Context Switch Overheads for Linux on ARM Platforms

Francis M. David
fdavid@uiuc.edu

Jeffrey C. Carlyle
jcarlyle@uiuc.edu

Roy H. Campbell
rhc@uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N Goodwin Ave
Urbana, IL 61801-2302

ABSTRACT

Context switching imposes a performance penalty on threads in a multitasking environment. The source of this penalty is both direct overhead due to running the context switch code and indirect overhead due to perturbation of caches. We calculate indirect overhead by measuring the running time of tasks that use context switching and subtracting the direct overhead. We also measure the indirect overhead impact on the running time of tasks due to processor interrupt servicing. Experiment results are presented for the Linux kernel running on an ARM processor based mobile device platform.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms

Experimentation,Measurement,Performance

Keywords

operating system, context switch overhead

1. INTRODUCTION

Context switching is the fundamental mechanism that is used to share a processor across multiple threads of execution. Each thread is associated with a processor state such as the program counter, general purpose registers, status registers and so on. A context switch is the act of saving the processor state of a thread and loading the saved state of another thread. If the threads are associated with different virtual address spaces, a context switch also involves switching the address translation maps used by the processor. In Linux, this happens when the threads belong to different user processes. Switching address spaces requires that relevant entries in the processor's address translation cache (TLB) are invalidated. If the instruction or data caches are tagged using virtual addresses, they would have to be emptied as well.

Context switching imposes a small performance penalty on threads in a multitasking environment. In addition to the direct overhead associated with the actual context switching code, there are several other factors that contribute to this penalty. The perturbation of processor caches like the instruction, data, address translation and branch-target buffers results in an additional indirect overhead. Yet another possible source of indirect overhead is operating system memory paging. A context switch can result in an in-use memory page being moved to disk if there is no free memory, thus hurting performance. In this paper, we do not consider overheads due to paging and assume that sufficient main memory is present to avoid thrashing.

We have described a context switch as a mechanism used to switch between two threads of execution. We do not consider a system call a context switch. This is like a simple function call and only involves switching the processor from unprivileged user mode to a privileged kernel mode. Memory maps are not switched. The transition back to userspace from the kernel during the return of the system call is similar to a function call return.

A processor interrupt causes the state of the currently executing task to be saved while an interrupt service routine is executed. When the interrupt service routine completes, the saved state is restored. While memory maps are not switched during interrupt servicing, it does perturb cache state and might also contribute some indirect overhead.

In this paper, we measure the indirect overhead of context switches inside the Linux kernel using pairs of tasks that perform cooperative multitasking. In a separate set of experiments, we also measure the indirect overhead introduced due to processor interrupt servicing.

We do not explore userspace implementations of threads and userspace context switching. The latest versions of the Linux kernel support the Native Posix Threading Library (NPTL) which implements user threads as kernel threads and context switching happens inside the kernel.

This study targets mobile device architectures and the hardware platform we use in our experiments is the OMAP1610 H2 Software Development Platform [8] cellular phone reference design from Texas Instruments. The OMAP1610 is powered by an ARM processor core.

The rest of this paper is organized as follows. Section 2 presents a quick introduction to the hardware platform that we use in our experiments. We discuss the experiment setup and results for context switch overhead measurements in section 3. The experiment setup

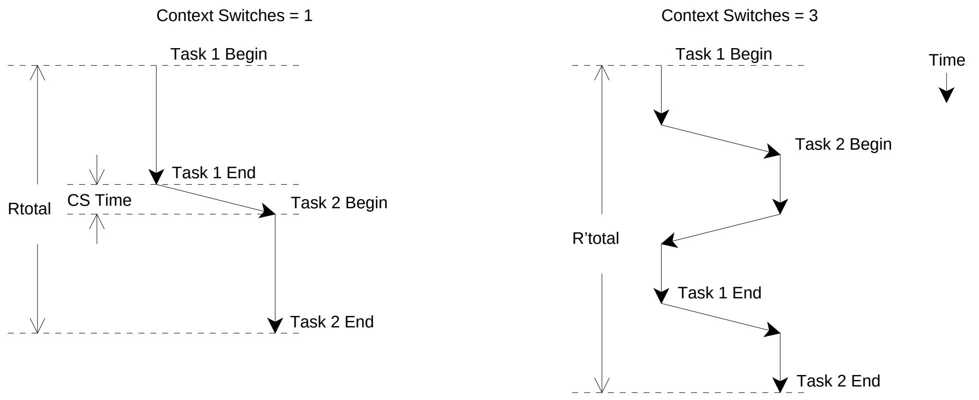


Figure 1: Context Switch Overhead Experiment Measurements

and results for interrupt servicing overhead measurements are presented in section 4. After exploring some related work in section 5, we conclude in section 6.

2. EXPERIMENTATION PLATFORM

ARM is a 32-bit RISC architecture. ARM processors are widely used in mobile devices because of their low power consumption. In this section, we briefly describe some features of the ARM architecture that are relevant to this research. Our implementations and experiments have been carried out on a processor core which belongs to the ARMv5 architecture generation. The ARM926EJ-S processor core that we use is part of the OMAP1610 chip from Texas Instruments.

Context switches require the saving of 16 general purpose registers (including the program counter) and one status register. A memory management unit (MMU) translates virtual addresses from the processor into physical addresses. A split (Harvard) memory cache is available in the processor, providing a 16 kilobyte, four-way set-associative instruction cache and a 8 kilobyte, four-way set-associative data cache. These caches are virtually tagged and therefore, have to be emptied when switching contexts. There are two TLBs - one for data and one for instructions. Each TLB holds 64 entries. TLB entries can be locked down in software, but we do not use any lockdowns for the experiments in this paper.

The clock fed to the processor can be configured during boot time. A 120 MHz processor clock is used for our experiments. Direct read access to a 32-bit on-chip timer running at 6 MHz (166.67 nanosecond measurement granularity) is used to measure time. The data cache is configured for write-back caching. 32MB of SDRAM with a 32-bit data path to the CPU is present on the OMAP1610 H2 board. We clock the SDRAM at the same frequency as the processor. The ARM processor also has a write buffer that queues pending writes to memory. We ensure that the write buffer is drained before running any experiments.

Processor performance monitoring counters are not available on the ARM processor we use. Therefore, we are unable to report statistics on the number of instructions executed. Our tests were run on the 2.6.20-rc5-omap1 version of the Linux kernel with some custom modifications for our experiments. Our modifications are described in section 3 and in section 4.

3. CONTEXT SWITCHING OVERHEAD

3.1 Experiment Setup

We added code into the Linux kernel to measure the running time of tasks performing deterministic computation with a controlled number of context switches and without external interference such as interrupts. In order to measure running time and the effects of a context switch accurately, the task code is built into the kernel and system calls are not used. We, however, configure the task with a unique `mmu_struct` to ensure that the page table mappings are reset during a context switch. This setup allows us to explore the impact of cache flushes and TLB invalidation during a context switch.

All measurements are performed starting with cold data and instruction caches, an empty address translation cache (TLB), and a fully drained write buffer. All data points in the graphs we present are an average over five measurements. The maximum (over all data points) coefficient of variation (standard deviation/average) of the running time measurement is marked in each graph as "Max CV".

In our experiments, we use two such tasks. The code for the tasks is modified to context switch between one another similar to cooperative multithreading. The scheduler is not used and we directly switch between the tasks by invoking the Linux `context_switch` function. The minimum possible combined running time of both tasks (R_{total}) is obtained when the first task is started and runs to completion, followed by a context switch to the second task which also runs to completion. This is illustrated in figure 1. The direct overhead for a single context switch (C) is constant for all experiments. This is obtained by measuring the time taken to execute the `context_switch` function with empty caches and a drained write buffer. The R_{total} measurement includes the overhead for one context switch. $R_{total} - C$ is the running time of both tasks without this overhead. The total running time of both tasks with n context switches is measured as R'_{total}. $R'_{total} - (R_{total} - C)$ is the total overhead due to the context switching. This consists of two parts. The first part is direct overhead due to the actual running time of the context switch code (register set save and restore, MMU switch etc). The second part is indirect overhead because of memory and translation cache pollution. The total direct overhead for an experiment with n context switches is $n * C$. The indirect overhead experienced by both tasks together due to the n context switches is then obtained as $I = R'_{total} - (R_{total} - C) - n * C$. When $n = 1$, $R'_{total} = R_{total}$ and therefore, $I = 0$.

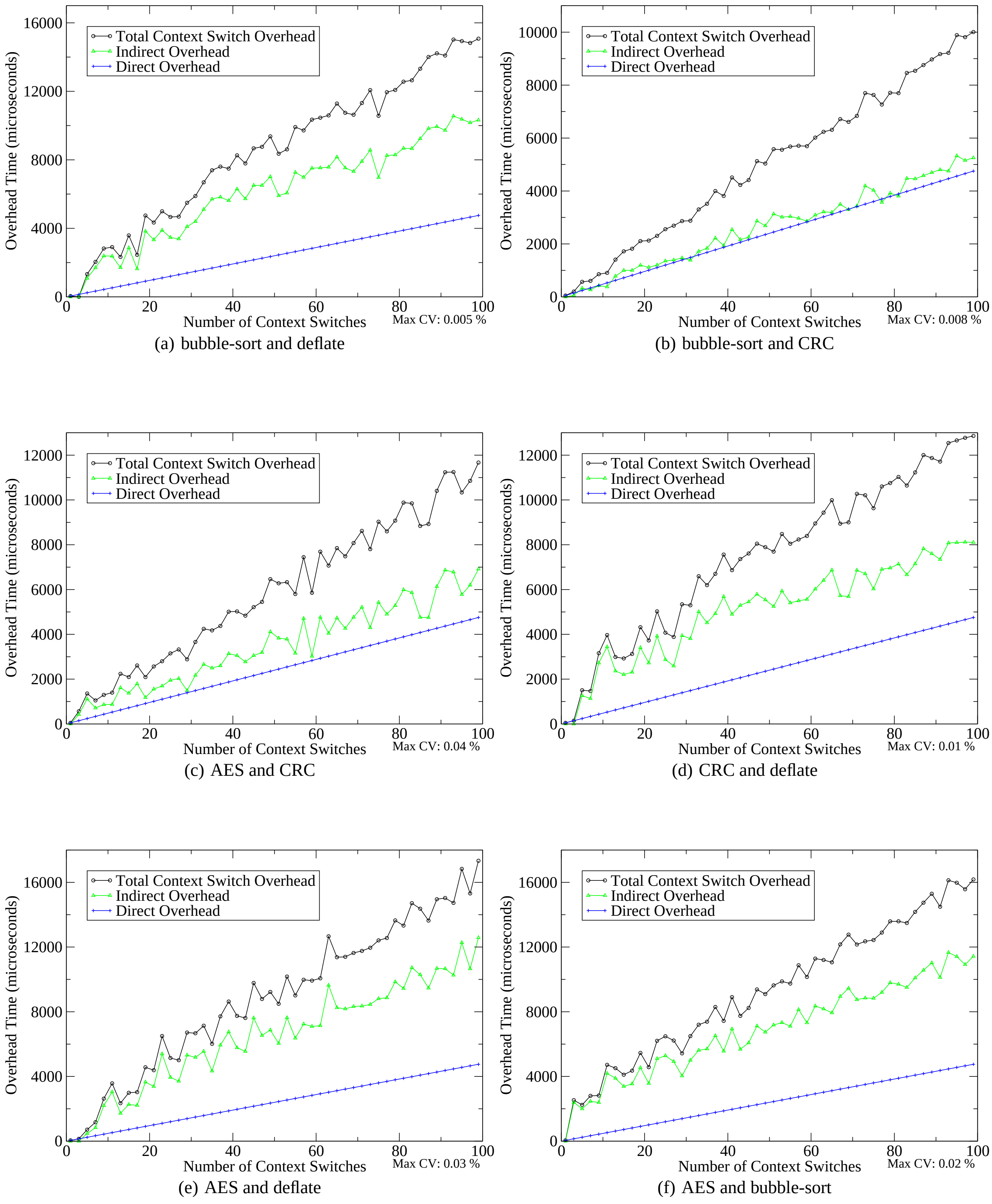
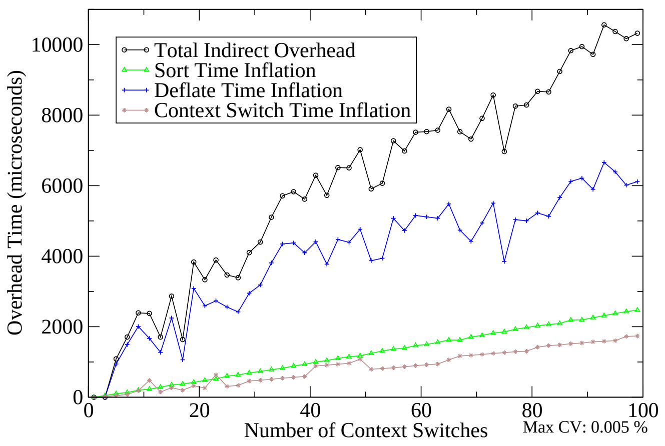
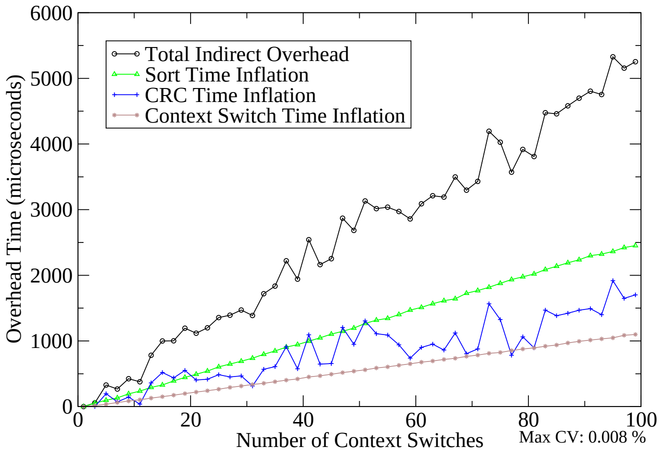


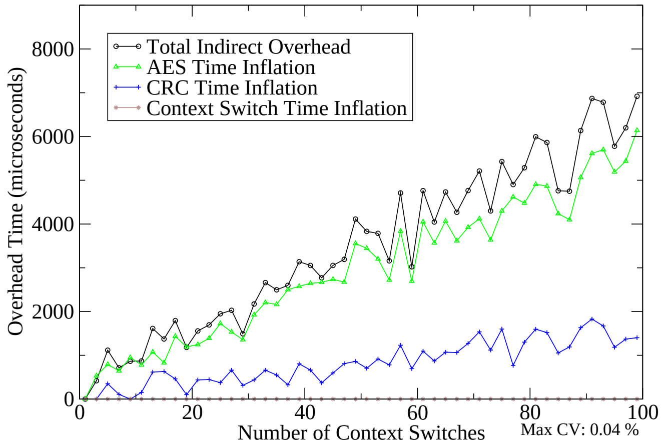
Figure 2: Context Switch Overheads - Direct and Indirect



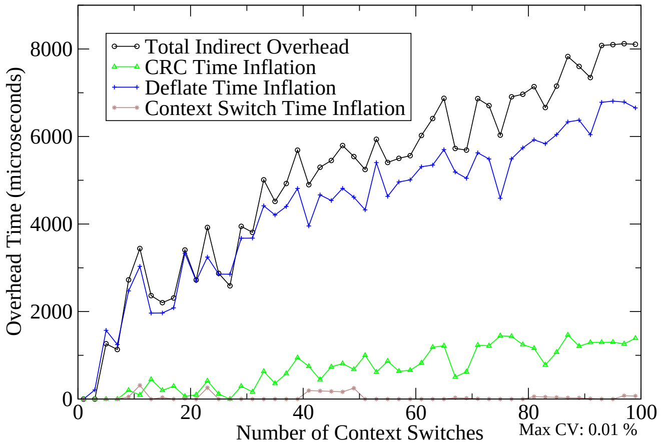
(a) bubble-sort and deflate



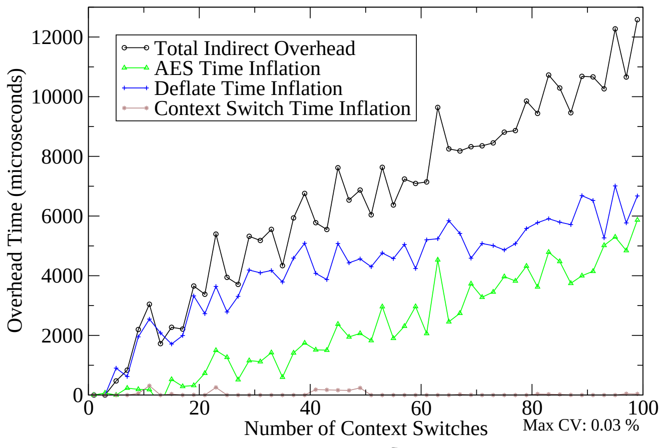
(b) bubble-sort and CRC



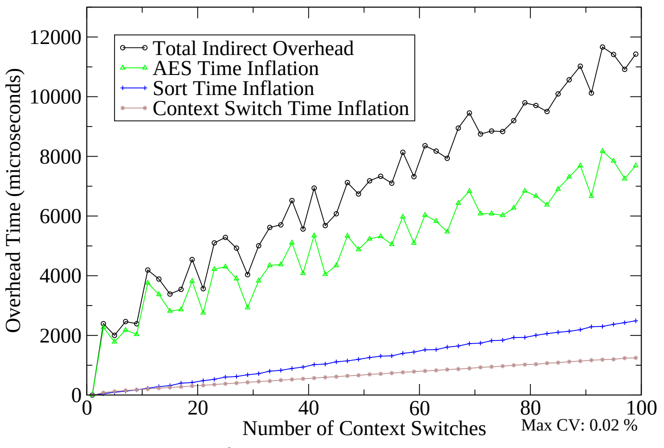
(c) AES and CRC



(d) CRC and deflate



(e) AES and deflate



(f) AES and bubble-sort

Figure 3: Indirect Context Switch Overhead Breakdown

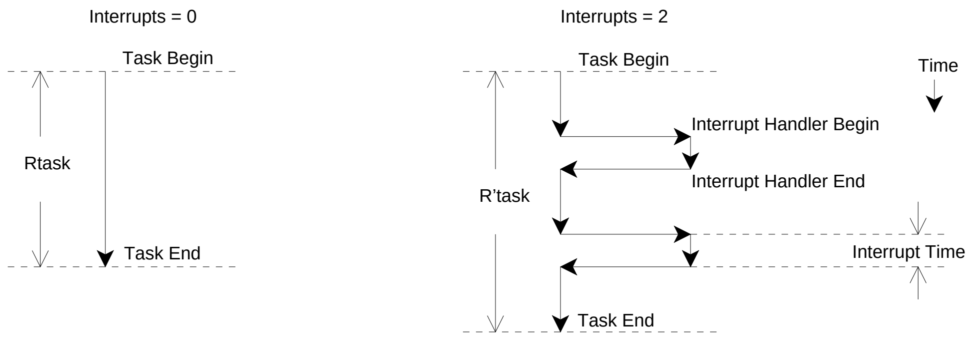


Figure 4: Interrupt Overhead Experiment Measurements

The tasks we pick for our experiments are processor and memory bound algorithms: bubble sort, deflate compression, AES encryption and CRC computation. These are designed to be non-interactive and use no device I/O. The data set used by the bubble sort algorithm is chosen to be the same size as the data cache and is aligned to fit completely within the cache. This maximizes data cache locality for the sort task and allows us to explore the impact of data cache pollution. All of the other tasks operate on large data streams and therefore, exhibit little data cache locality. The data set size for each task is chosen such that the task takes between 3 and 4 seconds to complete. The deflate, AES and CRC tasks are based on code libraries already inside the Linux kernel; the bubble sort algorithm was implemented by one of the authors.

3.2 Results

The direct overhead for a single context switch was measured to be 48 microseconds. Figure 2 compares the growth of the indirect and direct overheads for several pairs of tasks with varying values of n.

The indirect overhead for 99 context switches between bubble-sort and deflate adds about 10.3 milliseconds to their combined running time of 7.159 seconds (0.14% increase) For AES and CRC, the indirect overhead adds 6.9 milliseconds to their combined running time of 6.6 seconds (0.10% increase). The maximum increase we observed was with AES and deflate (0.18%). An interesting observation is that the indirect overhead at 99 context switches for all pairs of tasks is larger than the direct overhead (4.75 milliseconds).

Altogether (direct and indirect overheads), the context switching (99 times) adds between 0.17% and 0.25% to the running time for the various pairs of tasks.

Figure 3 shows additional details for the indirect overhead measurements. The total indirect overhead measurement is broken down into three components: the indirect overhead experienced by the individual tasks and the inflation in the context switch time because of the need to flush dirty cache lines to memory. The “sort” task measurements have very low variation and show a perfectly constant rate of overhead increase. The variation in the time measurements for the other tasks is higher and the plots are not smooth. However, the general trend for all tasks is an increase in the indirect overhead as the number of context switches increases. CRC experiences the least impact from context switching and overheads stay below 2 milliseconds for up to 99 context switches. This is followed by sort with 2.5 milliseconds. AES and deflate experience much higher overheads which vary between 6 and 8 milliseconds.

4. INTERRUPT SERVICING OVERHEAD

4.1 Experiment Setup

In order to measure overheads due to interrupt servicing, we modify the Linux kernel to run a task with context switching disabled and only one interrupt enabled. The interrupt we choose for this set of experiments is a timer interrupt. The actual hardware timer is disabled. We control the interrupts in this experiment by forcing an interrupt to be raised. This is done by writing two 32-bit words to a memory-mapped register on the interrupt controller. The interrupt is processed by the standard Linux kernel interrupt handling mechanisms. Our handler for the interrupt just increments a counter and returns.

We first measure the direct overhead involved in servicing the controlled timer interrupt as D. The value of D is computed as an average of 100 interrupt invocations in a loop. The write buffer is drained and the caches are warmed by raising an interrupt just before the start of the experiment.

The task is configured to periodically request interrupts while running. If the time taken to complete the task without interrupts is R_{task} and the time taken to complete the task with n interrupts is R'_{task}, the indirect overhead due to the interrupts is obtained by the equation $I = R'_{task} - R_{task} - n * D$. The measurement of R'_{task} is illustrated in figure 4.

The tasks we use for the interrupt overhead experiments are the same as the ones we use for the context switching experiments: bubble sort, deflate compression, AES encryption and CRC computation.

4.2 Results

Figure 5 shows the indirect overhead measurements for different rates of interrupts. The indirect overhead measurements are low and in the order of hundreds of microseconds. Unfortunately, even though the coefficient of variation of the actual measurements is low, the coefficient of variation for these calculated values of indirect overhead is high and only a general trend should be interpreted from all the graphs. A few data points in the graphs have been removed because the variation was extremely high. The “sort” task has the lowest variation and is a more accurate representation of interrupt indirect overhead behavior. For all tasks, the indirect overhead increases as the interrupt frequency is increased. Overheads are generally lower than for context switching because the caches and the TLB are not flushed.

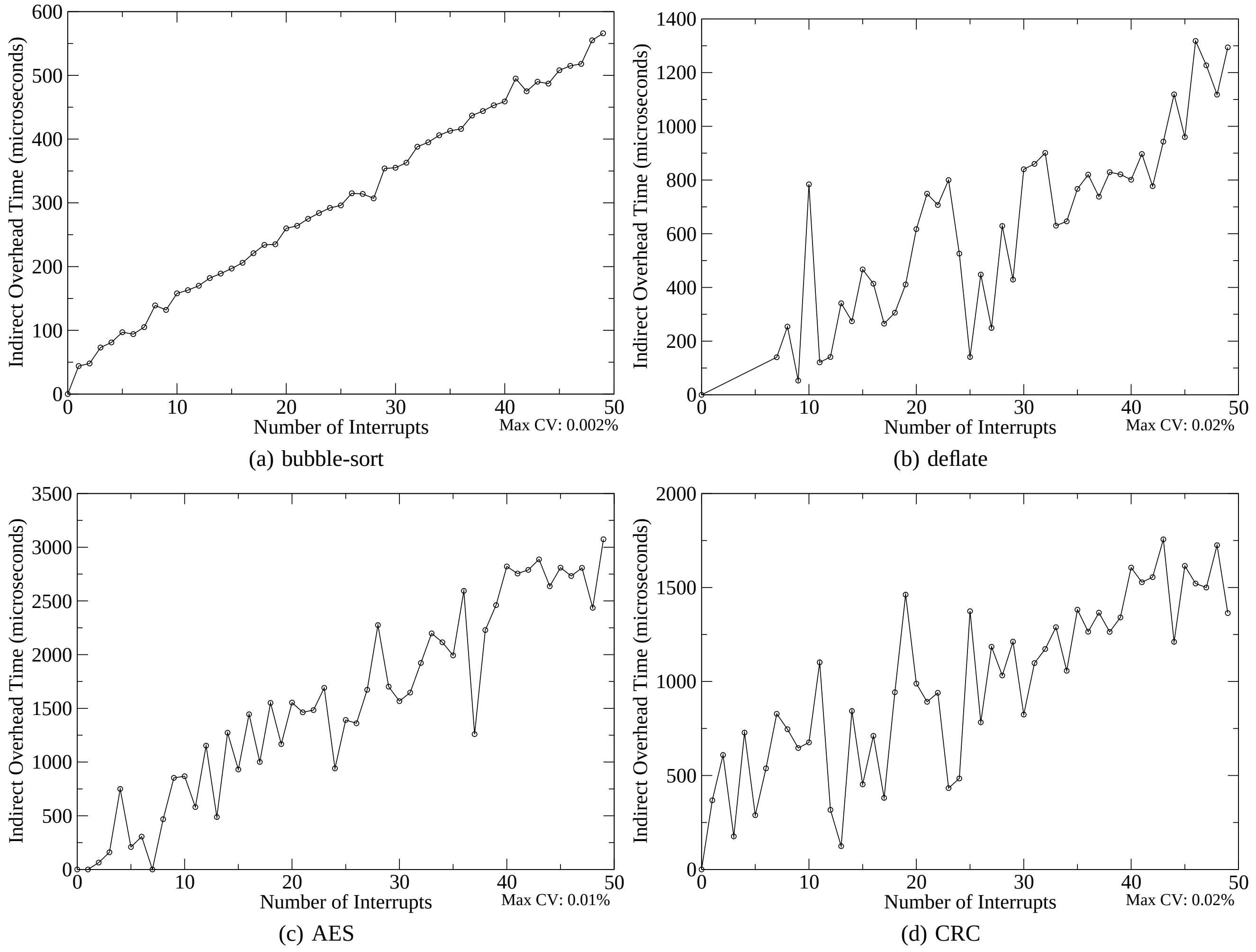


Figure 5: Indirect Interrupt Service Routine Overhead

The indirect overhead due to servicing 49 interrupts during the execution of “sort” adds 0.01% to its running time of 3.64 seconds. The values for deflate, AES and CRC are 0.02%, 0.09% and 0.05% respectively. For the interrupt experiments, the direct overhead measurements are larger than the indirect overhead measurements. We suspect that this is because our timer interrupt experiment does not significantly perturb the memory caches or the TLB.

Altogether (direct and indirect overheads), interrupt servicing adds between 0.28% and 0.38% to the total running time of the tasks for 49 interrupts.

5. RELATED WORK

Ousterhout attempted to measure the overhead associated with context switching by measuring the round trip time for passing a token between two processes using a pipe [6]. `lmbench` [4] improved upon this technique by eliminating system call overhead associated with Ousterhout’s measurements. Additionally, the cache footprint of processes were varied to determine how processor cache utilization affects context switching. Other researchers have also examined the relationship between caches and context switching [5, 7]. Our experiment framework can be used for a similar investigation of cache footprint effects. However, this is not investigated in this paper. Unlike our experiments, these studies do not attempt to differentiate between the direct and the indirect costs of a context switch.

Researchers have also examined the impact of context switching on branch prediction accuracy and found the effect to be minimal [2]. This implies that branch prediction failure is not a significant source of indirect context switching overhead.

Context switching overhead can be significantly reduced by exploiting hardware features on some architectures. Researchers have shown that the use of Fast Address-Space Switching (FASS) support on the StrongARM processor reduces context switching overheads by an order of magnitude [9]. The developers of the L4 microkernel used `lmbench` to show that because of the L4 support for FASS hardware extensions, the paravirtualized implementation of Linux on L4 has significantly lower context switching overheads than native Linux without FASS support [3]. Physically tagged caches supported by the ARMv6 architecture eliminate the need to perform cache flushes during context switching [1]. This has the potential to significantly reduce context switch overheads.

6. CONCLUDING REMARKS

The context switching results reported in this work are low level measurements of context switch overhead between two threads inside the operating system kernel. While we use in-kernel tasks for our experiments in order to ensure a controlled environment, we expect our results to closely reflect the behavior of userspace tasks during context switching. A context switch between two userspace processes involves both a timer interrupt (for an involuntary switch) and a scheduling decision in addition to the low level switch measured in this paper. The additional code executed when switching between userspace tasks may result in a small increase in the overheads due to additional cache pollution.

The interrupt overhead measurements for in-kernel tasks should be identical to the interrupt overheads experienced by a userspace task. These overheads will, however, vary for different types of interrupts which invoke different handlers. The results reported in this work are for a simple timer interrupt handler which just increments a counter after being signaled by standard Linux interrupt dispatch code.

The Linux kernel code used in our experiments is available on our website at `http://choices.cs.uiuc.edu/`

7. ACKNOWLEDGMENTS

We thank Ellick Chan for contributing to discussions on several technical aspects of this paper. We also thank the anonymous reviewers for helpful feedback. Part of this research was made possible by grants from DoCoMo Labs USA and generous support from Texas Instruments.

8. REFERENCES

[1] D. Brash. The ARM Architecture Version 6. *ARM White Paper*, Jan 2002.

[2] M. Co and K. Skadron. The Effects of Context Switching on Branch Predictor Performance. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 77–84, Nov 2001.

[3] L4 Performance. `http://ertos.nicta.com.au/research/l4/performance.pml`.

[4] L. W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[5] J. C. Mogul and A. Borg. The Effect of Context Switches on Cache Performance. In *Proceedings of the fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, New York, NY, USA, 1991. ACM Press.

[6] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *USENIX Summer*, pages 247–256, 1990.

[7] J. Stärner and L. Asplund. Measuring the Cache Interference Cost in Preemptive Real-Time Systems. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 146–154, New York, NY, USA, 2004. ACM Press.

[8] Texas Instruments OMAP Platform. `http://focus.ti.com/omap/docs/omaphomepage.tsp`.

[9] A. Wiggins, H. Tuch, V. Uhlig, and G. Heiser. Implementation of Fast Address-Space Switching and TLB Sharing on the StrongARM Processor. In *8th Asia-Pacific Computer Systems Architecture Conference*, Sep 2003.