Reducing System Call Performance Overhead on Paravirtualized L^{4}Linux

Ilias Stamatis

September 8, 2020

Master's Thesis

Supervised by:
Dr.-Ing. Michael Roitzsch
Dr.-Ing. Adam Lackorzynski
Dr.-Ing. Carsten Weinhold

Chair of Operating Systems
Department of Computer Science
Technical University of Dresden

Abstract

The L4Re microkernel has long been able to function as a hypervisor and host a slightly modified Linux kernel version called L^{4}Linux. Recently, L^{4}Linux introduced a new mode of operation which allows application threads of Linux programs to decouple themselves from the Linux scheduler. These threads can then run uninterrupted on dedicated CPU cores managed by the L4Re microkernel while L^{4}Linux keeps running on a separate core. When decoupled threads need to request Linux services using system calls, these requests have to be forwarded from the CPU cores hosting the decoupled threads to the one hosting L^{4}Linux. This forwarding requires expensive cross-processor communication which involves the microkernel and uses inter-processor interrupts that cause high latency.

This thesis introduces Memsc, a new system call forwarding mechanism implemented in the L^{4}Linux kernel. Memsc works by having L^{4}Linux perform memory polling on a set of special shared memory pages that registered processes use for posting system call entries, instead of passing system call arguments via CPU registers. Involvement of the L4Re microkernel is not required at any step in this process. With the help of a modified C library, applications can use Memsc transparently without requiring modification or recompilation. New applications willing to use the user-space API directly can further benefit from additional capabilities such as an asynchronous system call execution model. The experimental evaluation of Memsc, conducted using both custom microbenchmarks and general-purpose file system benchmarks, shows that it performs significantly better than the existing L^{4}Linux system call forwarding mechanism.

Acknowledgements

First and foremost, I would like to thank my tutors Adam Lackorzynski and Carsten Weinhold for supervising this thesis, answering questions, and helping me to understand Linux and L^{4}Linux internals. I also want to thank my former colleague Nick Sampanis for helping me identify a race condition bug in my code. Appreciation is due to my former mentor and good friend William Brown as well, for previously introducing me to many synchronization (and other) concepts having knowledge of which proved to be very useful in this work. Finally, I would like to thank my parents for making it easier for me to attend TU Dresden by funding a large part of my studies, but most importantly, for their unconditional and endless support in every moment of my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

System calls are the standard way through which user programs request services from the operating system. Every time an application needs to write to disk, send a network packet or obtain system information it needs to ask the operating system to do it on its behalf. While this segregation of duties is essential for security reasons, it also incurs performance overhead. This overhead can vary depending on different factors such as the underlying architecture, the system configuration, and whether or not the operating system runs in a virtualized environment.

A great deal of research for finding ways to minimize system call related overhead has been done already over the years [19][1][20][18]. Even though most research has been focused on system call batching techniques, a previous study [25] from the Portland State University claims to have achieved performance improvements by using dedicated user and kernel CPUs. This study however was not performed in the context of a virtualized operating system.

Recently, for different reasons, the para-virtualized L^{4}Linux kernel has introduced an execution mode called "decoupling" [13]. This mode allows user threads to run isolated on dedicated CPU cores others than the ones used for executing the L^{4}Linux kernel. In this case, it has resulted in reduced system call performance for decoupled applications. Finding ways to minimize this additional overhead is essential for allowing applications issuing large amounts of system calls to run as decoupled threads.

1.1 Thesis Objectives

The main objective of this thesis is to design and implement a forwarding mechanism that allows user threads bound to a certain CPU core to invoke system calls

on a Linux kernel running on a different core. The new method must be faster than the slow existing mechanism. Existing Linux programs should be able to use the new mechanism without requiring any modification to their source code.

A secondary objective is to investigate the potential performance improvements achieved using an asynchronous system call execution model and provide new software applications with a way to use it.

1.2 Document Structure

This document is organized as follows. Chapter 2 provides the reader with the necessary background for understanding the following chapters and discusses related work. Chapter 3 introduces the new forwarding mechanism and presents its overall architecture and main design points. Chapter 4 discusses a few interesting implementation aspects of the new mechanism. Chapter 5 presents the results obtained through the experimental evaluation of this work. Finally, Chapter 6 concludes the thesis and proposes future work directions.

Chapter 2

Fundamentals and Related Work

This chapter introduces the reader to the L4 operating system and the L^{4}Linux kernel that runs on top of it. The concept of operating systems noise is presented along with the L^{4}Linux decoupling mechanism which has been designed to combat this. Then, the fundamentals of how system calls are invoked are explained, together with the added overhead incurred by the decoupling mechanism. Finally, some known techniques for reducing system call performance overhead presented in related work are discussed.

2.1 The L4 Operating System

Originally L4 was developed by Jochen Liedtke as a microkernel-based operating system of high performance [14][8]. However, nowadays the name L4 does not refer to a single operating system, but a family of operating systems or operating system kernels. The L4 application binary interface has been re-implemented by operating system teams in different universities such as TU Dresden, Karlsruhe Institute of Technology, and the University of New South Wales. These implementations added their own characteristics to the microkernel as well. L4 does not aim to be Unix-like or POSIX compatible, however some compatibility layers are provided.

2.1.1 Fiasco.OC and L4Re

L4Re is the L4 implementation developed at TU Dresden and written in C++. The operating system is focused on security and performance and it offers real-time scheduling support with hard priorities. *Fiasco.OC* [24] is the name of the microkernel itself, which together with the set of libraries and user-level components that run on top of it, forms the *L4 Runtime Environment (L4Re)*. Fiasco.OC is

also simply called Fiasco, even though nowadays it is officially referred to as *The L4Re Microkernel*. These three names will be used interchangeably throughout this text.

As a microkernel, Fiasco aims to provide user-space programmers with a minimal interface for essential functionality. Specifically, the user can create address spaces and threads and it can assign address spaces to threads in order for programs to be executed. Additionally, the microkernel offers an inter-process communication (IPC) mechanism that tasks can use in order to talk to each other. Finally, Fiasco implements thread scheduling in-kernel too. This contradicts the microkernel philosophy that says that scheduling is a policy and scheduling implementations shall not be provided by the microkernel itself, however no efficient alternative has been developed as yet.

Other policies such as memory management are implemented in L4Re but run completely in user-space (they are not part of the microkernel codebase). The user-space part of L4Re also provides the infrastructure for loading applications and offers other system services, thus when combined with Fiasco it makes for a complete operating system. Development of Fiasco.OC and L4Re is tightly coupled and happens in parallel.

Fiasco.OC supports symmetric multi-processing (SMP), enabling parallel code execution in multi-processor systems. The microkernel does not migrate threads to different cores automatically, but thread migration is possible if explicitly requested. Fiasco is scalable and suitable for a wide range of systems, from small embedded devices to large and complex high-performance computing (HPC) environments.

2.1.2 L^{4}Linux

L^{4}Linux [11][8] is a variant of the Linux kernel that runs para-virtualized on top of the Fiasco microkernel. Contrary to full virtualization, para-virtualization is a technique that requires modification of the guest operating system which then runs while being aware that it is virtualized. Fiasco in that case acts as a hypervisor and L^{4}Linux can be characterized as a user-space server in microkernel parlance. L^{4}Linux is modified to the extent that it can run para-virtualized, but these modifications are kept to a minimum and L^{4}Linux remains binary compatible with the mainline Linux kernel. At the time of writing, L^{4}Linux supports the x86, x86_64, arm, and arm64 architectures.

The purpose of running a Linux kernel on top of L4 is to reuse legacy code. The Linux environment includes an enormous collection of drivers, libraries, and other services that L4 tasks would like to use. However, implementing them again just

for use in L4 would be very expensive and difficult. L^{4}Linux can run unmodified Linux programs as L4 tasks. Additionally, it is possible to write and run programs that use both Linux and L4 services, by issuing system calls for both kernels.

In order to achieve this para-virtualization Fiasco employes a scheme that uses *virtual CPUs* or *vCPUs* [12]. vCPUs are simply L4 threads that are interruptible. Any L4 thread can act as a vCPU and vCPUs appear to L^{4}Linux as real, physical CPU cores. L^{4}Linux uses these vCPUs to multiplex the execution of its tasks and it performs context switches by binding different address spaces to the vCPU.

Scheduling works as follows. The Fiasco microkernel manages all physical CPUs and is responsible for scheduling L4 threads on them. Some of these L4 threads can be vCPUs. The Linux scheduler on the other hand is responsible for scheduling Linux processes and it does so on the vCPUs which are assigned to it. This is depicted in Figure 2.1 (a). In this example, Fiasco manages 4 physical CPUs and runs a vCPU thread on the first 3 of them. These 3 vCPUs are visible to L^{4}Linux which can use them for scheduling work as it sees fit. L^{4}Linux cannot interfere with Fiasco's scheduling and vice versa.

2.2 Operating System Noise

Operating system noise or jitter is the overhead introduced by the operating system when it interrupts the progress of a running application. This overhead is normally caused by handling interrupts, by maintenance activities of the kernel such as page reclamation and internal synchronization, and also by preemption of the user application in order to schedule system daemons.

Operating system noise can be especially problematic on large high-performance computing (HPC) systems with multiple cores running parallel applications. In these systems delays introduced on a single CPU core can greatly reduce the overall performance of the system. Previous studies have shown that as the number of CPU cores increases, this performance overhead becomes more and more significant [22][21][9].

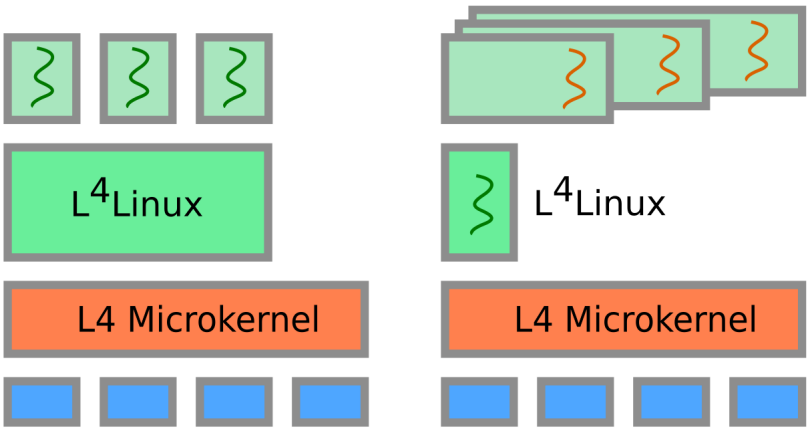
On the other hand, lightweight microkernels such as Fiasco, introduce minimal to no operating system noise at all. Fiasco can be configured in such a way that if only one thread runs on a core, even periodic timers interrupts are not delivered to that core. This is ideal for latency-critical applications. However, when running Linux applications on top of L4, it is the L^{4}Linux kernel which introduces the unwanted execution jitter.

2.3 Decoupling Thread Execution from L^{4}Linux

One solution that has been recently proposed and implemented for combatting OS noise on an L4 environment is decoupling thread execution from the noisy L^{4}Linux kernel [13].

The main idea behind the decoupling mechanism is to separate the execution of a thread in a Linux process from the vCPU it is normally running on. This decoupled thread can then run directly as a standalone L4 thread on a physical CPU different than the one that executes the vCPU thread. The result of this separation is that now L^{4}Linux cannot preempt this thread interrupting its progress.

The way this mechanism works is by creating a new native L4 thread that runs in the same address space as the thread to be decoupled. The execution of the new thread is then moved to a different CPU, outside the control of the Linux scheduler. This new thread runs only user-space code for that process and it can keep executing uninterrupted as it is now managed by the Fiasco microkernel. We can refer to this thread as the *user thread*.



(a) L^{4}Linux with three processes on vCPUs running on all cores (no decoupling)

(b) L^{4}Linux with three processes with decoupled threads on dedicated cores

Figure 2.1: Illustration of L^{4}Linux tasks running in standard and decoupled mode. Reprinted from [13].

The original Linux thread remains under L^{4}Linux's control, bound to one of its vCPUs, but it now executes kernel code only. This thread is put to sleep in order to ensure that the Linux scheduler won't pick it up for execution. It only wakes up and gets scheduled when the decoupled, user thread requests Linux services such as the execution of a L^{4}Linux system call or when it generates any kind of

exceptions. After servicing the request, it goes back to sleep until the user thread generates an exception again. We can refer to this thread as the corresponding *kernel-side context*.

Using the decoupling mechanism, we can employ a configuration where the execution of the L^{4}Linux kernel is isolated to a small subset of the total available physical CPU cores. For example, in a system with 32 available CPU cores, we can have 1 or 2 vCPU threads (bound to 1 or 2 physical CPUs correspondingly) running L^{4}Linux code. The remaining physical CPUs of the system can be used for running decoupled Linux threads or other native L4 tasks. When the decoupled threads require any services from the L^{4}Linux kernel, such as system call execution or page fault handling, these requests are forwarded to the appropriate core running L^{4}Linux. A similar scenario with 4 real CPUs and 1 vCPU is depicted in Figure 2.1 (b).

One side effect of this approach, if we take the microkernel out of the picture, is that it results in a deployment with dedicated user and kernel CPUs. Even though Fiasco still manages all the cores, it can be configured in a way that it doesn't interfere with the execution of user processes at all. And as previously mentioned, studies have shown that having dedicated user and kernel CPUs can result in significant performance gains [25].

Nonetheless, since the new decoupled thread and the corresponding kernel-context thread are bound to different physical cores, there are certain forwarding costs associated with the decoupling scheme. Eliminating this extra overhead, which is explained in detail in the following sections, served as the main motivation for the work in this thesis.

2.4 How System Calls Are Invoked

System calls are the way with which user programs can request services from the underlying operating system. They serve as an interface to the kernel which executes certain code on behalf of user processes. User-space programs can request hardware services, such as reading from or writing to a disk, communication with other processes, system information, and various other types of services. Some examples of Linux system calls are *open*, *read*, *write*, and *fork*. Each operating system offers its own set of system calls.

2.4.1 System Call Invocation on Non-Virtualized Linux

Most CPU architectures support different CPU modes of different privilege levels. Usually, user applications execute with the CPU operating on a less privileged

mode (referred to as user mode), while all kernel code executes on the most privileged CPU mode (referred to as kernel mode). On less privileged modes, certain hardware instructions will fail or they will behave differently. This is done for security reasons in order to ensure that user applications cannot do arbitrary things such as controlling the hardware for malicious purposes.

Since user programs cannot perform certain actions, they have to ask the operating system to do it on their behalf. In order for this to happen, there needs to be a transition from user mode to kernel mode. The CPU will automatically switch to kernel mode when certain things happen, for example when a page fault is generated, when an interrupt is received or when there has been an attempt to execute an illegal instruction. In any of the above cases, the CPU automatically switches its mode of operation and starts executing the kernel on a defined entry point. When the kernel wishes to return back to user mode, it can directly do that using a certain hardware instruction.

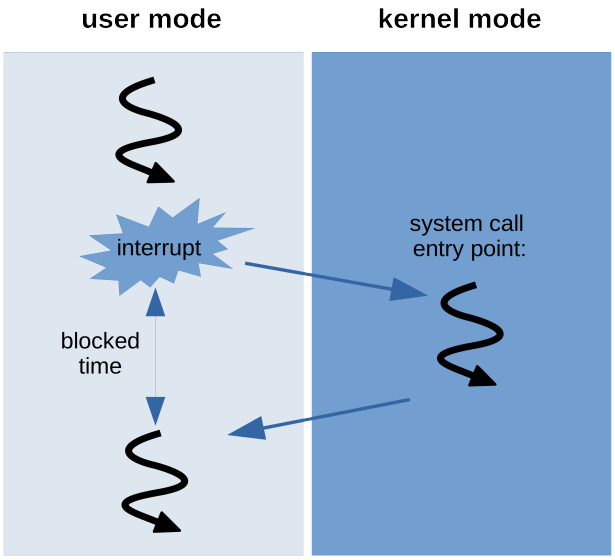


Figure 2.2: Illustration of the CPU mode switching happening a) when the program generates a software interrupt and b) when the kernel completes the execution of a system call and returns to user-space.

The way user programs voluntarily enter the kernel for system call execution is by deliberately generating a software interrupt or exception. This is illustrated in Figure 2.2. Prior to that, the program must place both the system call number and the system call arguments in certain CPU registers. When the exception is generated, the CPU switches to kernel mode and enters the kernel from a certain

point which indicates that a system call has been requested. The kernel can then examine the contents of the registers in order to determine which system call function it needs to invoke and with which arguments. The return value of the system call is also stored in a specific register. When the kernel is done executing the system call, another mode switch happens, and control returns back to the user space program.

In the x86 architecture, a user program needs to execute an *INT 0x80* instruction in order to invoke a system call in the kernel[1]. This instruction generates a software interrupt with an interrupt vector value of 0x80 which corresponds to system call exceptions. In the case of the ARM architecture, the *SVC* (formerly *SWI*) instruction is used in order to enter the *Supervisor mode*.

2.4.2 The C Library's Role

User programs normally do not employ code that invokes system calls directly by setting up the appropriate registers and generating an interrupt. This would be extremely inconvenient and also non-portable. Instead, they use wrapper functions offered by the C library which generate the appropriate assembly code for the target architecture. The C library offers wrappers for the majority of the available system calls plus a generic *syscall* function which can be used to easily invoke system calls for which a dedicated wrapper is not provided.

Figure 2.3 illustrates the steps involved in the execution of a system call using the example of the *read()* system call. The application code calls the C library wrapper which is responsible, among other things, for setting up the CPU registers in the way the kernel expects them and raising an exception by using inline assembly code. The exception results in a CPU mode switch and the invocation of the in-kernel system call handler. This handler after examining the system call number on the appropriate register, in turn dispatches the call to the appropriate system call service routine named *sys_read()* in this case. Once this routine completes, execution goes back to the system call handler which uses the *iret* instruction (on x86) in order to switch back to user mode. Control returns back to the C library wrapper function which finally forwards the return value of the system call to the user application.

[1] x86 additionally offers a way of performing "fast system calls" using the *sysenter/sysexit* or *syscall/sysret* pairs of instructions which are faster than using software interrupts. The mechanics are the same however, with the difference being that there is a separate entry point in the kernel dedicated to system calls. This way of performing system calls is now the standard while using a software interrupt is considered legacy.

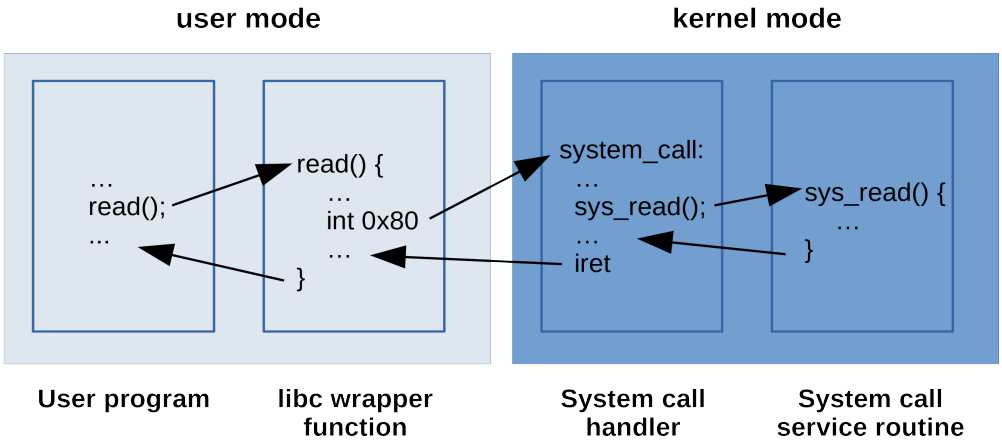


Figure 2.3: Steps in the invocation of a system call using the C library

2.4.3 System Call Invocation on Para-Virtualized L^{4}Linux

L^{4}Linux is able to directly run unmodified Linux programs that are not aware they are executed on a virtualized kernel. This means that these user programs will try to invoke L^{4}Linux system calls the way they know by raising exceptions. However, since L^{4}Linux is virtualized it does not run in a privileged CPU mode. The kernel running in privileged mode and directly managing the hardware is the Fiasco microkernel. That means that whenever an L^{4}Linux task raises an exception, control will be transferred to Fiasco and not L^{4}Linux.

Upon receipt of an interrupt, Fiasco can determine that the interrupt is intended to be caught by L^{4}Linux since it has been generated by a vCPU thread. In this case, it switches the address space of the vCPU to that of the L^{4}Linux kernel. L^{4}Linux has a predefined vCPU entry point for system calls and other asynchronous events such as page faults and hardware interrupts. Fiasco then switches to user mode and starts executing the vCPU from that entry point.

L^{4}Linux can now see and handle the exception. When it completes execution of the system call, it switches the address space of the vCPU to that of the process of whose behalf it executed the system call and it lets it run on the vCPU again. L^{4}Linux is able to switch the address space of the vCPU on its own through a special L4 system call.

2.4.4 System Call Invocation from Decoupled Threads

Decoupled L4 threads run on a possibly isolated core, managed by Fiasco, on which no vCPU is executing. That means that when a detached L4 thread wants

to invoke an L^{4}Linux system call, the execution of the system call needs to happen on a different CPU core on which L^{4}Linux is running (inside a vCPU).

In this case, system calls are essentially forwarded to some CPU hosting L^{4}Linux. The mechanics are similar to the one described in the previous section but this time cross-processor communication is required. Again, when the decoupled thread generates an exception, it traps into Fiasco which in this case creates an L4 IPC message.

This IPC message appears to come from the faulty thread and is directed to its exception handler which is the vCPU hosting the corresponding kernel-side context. Since the sender and receiver are in different cores, cross-core IPC is used which is significantly slower than core-local IPC. The reason it's slower is that it uses inter-processor interrupts as its notification mechanism in order to interrupt the execution of the remote processor.

When the interrupt hits the other processor, the Fiasco kernel running there forwards the exception to the appropriate vCPU. Once L^{4}Linux receives the exception (i.e. the IPC message), it wakes up the corresponding kernel-side context of the decoupled thread described in section 2.3. This thread, whose only purpose is to run L^{4}Linux kernel code handling exceptions on behalf of the decoupled thread, finally executes the system call. After completing the system call execution, L^{4}Linux puts this thread back to sleep and it doesn't wake it up until there is a new exception that it needs to handle for the decoupled thread.

Once the return value of the system call is ready, L^{4}Linux needs to inform the core hosting the decoupled thread by replying to the original IPC message. At this point, another inter-processor interrupt needs to be used for signaling. This cross-processor forwarding using inter-processor interrupts comes with a significant performance overhead over both native Linux and L^{4}Linux without decoupling. This can be prohibitive for time-critical or frequently used operations. It becomes clear that the L4 architecture needs a faster way of forwarding system calls from decoupled threads to L^{4}Linux.

2.5 Virtual System Calls and the vDSO

As explained in the previous sections, executing a system call requires at least one CPU mode switch. However, a few simple system calls that merely return information form the kernel are trivial enough that they can be executed without requiring a transition to kernel mode. That's why the Linux virtual Dynamic Shared Object (vDSO) [26] was created.

The Linux vDSO is a small shared library containing kernel code which the kernel maps into the user space part of the address space of all applications. Some simple frequently used system calls such as *gettimeofday* can then be executed without entering the kernel at all. Only read-only system calls qualify for this since user-space processes are not allowed to write into the kernel address space.

Due to the address space layout randomization (ASLR) security mitigation that is enabled in most systems, the location of vDSO is not fixed. The C library takes care of the task of locating the vDSO object in runtime by looking for an auxiliary ELF header in which the kernel writes its address when the program is loaded. The C library then uses the vDSO whenever possible and the whole process is transparent to applications.

The vDSO mechanism is supported in L^{4}Linux in both decoupled and no-decoupled mode [13]. That means that when a detached thread issues a vDSO-supported system call, no cross-processor communication is required. Nevertheless, the amount of system calls included in the vDSO is quite small. This depends on the architecture, but for example, on x86_64 only 4 system calls are supported; *clock_gettime*, *clock_getcpu*, *get_timeofday*, and *time*. Other architectures support more, but usually not more than 10.

Therefore, vDSO system calls alone are not enough to compensate for the extra performance overhead the decoupling mechanism introduces. Most applications will use a large variety of system calls, most of which not included in vDSO.

2.6 System Call Batching

Previous studies have highlighted the performance gains when using system call batching techniques [19][1][20][18]. Crossing the user-kernel boundary often can have significant overhead in terms of performance for reasons of mode switching and cache pollution.

Normally, when a user process issues a system call, the user mode pipeline is flushed, the privilege level of the CPU changes, and kernel code starts executing. Execution of kernel code results in the pollution of system caches. Once the kernel is done executing a system call, another mode switch happens and user code of the application starts executing again, but this time causing many more cache misses since the data and instruction caches have been filled with kernel data. These cache misses can affect performance significantly. Specifically, it can take up to 14,000 cycles of execution before the instructions per cycle rate of the application returns to the same level as before executing a system call [23].

The idea behind system call batching is that user programs can collect a number of system calls before they ask the kernel to execute them all at once. This way the system call related overhead is minimized since now only two CPU mode switches are required for executing each batch. The cache pollution because of the kernel execution also happens much less frequently.

One drawback is that existing applications need to be modified in some way in order to use system call batching techniques. Also, most operating systems do not inherently offer such a mechanism. Additionally, system call batching might not be feasible for some applications or algorithms at all. Many programs cannot issue further system calls or otherwise make progress before previously issued system calls complete. Nevertheless, new applications if written in a way that allows batching of system calls, and by using the appropriate libraries or tools, they can observe significant performance gains.

2.7 Exception-Less System Calls

A 2010 paper from Livio Soares and Michael Stumm has introduced the concept of exception-less system calls and a Linux implementation of it called FlexSC [23]. Using the FlexSC mechanism user applications can post system calls in memory, instead of using the traditional scheme with software exceptions and arguments passing via CPU registers. The Linux kernel then picks them up from there in order to execute them and writes the return value back to the shared memory area.

FlexSC is focused on highly threaded applications. The main idea is that different threads of the same process can post system calls in memory, one each. Once a thread posts a system call it goes to sleep and another thread of the same application gets user-level scheduled. All these threads could not execute in parallel anyway, since there is the limit of the physical CPU cores available in the system. When some or all of the threads have posted a system call and therefore cannot proceed, a traditional exception-based system call is used to indicate to the kernel that it needs to start processing the system calls that have already been posted to memory for this process.

This scheme allows for batching system calls, but instead of the applications having to prepare the batches themselves, the batches are automatically created from system calls of different threads that share the same address space. For this purpose, the authors of the paper have developed a user-space threading library compliant with POSIX Threads and binary compatible with NPTL, the default thread library on Linux. The difference is that the threads are managed

and scheduled by the library itself in user-space as in a green threads model. This library allows existing applications to transparently use the exception-less system calls mechanism without requiring modifications to their source code or recompilation.

While FlexSC seems to notably improve the performance of heavily multi-threaded applications with hundreds or thousands of threads, it does not seem to provide any performance benefits to existing single-threaded applications or multi-threaded applications that use a small number of threads. Nevertheless, the idea of invoking system calls using shared memory and without using software interrupts (which are a major bottleneck for decoupled threads as described previously), served as an inspiration for the creation of a faster system call forwarding mechanism for L^4Linux.

Chapter 3

Design

This chapter introduces *Memsc*, an alternative method for invoking L^{4}Linux system calls. The main design points are presented along with the overall architecture of the system. Additionally, the way Memsc can be used is explained and some example code snippets are given.

3.1 Overview

The current system call forwarding mechanism using inter-processor interrupts, described in Section 2.4.4, is a significant performance bottleneck for applications issuing large amounts of system calls. For this purpose, *Memsc* has been developed as an alternative way to forward system calls from decoupled L^{4}Linux threads using shared memory.

The general idea is that decoupled threads can post system calls in special memory pages while L^{4}Linux is constantly polling memory in order to detect changes in those pages. Once L^{4}Linux encounters new system calls, it immediately executes them on behalf of the appropriate thread. When execution is complete, the result is written back to memory for the requesting thread to pick it up from there.

Even though the main goal is faster system call forwarding in general, Memsc can provide applications willing to directly use its API with further benefits such as system call batching and asynchronous system calls. This works by allowing applications to post multiple system calls in memory at once before blocking and waiting for their processing.

In general, system calls need not happen in a synchronous matter. Applications don't necessarily have to block when issuing a system call. Instead, a new entry

can be simply written to memory while the application proceeds with other useful user-space work. At a later point in time, the application can check if a system call it has previously issued is complete in order to pick up its return value.

Using Memsc for system processes is optional. That means that some programs can opt to use Memsc, while others can still forward their system calls using the traditional way that uses L4 IPC. It is even possible to combine the two mechanisms and have some system calls be forwarded using one of them and some using the other. Detached threads that want to register for Memsc services simply need to invoke the new *memsc_register()* system call which was added to L^{4}Linux as part of Memsc. It is worth noting that even non-decoupled processes can use Memsc, however this only makes sense when they want to use its asynchronous system call execution model.

3.2 The Syspage

Every user process using Memsc posts its system calls in a special memory page called the *syspage*. The syspage is nothing more than a regular memory page that is shared between the user-space and the kernel-space in order to be readable and writeable from both L^{4}Linux and the corresponding detached user thread. Each user process registered for Memsc services has its own syspage.

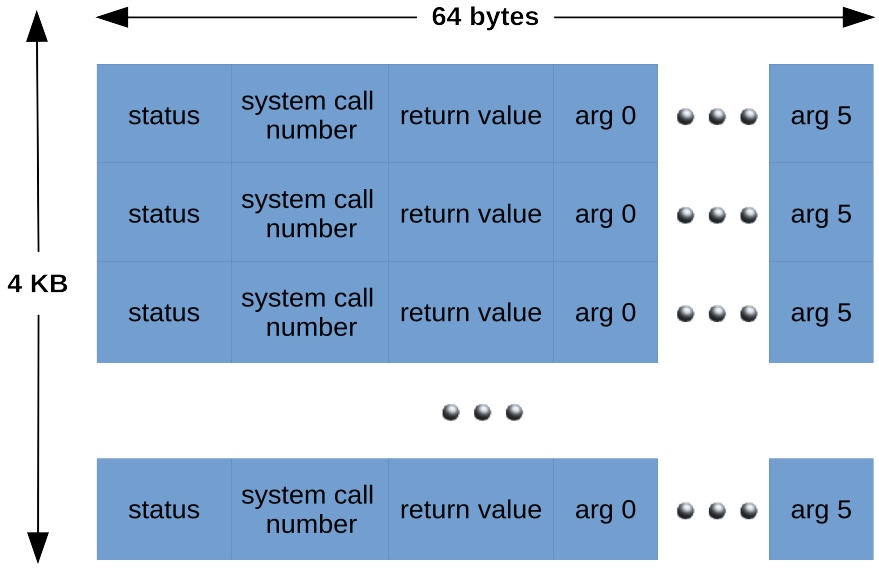


Figure 3.1: A syspage consisting of multiple system call entries

The syspage can be viewed as a table of adjacent system call entries. Each entry consists of an entry status, the system call number, the system call arguments (which can be up to 6 according to the Linux ABI), and the return value for when the system call has been processed by the kernel.

Entries of this table can be a) *free* meaning that that slot of the table is available for posting a new system call, b) *submitted* meaning that the user-space has already posted a system call but its result is not yet available, and c) *done* meaning that the system call has been processed and its result is available. This is determined by the status field of each entry.

Figure 3.1 illustrates a syspage in a system with a memory page size of 4KB which is the most common page size. Syspage entries have been designed to occupy exactly 64 bytes in order to match the cache line width of modern CPUs and avoid false sharing. As a result, in such a system, a maximum of 64 unprocessed system call entries could be posted before running out of slots in the syspage.

The syspage is implemented as a circular buffer also known as ring buffer. That means that entries are processed by the kernel in the same order they have been submitted. Hence, if a user thread posts multiple entries and later finds that one of them has been processed, it automatically knows that all entries posted before that one have been processed as well.

3.3 The Scanner Thread

As mentioned in the previous sections, L^{4}Linux needs to be constantly polling memory in order to detect newly posted system calls. This exactly is the job of the *scanner* thread, an L^{4}Linux kernel thread dedicated to this task.

The scanner is constantly checking all the registered syspages for updates in a round-robin fashion. Since as mentioned in the previous section a syspage is actually a ring buffer, only the head entry of that buffer needs to be checked on each iteration. Hence, the scanner does not need to perform a full scan of each syspage every time. It simply needs to "scan" all registered syspages by checking their head entry only.

No other task is performed by the scanner, nor does it execute system calls that it finds in the scanning process. Instead, once the scanner actually encounters an unprocessed entry, it simply dispatches the work to some other thread in the system and continues by scanning the rest of the pages.

The scanner thread does not need to run at all times. It is automatically spawned when needed (ie when Memsc services are initially requested) and it is

terminated once there are no more Memsc processes in order to not consume system resources. A single scanner thread is sufficient for a large number of registered processes, since as explained above the scanning process is fast.

3.4 Worker Threads

Memsc worker threads are L^{4}Linux kernel threads that execute system calls on behalf of user processes. Each worker thread is associated with a single user process and each user process has a unique worker thread executing its system calls. Worker threads are able to execute system calls for a certain process by accessing the same address space as that process.

These threads sleep for as long as there is no work for them to process. However, once the scanner thread detects a new system call on a syspage, it immediately wakes up the corresponding memsc worker thread in order to be scheduled as soon as possible.

Once a worker thread wakes up, it executes all pending system calls found in its ring buffer, not just the first one. This way, worker threads enable batch processing of system calls. When a worker finishes all the work that it finds on its syspage it goes back to sleep, waiting for the scanner thread to wake it up again once more work is available.

3.5 Architecture

A visual representation of the system architecture with Memsc is depicted in Figure 3.2. A system with 4 physical CPUs and 2 vCPUs sitting on the last 2 physical cores is used in this example. A decoupled thread that has registered for Memsc services runs on CPU 0, outside of L^{4}Linux's scope. Its Memsc worker thread along with its corresponding kernel-side context described in section 2.3 are managed by L^{4}Linux and their execution is multiplexed in vCPU1. All 3 threads can access the same memory and they run in the same context from the perspective of L^{4}Linux.

The decoupled thread might need to enter the L^{4}Linux kernel in the following cases; when it invokes system calls, when a hardware interrupt is received (for example the network card notifies that a new network packet has arrived) and when a software exception is raised (for example due to a page fault). Traditionally, all these kinds of events would be forwarded to L^{4}Linux in the way described in section 2.4.4. The software or hardware exception traps into Fiasco in CPU 0, which notifies the Fiasco instance running on CPU 3 using an inter-processor

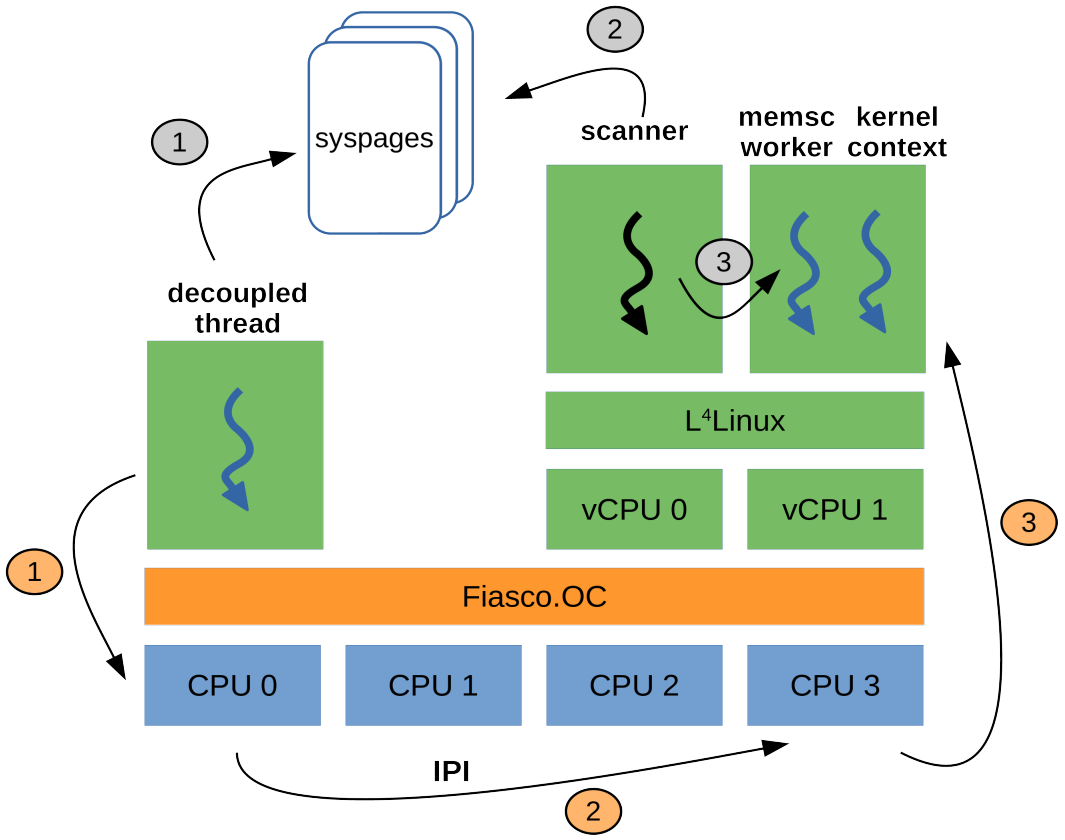


Figure 3.2: Memsc in action. System calls are forwarded via syspages (gray) while other exceptions are forwarded in the traditional way using L4 IPC (orange). Threads in blue run in the same context from the perspective of L^{4}Linux.

interrupt. Fiasco on that core then forwards it to vCPU1 and finally L^{4}Linux wakes up the appropriate kernel-side context to handle the exception.

The process for invoking system calls using Memsc is different. The decoupled thread first posts one or more system call entries on its syspage. The scanner thread running on its dedicated vCPU constantly scans all syspages. Once it detects a change, it immediately wakes up the corresponding worker thread in order to execute the system call. Note that neither the L4Re microkernel nor the kernel-side context are involved now in system call invocations. However, all kinds of exceptions[1] are still forwarded using the traditional way.

[1] system calls are now exception-less

Using this architecture requires an additional kernel thread for each decoupled L^{4}Linux process. Nonetheless, this does not really add any noteworthy overhead. The memory footprint of the extra thread is negligible anyway and Linux's Completely Fair Scheduler (CFS) requires constant time for choosing a thread for execution independently of the total amount of threads present in the system.

3.5.1 Parallel Execution of User and Kernel Code

An interesting aspect of this architecture is that it allows user-space code and kernel-space code execution of the same process in parallel. The decoupled thread, can post a few system call entries on its syspage and continue executing user-space code. At the same time, having being woken up by the scanner, the Memsc worker can execute system calls for the same process in parallel.

This eliminates any blocked time where the user thread cannot make any progress. Figure 3.3 depicts this and can be compared with Figure 2.2 where the user thread has to block and wait until the system call execution is complete.

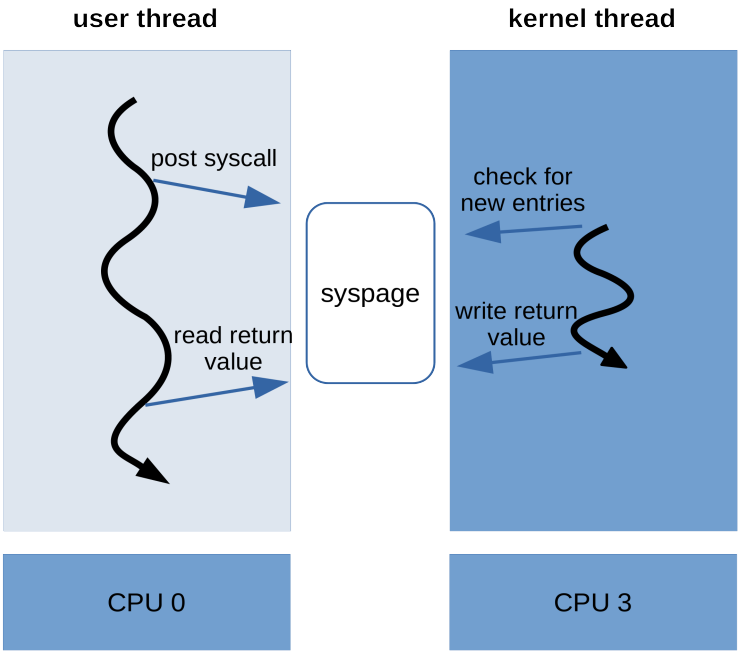


Figure 3.3: Parallel execution of user-space and kernel-space code for the same process. On the left is a decoupled L^{4}Linux thread and on the right its corresponding Memsc worker executing its system calls in parallel on a separate CPU.

Since system calls and exceptions are forwarded separately, it is even possible that different kernel code for the same process is executed in parallel. For example, if the decoupled thread posts a few system calls and then generates a page fault, there can be a situation where the memsc worker and the kernel-side context run in parallel. In order for this to happen, there would need to be enough available vCPUs running on separate physical CPUs. In that case, no user-space progress would be made since exceptions cannot be dealt with asynchronously.

3.6 Using Memsc

There are two ways of using the new Memsc forwarding mechanism. One is by directly invoking the Memsc user-space library and the other one is by using it indirectly via a modified version of the C library. This section presents memsclib and its API through which applications can take full advantage of all Memsc capabilities.

3.6.1 The Memsc User-Space Library

User applications can benefit from the new forwarding mechanism using *memsclib*, the Memsc user-space library. This library can be linked dynamically in programs or it can be statically linked during compilation time. Applications can then call the functions offered by its public interface directly.

Using memsclib requires source code modifications for existing applications. Therefore this approach is more suitable for new applications or for smaller existing programs for which porting them would not require significant efforts.

By using memsclib directly, an application can take full advantage of all of the Memsc mechanism's capabilities. In particular, it is the only way for an application to use system call batching and asynchronous system calls that allow parallel execution of user and kernel code as described in the previous section. Hence, using memsclib directly is especially suitable for applications that can be designed in advance (or be ported) to batch system calls and benefit from their asynchronous execution.

3.6.2 Application Interface (memsclib API)

User programs that are linked against memsclib and include the appropriate header file can make use of the following interfaces.

int memsc_register()

This function is used by a user process in order to register for Memsc services. It needs to be called prior to any other Memsc-related function. A new syspage is created as a result of that call along with a worker thread for that process.

On success 0 is returned, otherwise an integer indicating an error.

memsc_idx memsc_add(int sysno, ...)

This function is used for posting new system calls in the syspage. The new entry is added to the first free slot of the ring buffer.

On success, an identifier that refers to a specific slot in the syspage is returned. This identifier needs to be passed to other memsclib calls.

bool memsc_ready(memsc_idx idx)

This function returns true when the entry referred to by *idx* has been processed by the kernel and the result is available. A process can busy-wait by repeatedly calling this function if it cannot proceed otherwise without having the result of the system call.

long memsc_retval(memsc_idx idx)

This function returns the system call's return value stored by the kernel in the entry referred to by *idx*. This function must be called only after *memsc_ready()* for the same entry has returned true. It must be called only once for a given index as it results in its slot being freed.

void memsc_wait_all()

Calling this function blocks execution of the calling thread until all of the posted system calls have been processed by the kernel.

void memsc_wait_any()

Calling this function blocks execution of the calling thread until one or more of the posted system calls have been processed by the kernel.

3.6.3 Examples

Using the memsclib API presented above, applications can invoke system calls in both an asynchronous and a synchronous manner.

Figure 3.4 shows an example of issuing a traditional, blocking system call using memsclib. Before doing anything else, the application needs to register for Memsc

```
1  #include "memsclib.h"
2
3    ...
4
5    if (memsc_register())
6      return -1;
7
8    memsc_idx m_open = memsc_add(__NR_open, "file", 0, "r");
9
10   while (!memsc_ready(m_open));
11
12   int fd = memsc_retval(m_open);
13
14   ...
```

Figure 3.4: Example of a synchronous blocking system call invocation using Memsc

services using the *memsc_register()* call. If the call succeeds, the application proceeds with posting an *open* system call in its syspage using *memsc_add()* and then performs busy waiting in line 10 until the result of the system call invocation is available. Finally, it consumes the value by copying it back to its stack before proceeding with its execution.

```
1    ...
2
3    memsc_add(__NR_write, fd1, str, strlen(str));
4    memsc_add(__NR_write, fd2, str, strlen(str));
5    memsc_add(__NR_write, fd3, str, strlen(str));
6    memsc_add(__NR_write, fd4, str, strlen(str));
7
8    memsc_wait_all();
9
10   ...
```

Figure 3.5: Example of system call batching using Memsc

Figure 3.5 shows an example of system call batching using memsclib. This time the application posts 4 *write* system calls to different file descriptors it already holds. It then blocks using the *memsc_wait_all()* library call. Once this call returns, it means that all system calls posted until the time of the call have been completed

and their results can be collected.

3.7 Using Memsc Transparently

Since porting existing applications to use memsclib can often be prohibitive, using memsclib directly is not always an option. In this case, another way of using Memsc services without modifying existing programs is offered. This is achieved with the help of a slightly modified version of the C library that enables Memsc usage and has been adjusted for this purpose.

Thanks to the dynamic loading capabilities of Linux, the whole process is completely transparent to applications. Existing programs simply need to link against the modified C library and they can then call the library's system call wrappers as usual. These wrappers, instead of setting up the CPU registers and raising an exception, they make calls to memsclib.

Due to the fact that the C library semantics imply blocking system calls, using Memsc this way does not allow for asynchronous system calls and system call batching. However, even in this case, L^4Linux applications running decoupled on a separate core should experience improved performance nevertheless since all system calls are forwarded via the syspages.

Chapter 4

Implementation

This chapter discusses a few interesting aspects of the Memsc implementation such as establishing shared memory between the user and kernel space, spawning kernel threads, polling memory, and synchronizing accesses to the syspages.

4.1 Supported Architectures

Memsc has been implemented for the arm and aarch64[1] architectures. Nonetheless, porting Memsc to other architectures should be straightforward and require trivial effort. Specifically, only a single function needs to be ported. This short function, after extracting the system call arguments from the syspage, it dispatches the call using the system call table corresponding to that CPU architecture. Additionally, the new *memsc_register* system call needs to be appended to that same table.

Arm architecture's weak memory model introduces a few challenges related to memory polling and synchronization which are discussed in section 4.5.

4.2 Establishing Shared Memory Between User-Space and Kernel-Space

A syspage is simply a memory page shared by the kernel and a detached user thread that has registered for Memsc services. The kernel and the user thread can both read and modify its contents. Accessing the syspage is done on both

[1] arm's 64-bit extension

sides by simply dereferencing a pointer they hold to it. This memory can either be allocated in kernel-space using *kmalloc* or a similar kernel function, or it can be allocated in user-space using one of the memory allocator interfaces offered by the C library.

In case the memory is allocated from within the kernel, then the appropriate mappings must be established in the page tables of the user process. In general this is possible by providing a custom *mmap* implementation that maps the pages when page faults are generated. Chapter 15 of the LDD book [7] explains exactly how this is done and provides the reader with example code. Doing the opposite, allocating the memory in user-space and passing a reference to it to the kernel, appeared to be simpler and is the approach used in Memsc. To understand why it is simpler, we first need to understand how memory is organized in L^{4}Linux.

In L^{4}Linux the kernel runs on a separate address space than user processes. That means that a user pointer that is valid in the address space of a process is not valid when the kernel is scheduled. However, the kernel address space contains mappings for all memory belonging to user processes. The same physical pages are simply mapped in different locations in different address spaces. Therefore, if L^{4}Linux receives a user-space address it doesn't need to create a new kernel mapping for that memory since it already has one.

This is different from native Linux where both kernel-space mappings and user-space mappings exist in the address space of all processes[2]. The kernel doesn't have an address space of its own. Unlike L^{4}Linux, when the kernel needs temporary kernel mappings for user memory it has to create them.

Establishing a shared communication channel between the user-space and the kernel in Memsc works as follows. The user-space allocates one page's worth of memory using the *aligned_alloc()* library function. This function makes sure that the memory is allocated on a page boundary. This is important as it prevents the syspage from being physically split across two memory pages. The user-space then passes the address of the newly allocated page to the kernel as an argument to the *memsc_register()* system call.

Once the kernel receives the user-space pointer, it passes it to the *get_user_pages()* kernel function. This function does two things. First, it walks the page tables of the corresponding user process and returns a reference to the physical memory

[2]Native Linux also uses separate page tables in user mode and kernel mode when the Kernel Page Table Isolation (KPTI) feature is enabled. KPTI is used as a mitigation against the Meltdown security vulnerability.

page to which the user address belongs. Second, it makes sure that this page remains pinned in main memory and it doesn't get swapped out.

At this point the kernel already knows which physical page corresponds to the syspage for that process but it doesn't know how to refer to it. To get a working kernel pointer for it, it needs to use the *kmap()* function. Even though in native Linux *kmap* would create a new kernel mapping as previously explained, in L^{4}Linux it simply returns the existing mapping since it has already been established.

```
1  SYSCALL_DEFINE1(memsc_register, unsigned long __user,
       syspage_addr)
2  {
3    struct page *ph_pages[1];
4    void *sysp_addr;
5
6    ...
7
8    if (syspage_addr % PAGE_SIZE != 0)
9      return -MEMSC_ERR_PTR_NOT_ALIGNED;
10
11   /* locate the physical page and pin it in memory */
12   if (get_user_pages_fast(syspage_addr, 1, FOLL_WRITE,
       ph_pages) != 1)
13     return -MEMSC_ERR_INVALID_PTR;
14
15   /* find the existing mapping for the syspage in the
       kernel's address space */
16   sysp_addr = kmap(ph_pages[0]);
17
18   ...
```

Figure 4.1: Kernel code for obtaining a valid pointer to the syspage whose address is passed from the user-space

From that point on, the kernel can directly manipulate the syspage by simply dereferencing the pointer returned from *kmap*. Figure 4.1 shows the kernel code used in Memsc for achieving the desired result using the previously mentioned functions.

4.3 Spawning Memsc Workers

Worker threads do the main part of the work in Memsc. At the end of the day, they are the ones executing the system calls code. However, deciding how to create these threads, manage their lifetime, and make sure they can access the appropriate data was not straightforward.

4.3.1 Challenges with Spawning Kernel Threads

As explained in section 3.4, Memsc worker threads are threads that service system calls for their associated user process by executing in the same context (i.e. being able to access the same memory) as that process. Nevertheless, these threads are expected to execute in kernel-space during their lifetime. An intuitive solution for this would be to spawn new kernel threads from within the kernel. However it turns out that this is not that easy.

On native Linux, a kernel thread is defined as a thread that only ever executes in kernel-space and is not supposed to touch user memory [16][2]. Kernel threads can be spawned by calling either *kernel_thread()* or *kthread_create()*.

kernel_thread() is simply a direct wrapper for *do_fork()* which is the function used to create all threads/processes, user and kernel. By passing the appropriate clone flags to *kernel_thread()* or *do_fork()*, the caller can control what parts of the parent thread's address space and process descriptor the new thread will share.

That seems convenient for creating Memsc workers and indeed it could work with older kernels. However, in newer kernels a user thread cannot create new kernel threads even when running in kernel-space (e.g. executing a system call) [10]. Only existing kernel threads can create new threads using *kernel_thread()*. Hence, it is not possible to call *kernel_thread()* from within a system call such as *memsc_register()* in our case.

The other function for creating kernel threads is *kthread_create()*. Even though *kthread_create()* can be called from a non-kernel thread (ie a user thread currently executing in kernel mode), it does not allow for the caller to specify any clone flags. Therefore, it is not possible to control exactly what is being shared between the calling and the resulting thread.

Additionally, to be able to successfully execute all system calls on behalf of another process, just sharing its address space would not be sufficient for a Memsc worker. It would as well need to share other information stored on the process descriptor such as open files and the file descriptors table. Otherwise, system calls such as *read*, *write*, and *close* that need such information would fail.

4.3.2 Spawning Kernel Threads from User-Space

Facing the aforementioned challenges lead to the creation of a kind of a hybrid solution for spawning kernel threads for Memsc; spawning them from user-space. This might sound counter-intuitive at first but it significantly simplifies things. It also is consistent with the way shared memory for the syspage is established with the allocation of the memory happening in user-space instead of kernel-space. It might actually not be completely accurate to refer to these threads as "kernel threads" following the "by the book" definition of what a kernel thread is, but section 4.3.3 explains how these threads run almost exclusively in kernel mode.

The *clone()* system call is the mechanism that the Linux kernel offers to user processes for creating new threads and processes. That is also what library functions such as *fork()* and *pthread_create()* invoke under the hood. As mentioned in the previous section, *clone()* accepts a number of flags that allow the caller to control what is shared between the parent and the child thread.

From within memsclib's *memsc_register()* library function, *clone()* is invoked with the following flags in order to spawn a new Memsc worker thread:

```
int clone_flags = (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND |
                   CLONE_THREAD | CLONE_UNTRACED | 0);
```

Setting the CLONE_VM flag ensures that the resulting thread runs in the same address space as the calling thread. CLONE_THREAD additionally makes sure that calling *getpid()* from either thread will return the same value. CLONE_FS makes the two threads share filesystem information such as the root directory and the current working directory while CLONE_FILES makes them share the same file descriptor table. Similarly, CLONE_SIGHAND ensures sharing of the signal handlers table. Finally, CLONE_UNTRACED doesn't allow the resulting thread to be traced using the *ptrace* system call. More detailed documentation of these flags can be found on [6].

The decoupling mechanism has been implemented such as that new threads spawned by a decoupled process are not decoupled themselves by default. They instead run on the same vCPU as the kernel-side context of the process and they are managed by the L^{4}Linux scheduler as all normal L^{4}Linux threads. That means that a Memsc worker does not start as a decoupled thread which is convenient in our case because otherwise we would need to migrate the worker back to the L^{4}Linux core manually.

4.3.3 Running in Kernel Mode (Almost) Forever

Once a Memsc worker thread has been created in user-space, it immediately issues a *memsc_register()* system call in order to enter kernel-space and be registered as the corresponding worker thread for that user process. Since the Memsc mechanism has not been setup at that point, this system call needs to be issued in the traditional way using IPC. However, for a process that wants to exclusively use Memsc for forwarding its system calls, this can be the first and last system call that is communicated to the L^{4}Linux kernel by generating an interrupt.

When the worker thread enters kernel mode it stays there forever processing new system calls. Worker threads never leave the kernel after entering it, except if they enter it with the wrong system call arguments. In case no error condition is detected by *memsc_register()*, a worker thread is only ever going to terminate its execution if it receives a fatal signal.

A fatal signal is one that would result in the process being killed. The default action for many signals for which the user process hasn't specified a handler is to kill the process. *SIGKILL* and *SIGSEGV* are examples of such signals. However, as hinted, the user process could choose to ignore those signals or install user-space handlers that perform other actions than killing the thread. There is one signal nevertheless, *SIGKILL*, which in user-space[3] cannot be blocked nor caught by a handler. Thus *SIGKILL* is guaranteed to always terminate the process.

Now, in multithreaded environments, a signal that is sent to a process as a whole will be delivered to just one thread which is chosen on runtime among the threads that are not blocking that signal. However, in the case of a fatal signal generated or sent to a multithreaded application, the kernel kills all of its threads and not just the thread to which the signal has been delivered [2]. That means that whenever the user application needs to die for any reason, either because it voluntarily called *exit* or a fatal signal has been delivered to it, the corresponding Memsc worker thread will die with it too.

For example, if another thread of the process generates a page fault by illegally trying to access memory that doesn't belong to it, a *SIGSEGV* signal will be delivered to the process. Unless a handler has been installed for it, the default fatal effect of this signal will be "propagated" to all threads of the process, including the Memsc worker thread, resulting in their termination. This greatly simplifies the kernel-space code of the worker which doesn't have to otherwise determine when it would need to cease its execution.

[3] in kernel-space even SIGKILL and SIGSTOP can be blocked

Figure 4.2 shows the loop that a Memsc worker thread constantly executes until its termination. Initially, the worker sets its state to *TASK_KILLABLE*. *TASK_KILLABLE* is like *TASK_UNINTERRUPTIBLE* with the difference being that the kernel will wake up the thread if there is a fatal signal for it pending (but not any other signal). Thus, the thread essentially goes to sleep by updating its state and yielding the scheduler using the *schedule()* function.

The thread then can only wake up if some other thread wakes it up or if it receives a fatal signal. In the latter case, *signal_pending()* will return true and the worker thread will attempt to return back to user-space. Upon returning to user-space the kernel will notice the pending signal and will kill the worker thread. In case no fatal signal is pending, the worker processes any new system calls it finds in the syspage and then starts a new loop iteration and goes back to sleep.

```
1  while (1) {
2    set_current_state(TASK_KILLABLE);
3    schedule();
4
5    if (signal_pending(current))
6      return 0;
7
8    /* process new system calls */
9    ...
10 }
```

Figure 4.2: The loop that a Memsc worker thread constantly executes throughout its lifetime.

It now becomes clear that the amount of time spent by a Memsc worker thread in user-space is truly minimal. To be specific, it is only the very few CPU instructions that it executes after its creation and before issuing the *memsc_register()* system call which triggers the transition to kernel-space. This is why we refer to these threads as kernel threads in this document, even if they do not fulfill the conventional definition of a kernel thread. The only other time these threads would execute in user-space is if *memsc_register()* failed prematurely, in which case they would return back to user-space with an error code.

4.4 Polling Memory Non-Stop

A big question associated with the scanner thread is whether it should be constantly polling memory or if it could possibly sleep for short periods of time. By

constantly polling memory it doesn't allow other threads to be scheduled in its vCPU, while by taking breaks it risks reacting too late to new system call requests. To answer this question it is important to understand how Linux manages time and for how long threads can sleep.

Linux keeps track of time using jiffies, the kernel unit of time. The jiffies monotonic counter maintained in the kernel is increased once with every tick of the timer. The frequency of the timer is configurable but a typical value is 100Hz. That means that jiffies is incremented 100 times every second. In a system with a 1GHz processor that equals 10 million CPU cycles per jiffy.

Using the *schedule_timeout()* function threads can voluntarily go to sleep until the specified time has elapsed. The scanner thread could potentially use this function to sleep for some time between different scans of the syspages in order to not monopolize its vCPU. However, the least amount of time a thread can sleep this way is 1 jiffy. And since 1 jiffy equals millions of CPU cycles a sleeping scanner thread could stall the progress of decoupled threads waiting for their system calls to be completed.

It becomes clear now that in order for Memsc to be fast the scanner thread cannot sleep. Instead, it should be polling memory non-stop. In order for this to happen it must declare itself as non-preemptible using the *preempt_disable()* function. This way it doesn't allow any other thread to be scheduled on its vCPU. On the other hand, hardware interrupts can still be delivered there.

The downside of this is that the scanner thread is now monopolizing its vCPU. Nothing else can be scheduled there for as long as the scanner is running and until it voluntarily quits because there are no more registered Memsc processes. That essentially means that L^{4}Linux needs to dedicate an entire vCPU just for the scanner's execution.

The whole system now needs at least 2 vCPUs in order to function. One for the scanner and one for the remaining kernel threads. However, this is necessary in order to detect newly posted system calls as fast as possible and is compensated by a forwarding mechanism that performs significantly better.

4.5 Synchronizing Accesses to the Syspage

Special care needs to be taken when accessing the syspage. That is because different CPU cores, the one executing the decoupled user thread and the ones executing the scanner and worker threads, are accessing the same shared locations in system memory. The cache coherency protocol of the architecture ensures that different

CPU cores won't see different values for the same address and that writes will eventually propagate to all cores. However, it provides us with no ordering guarantees whatsoever.

Memory ordering is very important because sometimes we need to ensure that writes to memory become globally visible in a certain order, while in other cases stores are not re-ordered before earlier loads. For example, when the decoupled thread creates a new system scall entry in the syspage, an appropriate memory barrier needs to ensure that the *status* field is really written last. Otherwise, the kernel could mistakenly pick incomplete system call entries to execute. Similarly, the kernel needs to make sure that the *status* field is updated last when it's done with processing an entry, or else the user-space might read an incorrect return value for the system call.

Another example is related to memory polling. Polling is achieved via busy waiting or spinning on a specific memory location until we notice a change in its content. However, an optimizing compiler, being unaware of the fact that a different thread can update that value, will most probably alias the contents of the memory location into a CPU register resulting in the variable not be reloaded from memory on each iteration. A software barrier (also called compiler barrier) is needed to combat this problem by forcing the compiler to produce code that really reads the value from memory each time. Fortunately, CPU-level memory barriers always imply software barriers.

The Linux kernel internally used to offer only standalone memory barrier functions such as *smp_rmb()*, *smp_wmb()* and *smp_mb()*. However, at a later point, support for atomic operations with acquire and release semantics has been added as well. These operations are on a par with the acquire/release semantics introduced by the C11 standard [5]. Acquire and release semantics can often match the underlying CPU architecture model better since modern ISAs usually do not offer plain load and store barrier instructions [17]. Also, it worths mentioning that on some ISAs such as ARMv8, using relaxed operations along with separate standalone barriers is less efficient than directly using an atomic operation with the appropriate memory model.

The special situation in Memsc is that accesses to shared memory need to be synchronized between a user thread and one or more kernel threads. Studying the implementation of acquire and release operations in both L^{4}Linux and gcc showed that the implementations are identical or compatible. Thus in the Memsc implementation, *atomic_load_explicit(memory_order_acquire)* operations in user-space are paired with *smp_store_release()* in kernel-space and respectively *atomic_store_explicit(memory_order_release)* operations in user-space are paired

with *smp_load_acquire()* operations inside the kernel.

4.6 Preventing Timing Attacks

Normally, user and kernel memory contents are isolated. User code is not supposed to access kernel memory except in rare cases. One such case is the vDSO library where user-space is allowed to (only) read certain kernel memory contents. Memsc is even more exceptional since the syspages are meant to be accessed for reading and writing by both the kernel and the decoupled user threads. Special care must be taken in this case in order to avoid introducing any security-critical bugs.

All user input corresponding to system call arguments is forwarded by Memsc to the system call service routines as is. These routines are supposed to validate the user data before doing anything with it. The worst thing that can happen in case a user thread starts writing arbitrary data to the syspage is the generation of invalid system call requests that the kernel will reject.

However, another type of attack is possible. Exploiting a race condition type of bug called time-of-check to time-of-use. As the name suggests this attack works by modifying something (memory contents in this case) between the checking of the state of a part of the system and the use of the results of that check.

Such a bug could be introduced in Memsc in this way:

```
if (entry->sysno < __NR_syscalls)
    sys_call_table[entry->sysno](...)
```

Memsc, before attempting to execute a system call, needs to verify that the given system call number really corresponds to an entry in the system call table. If the number is valid, it is used as an index in that table. What an attacker could do in this case is change the value of the system call number in the syspage right after this check and just before it is used for indexing in the table. If this attack is successful it can possibly lead to arbitrary code execution in the kernel.

To prevent this, Memsc copies the data from the syspage to the kernel stack before using them. This way no data can be changed in an unexpected way.

4.7 Modifications to the C Library

Allowing applications to transparently use Memsc required the modification of uClibc-ng, a small-sized C library compatible with glibc. uClibc-ng was chosen for its simplicity and its suitability for an embedded environment as the one used in

the Memsc evaluation. The modified version of this library is dynamically linked against memsclib on runtime, allowing the latter to be altered or replaced without requiring recompilation of the former.

uClibc-ng has a wrapper function for each supported system call. All these wrapper functions are automatically generated by the C preprocessor just before compilation using 2 parameterized C macros. One of these macros is for system calls that can return an error and the other one is for system calls that never fail. Modifying these 2 C macros is enough for enabling Memsc for all system calls.

The patched libc version inserts a *memsc_add() / memsc_wait_all()* pair in each system call wrapper. This way, issuing any system call makes the calling thread block its execution until the result is available. Then the return value of the system call is retrieved using *memsc_retval()* and is forwarded back to the application.

The C library must also take care of the Memsc registration for each process. In the case of uClibc-ng, the first function called by the Linux program loader is *__uClibc_main()*. This function performs any necessary initialization of the execution environment before passing control to the main function of the user program. Inserting the *memsc_register()* call there ensures that Memsc is enabled before any application code is executed.

The process for patching any other C library implementation such as glibc should be similar to what described above and should not require big efforts.

Chapter 5

Evaluation

In this chapter the methods for performing the experimental evaluation of this work are presented and the results are discussed. The evaluation was split into two parts. The first part consists of a custom microbenchmark that uses the memsclib library directly. The second part consists of known programs making use of the new forwarding mechanism transparently via the modified C library.

5.1 System and Configuration

All measurements presented in this chapter were taken on a Raspberry Pi 3 Model B which uses a Broadcom BCM2837 system on a chip (SoC) with a 1.2 GHz 64-bit quad-core ARM Cortex-A53. The board's CPU contains 32KB of L1 cache (split into 16KB i-cache and 16KB d-cache), 512KB of L2 cache and the SoC is linked to a 1GB LPDDR2 memory module. Of this memory, 450MB was allocated to the para-virtualized L^{4}Linux instance.

Memsc was implemented in the L^{4}Linux kernel version 5.6.0 which for the purposes of this evaluation ran on top of the L4Re version 20.07. Due to incomplete support of Fiasco's decoupling mechanism for the ARM 64-bit architecture, the 32-bit version of all the aforementioned software was used. The CPU also ran in its 32-bit mode of operation.

By default the Raspberry Pi performs dynamic CPU frequency scaling. This means that the CPU frequency is automatically adjusted on the fly depending on the system workload in order to conserve power and reduce the amount of heat generated by the chip. Since this behavior can affect measurements, it was completely disabled. Instead, the system was setup to always operate at the maximum

frequency of 1.2GHz. Disabling the automatic CPU frequency scaling was achieved by setting the *force_turbo* option on Raspberry's *config.txt* configuration file.

Buildroot [3] was used for generating a small root filesystem containing only the necessary system binaries and libraries, benchmarking code, and the modified C library based on the uclibc-ng version 1.0.32. No periodic jobs or unnecessary services ran during benchmarking. The root file system of type rootfs[1] was loaded in memory and was not backed by a block device.

For measuring the passage of time, the ARM architecture provides a generic timer and a set of counters. Among these, there is a 64-bit per-core virtual timer whose value can be obtained by reading the contents of the CNTVCT counter register. This counter is incremented with a constant frequency determined by the value stored in the CNTFRQ register (19.2MHz by default on the Raspberry Pi 3). Fiasco is configured to allow user-level code to access these registers. The custom benchmark programs used the virtual timer in order to measure execution time.

To avoid measurement noise and wasted cycles caused by process migration to different cores, the benchmark processes were pinned on a single CPU core during their execution. This was achieved by setting the process affinity using the *taskset* utility. The first physical CPU was used in the case of native Linux and the first vCPU in the case of L^{4}Linux. The L4 vCPU threads are by default non-migratable, therefore they always execute on the same physical core as well.

For the L^{4}Linux experiments, two vCPUs were made available to L^{4}Linux always executing on the 3rd and 4th physical core correspondingly. When decoupling was used, the benchmark application was decoupled to the 1st physical core. That is, in the decoupling test cases the benchmark ran as a decoupled thread on the 1st core, forwarding its system calls to the 3rd core (hosting the 1st vCPU) of the Raspberry. In the Memsc case, the 4th core (hosting the 2nd vCPU) was used for running the scanner thread. This setup is identical to the one depicted in Figure 3.2 with the only difference being that the 2 vCPUs were swapped.

All values reported in this evaluation represent the average of 5 separate runs. The standard deviations of these runs were calculated and found to be low; less than 5% of the mean value for all experiments and less than 1% of the mean value for most file system tests.

[1] rootfs is a special instance of tmpfs, an in-memory file system

5.2 Custom Microbenchmarks Using Memsclib

For the first part of this experimental evaluation a simple microbenchmark that executes a number of getppid system calls in a loop and then exits was used. This custom program explicitly makes use of the Memsc interfaces by linking against memsclib directly, making it possible to use the asynchronous execution capabilities of Memsc as well. The getppid system call was chosen because it requires a minimal amount of work in the kernel allowing the focus to be on the round trip times of the forwarding mechanism (whenever cross-processor forwarding was required).

First the execution time using blocking system calls was measured, followed by experiments using batched execution of system calls. Finally, the system throughput with multiple decoupled threads running in parallel was computed.

5.2.1 Synchronous System Calls

For the first experiment, the benchmark was run for an increasing number of system calls. The program had to block and wait after issuing each system call until it was processed by the kernel and the result was ready. Performance was compared among native Linux, L^{4}Linux without decoupling, L^{4}Linux with decoupling, and Memsc-patched L^{4}Linux with decoupling.

Table 5.1 and Figure 5.1 present the results of these tests. The "dec" column indicates whether or not the decoupling mechanism was used in the corresponding experiment. As expected, L^{4}Linux being virtualized is always slower than native Linux. We also confirm that when the decoupling mechanism is used, there is a very significant performance overhead due to the cross-processor forwarding of system calls.

Blocking syscalls		Number of system calls					
OS kernel	dec	1	10	100	1,000	10,000	100,000
Linux	no	4	7	39	307	2,938	25,618
L^{4}Linux	no	10	41	348	3,423	34,260	342,381
L^{4}Linux	yes	36	278	2,701	26,949	269,664	2,705,248
L^{4}Linux-Memsc	yes	27	143	1,205	11,702	116,112	1,163,886

Table 5.1: Times (in microseconds) for completing a fixed number of blocking system calls per kernel and thread execution mode

What we are mostly interested in seeing is whether or not directly forwarding system calls using shared memory is faster than forwarding them via cross-

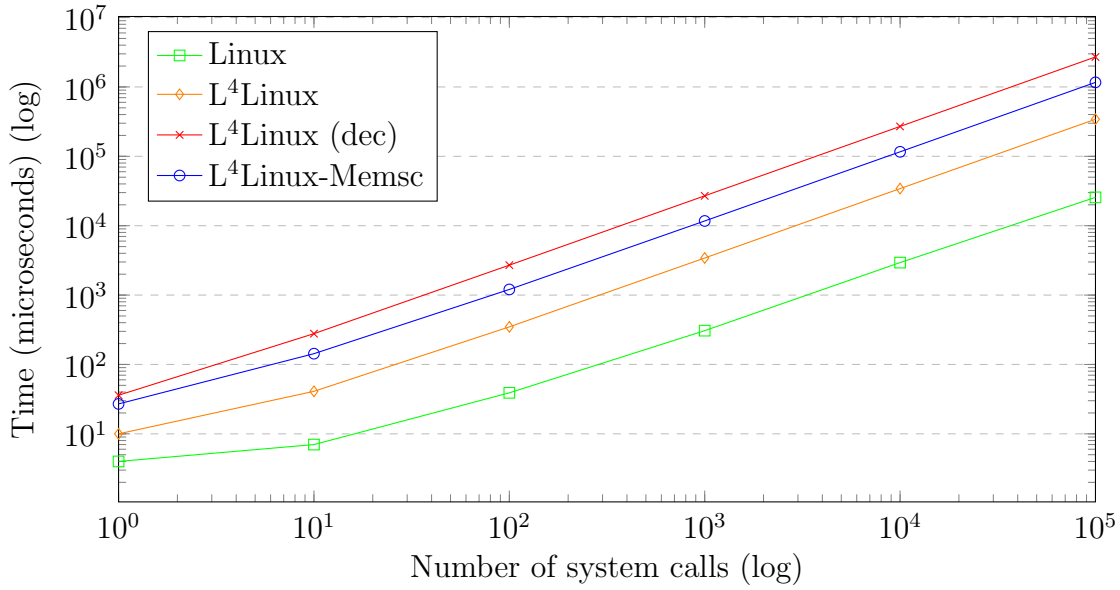


Figure 5.1: Comparison of blocking system calls performance among different configurations

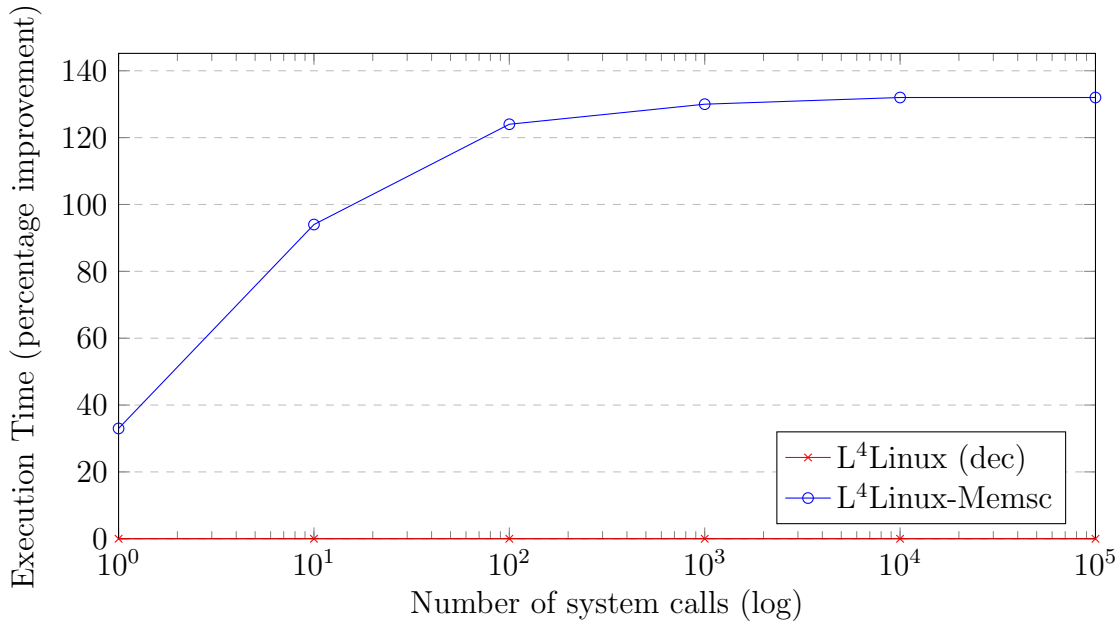


Figure 5.2: Execution time percentage improvement of L^{4}Linux-Memsc versus standard L^{4}Linux when using blocking system calls

processor IPC. We indeed observe that Memsc-patched L^{4}Linux performs significantly better than standard L^{4}Linux with decoupling for any number of system calls. In fact, for more than 10 system calls the execution is more than twice as fast.

Figure 5.2 shows the execution time percentage improvement[2] of Memsc-enabled L^{4}Linux over standard L^{4}Linux. What we can see in this diagram is that for more than 100 system calls there is a solid performance improvement of more than 120% which goes up to 133% for larger amounts of system calls.

5.2.2 Asynchronous System Calls and Batching

The previous section discussed the execution times for issuing blocking system calls. This section focuses on asynchronous system calls and the batched execution mode that the Memsc mechanism enables. In order to evaluate the batching capabilities of Memsc, the getppid benchmark was run with 100,000 iterations and was repeated for different batch sizes.

At this point, it is worth reminding the reader that when using Memsc the system calls are processed by the kernel as soon as possible after being posted to the syspage. Furthermore, the system call entries are posted to memory individually by the *memsc_add* library call, instead of been posted all together when a *memsc_wait_all* call is made. That means that by the moment the user program calls the *memsc_wait_all* function, some or all of the system calls posted might have already been completed.

The term batching is used a bit loosely here. It is about allowing the user-space to make progress and not have to block after each and every system call it issues. With the term "batch size" we refer to the number of system call entries that have been posted between two subsequent *memsc_wait_all* calls. However when exactly the kernel will pick up these entries for execution depends entirely on scheduling activity. The *memsc_wait_all* call serves as a barrier that doesn't allow the program to proceed until all posted system calls have been processed.

Having this explanation in mind, we can look at the obtained results displayed in Table 5.2 and Figure 5.3. A batch size of 1 is equivalent to synchronous execution as in the experiment of the previous section. The maximum batch size is 64, as this is the maximum amount of system call entries that fit in a 4K syspage at once

[2]Calculating percentage improvement for elapsed time where smaller is better can be confusing. The formula used for calculating the percentage is (old-new)/new*100%. In this case a percentage improvement of 100% means that the benchmark program ran twice as fast.

since each entry occupies 64 bytes.

Batch Size	1	2	3	4	8	16	32	64
Exec Time	1,163,886	1,195,358	855,631	300,568	195,536	114,533	82,314	76,956

Table 5.2: Times (in microseconds) for executing 100,000 system calls in batched mode for different batch sizes

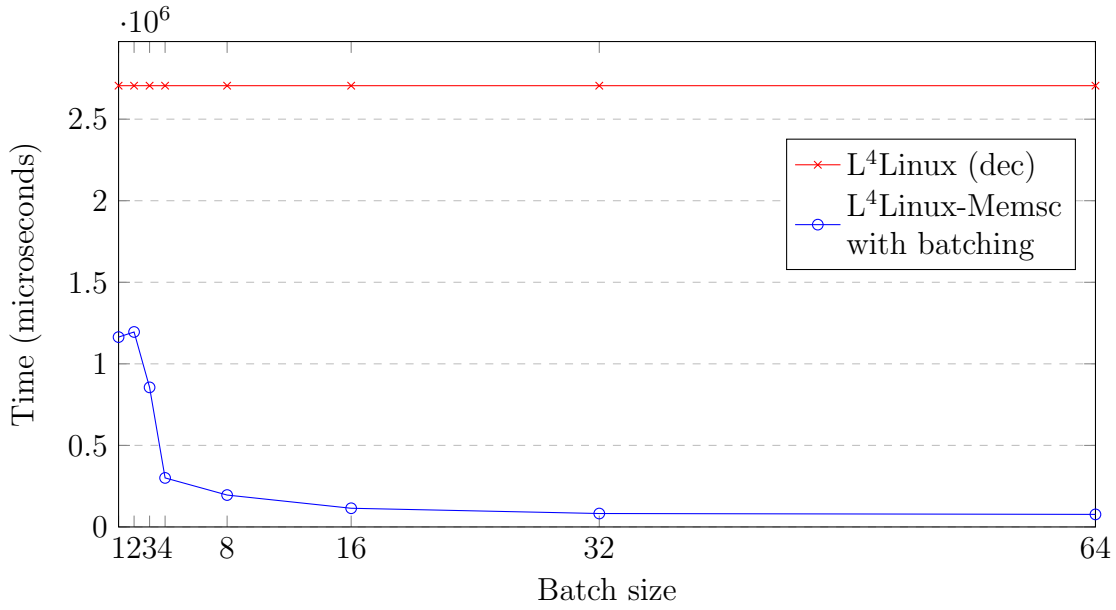


Figure 5.3: Comparison of batched system calls performance against standard L^{4}Linux for different batch sizes

Looking at these results the first thing we notice is that by executing the system calls in batches of 2 might not result in performance improvement. Instead, it might actually slightly worsen the execution time. This possibly happens due to unfavorable scheduling combinations between the decoupled thread and its Memsc worker.

However, with batches of 3 and 4 there is quite a large reduction of the execution time. For batches of larger sizes, execution time continues to decrease but the reduction is not as significant. Table 5.3 summarizes the percentage decrease in the execution time of the benchmark when compared to standard L^{4}Linux. We notice that by doing batches of 4 system calls there is already a reduction of almost 90% of the original execution time and the program actually even runs faster compared to when running in non-decoupled mode (Table 5.1). The time reduction percentage can go up to 97.2% for the maximum available batch size.

These results indicate that decoupled applications can significantly increase their performance by doing system calls in small batches of 3 or 4. This is an important observation since it is much easier for applications to prepare small batches consisting of a few system calls as opposed to batches of tens of system calls.

Batch Size	1	2	3	4	8	16	32	64
Exec Time	-57.0%	-55.8%	-68.4%	-88.9%	-92.8%	-95.8%	-97.0%	-97.2%

Table 5.3: Percentage decrease of execution time for different batch sizes over standard L^{4}Linux in decoupled mode

This experiment was repeated with system calls other than getppid and the results were found to be proportionally similar.

5.2.3 System Throughput

Next, the system call throughput of the system was evaluated. In this experiment 2 applications (2 instances of the microbenchmark) ran decoupled in parallel in the 1st and 2nd physical CPU correspondingly. The purpose of this experiment was to see how many system calls per second the L^{4}Linux kernel can handle with and without the Memsc patch.

To ensure that the applications started their execution roughly at the same time, an intermediate shell script was used. This short script works by simply sending a SIGSTOP signal to itself in order to transition to a sleep state. When it wakes up, it uses the *exec* command to load the benchmarking program on its address space and execute it. After launching both applications this way, the SIGCONT signal was sent to both of them with a single *killall* command in order to start their execution.

In order to measure the throughput, L^{4}Linux was modified and an atomic counter for counting all system call requests from all system processes was added. A new system call that returns the value of this counter was added as well. Using this system call, the main benchmarking program was able to calculate the system throughput by reading the atomic counter before and after executing the getppid programs. The throughput was measured for different time intervals (5, 10, 20, and 30 seconds) and the results were found to be consistent.

The results collected are presented in Table 5.4 and Table 5.5. Additionally, Figure 5.4 serves as a visual representation of these numbers. The system call throughput was calculated for standard L^{4}Linux with and without decoupling,

for L^{4}Linux-Memsc with synchronous system calls, and for L^{4}Linux-Memsc with asynchronous system calls and different batch sizes.

OS kernel	dec	Throughput
L^{4}Linux	no	290,341
L^{4}Linux	yes	37,455
L^{4}Linux-Memsc	yes	133,397

Table 5.4: Throughput (system calls per second) per kernel and execution mode

Batch Size	2	3	4	8	16	32	64
Throughput	136,260	197,806	485,253	603,365	1,036,953	1,600,792	2,205,457

Table 5.5: Throughput (system calls per second) per batch size for L^{4}Linux-Memsc

As with all previous experiments' results, we notice that Memsc with decoupling performs better than standard L^{4}Linux with decoupling. Specifically, the system call throughput is increased by 256% in this case.

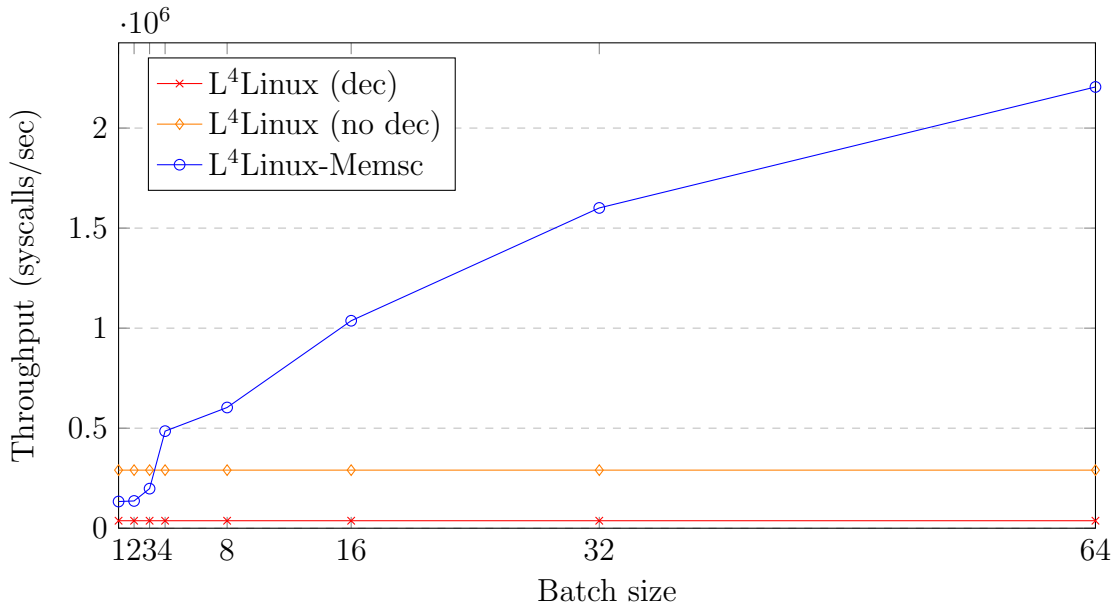


Figure 5.4: Comparison of system call throughput between L^{4}Linux with and without decoupling and L^{4}Linux-Memsc for different batch sizes

Additionally, from the Memsc results in batched mode, we notice that this time batches of larger sizes have a bigger impact on performance. This is because the

processes are allowed to run for more time and the individual improvements of each process are accumulated. For batches of 4, the system throughput is already higher than when standard L^{4}Linux without decoupling is used. In comparison to standard L^{4}Linux with decoupling, the throughput was increased by 1196% for batches of 4 and by 5788% for batches of 64.

This experiment was repeated for a larger number (4, 6, and 8) of decoupled threads equally divided between the two cores. However due to the limited physical resources, at any time only two out of these threads were able to execute in parallel. The system throughput seemed to increase slightly with more decoupled threads.

5.3 General-Purpose Benchmarks Using Uclibc-Memsc

The second part of the evaluation was done using the standard Unix dd and ls programs. The busybox [4] implementation of these utilities was used, as it was included in the buildroot image. The whole busybox binary was linked dynamically against uclibc-memsc, the modified C library, in order to assess its execution using Memsc. No modification of the binary was required.

Since the root file system was not backed by a block device, all file operations issued by these benchmarks were performed directly in memory. Measuring execution time for these benchmarks was done using the time program.

5.3.1 File Copying

The dd program was used for copying files of different sizes and determining whether file system operations are faster with Memsc. The program was called with bs=512 which is the default value. The bs parameter of dd specifies the number of bytes up to which dd reads and writes at a time. So one read and one write system call were issued for copying each chunk of 512 bytes from the original file to the new file.

The original file was created as a sparse file using the truncate command. Even though the original sparse file occupied no storage, the dd command had to allocate file system space (in this case memory) for the new file. The benchmark was repeated for different file sizes. The maximum file size in this benchmark was limited by the memory allocated to L^{4}Linux which was in turn limited by the amount of physical system memory.

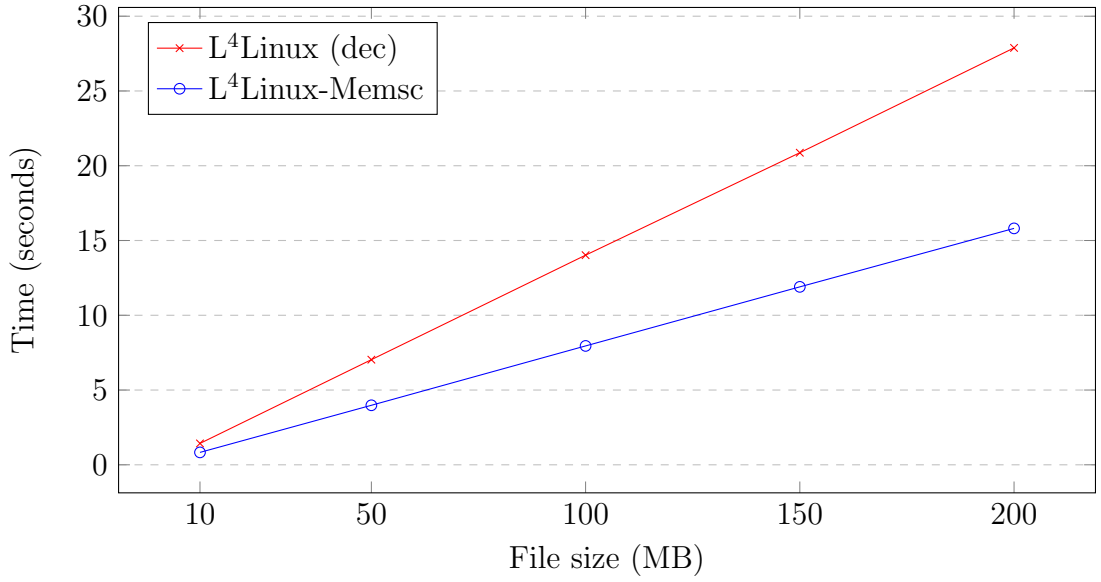


Figure 5.5: Execution times for the dd benchmark with and without Memsc

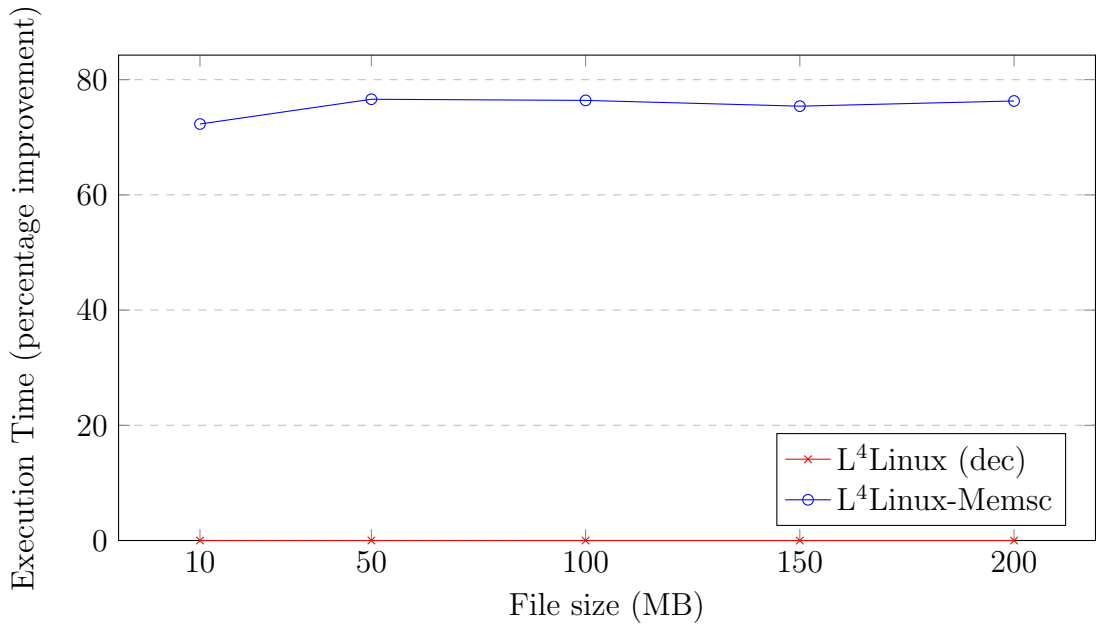


Figure 5.6: Execution time percentage improvement for the dd benchmark with Memsc compared to standard L4Linux with decoupling

It seems that even copying a sparse file in Linux populates its page cache with zero-filled pages. This results in subsequent copies of the same file being faster than the first one. For this reason, the original file was deleted and created again between each test in order to ensure a cold page cache.

As shown in Figure 5.5, Memsc outperformed standard L^{4}Linux for all file sizes. Figure 5.6 shows the percentage improvement over standard L^{4}Linux with decoupling. This remains pretty consistent independently of the file size and is around 76%.

5.3.2 Directory Listing

For the directory listing benchmark, ls was run in directories with different amounts of files. The ls program was run with the -l option in order to force it to issue a stat system call for each file listed. Its output was redirected to a file in order to avoid any overheads related to the terminal emulator, line buffering, and text rendering.

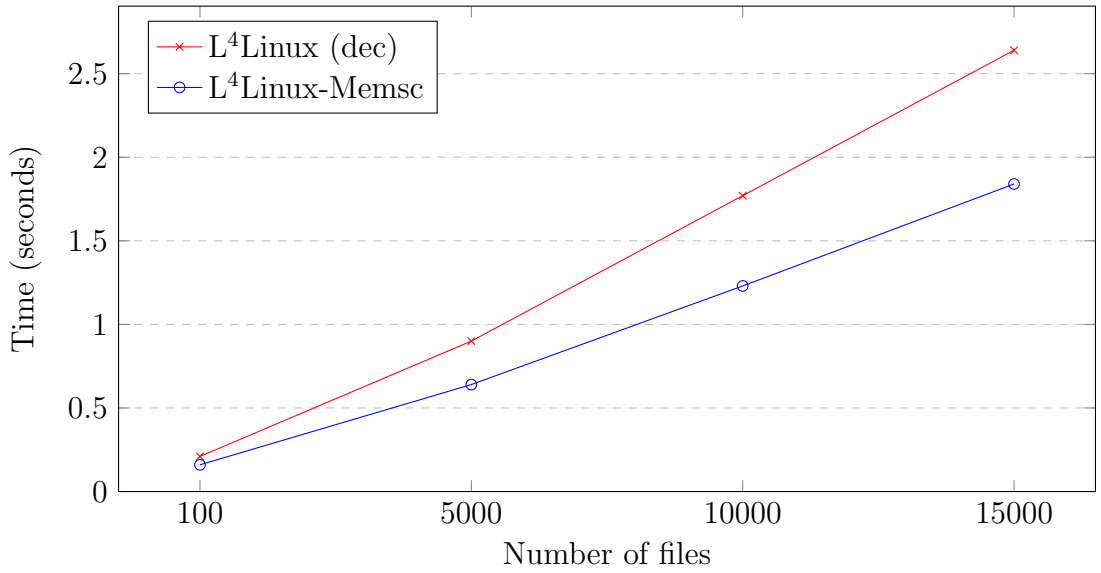


Figure 5.7: Execution times for the ls -l benchmark with and without Memsc

The resulting execution times are presented in Figure 5.7 as a graph. Once again, Memsc performed better for all test cases with the benefits increasing for a larger number of files. Figure 5.8 depicts execution time percentage improvements. Memsc shows an improvement of 31% over standard L^{4}Linux for listing 100 files and an improvement of 43% for listing 15000 files.

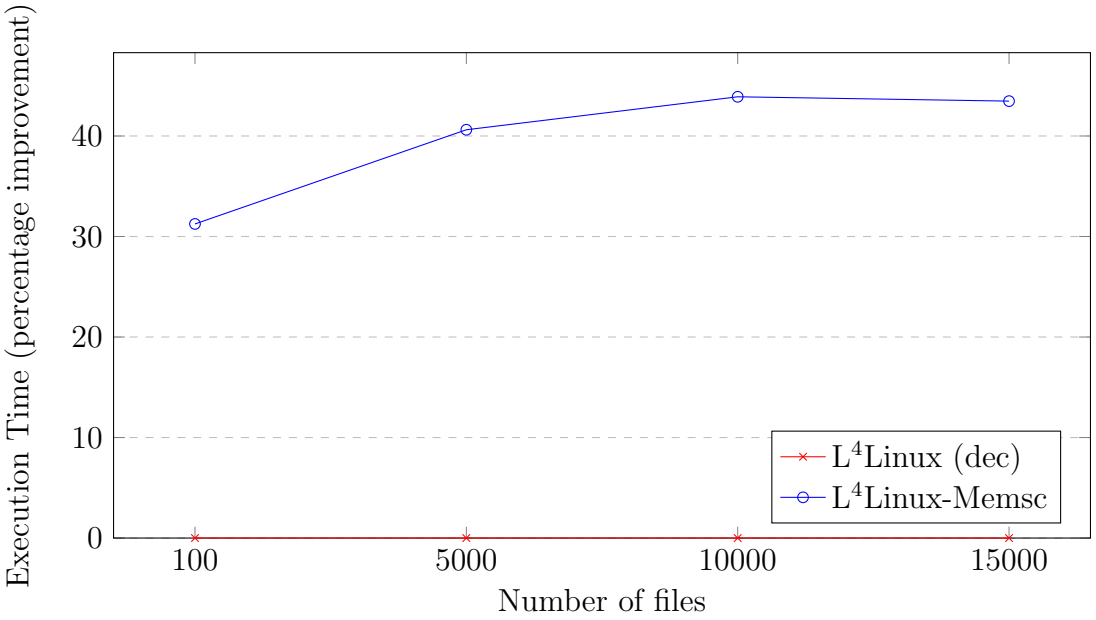


Figure 5.8: Execution time percentage improvement for the ls -l benchmark with Memsc compared to standard L^{4}Linux with decoupling

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This work introduced Memsc, a system call forwarding mechanism. Memsc uses shared memory in order to forward system calls from a decoupled L^{4}Linux thread running on a CPU core, to the L^{4}Linux kernel running on a separate core.

With Memsc, the L^{4}Linux kernel performs memory polling on a set of special memory pages that registered processes use for posting system calls. In contrast, the existing forwarding mechanism of L^{4}Linux uses standard L4 IPC which works by generating inter-processor interrupts in order to achieve this cross-processor communication.

Memsc can coexist with the traditional forwarding mechanism and it can be used by applications either directly or indirectly. When used indirectly, applications simply link against a modified C library version while no modification or recompilation of the original program is required. When used directly, applications use the Memsc user-space library which further allows them to use asynchronous system calls and batching capabilities.

Asynchronous system calls free applications from having to block after issuing each and every system call. Instead, they can proceed with doing other work after posting a system call and then return to check its result at a later point in time. Moreover, by grouping and submitting multiple system calls together, applications can further speed up their execution and minimize the time they spend being blocked.

The experimental evaluation of Memsc showed that it always performs better than the existing L^{4}Linux forwarding mechanism. The total system call through-

put of the system was increased by 256% while executing a custom getppid microbenchmark. General-purpose benchmarks using the dd and ls programs saw a performance improvement of 76% and 43% correspondingly.

The performance improvement is even more significant when asynchronous system calls and batching are used. Specifically, for the same getppid benchmark the system call throughput was increased by 1196% for constantly executing system calls in batches of 4, and by 5788% when using the maximum batch size of 64.

6.2 Future Work

Future work and research should focus on determining how well Memsc scales. Unfortunately, the available hardware for conducting the experimental evaluation was of limited physical resources. Further tests should be run in an environment with a larger amount of CPU cores and multiple decoupled processes running in parallel.

Additionally, it is worth investigating parallel execution of system calls. Even though Memsc can currently execute system calls of different processes at the same time, it cannot execute different system calls of the same process in parallel. In order to achieve this, multiple Memsc worker threads per process would need to be employed.

Finally, multi-threading support could be added to the user-space memsclib library. Achieving this should not be too complicated. In particular, the *memsc_add()* function is the only one not being thread-safe currently. This could be fixed by introducing a single lock in user-space to protect the syspage from concurrent insertions of new entries. The challenging part would be implementing proper thread cancellation and ensuring that no unneeded entries are left behind.

Bibliography

[1] Francisco Ballesteros, Ricardo Jimenez-Peris, Marta Patiño-Martínez, Fabio Kon, Sergio Arévalo, and Roy Campbell. "Using interpreted CompositeCalls to improve operating system services". In: *Softw., Pract. Exper.* 30 (May 2000), pp. 589–615. DOI: 10.1002/(SICI)1097-024X(200005)30:63.3.CO;2-B.

[2] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN: 0596005652.

[3] *Buildroot*. URL: https://buildroot.org.

[4] *Busybox*. URL: https://www.busybox.net/.

[5] *C11 - Atomic operations library*. URL: https://en.cppreference.com/w/c/atomic.

[6] *clone(2) manual - version 5.07 of the Linux man-pages project*.

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. 3rd ed. O'Reilly Media, 2005. Chap. 15.

[8] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. "The Performance of μ-Kernel-Based Systems". In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP '97. Saint Malo, France: Association for Computing Machinery, 1997, 66–77.

[9] Mati Karner. *Understanding noise introduced by operating systems and its effects on high performance computing*. URL: https://pdfs.semanticscholar.org/64f8/bf2735d7ab4da08470e364ed5f41b08d7b68.pdf.

[10] "kernel_thread() causes segfault". In: (2016). Discussion thread from the Linux kernel newbies mailing list. URL: https://www.spinics.net/lists/newbies/msg57445.html.

[11] Adam Lackorzynski. "L^{4}Linux on L4Env". MA thesis. Technical University of Dresden, 2002.

[12] Adam Lackorzynski, Alexander Warg, and Michael Peter. "Generic Virtualization with Virtual Processors". In: *Proceedings of Twelfth Real-Time Linux Workshop* (2010).

[13] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. "Decoupled: Low-Effort Noise-Free Execution on Commodity Systems". In: June 2016, pp. 1–8. DOI: 10.1145/2931088.2931095.

[14] J. Liedtke. "On Micro-Kernel Construction". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: Association for Computing Machinery, 1995, 237–250.

[15] *Linux Kernel Memory Barriers*. Part of linux-5.6.15 documentation. File: Documentation/memory-barriers.txt.

[16] Robert Love. *Linux Kernel Development*. 3rd. Addison-Wesley Professional, 2010. ISBN: 0672329468.

[17] Paul E. McKenney. *Memory Barriers: a Hardware View for Software Hackers*. Tech. rep. Linux Technology Center, IBM Beaverton, July 2010.

[18] Amit Purohit, Joseph Spadavecchia, Charles P. Wright, and Erez Zadok. "Improving Application Performance Through System Call Composition". In: 2003.

[19] M. Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. "System Call Clustering: A Profile-Directed Optimization Technique". In: 2005.

[20] Mohan Rajagopalan, Saumya Debray, Matti Hiltunen, and Richard Schlichting. "Cassyopia: Compiler Assisted System Optimization." In: *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS)* (2003).

[21] S. Seelam, L. Fong, J. Lewars, J. Divirgilio, B. F. Veale, and K. Gildea. "Characterization of System Services and Their Performance Impact in Multicore Nodes". In: *2011 IEEE International Parallel Distributed Processing Symposium*. 2011.

[22] S. Seelam, L. Fong, A. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea. "Extreme scale computing: Modeling the impact of system noise in multicore clustered systems". In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010.

[23] Livio Soares and Michael Stumm. "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls". In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010).

[24] *The Fiasco microkernel*. URL: http://os.inf.tu-dresden.de/fiasco/.

[25] Josh Triplett, Philip W. Howard, Eric Wheeler, and Jonathan Walpole. *Avoiding system call overhead via dedicated user and kernel CPUs*. Tech. rep. Portland State University. URL: http://web.cecs.pdx.edu/~walpole/papers/osdi2010paper.pdf.

[26] *vdso(7) manual - version 5.07 of the Linux man-pages project*.