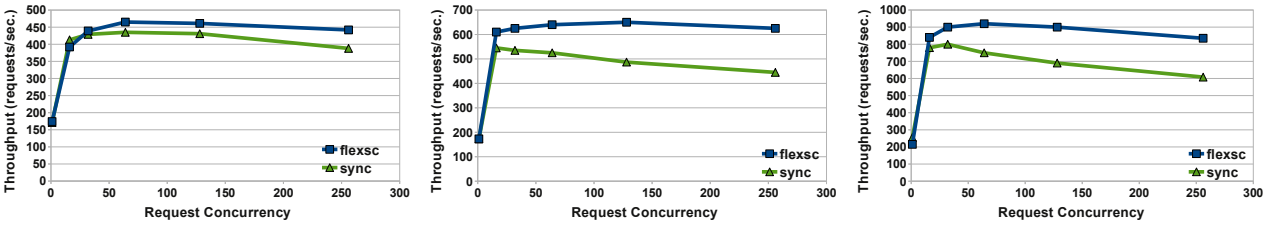
Figure 15 shows the average latencies of individual requests for MySQL execution with 256 concurrent clients. As is the case with Apache, the latency of requests on FlexSC is improved over execution on NPTL. Requests on FlexSC are satisfied within 70-88% of the time used by requests on NPTL.

5.4 Sensitivity Analysis

In all experiments presented so far, FlexSC was configured to have 8 system call pages per core, allowing up to 512 concurrent exception-less system calls per core.

Figure 16 shows the sensitivity of FlexSC to the number of available syscall entries. It depicts the throughput of Apache, on 1 and 4 cores, while servicing 2048 concurrent requests per core, so that there would always be more requests available than syscall entries. Uni-core performance approaches its best with 200 to 250 syscall entries

[6] For both NPTL and FlexSC, increasing the load on MySQL yields peak throughput between 32 and 128 concurrent clients after which throughput degrades. The main reason for performance degradation is the costly and coarse synchronization used in MySQL. MySQL and Linux kernel developers have observed similar performance degradation.



(a) 1 Core (b) 2 Cores (c) 4 Cores

Figure 14: Comparison of MySQL throughput of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores.

MySQL	User							Kernel						
Setup	**IPC**	L3	L2	d-cache	i-cache	TLB	Branch	**IPC**	L3	L2	d-cache	i-cache	TLB	Branch
sync (1 core)	**1.12**	0.6	21.1	34.8	24.2	3.8	7.8	**0.33**	16.5	125.2	209.6	184.9	3.9	17.4
flexsc (1 core)	**1.10**	0.8	19.6	36.3	23.6	5.4	6.9	**0.45**	23.2	55.1	131.9	86.5	3.7	13.6
sync (4 cores)	**0.55**	3.7	15.8	25.2	18.9	3.1	5.9	**0.36**	16.6	78.0	147.0	120.0	3.6	15.7
flexsc (4 cores)	**0.72**	2.7	16.7	30.6	20.9	4.1	6.5	**0.45**	18.4	46.6	104.4	63.5	2.5	11.5

Table 5: Micro-architectural breakdown of MySQL execution on uni- and quad-core setups. All values shown, except for IPC, are normalized using misses per kilo-instruction (MPKI): therefore, lower numbers yield more efficient execution and higher IPC.

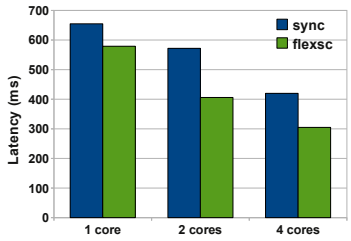


Figure 15: Comparison of MySQL latency of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores, with 256 concurrent requests.

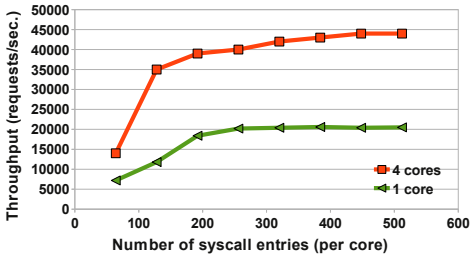


Figure 16: Execution of Apache on FlexSC-Threads, showing the performance sensitivity of FlexSC to different number of syscall pages. Each syscall page contains 64 syscall entries.

(3 to 4 syscall pages), while quad-core execution starts to plateau with 300 to 400 syscall entries (6 to 7 syscall pages).

It is particularly interesting to compare Figure 16 with figures 9 and 10. The direct cost of mode switching, exemplified by the micro-benchmark, has a lesser effect on performance when compared to the indirect cost of mixing user- and kernel-mode execution.

6 Related Work

6.1 System Call Batching

The idea of batching calls in order to save crossings has been extensively explored in the systems community. Specific to operating systems, *multi-calls* are used in both operating systems and paravirtualized hypervisors as a mechanism to address the high overhead of mode switching. Cassyopia is a compiler targeted at rewriting programs to collect many independent system calls, and submitting them as a single multi-call [18]. An interesting technique in Cassyopia, which could be eventually explored in conjunction with FlexSC, is the concept of a *looped multi-call* where the result of one system call can be automatically fed as an argument to another system call in the same multi-call. In the context of hypervisors, both Xen and VMware currently support a special multi-call hypercall feature [4][20].

An important difference between multi-calls and exception-less system calls is the level of flexibility exposed. The multi-call proposals do not investigate the possibility of parallel execution of system calls, or address the issue of blocking system calls. In multi-calls, system calls are executed sequentially; each system call must complete before a subsequent can be issued. With exception-less system calls, system calls can be executed in parallel, and in the presence of blocking, the next call can execute immediately.

6.2 Locality of Execution and Multicores

Several researchers have studied the effects of operating system execution on application performance [1, 3, 7, 6, 11, 13]. Larus and Parkes also identified processor inef-

ficiencies of server workloads, although not focusing on the interaction with the operating system. They proposed Cohort Scheduling to efficiently execute staged computations to improve locality of execution [11].

Techniques such as Soft Timers [3] and Lazy Receiver Processing [9] also address the issue of locality of execution, from the other side of the compute stack: handling device interrupts. Both techniques describe how to limit processor interference associated with interrupt handling, while not impacting the latency of servicing requests.

Most similar to the multicore execution of FlexSC is Computation Spreading proposed by Chakraborty et. al [6]. They introduced processor modifications to allow for hardware migration of threads, and evaluated the effects on migrating threads upon entering the kernel to specialize cores. Their simulation-based results show an improvement of up to 20% on Apache, however, they explicitly do not model TLBs and provide for fast thread migration between cores. On current hardware, synchronous thread migration between cores requires a costly inter-processor interrupt.

Recently, both Corey and Factored Operating System (fos) have proposed dedicating cores for specific operating system functionality [24, 25]. There are two main differences between the core specialization possible with these proposals and FlexSC. First, both Corey and fos require a micro-kernel design of the operating system kernel in order to execute specific kernel functionality on dedicated cores. Second, FlexSC can dynamically adapt the proportion of cores used by the kernel, or cores shared by user and kernel execution, depending on the current workload behavior.

Explicit off-loading of select OS functionality to cores has also been studied for performance [15, 16] and power reduction in the presence of single-ISA heterogeneous multicores [14]. While these proposals rely on expensive inter-processor interrupts to offload system calls, we hope FlexSC can provide for a more efficient, and flexible, mechanism that can be used by such proposals.

6.3 Non-blocking Execution

Past research on improving system call performance has focused extensively on blocking versus non-blocking behavior. Typically researchers have analyzed the use of threading, event-based (non-blocking), and hybrid systems for achieving high performance on server applications [2, 10, 17, 21, 22, 23]. Capriccio described techniques to improve performance of user-level thread libraries for server applications [22]. Specifically, Behren et al. showed how to efficiently manage thread stacks, minimizing wasted space, and propose resource aware scheduling to improver server performance. For an extensive performance comparison of thread-based and event-driven Web server architectures we refer the reader to [17].

Finally, the Linux community has proposed a generic mechanism for implementing non-blocking system calls, which is call asynchronous system calls [5]. In their proposal, system calls are still exception-based, and tentatively execute synchronously. Like scheduler activations, if a blocking condition is detected, they utilize a "syslet" thread to block, allowing the user thread to continue execution.

The main difference between many of the proposals for non-blocking execution and FlexSC is that none of the non-blocking system call proposals completely decouple the invocation of the system call from its execution. As we have discussed, the flexibility resulting from this decoupling is crucial for efficiently exploring optimizations such as system call batching and core specialization.

7 Concluding Remarks

In this paper, we introduced the concept of exception-less system calls that decouples system call invocation from execution. This allows for flexible scheduling of system call execution which in turn enables system call batching and dynamic core specialization that both improve locality in a significant way. System calls are issued by writing kernel requests to a reserved syscall page using normal store operations, and they are executed by special in-kernel syscall threads, which then post the results to the syscall page.

In fact, the concept of exception-less system calls originated as a mechanism for low-latency communication between user and kernel-space with hyper-threaded processors in mind. We had hoped that communicating directly through the shared L1 cache would be much more efficient than mode switching. However, the measurements presented in Section 2 made it clear that mixing user and kernel-mode execution on the same core would not be efficient for server class workloads. In future work we intend to study how to exploit exception-less system calls as a communication mechanism in hyper-threaded processors.

We presented our implementation of FlexSC, a Linux kernel extension, and FlexSC-Threads, a M-on-N threading package that is binary compatible with NPTL and that transparently transforms synchronous system calls into exception-less ones. With this implementation, we demonstrated how FlexSC improves throughput of Apache by up to 116% and MySQL by up to 40% while requiring no modifications to the applications. We believe these two workloads are representative of other highly threaded server workloads that would benefit from FlexSC. For example, experiments with the BIND DNS server demonstrated throughput improvements of between 30% and 105% depending on the concurrency of requests.

In the current implementation of FlexSC, syscall threads process system call requests in no specific order, opportunistically issuing calls as they are posted on syscall pages. The asynchronous execution model, however, would allow for different selection algorithms. For example, syscall threads could sort the requests to consecutively execute requests of the same type, potentially yielding greater locality of execution. Also, system calls that perform I/O could be prioritized so as to issue them as early as possible. Finally, if a large number of cores are available, cores could be dedicated to specific system call types to promote further locality gains.

8 Acknowledgements

This work was supported in part by Discovery Grant funding from the Natural Sciences and Engineering Research Council (NSERC) of Canada. We would like to thank the feedback from the OSDI reviewers, and to Emmett Witchel for shepherding our paper. Special thanks to Ioana Burcea for encouraging the work in its early stages, and the Computer Systems Lab members (University of Toronto), as well as Benjamin Gamsa, for insightful comments on the work and drafts of this paper.

References

[1] AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst. 6*, 4 (1988), 393–431.

[2] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Trans. Comput. Syst. 10*, 1 (1992), 53–79.

[3] ARON, M., AND DRUSCHEL, P. Soft timers: efficient microsecond software timer support for network processing. *ACM Trans. Comput. Syst. (TOCS) 18*, 3 (2000), 197–228.

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[5] BROWN, Z. Asynchronous system calls. In *Proceedings of the Ottawa Linux Symposium (OLS)* (2007), pp. 81–85.

[6] CHAKRABORTY, K., WELLS, P. M., AND SOHI, G. S. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006), pp. 283–292.

[7] CHEN, J. B., AND BERSHAD, B. N. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)* (1993), pp. 120–133.

[8] DREPPER, U., AND MOLNAR, I. The Native POSIX Thread Library for Linux. Tech. rep., RedHat Inc, 2003. http://people.redhat.com/drepper/nptl-design.pdf.

[9] DRUSCHEL, P., AND BANGA, G. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (1996), pp. 261–275.

[10] ELMELEEGY, K., CHANDA, A., COX, A. L., AND ZWAENEPOEL, W. Lazy asynchronous I/O for event-driven servers. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (2004), pp. 21–21.

[11] LARUS, J., AND PARKES, M. Using Cohort-Scheduling to Enhance Server Performance. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (2002), pp. 103–114.

[12] LI, T., JOHN, L. K., SIVASUBRAMANIAM, A., VIJAYKRISHNAN, N., AND RUBIO, J. Understanding and Improving Operating System Effects in Control Flow Prediction. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2002), pp. 68–80.

[13] MOGUL, J. C., AND BORG, A. The Effect of Context Switches on Cache Performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1991), pp. 75–84.

[14] MOGUL, J. C., MUDIGONDA, J., BINKERT, N., RANGANATHAN, P., AND TALWAR, V. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro 28*, 3 (2008), 26–41.

[15] NELLANS, D., BALASUBRAMONIAN, R., AND BRUNVAND, E. OS execution on multi-cores: is out-sourcing worthwhile? *SIGOPS Oper. Syst. Rev. 43*, 2 (2009), 104–105.

[16] NELLANS, D., SUDAN, K., BRUNVAND, E., AND BALASUBRAMONIAN, R. Improving Server Performance on Multi-Cores via Selective Off-loading of OS Functionality. In *Sixth Annual Workshorp on the Interaction between Operating Systems and Computer Architecture (WIOSCA)* (2010), pp. 13–20.

[17] PARIAG, D., BRECHT, T., HARJI, A., BUHR, P., SHUKLA, A., AND CHERITON, D. R. Comparing the performance of Web server architectures. In *Proceedings of the 2nd European Conference on Computer Systems (Eurosys)* (2007), pp. 231–243.

[18] RAJAGOPALAN, M., DEBRAY, S. K., HILTUNEN, M. A., AND SCHLICHTING, R. D. Cassyopia: compiler assisted system optimization. In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS)* (2003), pp. 18–18.

[19] REDSTONE, J. A., EGGERS, S. J., AND LEVY, H. M. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2000), pp. 245–256.

[20] VMWARE. *VMWare Virtual Machine Interface Specification*. http://www.vmware.com/pdf/vmi_specs.pdf.

[21] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why Events Are A Bad Idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS)* (2003).

[22] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 268–281.

[23] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 230–243.

[24] WENTZLAFF, D., AND AGARWAL, A. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev. 43*, 2 (2009), 76–85.

[25] WICKIZER, S. B., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).