

The ARM Fast Context Switch Extension for Linux

Gilles Chanteperdrix

gilles.chanteperdrix@xenomai.org

Richard Cochran

OMICRON electronics GmbH

Oberes Ried 1, A-6833 Klaus

richard.cochran@omicron.at

Abstract

The ARMv5 CPUs are inexpensive, low power, 32 bit processors widely used in embedded systems. Because of these processors' cache implementation, using memory protection on these systems incurs a performance penalty too large for many real time applications. By implementing the Fast Context Switch Extension, we achieved improved cache performance while retaining memory protection under the Linux 2.6 kernel. We briefly discuss the problem, explain the necessary changes to the Linux memory management system, and present performance measurements taken from artificial and real world applications.

1 Introduction

The ARMv5 architecture specifies virtually indexed, virtually tagged (VIVT) cache units. This means that the central processing unit (CPU) accesses the memory cache by means of virtual addresses. When the CPU issues a load or store instruction, the desired virtual address is presented to the memory management unit. If this address is present in the cache, then the unit immediately returns the cached data to the CPU. In a memory protected, multi-tasking operating system like Linux, each process has its own virtual address space. Consequently, the cache must be invalidated during a context switch in order to ensure that each process can only access its own data.

This cache flush also has the side effect of ensuring coherence between writable shared mappings mapped at different virtual addresses in *different* address-spaces. Furthermore, coherence between writable shared mappings mapped at different virtual addresses in *the same* address space is ensured in Linux by mapping them without cache.

As a result of the cache invalidation, each time a process is scheduled, it must reload all its working data from the main memory. Reading data from main memory is an expensive operation. Depending on the particular CPU type, the memory speed, and the program's data access pattern, the cost of the cache invalidation during a context switch can be

on the order of 200 microseconds. This performance penalty is great enough to preclude using such CPU types for certain time critical applications, for example when the reaction time to external events is required to be less than one millisecond.

One approach to address this problem, used for example in vxWorks or μ Clinux, is to use one "flat" address space, shared by the operating system kernel and all user processes. This avoids the context switch penalty while sacrificing memory protection. The obvious drawback under this scenario is that any one misbehaving program can corrupt the kernel or other programs.

Another solution is to use the Fast Context Switch Extension (FCSE) available on ARMv5 processors. The FCSE may be understood as a workaround to the VIVT cache, where a single virtual address space is divided up among several processes. This is done in such a way that no two processes share the same range of virtual addresses, thus obviating the need to invalidate the cache during a context switch.

The work described in this paper implements the FCSE under Linux kernel 2.6, removing some of the performance problems associated with the VIVT cache. Our goal was to create a minimally intrusive solution that would be viable for acceptance into the main line kernel.

2 Previous Work

Earlier work has highlighted the context switch performance problem running Linux on ARMv5 computers. Hyok-Sung Choi and Hee-Chul Yun [1] report dramatically improved performance when moving from plain Linux to μ Clinux. David, Carlyle, and Campbell [2] instrumented a Linux 2.6 kernel to investigate the direct and indirect latency caused by context switches while running four different algorithms. They found that 99 context switches during a three to four second test add between 0.17% and 0.25% to the running time.

Our work was not the first attempt to implement the ARM FCSE for Linux. As part of his student work at the University of New South Wales, Adam Wiggins implemented the FCSE for Linux kernel 2.4. The changes in Wiggins' Linux patch [3] are considerably more extensive than just adding the FCSE. To avoid changing the page directories during a context switch, Wiggins employed the ARMv5 domain mechanism to create a single, global Translation Lookaside Buffer (TLB) in software. This approach can avoid invalidating the TLB during a context switch, in addition to avoiding cache invalidation. However, the performance gain comes at the cost of some complexity in the ARM Linux kernel code. Wiggins termed the combination of using ARM domains and FCSE as Fast Address Space Switching (FASS), and he had previously [4] implemented FASS for the L4 microkernel.

Van Schaik and Heiser [5] describe a paravirtualized Linux 2.6 kernel called "Wombat" running under the L4 microkernel. They report Wombat having greatly reduced context switching times compared with native ARM Linux, under the restriction of a 32 MB virtual memory space for user programs.

Sebastian Smolorz [6] attempted a port of Wiggins' FASS work to Linux kernel 2.6. Smolorz's work helps to highlight the ARMv5 context switch performance problems, and it presents one possible approach toward a solution. Although we did copy one useful header file from this source, in our view this port suffers from two major drawbacks. First, the FASS changes are far too intrusive to ever be accepted into the main line kernel, since they make fairly complex use of ARM domains, introducing a completely new concept into the ARM Linux memory system. Second, the port is not fully working and is no longer being developed.

3 The ARM FCSE

A simplified schematic of the ARMv5 Memory Management Unit (MMU) is depicted in Figure 1. The

CPU issues a virtual address (marked VA in the figure) to the instruction cache (I-cache), data cache (D-cache), and the TLB. The TLB is a cache of virtual to physical address mappings. If the translation is not cached, then a hardware page table walker obtains the physical address (marked PA in the figure). The cache unit returns the desired data to the CPU if the requested address is in the cache.

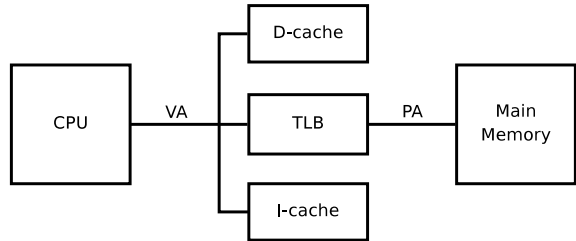


FIGURE 1: *Memory Management Unit*

Not shown in the figure is the fact that the cache units consult permission bits stored within the TLB. Since Linux uses the same address space layout for every user program, this arrangement of the MMU requires invalidating the caches and the TLB during every context switch.

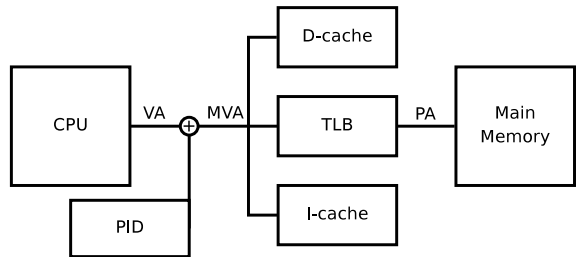


FIGURE 2: *FCSE PID Relocation*

The FCSE provides a "Process ID" register (PID) for sharing a single 32 bit virtual address space. This register contains a 7 bit process identifier that can be combined with a 25 bit virtual address, as shown in Figure 2. When the PID is set to a non-zero value and the 7 high order bits of a virtual address are cleared, a modified virtual address (shown as MVA) is generated by adding the PID value to the virtual address according to the following formula:

$$MVA = VA + (PID \ll 25)$$

Using this method allows 128 distinct 32 MB address spaces, so that up to 128 processes may safely coexist without having to flush the caches at each context switch. However, we note that it is still necessary to invalidate the TLB during a context switch in order to enforce memory protection. The resulting limitation of 128 processes and 32 MB address space is acceptable for some embedded systems.

4 Implementation

The implementation of the Linux memory management subsystem¹ is spread among many files, some of which are architecture independent, and some architecture specific. Moreover, the actual logic can be quite tricky and contains a fair amount of “black magic.”

We endeavored to implement the FCSE in a minimally intrusive way, without requiring major changes in the existing memory management code. The current patch [8] changes 23 files and affects about 300 lines in the kernel. In addition, the patch adds two new, FCSE-specific files with about 260 lines total. Our implementation affects only the ARM specific architecture sources. No changes in the main Linux memory management subsystem were necessary.

4.1 Guaranteed Mode

The virtual memory layout for ARM Linux is fixed in such a way that all virtual addresses above `TASK_SIZE` (3 GB) are always interpreted as kernel addresses. Adapting the FCSE to this model thus required reducing the number of useable PIDs to 95. When the FCSE is active, it effectively partitions a single virtual memory space into one kernel and 95 user regions, as shown in Table 1.

Virtual Address	Use
⋮	
0xC000.0000	KERNEL reserved
0xBE00.0000	
0xBC00.0000	PID 94
⋮	
0x0400.0000	PID 2
0x0200.0000	PID 1
0x0000.0000	PID 0

TABLE 1: *The FCSE memory layout*

Although the original intent of the ARMv5 PID register appears to have been to allow the operating system to keep the current process ID in this register, our implementation decouples the ARM PID from the Linux process ID. Linux gives each thread its own process ID, but all the threads within one process share the same virtual memory space. Therefore, we associate one ARM PID with the `mm_struct` of each process.

With FCSE enabled, the CPU automatically converts a virtual address in the range from zero to `0x2000000` into a modified virtual address within one

of the 95 possible slots. This creates situations where the kernel expects an unmodified VA, but the hardware MMU expects a MVA. The patch provides a pair of functions² to convert between a VA and a MVA. A good deal of the effort in implementing the FCSE was simply to identify the spots where such a conversion is required.

The only place where a MVA to VA conversion occurs is in the fault handler. In this case, the CPU has issued an instruction that has triggered a fault. The hardware gives the MVA as the source of the fault, but the kernel requires the original VA in order to properly handle the fault.

In contrast, the VA to MVA conversion occurs several times, and for two different reasons. When flushing the cache and the TLB, the kernel has a VA in its Virtual Memory Area (VMA) data structures, but the cache and TLB contain the MVA that the CPU generated. Thus the cache and TLB flushing code must present the MVA to the hardware. These cases account for almost all of the VA to MVA conversions.

The one other important VA to MVA conversion occurs in the `pgd_offset()` macro. This macro is used by kernel code that needs to access the page tables. By modifying the VA, this macro now ensures that page tables will be created or adjusted using a correctly relocated MVA.

Besides accounting for the VA to MVA conversions, the FCSE code must also enforce the 32 MB limit on the virtual address space and reprogram the PID register during a context switch. Limiting the address space is accomplished in the `arch_get_unmapped_area` function, where any mapping that would cause a task’s space to exceed 32 MB is denied. Reprogramming the PID register amounts to a one line addition to the `switch_mm` function. In order to actually benefit from the FCSE, we also must disable the cache invalidation code normally executed during a context switch. The current patch implements this for Xscale, ARM920, and ARM926, by adding appropriate `#ifdefs` to the machine specific assembler files under `arch/arm/mm`.

As mentioned in the introduction, the cache flush made by Linux also has an effect on shared mappings, so, to compensate, we modified the function `make_coherent`, used by the kernel to detect shared mappings in the same address-space and mark them non-cacheable, to mark as non-cacheable all writable mappings.

¹Mel Gorman’s book [7] offers a good, if somewhat dated, introduction to the Linux memory management system.

²The functions are defined in such a way that, if the FCSE option is not selected at compile time, they do nothing at all.

4.2 Best Effort Mode

After presenting initial versions of the FCSE patch to the ARM Linux kernel mailing list, one repeated criticism was that allowing only 95 processes, each with a 32 MB virtual address space, is an unacceptable limitation. There was a desire for a way to enable the FCSE but, at the same time, to fall back to the standard, unlimited memory management scheme dynamically. To fulfill this requirement, the current version of the patch offers a compilation option to enable either a “guaranteed” mode, or a “best effort” mode. The guaranteed mode operates as described above, with the mentioned limitations, and never flushes the caches during a context switch.

When the best effort mode is enabled, there are no artificial limits to the number of processes or the virtual address space. If the run time condition of the user space programs permits, then the system will benefit from the performance advantage that the FCSE offers. Under the best effort mode, the cache flush during a context switch is only performed when necessary to ensure the integrity of the virtual memory system.

For the best effort mode, we overcome the 95 processes limitation by reusing the FCSE PIDs. Each Linux process is assigned to one of 95 groups, and each group shares one PID. A context switch between processes belonging to two different groups does not necessarily trigger a cache flush. Instead, each group has a “dirty” bit which is set whenever a process in that group is scheduled. Also, each group remembers which of its processes was last scheduled. A cache flush is needed when the new process’s group is “dirty” and a different process was last scheduled in that group. In case of cache flush for any reason, the “dirty” bit of all pids is cleared, avoiding additional, unnecessary cache flushes.

In order to overcome the 32 MB virtual memory limitation, the best effort mode reserves the zero PID for those processes which require a larger space. Once a process requests more than 32 MB, it is relocated from its initial PID to PID zero by copying its page tables. The FCSE accounts for mappings which exceed the 32 MB area, and if and only if there are any, flushing the cache is required every time such a process is scheduled or preempted. This allows avoiding cache flushes if such a process returns below the 32MB limit.

The issue with shared mappings is solved differently than with the guaranteed mode. For each process, we account for writable shared mappings which have been written to, and if such mappings exist, we flush cache when switching from and to this process.

5 Performance Evaluation

In order to measure the effectiveness of the FCSE, we benchmarked a recent Linux kernel both with and without the FCSE. This section presents results of performance measurements on artificial and real world applications. These measurements were performed on an Intel IXP425 clocked at 400 MHz, with 128 MB main memory. The IXP425 has separate data and instruction caches, each 32 KB in size.

5.1 Artificial Applications

5.1.1 `cyclictest`

Thomas Gleixner’s `cyclictest` program [9] was developed to measure the scheduling latency of the Linux kernel. The program repeatedly attempts to sleep (yield the CPU) for a certain time interval, and measures the actual duration of the sleep to infer the latency. We ran `cyclictest` first under plain Linux, and then under FCSE Linux in both guaranteed and best effort mode, with the following command line arguments.

```
cyclictest -D 60 -p 50 -q -h 1000
```

In order to create a load on the system, we ran the following program in the background before running the `cyclictest` program.

```
while [ 1 ]; do
    find / ;
    cat /dev/mem > /dev/null ;
done
```

Table 2 shows the measured scheduling latencies for each of the three test cases. On average, the guaranteed FCSE mode had 100 μ s less latency than Linux without the FCSE, while the best effort mode showed about 50 μ s better average performance than plain Linux. The worst measured latency exceeded 550 μ s for all three test cases, but the FCSE test cases outperformed plain Linux here as well.

Kernel	Min	Ave	Max
Plain Linux	97	194	682
Guaranteed FCSE	26	94	611
Best Effort FCSE	37	143	569

TABLE 2: *Scheduling Latency (μ s)*

Figures 3 and 4 show the histograms of the `cyclictest` latency measurements. In Figure 3 the results from the guaranteed FCSE mode, on the left hand side, are presented along with plain Linux, on the right hand side.

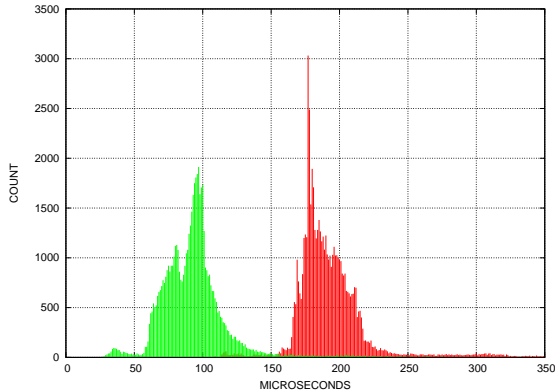


FIGURE 3: *FCSE versus Plain Linux*

Figure 4 shows the results from the best effort FCSE mode. Here one can recognize that the best effort mode appears as a combination of plain Linux and FCSE Linux, as it were. It is interesting to notice an additional latency region at about 250 μ s. Although the average performance is clearly improved over plain Linux, the best effort mode appears to have increased latencies in some code paths.

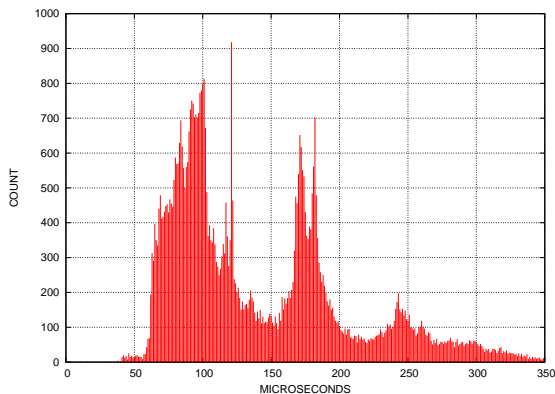


FIGURE 4: *Best Effort FCSE*

5.1.2 lmbench

Larry McVoy’s `lmbench` [10] comprises a suite of programs that measure various system performance benchmarks. In our view, `lmbench` is not as useful as other tools for measuring latency since it reports only an average value, rather than a set of statistics or a histogram. However, since both Choi [1] and Van Schaik [5] report results obtained using `lmbench`, we also include measurements made with that program.

Since neither of those two papers mention using any kind of system load, we assume that their `lmbench` measurements were conducted with an otherwise idle system. Thus, in contrast to the `cyclictest` measurements, we ran the tests without any system load, with the command line as follows.

```
for s in 0 1 4 16; do
    lat_ctx -s $s 2 4 6 8 10 12 14 16;
done
```

We compared `lmbench` performance for plain Linux and FCSE Linux in both guaranteed and best effort mode. For each variety of Linux, we ran the `lat_ctx` test four times, using a different work buffer size each time, and varying the number of processes participating in the test. Following Choi, [1] we used work buffer sizes of 0, 1, 4, and 16 KB. A larger work buffer causes greater pressure on the data cache. For each buffer size, the test connects a number of processes in a ring of Unix pipes. Each process reads a token from its pipe and then writes the token to the next process. In the graphs, the x-axis shows the number of processes, the y-axis shows the measured latency. The result for each work buffer size appears as one line.

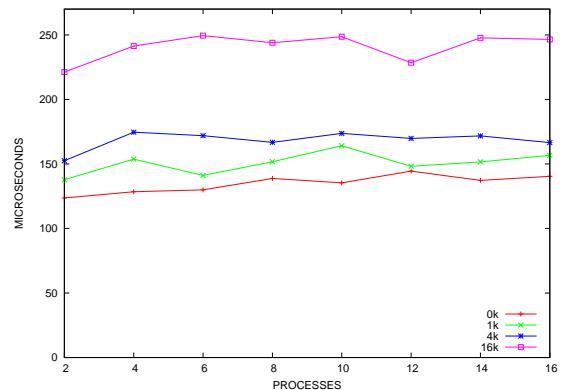


FIGURE 5: *Plain Linux - lmbench*

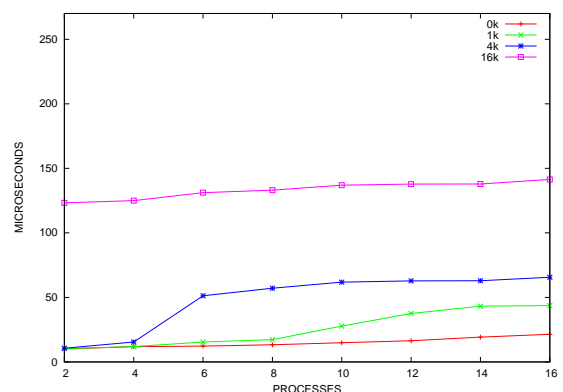


FIGURE 6: *FCSE Linux - lmbench*

Figure 5 shows the results of the `lmbench` measurement for plain Linux. For this test case, the scheduling latency stayed well above 100 μ s in every case. The fact that the number of processes made little difference in the latency comes as no surprise, since, by flushing the cache during a context switch,

plain Linux always provokes the worst case performance.

Figure 6 shows the results for FCSE Linux in guaranteed mode, and Figure 7 shows the best effort mode. For these test cases, the scheduling latency never exceeded $150 \mu\text{s}$. Here we notice that increasing the number of processes did indeed affect the latency. It is also interesting that the two FCSE modes yielded almost identical performance for this test. We expect that running the test under a loaded system, as we did for the `cyclictest`, would better highlight the difference between the two modes.

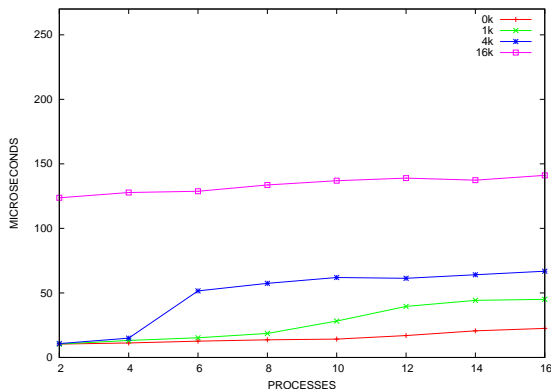


FIGURE 7: *Best Effort FCSE - lmbench*

5.1.3 hackbench

The `hackbench` test is part of the LTP testsuite and written by a few kernel developers. It tests scheduler performance and causes many context switches, which makes it a good test to evaluate the performances of the FCSE-enabled kernel in best-effort mode.

The result provided by the `hackbench` test is its running time for a given number of processes. A shorter running time indicates increased system performance.

We ran 50 instances of this test with 400 processes on a kernel without FCSE, and on a kernel with FCSE in best-effort mode. The test ran without any additional load, on a AT91RM9200, ARMv4 processor clocked at 180 MHz.

Figure 8 shows that a kernel with FCSE in best-effort mode with 400 processes, even though there are some cache flushes from time to time (400 processes means that for each PID, at least 4 processes are using the same PID), manages to run the `hackbench` test in almost half the time needed to run it without FCSE.

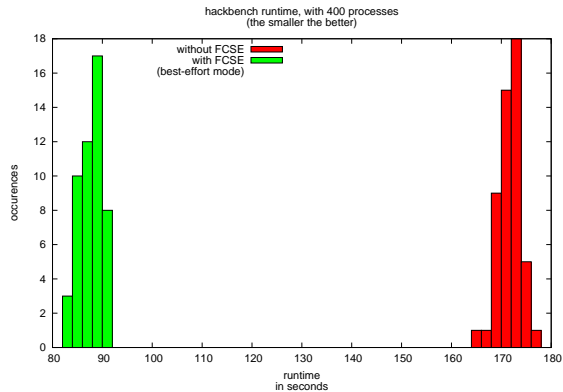


FIGURE 8: *hackbench*

5.2 Real World Applications

Our original motivation for implementing the FCSE was to improve the performance of ARMv5 machines on real world applications. One recent trend in various industries is to run time critical applications over standard Ethernet. For example, Ethernet networks are being used as field bus replacements (like EtherCAT), for distributed real time computing, and for audio/video streaming. We present performance results for two real world applications that send or receive time critical messages in the form of Ethernet packets.

We used the Beckhoff ET200 Industrial Ethernet Multichannel Probe in order to obtain impartial results, external to the device under test. The ET2000 is able to time stamp Ethernet packets with a resolution of 10 nanoseconds. Since we measure the interval with an Ethernet tap outside of the device under test, the measured time includes all of the delays from both the hardware and software of the system under test.

5.2.1 Packet Message Reply

In this application, the program expects incoming Ethernet packets containing control commands. After receiving a command packet, the application updates its internal state and sends a status reply packet. It is important that the program sends the reply as soon as possible. We measured the interval from the time the command packet appears on the wire to the time the reply packet appears.

Kernel	Min	Ave	Max	Std Dev
Plain Linux	313	364	771	29.46
Guaranteed FCSE	211	283	840	30.50
Best Effort FCSE	210	284	718	31.59

TABLE 3: *Packet Reply Latency (μs)*

We tested this application under plain Linux and FCSE Linux in both guaranteed and best effort mode. During these tests we created a system load using the method described in Section 5.1.1. For this test, the best effort FCSE mode performed nearly as well as the guaranteed mode, as shown in Table 3. Both of the FCSE modes outperformed plain Linux by $80 \mu\text{s}$ on average. Figure 9 presents a histogram of the reply times using guaranteed FCSE mode, on the left hand side, along with the times using plain Linux, on the right hand side.

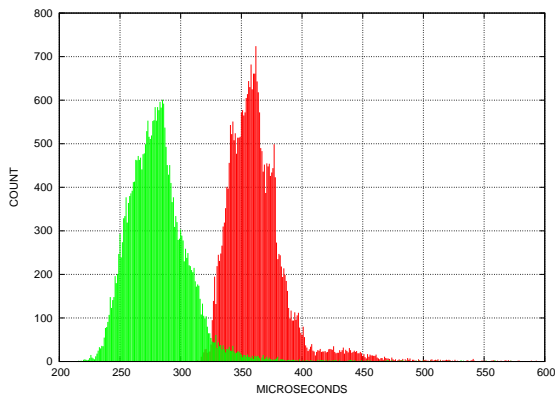


FIGURE 9: *FCSE vs Plain Linux*

5.2.2 Periodic Packet Transmission

This application periodically transmits a single Ethernet packet containing sampled data. The program attempts to send one packet every $500 \mu\text{s}$ exactly. On the test platform, keeping such an extremely short interval is not possible using a standard Linux kernel, so we used Xenomai [11] to achieve better real time performance. The program under test creates one high priority Xenomai thread which runs a periodic timer set at $500 \mu\text{s}$. In order to demonstrate the effect of the FCSE, we also modified the `trivial-periodic` example program to run its timer in a one millisecond period. This second Xenomai thread was executed with a lower priority than the program under test.

Xenomai Kernel	Min	Ave	Max	Std Dev
Plain Linux	458	500	542	17.157
Guaranteed FCSE	456	500	534	1.969

TABLE 4: *Packet Period (μs)*

Xenomai Linux is able to deliver the correct average period both with and without the FCSE, as the results in Table 4 show. However, the standard deviation is noticeably higher without the FCSE.

Figure 10 shows the histogram of the packet transmission period when using plain Xenomai. Notice that the y-axis is logarithmically scaled in order

to highlight the worst case performance. The symmetrical form of this graph demonstrates how the Xenomai scheduler is able to keep a periodic timer on track. Whenever the timer overshoots a deadline, the induced delay is always compensated in the next period. This graph has two maxima around the average interval, offset about $15 \mu\text{s}$. In addition, there are two local maxima offset about $40 \mu\text{s}$ from the center.

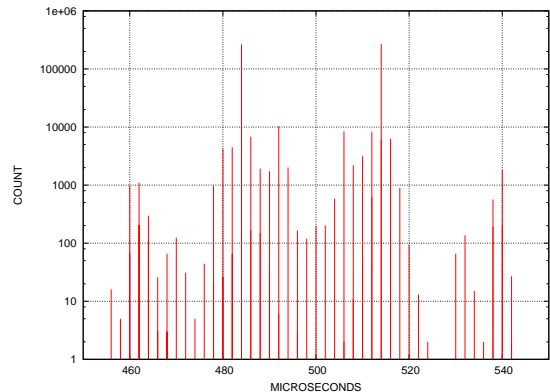


FIGURE 10: *Periodic Task - Plain*

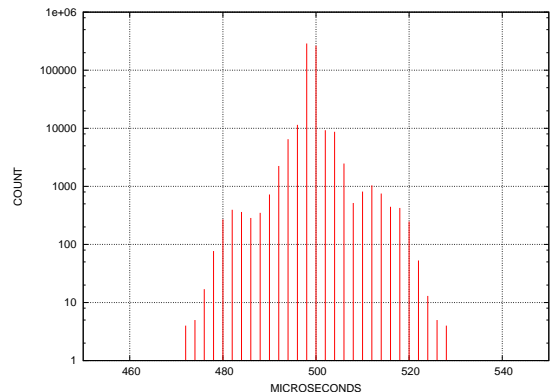


FIGURE 11: *Periodic Task - FCSE*

Figure 11 shows the histogram when the FCSE is enabled. In contrast to the previous graph, here the histogram has a wedge shape with only one maximum in the center. Comparing the shapes of Figures 10 and 11 illustrates the consequence of the heavy handed cache flushes which are necessary when the FCSE is not enabled.

6 Conclusions and Future Work

The ARM FCSE offers a method for avoiding costly cache flushing during context switches. We presented a minimally intrusive implementation of the FCSE for the Linux kernel. Experimental evidence from artificial and real world tests shows that enabling the

FCSE can reduce the time to schedule processes by dozens or even hundreds of microseconds. This performance gain is significant enough to bring certain real time applications within reach of Linux running on ARMv5 processors.

The FCSE imposes fairly restrictive limitations on programs. However, for systems which cannot accept the FCSE limitations, the benefits of the FCSE can still be exploited by using our patch in the dynamic, best effort mode.

It is the authors' wish to have the FCSE merged

into the mainstream ARM Linux kernel. The patch has previously been presented on the mailing list, and it will be resubmitted in small, more easily digestible pieces in the near future. Some further work needs to be done, namely identifying the other ARMv5 machines supported by Linux and adapting their machine specific assembler files to benefit from the FCSE. One interesting possibility not covered here would be to combine the FCSE with the PREEMPT_RT patch and gauge its effectiveness in that configuration.

References

- [1] Hyok-Sung Choi and Hee-Chul Yun.
Context switching and IPC performance comparison between uClinux and Linux on the ARM9 based processor.
Technical report, Samsung Electronics, 2004.
http://opensrc.sec.samsung.com/document/uc-linux-04_sait.pdf.
- [2] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell.
Context switch overheads for Linux on ARM platforms.
ExpCS, 13-14, June 2007.
- [3] Adam Wiggins.
Fast address-space switching for ARM Linux kernels.
<http://www.ertos.nicta.com.au/software/fass>, March 2004.
- [4] Adam Wiggins.
The design and implementation of the L4 microkernel on the StrongARM SA-1100.
Bachelor thesis, University of New South Wales, November 1999.
- [5] Carl van Schaik and Gernot Heiser.
High-performance microkernels and virtualisation on ARM and segmented architectures.
Technical report, University of New South Wales, 2007.
<http://www.ertos.nicta.com.au/software/fass>.
- [6] Sebastian Smolorz.
linux-2.6 FASS.
<http://git.opensource.emlix.com/cgi-bin/gitweb.cgi?p=kernel/fass/linux-2.6.git>, November 2007.
- [7] Mel Gorman.
Understanding the Linux Virtual Memory Manager.
Prentice Hall, 2004.
<http://www.kernel.org/doc/gorman>.
- [8] Gilles Chanteperdrix.
FCSE for Linux.
<http://git.denx.de/?p=ipipe-2.6.git;a=commitdiff;h=607cdaaa4447068ac81e04ad79af3feb04858fd5>,
May 2009.
- [9] Thomas Gleixner.
Cyclictest.
<http://www.kernel.org/pub/linux/kernel/people/tglx/rt-tests>, August 2009.
- [10] Larry McVoy.
Lmbench - tools for performance analysis.
<http://www.bitmover.com/lmbench>, August 2009.
- [11] Philippe Gerum.
Xenomai: Real-time framework for Linux.
<http://www.xenomai.org>, August 2009.