Exception-Less System Calls for Event-Driven Servers

Livio Soares Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto

Abstract

Event-driven architectures are currently a popular design choice for scalable, high-performance server applications. For this reason, operating systems have invested in efficiently supporting non-blocking and asynchronous I/O, as well as scalable event-based notification systems.

We propose the use of *exception-less system calls* as the main operating system mechanism to construct high-performance event-driven server applications. Exception-less system calls have four main advantages over traditional operating system support for event-driven programs: (1) any system call can be invoked asynchronously, even system calls that are not file descriptor based, (2) support in the operating system kernel is non-intrusive as code changes are not required for each system call, (3) processor efficiency is increased since mode switches are mostly avoided when issuing or executing asynchronous operations, and (4) enabling multi-core execution for event-driven programs is easier, given that a single user-mode execution context can generate enough requests to keep multiple processors/cores busy with kernel execution.

We present *libflexsc*, an asynchronous system call and notification library suitable for building event-driven applications. Libflexsc makes use of exception-less system calls through our Linux kernel implementation, FlexSC. We describe the port of two popular event-driven servers, memcached and nginx, to libflexsc. We show that exception-less system calls increase the throughput of memcached by up to 35% and nginx by up to 120% as a result of improved processor efficiency.

1 Introduction

In a previous publication, we introduced the concept of **exception-less system calls** [28]. With exception-less system calls, instead of issuing system calls in the traditional way using a trap (exception) to switch to the kernel for the processing of the system call, applications issue kernel requests by writing to a reserved **syscall page**, shared between the application and the kernel, and then switching to another user-level thread ready to execute without having to enter the kernel. On the kernel side, special in-kernel **syscall threads** asynchronously execute the posted system calls obtained from the shared syscall page, storing the results to the syscall page after their servicing. This approach enables flexibility in the scheduling of operating system work both in the form of kernel request batching, and (in the case of multi-core processors) in the form of allowing operating system and application execution to occur on different cores. This not only significantly reduces the number of costly domain switches, but also significantly increases temporal and spacial locality at both user and kernel level, thus reducing pollution effects on processor structures.

Our implementation of exception-less system calls in the Linux kernel (**FlexSC**) was accompanied by a user-mode POSIX compatible thread package, called **FlexSC-Threads**, that transparently translates legacy synchronous system calls into exception-less ones. FlexSC-Threads primarily targets highly threaded server applications, such as Apache and MySQL. Experiments demonstrated that FlexSC-Threads increased the throughput of Apache by 116% while reducing request latencies by 50%, and increased the throughput of MySQL by 40% while reducing request latencies by roughly 30%, requiring no changes to these applications.

In this paper we report on our subsequent investigations on whether exception-less system calls are suitable for event-driven application servers and, if so, whether exception-less system calls are effective in improving throughput and reducing latencies. Event-driven application server architectures handle concurrent requests by using just a single thread (or one thread per core) so as to reduce application-level context switching and the memory footprint that many threads otherwise require. They make use of non-blocking or asynchronous system calls to support the concurrent handling of requests. The belief that event-driven architectures have superior performance characteristics is why this architecture has been widely adopted for developing high-performant and scalable servers [12, 22, 23, 26, 30]. Widely used application servers with event-driven architectures include memcached and nginx.

The design and implementation of operating system support for asynchronous operations, along with event-based notification interfaces to support event-driven architectures, has been an active area of both research and development [4, 8, 7, 11, 13, 14, 16, 22, 23, 30]. Most of the proposals have a few common characteristics. First, the interfaces exposed to user-mode are based on file descriptors (with the exception of *kqueue* [4, 16] and LAIO [11]). Consequently, resources that are not encapsulated as descriptors (e.g., memory) are not supported. Second, their implementation typically involved significant restructure of kernel code paths into an asynchronous state-machine in order to avoid blocking the user execution context. Third, and most relevant to our work, while the system calls used to request operating system services are designed not to block execution, applications still issue system calls *synchronously*, raising a processor exception, and switching execution domains, for every request, status check, or notification of completion.

In this paper, we demonstrate that the exception-less system call mechanism is well suited for the construction of event-based servers and that the exception-less mechanism presents several advantages over previous event-based systems:

1. **General purpose.** Exception-less system call is a general mechanism that supports any system call and is not necessarily tied to operations with file descriptors. For this reason, exception-less system calls provide asynchronous operation on any operating system managed resource.
2. **Non-intrusive kernel implementation.** Exception-less system calls are implemented using light-weight kernel threads that can block without affecting user-mode execution. For this reason, kernel code paths do not need to be restructured as asynchronous state-machines; in fact, no changes are necessary to the code of standard system calls.
3. **Efficient user and kernel mode execution.** One of the most significant advantages of exception-less system calls is its ability to decouple system call invocation from execution. Invocation of system calls can be done entirely in user-mode, allowing for truly asynchronous execution of user code. As we show in this paper, this enables significant performance improvements over the most efficient non-blocking interface on Linux.
4. **Simpler multi-processing.** With traditional system calls, the only mechanism available for applications to exploit multiple processors (cores) is to use an operating system visible execution context, be it a thread or a process. With exception-less system calls, however, operating system work can be issued and distributed to multiple remote cores. As an example, in our implementation of memcached, a single memcached thread was sufficient to generate work to fully utilize 4 cores.

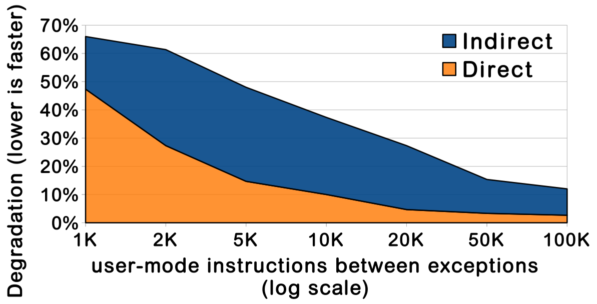


Figure 1: System call (`pwrite`) impact on user-mode instructions per cycle (IPC) as a function of system call frequency for Xalan.

Specifically, we describe the design and implementation of an asynchronous system call notification library, **libflexsc**, which is intended to efficiently support event-driven programs. To demonstrate the performance advantages of exception-less system calls for event-driven servers, we have ported two popular and widely deployed event-based servers to libflexsc: memcached and nginx. We briefly describe the effort in porting these applications to libflexsc. We show how the use of libflexsc can significantly improve the performance of these two servers over their original implementation using non-blocking I/O and Linux's `epoll` interface. Our experiments demonstrate throughput improvements in memcached of up to 35% and nginx of up to 120%. As anticipated, we show that the performance improvements largely stem from increased efficiency in the use of the underlying processor.

2 Background and Motivation: Operating System Support for I/O Concurrency

Server applications that are required to efficiently handle multiple concurrent requests rely on operating system primitives that provide I/O concurrency. These primitives typically influence the programming model used to implement the server. The two most commonly used models for I/O concurrency are threads and non-blocking/asynchronous I/O.

Thread based programming is often considered the simplest, as it does not require tracking the progress of I/O operations (which is done implicitly by the operating system kernel). A disadvantage of threaded servers that utilize a separate thread per request/transaction is the inefficiency

Server (workload)	Syscalls per Request	User Instructions per Syscall	User IPC	Kernel Instructions per Syscall	Kernel IPC
Memcached (memslap)	2	3750	0.80	5420	0.59
nginx (ApacheBench)	12	1460	0.46	6540	0.49

Table 1: The average number of instructions executed on different workloads before issuing a syscall, the average number of system calls required to satisfy a single request, and the resulting processor efficiency, shown as instructions per cycle (IPC) of both user and kernel execution. Memcached and nginx are event-driven servers using Linux's `epoll` interface.

of handling a large number of concurrent requests. The two main sources of inefficiency are the extra memory usage allocated to thread stacks and the overhead of tracking and scheduling a large number of execution contexts.

To avoid the overheads of threading, developers have adopted the use event-driven programming. In an event-driven architecture, the program is structured as a state machine that is driven by progress of certain operations, typically involving I/O. Event-driven programs make use of non-blocking or asynchronous primitives, along with event notification systems, to deal with concurrent I/O operations. These primitives allow for uninterrupted execution that enables a single execution context (e.g., thread) to fully utilize the processor. The main disadvantage of using non-blocking or asynchronous I/O is that it entails a more complex programming model. The application is responsible for tracking the status of I/O operations and availability of I/O resources. In addition, the application must support multiplexing the execution of stages of multiple concurrent requests.

In both models of I/O concurrency, the operating system kernel plays a central role in supporting servers in multiplexing execution of concurrent requests. Consequently, to achieve efficient server execution, it is critical for the operating system to expose and support efficient I/O multiplexing primitives. To quantify the relevance of operating system kernel execution experimentally, we measured key execution metrics of two popular event-driven servers: memcached and nginx.

Table 1 shows the number of instructions executed in user and kernel mode, on average, before changing mode, for nginx and memcached. (Sections 5 and 6 explain the servers and workloads in more detail.) These applications use non-blocking I/O, along with the Linux `epoll` facility for event notification. Despite the fact that the `epoll` facility is considered the most scalable approach to I/O concurrency on Linux, management of both I/O requests and events is inherently split between the operating system kernel and the application. This fundamental property of event notification systems imply that there is a need for continuous communication between the application and the operating system kernel. In the case of nginx, for example, we observe that communication with the kernel occurs, on average, every 1470 instructions.

We argue that the high frequency of mode switching in these servers, which is inherent to current event-based facilities, is largely responsible for the low efficiency of user and kernel execution, as quantified by the instructions per cycle (IPC) metric in Table 1. In particular, there are two costs that affect the efficiency of execution when frequently switching modes: (1) a *direct* cost that stems from the processor exception associated with the system call instruction, and (2) an *indirect* cost resulting from the pollution of important processor structures.

To quantify the performance interference caused by frequent mode switching, we used the Core i7 hardware performance counters (HPC) to measure the efficiency of processor execution while varying the number of mode switches of a benchmark. Figure 1 depicts the performance degradation of user-mode execution, when issuing varying frequencies of `pwrite` system calls, on a high IPC workload, Xalan, from the SPEC CPU 2006 benchmark suite. We used a benchmark from SPEC CPU 2006 as these benchmarks have been created to avoid significant use of system services, and should spend only 1-2% of time executing in kernel-mode. 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 0.50 and 0.58.

The figure also depicts 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 by 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, which is the case for nginx), 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.

Given the inefficiency of event-driver server execution due to frequent mode switches, we envision that these servers could be adapted to make use of exception-less system calls. Beyond the potential to improve server performance, we believe exception-less system calls is an appealing mechanism for event-driven programming, as: (1) it is as simple as asynchronous I/O to program to (no retry logic is necessary, unlike non-blocking I/O), and (2) more generic than asynchronous I/O, which mostly supports descriptor based operations and which are only partially supported on some operating systems due to their implementation complexity (e.g., Linux does not offer an asynchronous version of the zero-copy `sendfile()`).

One of the proposals that is closest to achieving the goals of event-driven programming with exception-less system calls is *lazy asynchronous I/O* (LAIO), proposed by Elmeleegy et al. [11] However, in their proposal, system calls are still issued synchronously, using traditional exception based calls. Furthermore, a completion notification is also needed whenever an operation blocks, which generates another interruption in user execution.

3 Exception-Less System Call Interface and Implementation

In this work, we argue for the use of exception-less system calls as a mechanism to improve processor efficiency while multiplexing execution between user and kernel modes in event-driven servers. Exception-less system call is a mechanism for requesting kernel services that does not require the use of synchronous processor exceptions [28]. 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.

Exception-less system calls have been shown to improve the performance of highly threaded applications, by using a specialized user-level threading package that transparently converts synchronous system calls into exception-less ones [28]. The goal of this work is to extend the original proposal by enabling the explicit use of exception-less system calls by event-driven applications.

In this section, we briefly describe the original exception-less system call implementation (FlexSC) for the benefit of the reader; those readers already familiar with exception-less system calls may skip to Section 4. For space considerations, this is a simple overview of exception-less system calls; for more information, we refer the reader to the original exception-less system calls proposal [28].

3.1 Exception-Less 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.

3.1.1 Exception-Less Syscall Interface

The interface for exception-less system calls is simply a set of memory pages that is shared between user and kernel space. The shared memory pages, henceforth referred to as **syscall pages**, contain exception-less system call entries. Each entry records the request status, system call number, arguments, and return value (Figure 2).

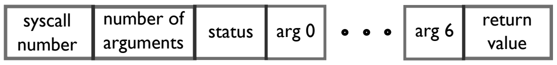
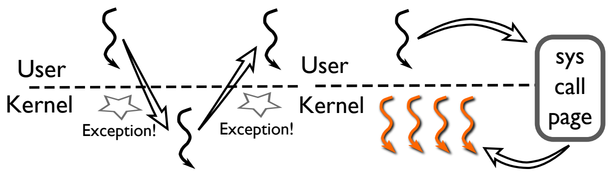


Figure 2: 64-byte syscall entry from the syscall page.



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

Figure 3: 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.

The traditional invocation of a system call occurs by populating predefined registers with call information and issuing a specific machine instruction that immediately raises an exception. In contrast, to issue an exception-less system call, user-mode must find a free entry in the syscall page and populate the entry with the appropriate values using regular store instructions. The user thread can then continue executing without interruption. It is the responsibility of the user 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.

3.1.2 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 thread. Figure 3 illustrates the difference between traditional synchronous system calls, and our proposed split system call model.

3.2 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. 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. 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-mode execution, namely what to do when the user-space thread has nothing more to execute and is waiting on pending system calls. 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 (this is akin to `aio_suspend()` or `epoll_wait()` calls). FlexSC will later wake up the user-space thread when at least one of the posted system calls are complete.

3.2.1 Syscall Threads

Syscall thread 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 the 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 a system call does not block, then 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, and another thread wakes up to immediately start 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.

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 *Libflexsc*: Asynchronous system call and notification library

To allow event-driven applications to interface with exception-less system calls, we have designed and implemented a simple asynchronous system call notification library, **libflexsc**. Libflexsc provides an event loop for the program, which must register system call requests, along with callback functions. The main event loop on libflexsc invokes the corresponding program provided callback when the system call has completed.

The event loop and callback handling in libflexsc was inspired by the *libevent* asynchronous event notification library [24]. The main difference between these two libraries is that *libevent* is designed to monitor low-level events, such as changes in the availability of input or output, and operates at the file descriptor level. The application is notified of the availability, but its intended operation is still not guaranteed to succeed. For example, a socket may contain available data to be read, but if the application requires more data than is available, it must restate interest in the event to try again. With libflexsc, on the other hand, events correspond to the completion of

```c
conn master;

int main(void)
{
   /* init library and register with kernel */
   flexsc_init();

   /* not performance critical,
                  do synchronously */
   master.fd = bind_and_listen(PORT_NUMBER);

   /* prepare accept */
   master.event->handler = conn_accepted;
   flexsc_accept(&master.event, master.fd,
                 NULL, 0);

   /* jump to event loop */
   return flexsc_main_loop();
}

/* Called when accept() returns */
void conn_accepted(conn *c)
{
   conn *new_conn = alloc_new_conn();

   /* get the return value of the accept() */
   new_conn->fd = c->event->ret;
   new_conn->event->handler = data_read;

   /* issue another accept on the master socket */
   flexsc_accept(&c->event, c->fd, NULL, 0);

   if (new_conn->fd != -1)
      flexsc_read(&new_conn->event, new_conn->fd,
                  new_conn->buf, new_conn->size);
}

void data_read(conn *c)
{
   char *reply_file;

   /* read of 0 means connection closed */
   if (c->event->ret == 0) {
       flexsc_close(NULL, c->fd);
       return;
   }

   reply_file = parse_request(c->buf, c->event->ret);

   if (reply_file) {
      c->event->handler = file_opened;
      flexsc_open(&c->event, c->fd, reply_file,
                  O_RDONLY);
   }
}

void file_opened(conn *c)
{
    int file_fd;

    file_fd = c->event->ret;
    c->event->handler = file_sent;
    /* issue asynchronous sendfile */
    flexsc_sendfile(&c->event, c->fd, file_fd,
                    NULL, file_len);
}

void file_sent(conn *c)
{
    /* no callback necessary to handle close */
    flexsc_close(NULL, c->fd);
}
```

Figure 4: Example of network server using libflexsc.

a previously issued exception-less system call. With this model, which is closer to that of asynchronous I/O, it is less likely that applications need to include cumbersome logic to retry incomplete or failed operations.

Contrary to common implementations of asynchronous I/O, FlexSC does not provide a signal or interrupt based completion notification. Completion notification is a mechanism for the kernel to notify a user thread that a previously issued asynchronous request has completed. It is often implemented through a signal or other upcall mechanism. The main reason FlexSC does not offer completion notification is that signals and upcalls entail the same processor performance problems of system calls: direct and indirect processor pollution due to switching between kernel and user execution.

To overcome the lack of completion notifications, the libflexsc event main loop must poll the *syscall pages* currently in use for completion of system calls. To minimize overhead, the polling for system call completion is performed only when all currently pending callback handlers have completed. Given enough work (e.g., handling many connections concurrently), polling should happen infrequently. In the case that all callback handlers have executed, and no new system call has completed, libflexsc falls back on calling `flexsc_wait()` (described in Section 3.2).

4.1 Example server

A simplified implementation of a network server using libflexsc is shown in Figure 4. The program logic is divided into states which are driven by the completion of a previously issued system call. The system calls used in this example that are prefixed with "`flexsc_`" are issued using the exception-less interface (`accept`, `read`, `open`, `sendfile`, `close`). When the library detects the completion of a system call, its corresponding callback handler is invoked, effectively driving the next stage of the state machine. During normal operation, the execution flow of this example would progress in the following order: (1) `main`, (2) `conn_accepted`, (3) `data_read`, (4) `file_opened`, and (5) `file_sent`. As mentioned, file and network descriptors do not need to be marked as non-blocking.

It is worth noting that stages may generate several system call requests. For example, the `conn_accepted()` function not only issues a read on the newly accepted connection, it also issues another accept system call on the master listening socket in order to pipeline further incoming requests. In addition, for improved efficiency, the server may choose to issue *multiple* accepts concurrently (not shown in this example). This would allow the operating system to accept multiple connections without having to first execute user code, as is the case with traditional event-based systems, thus reducing the number of mode switches for each new connection.

Finally, not all system calls must provide a callback, as a notification may not be of interest to the programmer. For example, in the `file_sent` function listed in the simplified server code, the request to close the file does not provide a callback handler. This may be useful if the completion of a system call does not drive an additional state in the program and the return code of the system call is not of interest.

4.2 Cancellation support

A new feature we had to add to FlexSC in order to support event-based applications is the ability to cancel submitted system calls. Cancellation of in-progress system calls may be necessary in certain cases. For example, servers typically implement a *timeout* feature for reading requests on connections. With non-blocking system calls, reads are implemented by waiting for a notification that the socket has become readable. If the event does not occur within the timeout grace period, the connection is closed. With exception-less system calls, the read request is issued before the server knows if or when new data will arrive (e.g., the `conn_accepted` function in Figure 4). To properly implement a timeout, the application must cancel pending reads if the grace period has passed.

To implement cancellation in FlexSC, we introduced a new *cancel* status value to be used in the status field of the syscall entry (Figure 2). When syscall threads, in the kernel, check for new submitted work, they now also check for entries in *cancel* state. To cancel the in-progress operation, we first identify the syscall thread that is executing the request that corresponds to the cancelled entry. This is easily accomplished since each core has a map of syscall entries to syscall threads for all in-progress system calls. Once identified, a signal is sent to the appropriate syscall thread to interrupt its execution. In the Linux kernel, signal delivery that occurs during system call execution interrupts the system call even if the execution context is asleep (e.g., waiting for I/O). When the syscall thread wakes up, it sets the return value to `EINTR` and marks the entry as *done* in the corresponding syscall entry, after which, the user-mode process knows that the system call has been cancelled and the syscall entry can be reused.

Due to its asynchronous implementation, cancellation requests are not guaranteed to succeed. The window of time between when the application modifies the status field and when the syscall thread is notified of cancellation may be sufficiently long for the system call to complete (successfully). The application must check the system call return code to disambiguate between successfully completed calls and cancelled ones. This behavior is analogous to cancellation support of asynchronous I/O implemented by several UNIX systems (e.g., `aio_cancel`).

Server	**Total lines of code**	**Lines of code modifiied**	**Files modified**
memcached	8356	293	3
nginx	82819	255	16

Table 2: Statistics regarding the code size and modifications needed to port applications to libflexsc, measured in lines of code and number of files.

5 Exception-Less Memcached and nginx

This section describes the process of porting two popular event-based servers to use exception-less system calls. In both cases, the applications were modified to conform to the libflexsc interface. However, we strived to maintain the structure of code as similar to the original as possible, to make performance comparisons meaningful.

To reduce the complexity of porting these applications to exception-less system calls, we exploited the fact that FlexSC allows exception-less system calls to co-exist with synchronous ones in the same process. Consequently, we have not modified *all* system calls to use exception-less versions. We focused on the system calls that were issued in the code paths that are involved in handling requests (which correspond to the *hot paths* during normal operation).

5.1 Memcached - Memory Object Cache

Memcached is a distributed memory object caching system, built as an in-memory key-value store [12]. It is typically used to cache results from slower services such as databases and web servers. It is currently used by several popular web sites as a way to improve the performance and scalability of their web services. We used version 1.4.5 as a basis for our port.

To achieve good I/O performance, memcached was built as an event-based server. It uses *libevent* to make use of non-blocking execution available on modern operating system kernels. For this reason, porting memcached to use exception-less system calls through libflexsc was the simplest of the two ports. Table 2 lists the number of lines of code and the number of files that were modified. For memcached, the majority of the changes were done in a single file (`memcached.c`), and the changes were mostly centered around modifying system calls, as well as calls to *libevent*.

Multicore/multiprocessor support has been introduced to memcached, despite most of the code assuming single-threaded execution. To support multiple processors, memcached spawns worker threads which communicate via a pipe to a master thread. The master thread is responsible for accepting incoming connections and handing them to the worker threads.

5.2 nginx Web Server

Nginx is an open-source HTTP web server considered to be light-weight and high-performant; it is currently one of the most widely deployed open-source web servers [26]. Nginx implements I/O concurrency by natively using non-blocking and asynchronous operations available in the operating system kernel. On Linux, nginx uses the `epoll` notification system. We based our port on the 0.9.2 development version of nginx.

Despite having had to change a similar number of lines as with memcached, the port to nginx was more involved, evidenced by the number of files changed (Table 2). This was mainly due to the fact that nginx's core code is significantly larger than that of memcached's (about 10x), and its state machine logic is more complex.

We substituted all system calls that could potentially be invoked while handling client requests to use the corresponding version in libflexsc. The system calls that were associated with a file descriptor based event handler (such as `accept`, `read` and `write`) were straightforward to implement, as these were already programmed as separate stages in the code. However, the system calls that were previously invoked synchronously (e.g., `open`, `fstat`, and `getdents`) needed more work. In most cases, we needed to split a single stage of the state machine into two or more stages to allow asynchronous execution of these system calls. In a few cases, such as `setsockopt` and `close`, we executed the calls asynchronously, without a callback notification, which did not required a new stage in the flow of the program.

Finally, for system calls that not only return a status value, but also fill in a user supplied memory pointer with a data structure, we had to ensure that this memory was correctly managed and passed to the newly created event handler. This requirement prevented the use of stack allocated data structures for exception-less system calls (e.g., programs typically use stack allocated "`struct stat`" data structure to pass to the `fstat` system call).

6 Experimental Evaluation

In this section, we evaluate the performance of exception-less system call support for event-driven servers. We present experimental results of the two previously discussed event-driven servers: memcached and nginx.

FlexSC was implemented in the Linux kernel, version 2.6.33. The baseline measurements were collected using unmodified Linux (same version), with the servers configured to use the `epoll` interface. In the graphs shown, we identify the baseline configuration as "**epoll**", 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.

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.

For the experiments involving both servers, 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.

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

6.1 Memcached

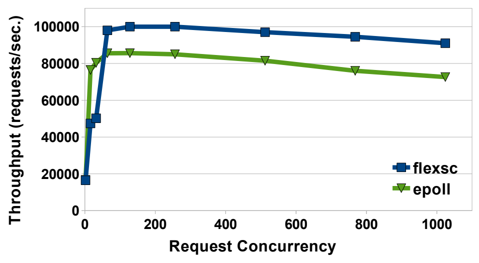
The workload we used to drive memcached is the *memslap* benchmark that is distributed with the libmemcached client library. The benchmark performs a sequence of memcache `get` and `set` operations, using randomly generated keys and data. We configured memslap to issue 10% of set requests and 90% of get requests.

For the baseline experiments (Linux `epoll`), we configured memcached to run with the same number of threads as processor cores, as we experimentally observed this yielded the best baseline performance. For our exception-less version, a single memcached thread was enough to generate enough kernel work to keep all cores busy.

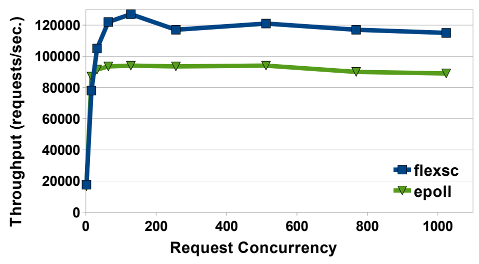
Figure 5 shows the throughput obtained from executing the baseline and exeception-less memcached on 1, 2 and 4 cores. We varied the number of concurrent connections generating requests from 1 to 1024. For the single core experiments, FlexSC employs system call batching, and for the multicore experiments it additionally dynamically distributed system calls to other cores to maximize core locality.

The results show that with 64 or more concurrent requests, memcached programmed to libflexsc outperforms the version using Linux `epoll`. Throughput is improved by as much as 25 to 35%, depending on the number of cores used.

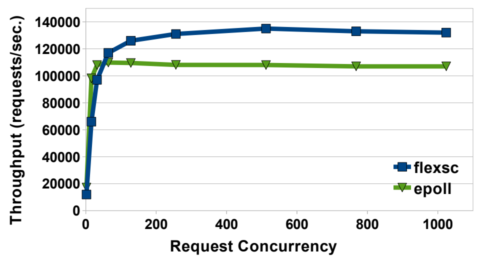
To better understand the source of performance improvement, we collected several performance metrics of the processor using hardware performance counters. Figure 6 shows the effects of executing with FlexSC, while



(a) 1 Core



(b) 2 Cores



(c) 4 Cores

Figure 5: Comparison of Memcached throughput of Linux `epoll` and FlexSC executing on 1, 2 and 4 cores.

servicing 768 concurrent memslap connections. The most important metric listed is the cycles per instruction (CPI) of the user and kernel mode for the different setups, as it summarizes the efficiency of execution (the lower the CPI, the more efficient the execution). The other values listed are normalized values of *misses* on the listed structure (the lower the misses, the more efficient the execution).

The CPI of both kernel and user execution, on 1 and 4 cores, is improved with FlexSC. On a single core, user-mode CPI decreases by as much as 22%, and on the 4 cores, we observe a 52% decrease in user-mode CPI. The data shows that, for memcached, the improved execution comes from significant reduction in misses in the performance sensitive L1, both in the data and instruction part (labelled as *d-cache* and *i-cache*).

The main reason for this drastic increase of user CPI on 4 cores is that with traditional system calls, a user-mode thread must occupy each core to make use of it. With FlexSC, however, if a single user-mode thread generates many system requests, they can be distributed and serviced to remote cores. In this experiment, a single memcached thread was able to generate enough requests to occupy the remaining 3 cores. This way, the core executing the memcached core was predominantly filled with state from the memcached process.

6.2 nginx

To evaluate the effect of exception-less execution of the nginx web server, we used two workloads: ApacheBench and a modified version of httperf. For both workloads, we present results with nginx execution on 1 and 2 cores. The results obtained with 4 cores were not meaningful as the client machine could not keep up with the server, making the client the bottleneck. For the baseline experiments (Linux `epoll`), we configured nginx to spawn one worker process per core, which nginx automatically assigns and pins to separate cores. With FlexSC, a single nginx worker thread was sufficient to keep all cores busy.

6.2.1 ApacheBench

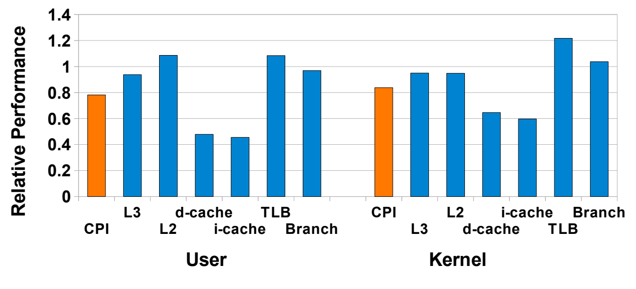
ApacheBench is a HTTP workload generator that is distributed with Apache. It is designed to stress-test the Web server determining the number of requests per second that can be serviced, with varying number of concurrent requests.

Figure 7 shows the throughput numbers obtained on 1 and 2 cores when varying the number of concurrent ApacheBench client connections issuing requests to the nginx server. For this workload, system call batching on one core provides significant performance improvements: up to 70% with 256 concurrent requests. In the 2 core execution, we see that FlexSC provides a consistent improvement with 16 or more concurrent clients, achieving up to 120% higher throughput, showing the added benefit of dynamic core specialization.

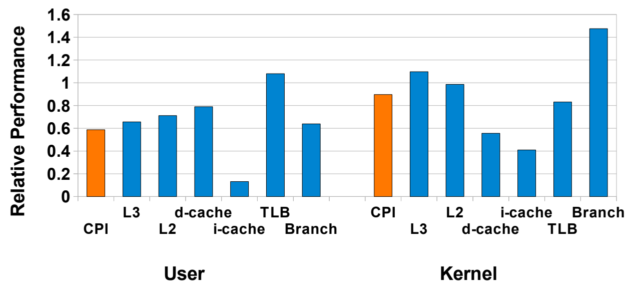
Besides aggregate throughput, latency of individual requests is an important metric when evaluating performance of web servers. Figure 8 reports the mean latency, as reported by the client, with 256 concurrent connections. FlexSC reduces latency by 42% in single core execution, and 58% in duo core execution. It is also interesting to note that adding a second core helps to reduce the average latency of servicing requests with FlexSC, which is not the case when using the `epoll` facility.

6.2.2 httperf

The *httperf* HTTP workload generator was built as a more realistic measurement tool for web server performance [19]. In particular, it supports session log files, and models a *partially open* system (in contrast to ApacheBench, which models a *closed* system) [27]. For this reason, we do not control the number of concurrent

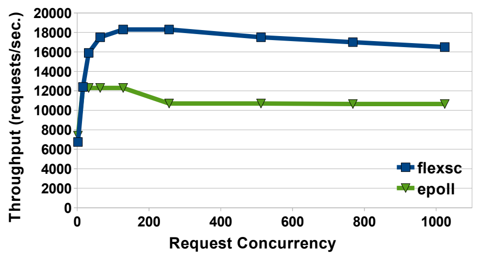


(a) 1 Core

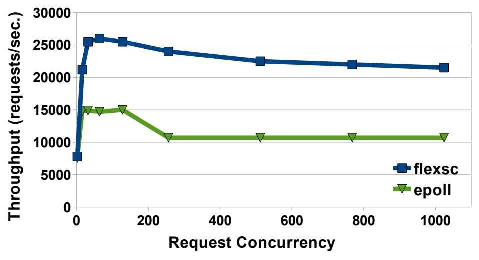


(b) 4 Cores

Figure 6: Comparison of processor performance metrics of Memcached execution using Linux `epoll` and FlexSC on 1 and 4 cores, while servicing 768 concurrent memslap connections. All values are normalized to baseline execution (`epoll`). The CPI columns show the normalized cycles per instruction, while the other columns depict the normalized misses of each processor structure (lower is better in all cases).



(a) 1 Core



(b) 2 Cores

Figure 7: Comparison of nginx performance with the ApacheBench when executing with Linux `epoll` and FlexSC on 1 and 2 cores.

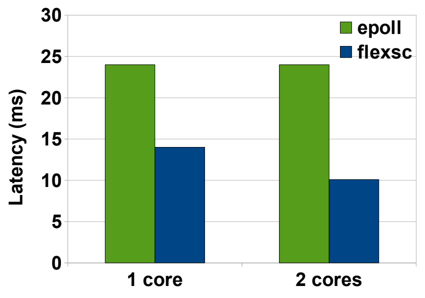


Figure 8: Comparison of nginx latency replying to 256 concurrent ApacheBench requests when executing with Linux `epoll` and FlexSC on 1 and 2 cores.

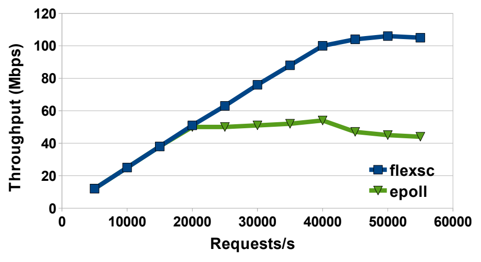
connections to the server, but instead the request arrival rate. The number of concurrent connections is determined by how fast the server can satisfy incoming requests.

We modified httperf (we used the latest version, 0.9.0) in order for it to properly handle large number of concurrent connections. In its original version, httperf uses the `select` system call to manage multiple connections. On Linux, this restricts the number of connections to 1024, which we found insufficient to fully stress the server. We modified httperf to use the `epoll` interface, allowing it to handle several thousand concurrent connections. We verified that the results of our modified httperf were statistically similar to the original httperf, when using less than 1024 concurrent connections.

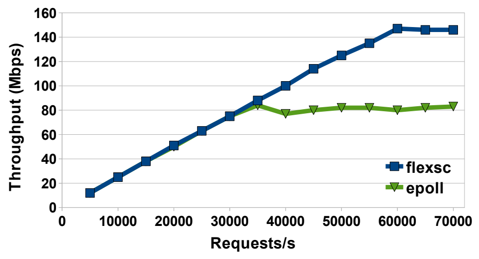
We configured httperf to connect using HTTP 1.1 protocol, and issue 20 requests per connection. The session log contained requests to files ranging from 64 bytes to 8 kilobytes. We did not add larger files to the session as our network infrastructure is modest, at 1Gpbs, and we did not want the network to become a source of bottleneck.

Figure 9 shows the throughput of nginx executing on 1 and 2 cores, measured in megabits per second, obtained when varying the request rate of httperf. Both graphs show that the throughput of the server can satisfy the request rate up to a certain value. After that the throughput is relatively stable and constant. For the single core case, the throughput of Linux `epoll` stabilizes after 20,000 requests per second, while with FlexSC, throughput increases up to 40,000 requests. Furthermore, FlexSC outperforms Linux `epoll` by as much as 120%, when httperf issues 50,000 requests per second.

In the case of 2 core execution, nginx with Linux `epoll` reaches peak throughput at 35,000 requests per second, while FlexSC sustains improvements with up to 60,000 requests per second. In this case, the difference in throughput, in megabits per second, is as much as 77%.

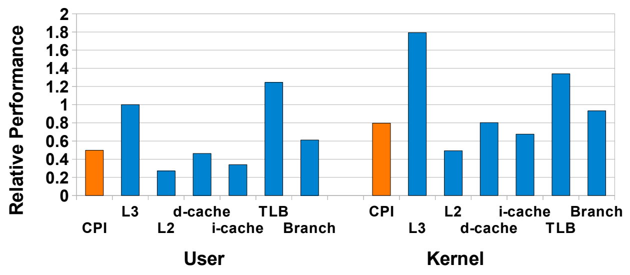


(a) 1 Core

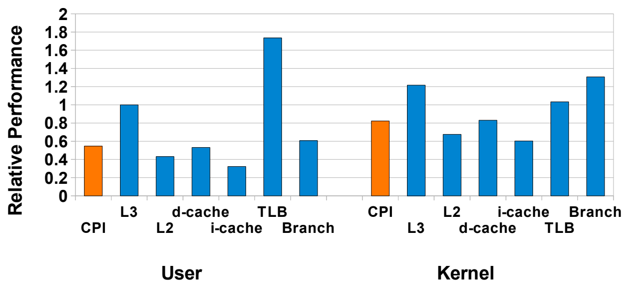


(b) 2 Cores

Figure 9: Comparison of nginx performance with the httperf when executing with Linux `epoll` and FlexSC on 1 and 2 cores.



(a) 1 Core



(b) 2 Cores

Figure 10: Comparison of processor performance metrics of nginx execution using `epoll` and FlexSC on 1 and 2 cores, while servicing 40,000 and 60,000 req/s, respectively. Values are normalized to baseline execution (`epoll`). The CPI columns show the normalized cycles per instruction, while the other columns depict the normalized misses of each processor structure (lower is better in all cases).

Similarly to the analysis of memcached, we collected processor performance metrics using hardware performance counters to analyze the execution of nginx with httperf. Figure 10 shows the several metrics, normalized to the baselinux (Linux `epoll`) execution. The results show that the efficiency of user-mode execution doubles, in the single core case, and improves by 83% on 2 cores. Kernel-mode execution improves efficiency by 25% and 21%, respectively. For nginx, not only are the L1 instruction and data caches better utilized (we observe less than half of the miss ratio in these structures), but the private L2 cache also observes miss rate reduction of less than half of the baseline.

Although we observe increase of some metrics, such as the TLB and kernel-mode L3 misses, the absolute values are small enough that it does not affect performance significantly. For example, the increase in 80% of kernel-mode L3 misses in the 1 core case corresponds to the misses per kilo instructions increasing from 0.51 to 0.92 (that is, for about every 2,000 instructions, an extra L3 miss is observed). Similarly, the 73% increase in misses of the user-mode TLB in the 2 core execution corresponds to only 4 extra TLB misses for every 1,000 instructions.

The values for the hardware performance information collected during execution driven by ApacheBench showed similar trends (not shown in the interest of space).

7 Discussion: Scaling the Number of Concurrent System Calls

One concern not addressed in this work is that of efficiently handling applications that require a large number of concurrent outstanding system calls. Specifically, there are two issues that can hamper scaling with the number of calls: (1) the exception-less system call interface, and (2) the requirement of one syscall thread per outstanding system call. We briefly discuss mechanisms to overcome or alleviate these issues.

The exception-less system call interface, primarily composed of *syscall entries*, requires user and kernel code to perform linear scans of the entries to search for status updates. If the rate of entry modifications does not grow in same proportion as the total number of entries, the overhead of scanning, normalized per modification, will increase. A concrete example of this is a server servicing a large number of slow or dormant clients, resulting in a large number of connections that are infrequently updated. In this case, requiring linear scans on syscall pages is inefficient.

Instead of using syscall pages, the exception-less system call interface could be modified to implement two shared message queues: an incoming queue, with system calls requests made by the application, and an outgoing queue, composed of system call requests serviced by the kernel. A queue based interface would potentially complicate user-kernel communication, but would avoid the overheads of linear scans across outstanding requests.

Another scalability factor to consider is the requirement of maintaining a syscall thread per outstanding system call. Despite the modest memory footprint of kernel threads and low overhead of switching threads that share address spaces, these costs may become non-negligible with hundreds of thousands or millions of outstanding system calls.

To avoid these costs, applications may still utilize the `epoll` facility, but through the exception-less interface. This solution, however, would only work for resources that are supported by `epoll`. A more comprehensive solution would be to restructure the Linux kernel to support completely non-blocking kernel code paths. Instead of relying on the ability to block the current context of execution, the kernel could enqueue requests for contended resources, while providing a mechanism to continue the execution of enqueued requests when resources become available. With a non-blocking kernel structure, a single syscall thread would be sufficient to service any number of syscall requests.

One last option to mitigate both the interface and threading issues, that does not involve changes to FlexSC, is to require user-space to throttle the number of outstanding system calls. In our implementation, throttling can be enforced within the libflexsc library by allocating a fixed number of syscall pages, and delaying new system calls whenever all entries are busy. The main drawback of this solution is that, in certain cases, extra care would be necessary to avoid a standstill situation (lack of forward progress).

8 Related Work

8.1 Operating System Support for I/O Concurrency

Over the years, there have been numerous proposals and studies exploring operating system support for I/O concurrency. Due to space constraints, we will briefly describe previous work that is directly related to this proposal.

Perhaps the most influential work in this area is Scheduler Activations that proposed addressing the issue of preempting user-mode threads by returning control of execution to a user-mode scheduler, through a scheduler activation, upon experiencing a blocking event in the kernel [2].

Elmeleegy et al. proposed lazy asynchoronous I/O, a user-level library that uses Scheduler Activations to support event-driven programming [11]. LAIO is the proposal that most closely resembles ours. However, in LAIO, system calls are still exception-based, and tentatively execute synchronously. Since LAIO makes use of scheduler activations, if a blocking condition is detected, a continuation is created, allowing the user thread to continue execution. Recently, the Linux community has proposed a mechanism similar to LAIO for implementing non-blocking system calls [8].

Banga et al. are among the first to explore the construction of generic event notification infrastructure under UNIX [4]. Their work inspired the implementation of the *kqueue* interface available on BSD and Linux kernels [16]. While their proposal does encapsulate more resources than descriptor based ones, explicit kernel support is needed for each type of event. In contrast, exception-less system calls supports all system calls without code specific to each system call or resource.

The main difference between many of the proposals for non-blocking or asynchronous 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.

8.2 Locality of Execution and Multicores

Several researchers have studied the effects of operating system execution on application performance [1, 3, 9, 15, 17]. Larus and Parkes also identified processor inefficiencies 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 [15].

Techniques such as Soft Timers [3] and Lazy Receiver Processing [10] 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.

Computation Spreading, proposed by Chakraborty et al., is similar to the multicore execution of FlexSC [9]. They introduced processor modifications to allow for hardware migration of threads, and evaluated the effects on migrating threads to specialize cores when they enter the kernel. 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 interprocessor interrupt.

Recently, both Corey and Factored Operating System (*fos*) have proposed dedicating cores for specific operating system functionality [31, 32]. 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 [20, 21] and power reduction in the presence of single-ISA heterogeneous multicores [18]. 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.

Zeldovich et al. introduced *libasync-smp*, a library that allows event-driven servers to execute on multiprocessors by having programmers specify events that can be safely handled concurrently [33]. *Libasync-smp* was designed to use existing asynchronous I/O facilities of UNIX kernels, but could be extended to rely on exception-less system calls instead.

8.3 System Call Batching

The idea of batching calls in order to save crossings has been extensively explored in the systems community. Closely related to this work is the work by Bershad et al. on user-level remote procedure calls (URPC) [6]. In particular, the use of shared memory to communicate requests, allied with the use of light-weight threads is common in both URPC and FlexSC. In this work, we explored directly *exposing* the communication mechanism to the application thereby removing the reliance on user-level threads.

Also related to exception-less system calls are *multi-calls*, which 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 [25]. 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 [5][29].

9 Concluding Remarks

Event-driven architectures continue to be a popular design option for implementing high-performance and scalable server applications. This paper proposes the use of *exception-less system calls* as the principal operating system primitive for efficiently supporting I/O concurrency and event-driven execution. We describe several advantages of exception-less system calls over traditional support for I/O concurrency and event notification facilities, including: (1) any system call can be invoked asynchronously, even system calls that are not file descriptor-based, (2) support in the operating system kernel is non-intrusive as code changes are not required to each system call, (3) processor efficiency is high since mode switches are mostly avoided when issuing or executing asynchronous operations, and (4) enabling multicore execution for event-driven programs is easier, given that a single user-mode execution context can generate a sufficient number of requests to keep multiple processors/-cores busy with kernel execution.

We described the design and implementation of *libflexsc*, an asynchronous system call and notification library that makes use of our Linux exception-less system call implementation, called FlexSC. We show how libflexsc can be used to support current event-driven servers by porting two popular server applications to the exception-less execution model: memcached and nginx.

The experimental evaluation of libflexsc demonstrates that the proposed exception-less execution model can significantly improve the performance and efficiency of event-driven servers. Specifically, we observed that exception-less execution increases the throughput of memcached by up to 35%, and that of nginx by up to 120%. We show that the improvements, in both cases, are derived from more efficient execution through improved use of processor resources.

10 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 valuable feedback from the reviewers and our shepherd, Richard West. Special thanks to Ioana Burcea for innumerous discussions that influenced the development of this work.

References

[1] Agarwal, A., Hennessy, J., and Horowitz, M. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst. (TOCS) 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] Banga, G., Mogul, J. C., and Druschel, P. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1999), USENIX Association, pp. 19–19.

[5] 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.

[6] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst. 9* (May 1991), 175–198.

[7] Bhattacharya, S., Pratt, S., Pulavarty, B., , and Morgan, J. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Ottawa Linux Symposium* (2003), pp. 371–386.

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

[9] 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.

[10] 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.

[11] 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.

[12] Fitzpatrick, B. Distributed Caching with Memcached. *Linux Journal* (2004).

[13] Gammo, L., Brecht, T., Shukla, A., and Pariag, D. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of 6th Annual Ottawa Linux Symposium* (2004), pp. 215–225.

[14] Krohn, M., Kohler, E., and Kaashoek, M. F. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), USENIX Association, pp. 7:1–7:14.

[15] 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.

[16] Lemon, J. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 141–153.

[17] 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.

[18] 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.

[19] Mosberger, D., and Jin, T. httperf – A Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev. 26* (December 1998), 31–37.

[20] 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.

[21] 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.

[22] Pai, V. S., Druschel, P., and Zwaenepoel, W. Flash: an efficient and portable web server. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1999), USENIX Association, pp. 15–15.

[23] 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.

[24] Provos, N. libevent - An Event Notification Library. `http://www.monkey.org/~provos/libevent`.

[25] 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.

[26] Reese, W. Nginx: the High-Performance Web Server and Reverse Proxy. *Linux Journal* (2008).

[27] Schroeder, B., Wierman, A., and Harchol-Balter, M. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3* (Berkeley, CA, USA, 2006), NSDI'06, USENIX Association, pp. 18–18.

[28] Soares, L., and Stumm, M. Flexsc: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010), pp. 33–46.

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

[30] 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.

[31] 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.

[32] 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).

[33] Zeldovich, N., Yip, A., Dabek, F., Morris, R., Mazières, D., and Kaashoek, F. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX)* (June 2003).