Quantifying The Cost of Context Switch*

Chuanpeng Li
Dept. of Computer Science
University of Rochester
cli@cs.rochester.edu

Chen Ding
Dept. of Computer Science
University of Rochester
cding@cs.rochester.edu

Kai Shen
Dept. of Computer Science
University of Rochester
kshen@cs.rochester.edu

ABSTRACT

Measuring the indirect cost of context switch is a challenging problem. In this paper, we show our results of experimentally quantifying the indirect cost of context switch using a synthetic workload. Specifically, we measure the impact of program data size and access stride on context switch cost. We also demonstrate the potential impact of OS background interrupt handling on the measurement accuracy. Such impact can be alleviated by using a multi-processor system on which one processor is employed for context switch measurement while the other runs OS background tasks.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Measurements*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques*

General Terms

Experimentation, Measurement, Performance

Keywords

Context switch, Cache interference

1. INTRODUCTION

For a multitasking system, context switch refers to the switching of the CPU from one process or thread to another. Context switch makes multitasking possible. At the same time, it causes unavoidable system overhead.

The cost of context switch may come from several aspects. The processor registers need to be saved and restored, the OS kernel code (scheduler) must execute, the TLB entries need to be reloaded, and processor pipeline must be flushed [2]. These costs are directly associated with almost every context switch in a multitasking system. We call them direct costs. In addition, context switch leads to cache sharing between multiple processes, which may result in performance degradation. This cost varies for different workloads with different memory access behaviors and for different architectures. We call it cache interference cost or indirect cost of context switch.

The report now proceeds to describe the approach and result of our measurement on the direct and indirect cost of context switching.

2. THE MEASUREMENT APPROACH

In previous work, Ousterhout measured the direct cost of context switch using a benchmark with two processes communicating via two pipes [6]. McVoy *et al.* measured the cost of context switch between multiple processes using lmbench [4]. Based on these traditional methods, we also used pipe communication to implement frequent context switches between two processes. We first measure the direct time cost per context switch (c_1) using Ousterhout's method where processes make no data access. Then we measure the total time cost per context switch (c_2) when each process allocates and accesses an array of floating-point numbers. Note that the total cost (c_2) includes the indirect cost of restoring cache state. The indirect cost is estimated as $c_2 - c_1$.

2.1 The direct cost per context switch (c_1)

Following Ousterhout's method, we have two processes repeatedly sending a single-byte message to each other via two pipes. During each round-trip communication between the processes, there are two context switches, plus one *read* and one *write* system call in each process. We measure the time cost of 10,000 round-trip communications (t_1).

We use a single process simulating two processes' communication by sending a single-byte message to itself via one pipe. Each round-trip of the simulated communication includes only one *read* and one *write* system call. Therefore, the simulation process does half of the work as the two communicating processes do except for context switches. We measure the time cost of 10,000 simulated round-trip communications (t_2), which include no context switch cost. We get the direct time cost per context switch as $c_1 = t_1/20000 - t_2/10000$.

2.2 The total cost per context switch (c_2)

The control flow of this test program is similar to that of section 2.1. However, after each process becomes runnable, it will access an array of floating-point numbers before it

*This work was supported in part by the National Science Foundation (NSF) grants CCF-0448413, CNS-0509270, CNS-0615045, and CCF-0621472.

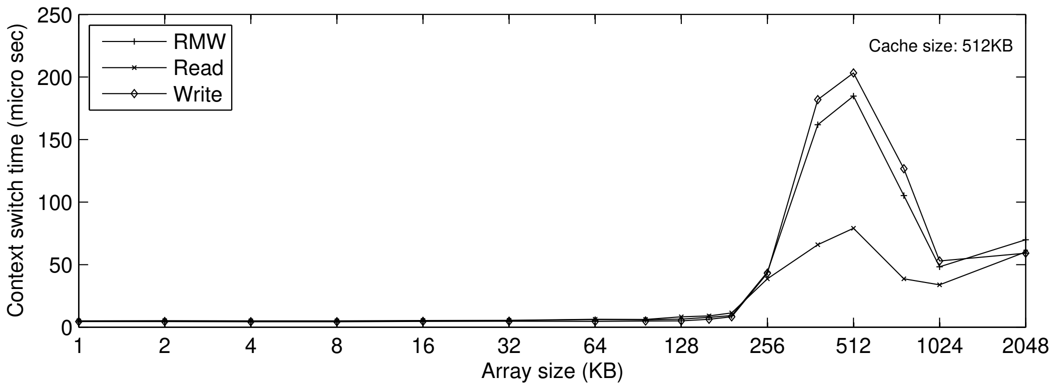

Figure 1: The effect of data size on the cost of the context switch

writes a message to the other process and then blocks on the next read operation. We still have a single process simulating the two processes' behavior except for context switches. The simulation process will do the same amount of array accesses as each of the two communicating processes. Assuming the execution time of 10,000 round-trip communications between the two test processes is (s_1) and the execution time of 10,000 simulated round-trip communications (s_2), we get the total time cost per context switch as $c_2 = s_1/20000 - s_2/10000$.

We change the following two parameters during different runs of our test.

- *Array size*: the total data accessed by each process.
- *Access stride*: the size of the strided access.

2.3 Avoid potential interference

To avoid potential interference from background interrupt handling of the OS or from other processes in the system, we use a dual-processor machine for our experiment. Since most OS interrupt handlers are bound to one default processor, we assign the communicating processes and the simulation process in our experiment to the other processor. We also set up our test processes with *real-time* scheduling policy SCHED_FIFO and give them the maximum priority. Presumably, most OS routine tasks and unexpected event handling will not interfere with our measurement. And no other process can preempt any of our test processes as long as our process is runnable. Linux system calls *sched_setaffinity()* and *sched_setscheduler()* are used to effect the design.

2.4 Time measurement

The timer we use is a high resolution timer that relies on a counting register in the CPU. It measures the time by counting the number of cycles the CPU has gone through since startup. When the time length of the measured event is extremely short, the overhead of the timer itself may cause some error. Therefore, we measure the cost of a large number (20,000) of context switches and then report the average cost.

3. EXPERIMENTAL RESULTS

The machine we use is an IBM eServer with dual 2.0 GHz Intel Pentium Xeon CPUs. Each processor has 512KB L2 cache and the cache line size is 128B. The operating system is Linux 2.6.17 kernel with Redhat 9. The compiler is gcc 3.2.2. We do not use any optimization option for compilation. Our source code can be found at *http://www.cs.rochester.edu/u/cli/research/switch.htm*.

The average direct context switch cost (c_1) in our system is 3.8 microsecond. The results shown below are about the total cost per context switch (c_2). In general, c_2 ranges from several microseconds to more than one thousand microseconds. The indirect context switch cost can be estimated as $c_2 - c_1$.

In the following subsections, we first discuss effects of data size and access stride on the cost of context switch. Then we discuss the effect of experimental environment on measurement accuracy.

3.1 Effect of data size

We show the effect of data size on the cost of context switch in figure 1. In this experiment, each process sequentially traverses an array of size d (in byte) between context switches. Each process does a read, write, or read-modify-write (RMW) operation on each array element.

With the increment of the array size d, the three curves in figure 1 show similar changing patterns. Each curve falls into three regions. The first region ranges from array size 1KB to array size about 200KB. In this region, all the curves are relatively flat, with context switch times ranging from $4.2\mu s$ to $8.7\mu s$. This is because the entire dataset of our benchmark (including the two communicating processes and the simulation process) can fit into the L2 cache, and the context switch does not cause any visible cache interference.

The second region ranges from array size 256KB to array size 512KB. Because the dataset of the simulation process fits in L2 cache but the combined dataset of the two communicating processes does not, the cost of context switch increases dramatically, from $38.6\mu s$ to $203.2\mu s$, with the increment of array size. Upon each context switch, the newly scheduled process need to refill the L2 cache with its own data. We believe the additional cost is due to frequent cache warm-ups. At array size 384KB, the cost starts to differ significantly depending on the type of data access. Since the test program incurs cache misses on contiguous memory locations, the execution time is mostly bounded by the memory bandwidth. Data writes consume twice the bandwidth,

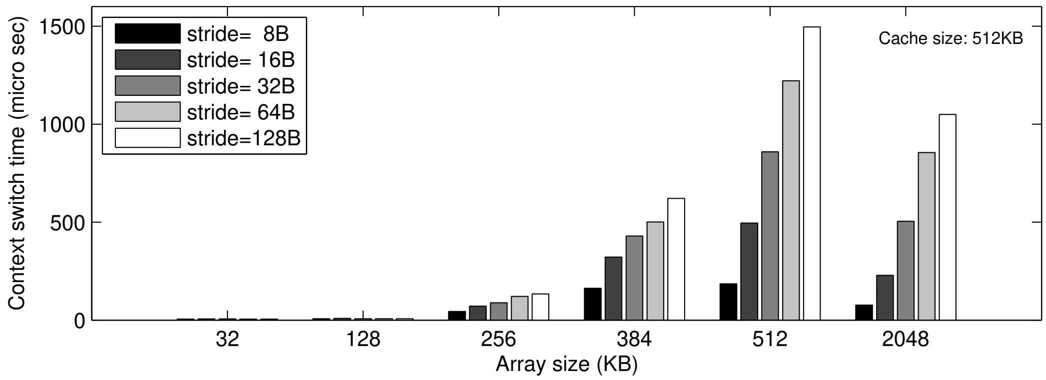

Figure 2: The effect of the access stride on the cost of context switch

hence the cases of RMW and write showing twice the cost as the case of read at the next three array sizes.

The third region starts from the array size 512KB. Here the dataset for both the communicating processes and the simulation process is larger than the size of L2 cache. The cost of context switch is still high compared to the first region, showing the presence of cache interference. But the curves do not increase monotonously with the array size. This is because context switch is not the only reason for cache misses any more. Since the dataset of each process is too large to fit in cache, cache misses will happen even when there is no context switch.

These results show the nonlinear effect of cache sharing. One may question whether it is proper to count this as part of the indirect cost of a context switch, because contention to the cache resource also happens when multiple processors share the same cache, regardless whether they incur context switches or not. However, we note that the overhead for a time-shared cache is not the same as that for a concurrently shared cache. Take the simple example of two concurrent processes writing to the same data block. The cost of their cache interference at each context switch is the re-loading of the cache block, which is very different from the cost of parallel access. In general, the interference manifests as cache warm-ups in the case of context switch. A number of past studies have examined this cost in detail including the relation with the cache size and other parameters, the workload, and the length of CPU quanta [5, 2, 7, 1]. Further work to compare time-shared and concurrently shared caches would be interesting.

3.2 Effect of access stride

We show the effect of access stride on the cost of context switch in figure 2. In this experiment, each process accesses an array of floating-point number numbers in a strided pattern. Suppose the access stride size is s. Starting from the first element, it accesses every s-th element. Then starting from the second element, it accesses every next s-th element. The process repeats striding until every element of the array is accessed. We show the array access behavior in the following code.

```
for (i=0; i<s; i++)
    for (j=i; j<array_size; j=j+s)
        array[j]++;
```

If s is 1, the access pattern is actually sequential. Each process does a read-modify-write operation on each element of the array. We show results on arrays of size between 32KB and 2MB, which include arrays from all the three regions described in section 3.1.

For array size of 32KB and 128 KB, since datasets can fit into cache, there is not much difference for the cost of context switch when we change the access stride. However, when the datasets do not fit into cache, the cost of context switch increases significantly with the increment of the stride size. When the access stride is 8B, the cost ranges between 44.1μs and 183.8μs with the mean 116.5μs. When the access stride is 128B, the cost ranges between 133.8μs and 1496.1μs with the mean 825.3μs. This substantial difference in the cost of context switch is caused only by the increase in the access stride. In other words, the data access pattern can affect the cost of context switch significantly. The reason is that the stride affects the cost of cache warm-ups in a similar way it affects program running time. For contiguous memory access, the hardware prefetching works well, and the cost is relatively lower than in the case of strided access.

3.3 Effect of experimental environment on measurement accuracy

All the above results are measured on a dual-processor machine. According to our design, the two communicating processes are bound to the same processor and they have the maximum real-time priority. This design aims to avoid the interference from background interrupt handling or from other processes in the system. We call it the *augmented* design for interfered measurement environment.

We evaluate the effectiveness of this design by comparing the execution time of the two-process communication program in the following experimental settings.

- *dual-processor:* The program with our augmented design runs in the dual-processor environment as described in our experiment.
- *single-processor:* The program without our augmented design runs in the single-processor environment we create on the same machine by disabling multiple-processor support in the Linux kernel.

We run the programs with 3 different array size in both settings for 6 times. Assuming the measured results follow a normal distribution, we report the 90% confidence

Array size	dual-processor	single-processor
256KB	(242.07, 246.46)	(221.00, 226.35)
384KB	(462.78, 474.43)	(461.80, 474.40)
512KB	(614.68, 629.87)	(614.69, 634.78)

Table 1: Confidence intervals of the execution time in a quiet environment

Array size	dual-processor	single-processor
256KB	(237.45, 245.54)	(220.72, 263.79)
384KB	(459.38, 470.03)	(510.91, 555.67)
512KB	(623.28, 630.77)	(635.75, 683.84)

Table 2: Confidence intervals of the execution time in the presence of external (network) interference

intervals [3] for the mean of each test. Generally, when the confidence interval is wide, the results are unstable.

When the machine has nothing else to run and there is no outside interference, we say it is in a quiet environment. Table 1 reports confidence intervals of six tests in such a environment. We can see the width of each confidence interval in the dual-processor setting is similar to the width of the corresponding confidence interval in the single-processor setting. This means both settings generate relatively stable results. The confidence interval boundary values for the two settings are a little different. This is because the Linux kernel scheduler code for the two settings is not the same. Remember that single-processor has the multiple-processor support disabled in the Linux kernel.

However, in reality, we can not always guarantee our experimental environment is not interfered. Thus, we simulate outside interference by sending "ping" packets to the test machine with a variable waiting intervals (between 0 and 200 milliseconds). We show the confidence intervals in the interfered environment in table 2. The confidence intervals obtained from single-processor is much larger than the intervals from dual-processor. This shows the instability of the results of single-processor in the interfered environment. Comparing the results of single-processor in table 2 to the corresponding results in table 1, we can see the measurement inaccuracy of the single-processor setting in the interfered environment.

4. SUMMARY

We summarize our observations from the experiment as follows.

- In general, the indirect cost of context switch ranges from several microseconds to more than one thousand microseconds for our workload.
- When the overall data size is larger than cache size, the overhead of refilling of L2 cache have substantial impact on the cost of context switch. The cost increases monotonously when the data size increases in some cases as described in section 3.1.
- When the overall data size is larger than cache size, the effect of access stride on the cost of context switch is significant. The larger the stride is, the larger the cost of context switch is.
- Experimental environment may affect the measurement accuracy. Our suggested augmented design can help to avoid the interference from background interrupt handling or from other processes in the system.

5. ACKNOWLEDGMENTS

We wish to thank Linlin Chen for demonstrating how to to use a statistical analysis package and Trishul Chilimbi for pointing out a related paper in ACM TOCS. We thank the anonymous reviewers of the ExpCS workshop and our colleagues at Rochester in particular Michael Huang and Michael Scott for their comments on the work and the presentation.

6. REFERENCES

[1] Anant Agarwal, John L. Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, 1988.

[2] R. Fromm and N. Treuhaft. Revisiting the cache interference costs of context switching. http://citeseer.ist.psu.edu/252861.html.

[3] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. John Wiley & Sons, 2001.

[4] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *In Proc. of the USENIX Annual Technical Conference*, pages 279–294, San Diego, CA, January 1996.

[5] J. C. Mogul and A. Borg. The Effect of Context Switches on Cache Performance. In *In Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, CA, April 1991.

[6] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast As Hardware ? In *In Proc. of the USENIX Summer Conference*, pages 247–256, Anaheim, CA, June 1990.

[7] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the International Conference on Supercomputing*, pages 1–12, 2001.