Deep Dive into the Cost of Context Switch

Shibo Chen, Yu Wu, Xinyun Jiang, Wen-Jye Hu
EECS Department, University of Michigan

Introduction

Today's CPU and operating systems are good at coping with latency at nanosecond and millisecond scale, while modern high-performance networking and flash I/O often have microsecond scale data access latency. Neither hardware nor software can provide efficient mechanisms to hide these latency. One of the conventional ways to hide millisecond-level latency is context switching, thus it is appealing to use the same technique to hide microsecond scale latency.
This project aims at measuring and analyzing the overhead of the kernel-level context switch under various working sets and data access patterns with comparison to user-level context switch. L1 cache misses and dTLB misses are identified as key components contributing to the overhead of kernel-level context switch.

Methodology & Experiment Setup

We referred to the approach proposed in Li's work [1] as our starting point. The general idea is that we first measure the average execution time of one unit of work and then measure the average execution time of one unit of work plus the overhead of context switch.
To do so, we first create a single thread traversing the array of size *M* bytes by stride size *S* bytes for *N* times, as shown in algorithm 1. Then we create two threads, each of which traverses the array of size *M* bytes by stride size *S* bytes in the same manner shown in algorithm 1. After traversing the array, it will switch to the second thread to continue the work. Figure 1 illustrates the context switch between two threads, where the threads take turns to execute. When one thread is stalled due to system call in kernel-level or *yield()* function called in user-level, the thread relinquishes the resource and allow the other thread to execute.

Algorithm 1 Traverse Array

```
for (i = 0; i < Stride_Size; i++) do
    for (j = i; j < Array_Size; j+= i) do
        ARRAY[j] ← ARRAY[j] + 1
    end for
end for
```

In the case of user-level context switch, we use *gtthread* library to implement a user-level context switch between two threads without using kernel-level libraries.
In order to rule out undesired factors affecting the results, multiple efforts and sanity checks have been made.
Experiments were run on Intel Xeon D-1541.

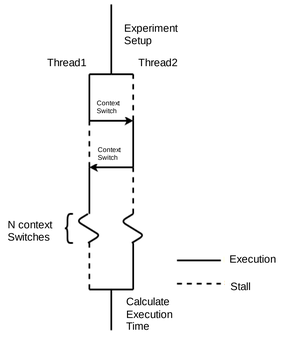

Figure 1: Context switch between two threads

Results and Discussion

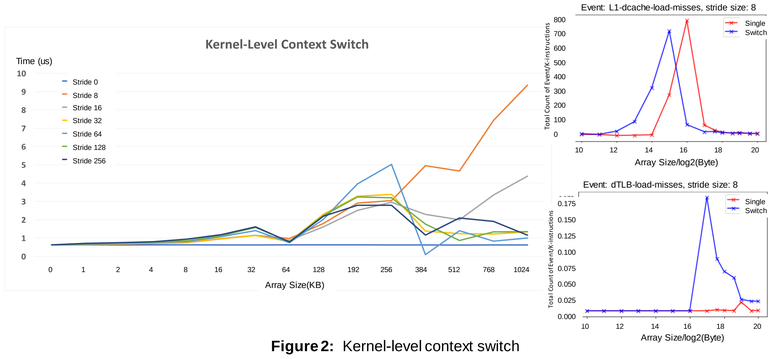

Figure 2: Kernel-level context switch

The overhead of the kernel-level context switch is shown in figure 2. The direct cost is 0.62µs. When the stride size is larger than 0, the overhead is composed of both direct cost and indirect cost induced by resource contention. The first local maximum appears when array size is 32KB, which equals to L1 cache capacity. And it then decreases because that the dCache load miss rate of single-thread case exceeds that of the two-thread switching case. This diminishing of the dCache load misses suggests that there might be a better cache replacement or prefetch policy coming into play and the data locality is better exploited when L2 gets involved. The pattern, when the array size is larger than 64KB, conforms to the dTLB load miss rate, which dominants the overhead, because dTLB misses result in larger penalty than cache misses.

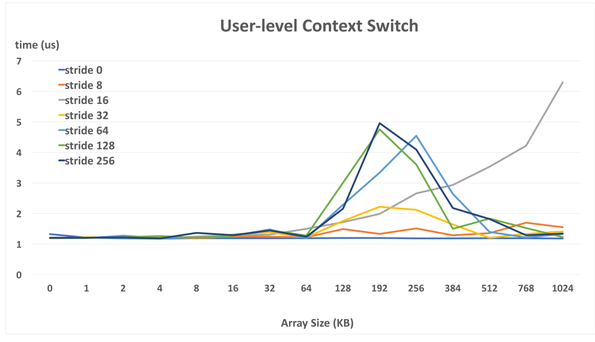

Figure 3: User-level context switch

Figure 3 shows the overhead of the user-level context switch. When the array size is smaller than 64KB, the overhead stays between 1.2 and 1.4 µs. The rest of the overhead pattern is similar to that of the kernel-level context switch.
Figure 4a and 4b compare the L1 iCache load misses in kernel-level context switch and that in user-level context switch. Due to the inclusion of the L2 cache, instructions can be evicted from L1 iCache when they got replaced in the L2 cache. Thus the number of iCache misses increases along with the array size. There are gaps between single-thread process and two-thread switching process in both figure 4a and 4b, and the kernel-level case has larger gap because that the OS has a more complex scheduling algorithm thus leaving a larger memory footprints in both iCache and dCache. The figure 4c shows a large gap in dTLB load misses, which is because that two kernel threads have two different virtual address spaces, and together they occupy more pages and dTLB entries than the single thread process.

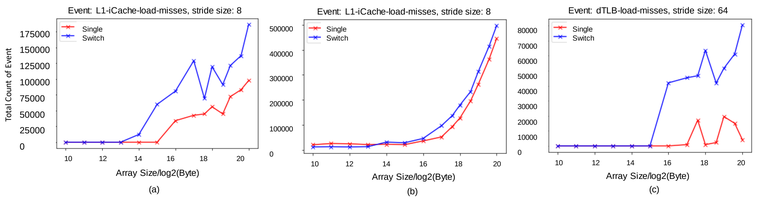

Figure 4: Perf results (a) kernel-level threads iCache load misses (b) user-level threads iCache load misses (c) kernel-level threads dTLB load misses

Conclusions

We measure the context switch overhead under different working set sizes and memory access stride sizes. In kernel level experiment, OS controls context switch with several system calls to ping-pong messages between two threads; in user level experiment, we go through the same flow as kernel level does. Our results show the user-level context switch time is larger than that of kernel level due to non-customized *gtthread* library but has less variance. The peaks in kernel-level context switch are due to L1 dCache load miss and dTLB load miss. We show not only the cache and TLB misses, but also cache inclusion policy contribute to the most part of the indirect cost of context switch. To reduce the context switch time, enabling better cache policies and enlarging hardware size of TLB or page size can be good solutions.

Acknowledgement

We would like to thank Prof. Thomas Wenisch and Prof. Todd Austin who helped us with great suggestions and experiment equipment throughout the project.

References

[1] Li Chuanpeng, Chen Ding, and Kai Shen. "Quantifying the cost of context switch." Proceedings of the 2007 workshop on Experimental computer science. ACM, 2007.