ADDING SMP SUPPORT TO FASTPATHS IN AN L4 MICROKERNEL

Petre Eftime[1], Lucian Mogosanu[2], Mihai Carabas[3], Laura Gheorghe[4], and Razvan Deaconescu[5]

[1]*University POLITEHNICA of Bucharest, Romania, petre.eftime@cti.pub.ro*
[2]*University POLITEHNICA of Bucharest, Romania, lucian.mogosanu@cs.pub.ro*
[3]*University POLITEHNICA of Bucharest, Romania, mihai.carabas@cs.pub.ro*
[4]*University POLITEHNICA of Bucharest, Romania, laura.gheorghe@cs.pub.ro*
[5]*University POLITEHNICA of Bucharest, Romania, razvan.deaconescu@cs.pub.ro*

ABSTRACT. Fastpaths are a method of optimization which relies on treating the most commonly executed cases of certain functions in a privileged manner, such that behaviour is not modified, but execution time is reduced. Fastpaths play an important role on improving paravirtualization performance offered by an L4 microkernel. In this article we redesign two existing fastpaths in an L4 microkernel for the purpose of adding SMP support. We then put these fastpaths through a series of regression and performance tests to determine if the design is correct and what performance benefits we can expect by using them on a multiprocessor system.

Keywords: L4 kernel, ARM, SMP, fastpath

INTRODUCTION

Operating systems is one of the areas where performance can be important to usability. Ideally, the operating system kernel should perform all operation without adding any overhead, but in practice this is not possible. However, performance can be improved in certain scenarios through the use of fastpaths, i.e. optimizations which rely on various assumptions regarding the system's state in order to speed up execution time.

The microkernel we improve belongs to the L4 family of microkernels, based around the ideas of Jochen Liedtke. L4 microkernels were built from the ground up to have good performance through careful design of the Application Programming Interface (API) and implementing it using the best available algorithms for the targeted systems (Liedtke, 1994) (Liedtke, 1995).

However, even the best API and algorithms can be improved by optimizing the code for various architectures or use cases, or, as is our case, to optimize for certain system states without having to reduce abstraction (Elphinstone & et al., 2007). Optimizations can reduce code readability and increase maintenance cost, so they must add a tangible benefit to performance in order for them to be implemented or even kept in the kernel. They must also not harm system security or behavior (Klein et al., 2014).

In this paper we discuss two optimizations available in our L4 based microkernel, which we update to work on Symmetric multiprocessing (SMP) systems, namely the system call exception fastpath and the Inter-process communication (IPC) system call fastpath. Section

Related Work explains what fastpaths are and why these two were chosen for improvement. Their functionality and design is explained in Section Design and Implementation. In Section Evaluation the benefits brought by these optimizations are measured. The last section draws conclusions on the importance and maintability of these fastpaths, as well describing some future work which could be done for further improving them.

RELATED WORK

Optimization refers to the process of improving execution time of a certain functionality without affecting the observed behaviour of that functionality. Optimizations can come at multiple levels: algorithmic (an algorithm is either replaced or tailored to better suit its use cases), instruction set (a better set is used to implement the same algorithm), architectural (hardware specific functionality is added or simply employed).

Optimization must be carefully employed, unless some tool (e.g. a compiler) does this optimization automatically, since it always comes at the cost of either code maintainability, deployability or even worse, correctness (Dannowski, 2007). Therefore, it is generally preferred to start optimization at high levels of abstraction and only use architectural improvements when it is required or if they offer significant advantages.

Fastpaths are an optimization method which rely on treating one or more commonly executed code paths in an special manner, such that these common cases take less time to complete. If the execution could lead to fewer common cases or errors the normal path, or slowpath, is taken instead. While the goal is to reduce execution time by having these common cases execute as fast as possible, fastpaths do add overhead when the less common cases must be executed, falling back to the slowpath. With or without the fastpath, the execution must have identical (observed) behavior (Blackham & Heiser, 2012).

Fastpaths are usually employed in applications where performance is required, such as networking, operating system kernels and servers. Kernels in the L4 family include a fastpath for IPC system calls, and try to avoid doing IPC syscalls entirely if possible, since IPCs can have relatively complex code associated with them and are the main mechanism for communication and synchronization between processes and even threads. Indeed, (Liedtke, 1994), the father of the L4 microkernels suggested that IPC performance is very important and built L4 around this idea.

Previously, our L4 microkernel had fastpaths for both system call exceptions (necessary for paravirtualization) and IPCs, written in assembly language and later translated to C/C++ for the purpose of portability. The performance improvements brought by these fastpaths were rather promising and because they were written in a high level programming language, the cost of updating them to support SMP was thought to be minimal.

DESIGN AND IMPLEMENTATION

The purpose of fastpaths are reducing execution time, by optimising the most common case or cases. After determining these cases, the fastpath will implement one or more of them, depending on their complexity. The fastpath will start with a series of tests to determine whether it's currently in one of the selected cases, otherwise it will fall back to the slowpath.

Fastpaths have been used before in L4 microkernels, but our implementation is on SMP, which brings a few complications with it: one cannot make assumptions about the state of other threads, since they could be currently running on another CPU and their state might change. Locking these threads is essential to correctness, and the scheduler must be involved in making the decision of what thread will run next and on what CPU. Locking is fine grained in our L4 microkernel: there are a few global locks (e.g. scheduler, thread list), but most of

the locks are per object (e.g. thread, space). The locks in the kernel are basic spinlocks and ticketlocks, making deadlocks a possiblity if the threads are not acquired in a certain order: global locks first, then object locks in the order of their address.

A brief explanation of what the fastpaths already available in the microkernel is in order. In the case of our L4 microkernel system calls (and other exceptions) can be redirected to a userspace thread as IPCs, a mechanism which is required for paravirtualization. If the thread which processes and serves these system calls is waiting for an IPC and is idle (which should be the common case), then the microkernel can send the IPC to the exception handler directly, without queueing the exception and waiting for that thread to become available, since it already is. This is done by copying the necessary information and setting the exception handler thread as running in place of the thread that raised the exception. The IPC fastpath deals with a similar case, for an explicit IPC system call made by a user thread and with some additional verifications and complexity, since IPCs are a general purpose mechanism and can be used for multiple purposes such as synchronization or communication, but the core idea remains the same.

The non-SMP implementations were written in C/C++ as modern compilers are good at generating optimized assembly code, especially with some aid from the programmer (Blackham & Heiser, 2012). This made updating to SMP cost-efficient, but because locking can be an expensive process, special care was required. Test cases which do not require a lock, or require a single lock, were prioritized, as failing faster reduces overhead when having to enter the slowpath. Releasing and reacquiring locks was avoided, since fastpath executes quickly and would not cause other threads to wait, but other threads might cause the fastpath to wait (multiple times) if contention were to occur. Another reason to avoid reacquiring locks is that another thread might modify the state of the exception handler thread, which would have required additional tests, and thus would add undesirable complexity to the fastpath. Finally, locks in the microkernel have to be acquired in a defined order (e.g. scheduler lock before thread locks, thread locks in order of their address), to prevent deadlocks, which forced some reordering of the tests. Some additional fixes and checks were also added during this process, to guarantee safe operations. Even though this did not seem to be an issue previously, SMP systems are more susceptible to bugs because of increased complexity.

EVALUATION

In this section, we assess the correctness of fastpath using two kind of unit tests: ktest at the microkernel level and Linux Test Project (LTP) at the paravirtualized Linux kernel level. Than we measure the performance gain obtained by fastpath mechanism. The SMP-enabled fastpath must perform the same as the slowpath in terms of behavior (Blackham & Heiser, 2012), but should result in better performance. The test platform used was a Pandaboard containing an OMAP4430 System on a chip (SoC) with an ARMv7 Cortex A9 dual-core CPU and 1 GB of RAM. The version of Linux used was a paravirtualized Linux 3.4.0 with a root filesystem with Busybox generated by Buildroot.

ktest is a test suite used for testing the correct implementation of the system call interface and mechanisms of our L4 microkernel (Mogosanu, 2013). This is the first step towards validating correctness, as it tests the microkernel mechanisms directly, making sure the implementation conforms to the specifications, but it is not necessarily a stress-test and might not uncover all the possible SMP issues such as race conditions or deadlocks, even though it can be useful in detecting at least some of these issues (Condurache & Eftime, 2014). Linux Test Project (*Linux Test Project*, n.d.) is a tool used to verify the Linux kernel and libraries (as ktest is used for our L4 microkernel) for conformance. This is considered a standard test tool and covers many features of the Linux kernel and standard libraries. Since one of the uses of

important uses of our L4 microkernel is paravirtualization and since it is a complex test scenario, LTP was used to verify the fastpaths, as used by Linux, and the same tests passed with and without the fastpaths activated.

Profiling the fastpaths under paravirtualized Linux

To check the performance improvements and potential issues that the fastpaths have, measuring cycles (or a few other performance parameters) between entry and exit was necessary. A mechanism for such measurements already existed in the microkernel, under the name of Performance Management Unit, but it required modification of either the microkernel or userspace programs. Modifying the userspace was difficult in the case of Linux, since the results had to be printed or memorized after each system call. We opted to add a new mechanism to measure IPCs and syscalls exceptions (which is a type of IPC sent by the microkernel).

At the points of entry in the microkernel for the two fastpaths and slowpaths a cycle counter was started and at the points of exit, the cycle counter was logged in the microkernel's tracebuffers, alongside the point of exit from the fastpath (the tracebuffers are a mechanism in the microkernel used for storing messages directly in physical memory). This allowed a direct comparison between the fastpath-enabled and normal microkernel in terms of cycles spent treating an IPC or an syscall exception, with minimal overhead.

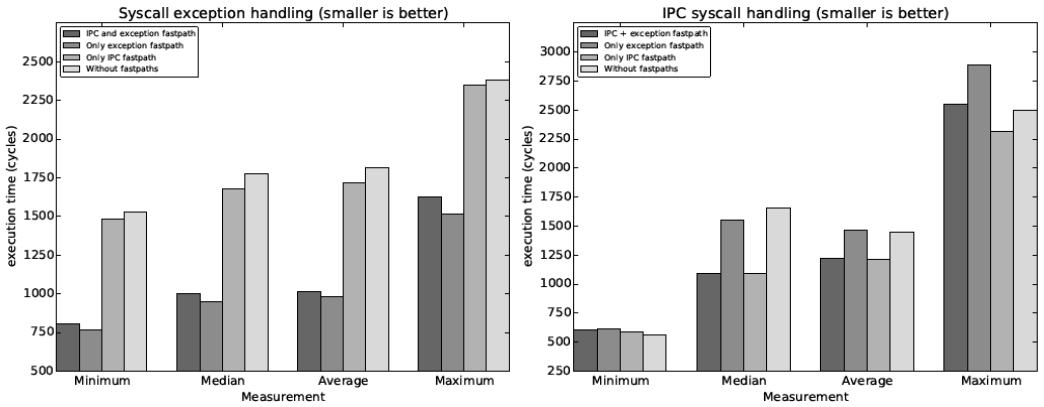

Figure 6a. Syscall exception handling **Figure 7b. IPC exception handling**

One of the difficulties with this new mechanism was that, while all IPCs end mostly in the same point, they can be synchronously interrupted and parts of them are executed later, a mechanism known as continuations. This meant that the measurements had to be interrupted and resumed later, otherwise the results would have included unwanted pieces of code, such as how much time it takes for a userspace thread to treat a syscall exception or reply to an IPC.

Under Linux three I/O heavy processes and three CPU intensive processes were started on each CPU, to simulate a system under load. It is noticeable from Figure 1a that, in the case of the system call exception fastpath is used exclusively by Linux and it performs more than 1.6 times faster on average than the slowpath. It is used exclusively because only one user thread per CPU can be active at any one time in Linux, which means that the exception handler in Linux should be waiting for an IPC whenever an exception occurs and no more than one can occur at a single point in time. Improvements on the IPC system call are visible as well in Figure 2a, about 1.3 times faster on average. While the improvement is smaller than that brought by the system call exception fastpath, it can have a big impact on system performance, because its' importance to L4 kernels in general and to our paravirtualized Linux spe-

cifically, where IPCs are used for IRQ delivery by the microkernel and then by the interrupt handler on the first CPU, which forwards IRQs to the other interrupt handlers on other CPUs when it is required (e.g. the timer interrupt, which is used for scheduling).

Benchmarking the syscall exception fastpath

The paravirtualized Linux did not stress the syscall exception mechanism enough for drawing conclusions about how it would perform under a more general work scenario. A test in which the number of threads per exception handler and wait time between syscall exceptions was varied in a controlled way was devised for this purpose. The mechanism for measuring performance is the same as the one used in the previous section.

The waiting time between each syscall generated by a thread is randomized using a pseudo-random number generator and the average waiting time is manipulated by multiplying this random number with a factor which is set on each test. This waiting time represents a counter for a loop, and after the loop finishes executing a syscall is generated, then the thread is reset. The syscall exception handler then catches a predefined number of events (3000) after which the wait factor is increased and the test runs again.

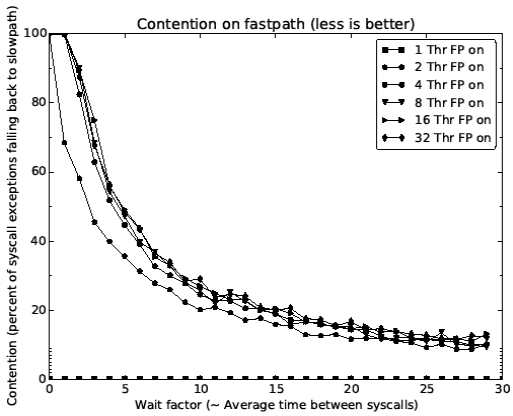

Figure 2a. Contention on fastpath

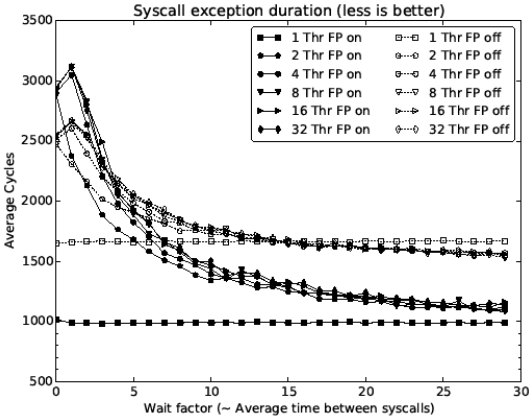

Figure 2b. Syscall exception duration

It is noticeable that waiting time has a much bigger impact on performance than the number of threads, as the curves from the two graphics in Figure 2a and Figure 2b are extremely similar, for two threads, for example, the correlation coefficient between contention and execution time is *0.9998*. Contention, generated by the having a short wait time between syscalls, follows a similar path as the one before, as can be seen in Figure 2a. This is thought to be caused by scheduling: increasing the number of threads also increases the waiting time between syscalls indirectly, effectively rate-limiting the number of syscalls generated in this test scenario. We do notice in Figure 2b that when only one thread is started, the results are as we expected: a horizontal line just below any of the other lines. This is in line with results from Figure 1a and since paravirtualization is an important use case for this particular L4 microkernel, it is a good sign that Linux performance is indeed improved by the fastpath.

Benchmarking the IPC exception fastpath

We employed the same measuring methodology and test program as in previous section, but toggled the IPC fastpath on and off instead. The improvements are only marginal in our benchmark. We notice that IPC duration is improved considerably when receiving from and replying to a single thread, however, when this is not the case, the presence of the fastpath only marginally decreases duration, when contention is low, and behaves worse when conten-

tion is high. From Figure 1 we know that under Linux the improvement is not negligible, but, as explained before, only one thread at a time is active. Further work is necessary to make the IPC fastpath behave well under more varied conditions.

CONCLUSION AND FURTHER WORK

The improvements made by these fastpaths are important for performance, even if their individual contribution is on the scale of a few percent improvements, and since they are written in a high level programming language they are very portable and easy to debug. The previous version had the issue of not being SMP-compatible and since CPUs come with an ever-increasing number of cores, it limited their usability. While our main concern is paravirtualization, the in-depth look at the syscall exception fastpath raised some interesting questions about how cycles are spent inside the microkernel and offered some hints as to where performance improvements could be made in the future.

It would be interesting to see exactly what are the test cases which fail on each fastpath more often than the others, and prioritize them, such that dropping back to the slowpath would have minimal overhead. In addition to this, it may be possible that some tests are linked to each other, in the sense that passing one would mean passing the other as well, and removing extraneous tests could improve performance. It might be possible to cover additional cases with the IPC fastpath at no cost, such as sending notifications. This seemed possible in the past, in the assembly version of the fastpath, but the current C/C++ version is not capable of doing this, and the direct translation of the assembly version produces some errors which are not yet investigated.

ACKNOWLEDGMENTS

The work has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/134398.

REFERENCES

Blackham, B., & Heiser, G. (2012). Correct, fast, maintainable: choose any three! In *Proceedings of the asia-pacific workshop on systems* (p. 13).

Condurache, C., & Eftime, P. (2014). *Enhancing virtualization on top of the vmxl4 kernel.*

Dannowski, U. (2007). Automated object layout optimization in a portable microkernel. In *Proceedings of the MIKES 2007: First International Workshop on MicroKernels for Embedded Systems*.

Elphinstone, K., Greenaway, D., & Ruocco, S. (2007). Lazy queueing and direct process switch - merit or myths?. In *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*.

Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., & Heiser, G. (2014). Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1), 2.

Liedtke, J. (1994). Improving ipc by kernel design. *ACM SIGOPS operating systems review*, 27, 175–188.

Liedtke, J. (1995). On micro-kernel construction. *Proceedings of the fifteenth ACM symposium on Operating systems principles*, 237-250

Linux Test Project. (n.d.). http://linux-test-project.github.io/. (Last accessed on 1 February 2015)

Mogosanu, L. (2013, July). Evaluating Virtualization on Top of the VMXL4 Microkernel, *Master thesis.*