Deep Dive Into the Cost of Context Switch

Shibo Chen, Yu Wu, Xinyun Jiang, Wen-Jye Hu
{chshibo, wuyumay, xinyunj, huwenjye}@umich.edu

***Abstract*—Although modern operating systems and architectures have already made it easy for programmers to hide the nanosecond and millisecond scale latency resulting from conventional memory and storage devices, there are very few software or hardware techniques that are effective in hiding the latency on microsecond level. Microsecond scale latency has become increasingly common in data centers and warehouse-scale computing environment. Inability to hide such latency leads to low resource utilization and performance degradation. Since the "killer-microsecond" problem gained widespread attention, a number of possible solutions have been proposed to hide microsecond scale latency, and enabling fast context switching is a noteworthy example. However, the conventional operating system managed context switching mechanism has significant overhead and is difficult to optimize, which gives us the motivation to find out the bottleneck of the context switch.**

In this study, we measured the cost of both the kernel-level and user-level context switch, and we compared them to show the differences. By analyzing the results in great length, we provide explanations about the components of the overhead and also the insight into the bottleneck of context switch.

***Index Terms*—context switch, kernel-level thread, user-level thread**

I. Introduction

Today's CPU and operating systems are very good at dealing with latency at nanoseconds and milliseconds 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. Previous works, such as Denelcor HEP15 [4] and Tera MTA computers [5], enable fast context switch by giving up single thread performance. Other works, like Erlang [6] and Go [7], propose using light-weighted threads and context switching in order to avoid high-cost operating system context switching, but such solutions fall back to conventional operating system context switching when dealing with I/O request. None of the aforementioned approaches is appealing in warehouse scale environment. It would be plausible to incorporate modifications into either hardware or software to speed up context switch. However, even though context switch is well understand on software level, its implications on hardware-level events are not fully understand, which makes finding a low-cost and universal solution with high single thread performance impossible. Therefore, in this study, we propose to find the bottleneck of the speed of context switching in hardware design. Inspired by Li's work [1] which measured the indirect kernel-level context switch time back in 2007, we expect the kernel-level context switch time would be improved by a large amount due to the development of multi-core processors in the past ten years. Therefore, we would like to re-visit the bottleneck of context switch in modern processor and to explore potential opportunities for speed up. In addition, we are interested to see how user-level context switch performs compared to kernel-level one, thus specific `perf` hardware events, such as Instruction Cache (icache) misses and Translation Lookaside Buffer (TLB) misses, have been collected and profiled during the experiment. By profiling access patterns and characteristics during the context switch, we provide insights into the context switch and set up the stage for future works. Based on our observations, the average direct cost of the kernel-level context switch is $0.62\mu s$ and the average direct cost of user-level context switch time is $1.18\mu s$. Although the average total cost of the kernel-level context switch varies over different stride size and working array size, we are able to identify two components of the overhead in kernel-level context switch: (1) L1 cache misses, and (2) dTLB misses.

The paper is organized as follows: in section two, we explain the high-level ideas and implementation details of our measurements; in section three, we explain the experiment setups; in section four, we present our results and discuss the implications of the results; in section five, section six and section seven, we introduce the related works, conclude our findings and present the our contributions in this work.

II. Methodology

We referred to the approach proposed in Li's work [1] as our starting point. Since a context switch only takes a few microseconds on average, in order to statistically measure the overhead of one context switch, we need to measure a large number(N) of context switches and then divide the total overhead by N. Moreover, to make the measurement more realistic and closer to the production environment, we pad each context switch with one unit of work. 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 introduced by the context switch. To be more specific, we first measure the time t_{single} of a single thread sending message to itself for N times. Between each message communication, the thread would complete one unit of work. We then measure the execution time, t_{switch}, of two threads passing messages and switching between each other for N times. Between each message communication,

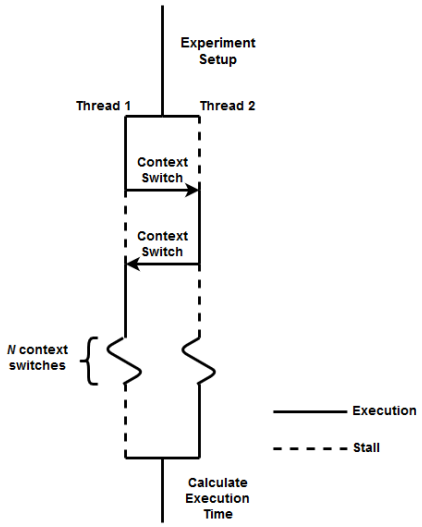


Fig. 1: Thread execution

each thread would do one unit of work and one context switch. The average time of context switch should be

$$t_{avg} = (t_{switch}/2 - t_{single})/N$$

where N is the total count of switching (100,000 in this 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.

To make sure that the two replicated threads are executed sequentially and in turn, we need to schedule them on the same virtual core. In order to do so, we restricted our program on one core by calling `sched_setaffinity()`. And with the help of real-time scheduling policy `SCHED_FIFO`, we granted our process with the highest scheduling priority and prevented them from being interrupted and swapped out by other threads.

The total cost of context switch consists of two parts. The first part, which is called direct cost, is contributed mainly by storing and restoring the processor states, flushing the pipeline, thread scheduling and executing the OS kernel code. The other part, which is called indirect cost, is resulted from the resource contention between multiple processes including but not limited to memory contention, TLB contention and BTB contention.

A. Kernel-level context switch

To measure the direct cost of context switch, we take the time difference between the two threads repeatedly sending messages to each other and one thread repeatedly sending messages to itself. The thread passes a message with size of 1 bit via `pipe`. The pipe has a read end and a write end. We first obtained the time for single-thread message passing. By calling `write()` and then `read()` the main thread performs one round of message passing. To obtain the time for two threads sending messages to each other, a child process is first created from main process via `fork()`. Two pipes were created where one thread writes in $\texttt{pipe}_1$ and reads in $\texttt{pipe}_2$ while the other thread does in opposite. Read from a pipe cannot be performed unless there is a message written to the pipe. This triggers context switch between the two threads.

To measure the indirect cost of context switch, an array traversal is performed between each round of message passing. To measure the cost under different access pattern, we traverse the array in different stride size. Figure 2 shows an example of traversing a 32 byte array in stride size 2 or 4. Algorithm 1 shows the the code we use to traverse the array.

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
```

B. User-level context switch

To show the impact of using system call, we implemented a user-mode program to compare its performance with that of the kernel-mode program, using an open source user level thread library `gtthread` [2], which has similar semantics as `pthread` library. The `gtthread_create()` function handles heap allocation for parent thread and stack allocation for the new context. The scheduling of the created user-level threads are implemented using a queue. When a thread calls `gtthread_yield()`, the calling thread will be put at the tail of the queue, letting the ready thread at the head of the queue start executing. We made some necessary modification to the library in order to imitate the scenario in our kernel-level process. For each thread, it also walks through an array before sending message as shown in algorithm 1. We need to implement synchronization and message exchanging functionalities similar to the program behaviors we achieved by using `read()` and `write()` pair. However, we do not want to hand the control over to the kernel in order to avoid undesirable kernel-level context switch, thus we pass messages by doing atomic read and write to two shared memory location with the help of synchronization semantics implemented by `gtthread`. Our program acquires lock before reading and then releases the lock by calling the functions `gtthread_mutex_lock()` and `gtthread_mutex_unlock()` provided by the library. Under the circumstances that it finishes execution or cannot proceed, it releases the lock and calls `gtthread_yield()` to put itself at the tail of a queue of all the threads, and allow the thread at the head of the queue to proceed.

III. Experiment

We performed the simulation on a Xeon processor. Table I shows the detailed configuration of the server.

Naively running the program would lead to changing performance caused by unpredictable factors like dynamic frequency

Parameter	Value
Core	Xeon D-1541 (Broadwell)
Max Clock Frequency	2.1GHz
L1 iCache/dCache size	32kB
L1 iCache/dCache line size	64B
L2 iCache size	256KB
L2 iCache line size	64B
dTLB	4KB, 4-way, 64 entries

TABLE I: Server Configuration

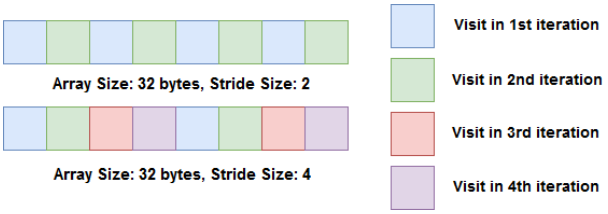


Fig. 2: Access pattern

scaling, temperature variation. For example, if the processor is running at a lower frequency when executing the single-thread program and dynamically switches to higher frequency when executing the multi-thread version because of more intense resource contention, we may observe a negative context switch cost due to the unfairness described beforehand. In order to rule out such undesirable factors, multiple efforts and sanity checks have been made during the experiment setup stage.

In order to limit the effect of dynamic frequency scheduling, we set the cpu scaling governor to *performance*. To minimize the influence of temperature and other resources of interference, we sampled 10 iterations for each combination of stride and array size together as a group and ran 20 groups of experiments. We took the average of these 200 samples to improve the reliability of our results. To make sure that there is no unexpected extra context switch, for instance, non-voluntary context switching caused by OS when a single thread is passing message to itself and when two threads are passing message via system call, we verified the total number of context switching counted by OS in `/proc/<pid>/status` is consistent with our estimation.

To show the impact of accessing pattern, as figure 2 indicates, we ran experiments with stride size from 2^3 to 2^8. Stride size 0 is used to calculate the direct cost of context switch since we don't access the array between the message passing. Note that since we use type `double` as our base element, incrementing the array index by one means we jump over 8 bytes. And we also compared the performance with different array size from 2^{10} to 2^{20} bytes.

IV. Results

We first measured the overhead induced by kernel-level context switches. The overhead of the kernel-level context switches over the size of the working data set is shown in figure 3. When the stride size is 0 byte, the overhead is the direct cost of the context switch. The direct cost of one context switch is $0.62\mu s$. When the stride size is larger than 0 byte, the overhead is composed of direct cost of the context switch and indirect cost of the context switch induced by resource contention. When the stride size is larger than 0 byte, the kernel-level context switch overhead increases along with the array size when the array size is smaller than 32KB. After hitting the local maximum, about $1.5\mu s$, when the array size is 32KB, the overhead decreases as we increase the array size to 64KB. This pattern conforms to the the pattern of dcache miss rate. Figure 4a shows the number of dCache load misses per thousand instructions. At array size of 32KB, which equals L1 cache capacity, the rate of dCache load misses in context-switch case is larger than that of single-thread case. However, when the array size increases to 64KB, the situation reverses. This alternation in the rate of dCache load misses could contribute to the local maximum of context switch time in figure 3 at array size of 32KB.

When the array size is larger than 64KB, context switch overhead begins to increase again until it hits the global maximum at array size of 256KB. At array size 256KB, the context switch overhead is around $3\mu s$ for stride size of 8 byte to 256 byte, except for stride size of 64 byte whose maximum context switch overhead is $5\mu s$, significantly greater than the others. As we increase the array size beyond 256KB, the overhead, when the stride size is greater than 16 byte, decreases to around $1.5\mu s$ and stays stable. This pattern conforms to the pattern of dTLB load miss rate. Figure 4b shows the number of dTLB load misses per thousand instructions. When array size is larger than 64KB, the rate of dTLB load misses of context-switch case is much higher than that of single-thread case. Though the rate of dTLB load misses is smaller than the rate of dCache load misses, dTLB load misses can result in a larger latency than dCache load misses. So as the rate of dTLB load misses increases, the context switch time in figure 3 increases accordingly. When the rate of dTLB load misses in context-switch case decreases, the context switch time in figure 3 also decreases.

Noticeably, the overhead has an increasing tendency when the stride size is 8 byte or 16 byte. At this point, we still cannot figure out any reasonable explanation.

The local maximum of context switch time occurred at L1 cache capacity shown in figure 3 and then diminished suggested that this might be due to a better cache replacement algorithm or cache line locality improvement when L2 cache is involved. Therefore this phenomenon might be exploited and researched on to help eliminate context switch time.

Furthermore, using larger page or increasing the TLB entries would also help to mitigate the context switch overhead when the working data size is small. Using `pid` in TLB to avoid flushing TLB entries entirely during the context switch may also help, even though it may increase the TLB hit latency.

Then, we measured the overhead using user-level context switch library. The results are shown in figure 5. As we can see, the context switch overhead of user-level context switch is constant and relatively stable at about $1.2\mu s$ comparing to the kernel-level context switch overhead when the array size is smaller than 64KB. As we increase the array size over 64KB,

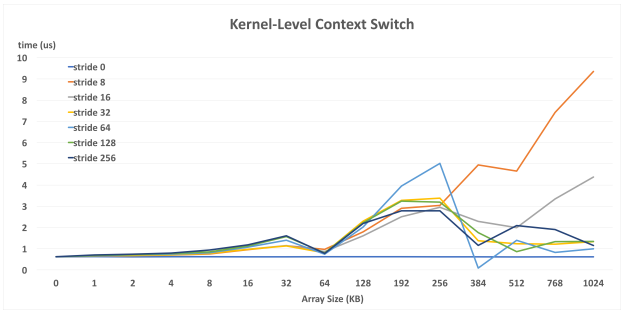
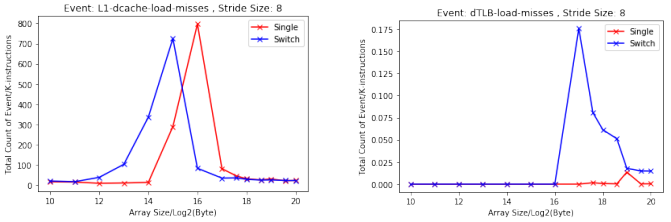


Fig. 3: Kernel level context switch time



(a) dCache load miss per 1000 instructions, stride size=8

(b) dTLB load miss per 1000 instructions, stride size=8

Fig. 4: Kernel level load misses per 1000 instructions in (a) dCache, (b) dTLB

the overhead increases along with the array size. When the stride size is 128 byte, 256 byte or 32 byte, the overhead reaches its maximum ($5\mu s$, $4.8\mu s$, $2.2\mu s$ respectively) at array size of 192 KB. When the stride size is 64 byte or 8 byte, the overhead hits its maximum ($4.5\mu s$, $1.4\mu s$ respectively) at array size of 256 KB. After hitting the maximum, the overhead of all stride sizes would decrease to around $1\mu s$ to 2 μs. The only exception is when the stride size is 16 byte. When the stride size is 16 byte, the overhead increases monotonically.

The analysis of the following sections are based on results obtained from `perf`. Section *A* discussed the impact of load misses incurred in L1 instruction cache and L1 data cache respectively on both kernel-level and user-level context switch implementation. Section *B* discussed the impact of TLB misses and section *C* presents the compare and contrast between kernel-level and user-level context switch.

A. Impact of Cache Miss

1) L1 iCache Load Miss: Figure 6 shows the result of kernel-level iCache load misses with stride size 8, and 64. There is a gap in the number of iCache misses between single-thread case and context-switch case, although we have already normalized the counts based on the amount of work they do. We observed that there are non-zero iCache load misses in context-switch case when the array size is larger than 2^{14} bytes, which is half of the L1 cache size. In addition, the difference of count increases as the array size increases. Due to inclusive L2 cache and the fact that the total working set of context-switch case is twice as that of single-thread case, as more data is put into the L2 cache, instructions will be more

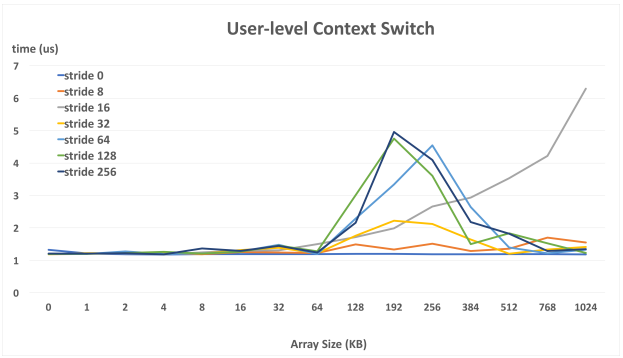
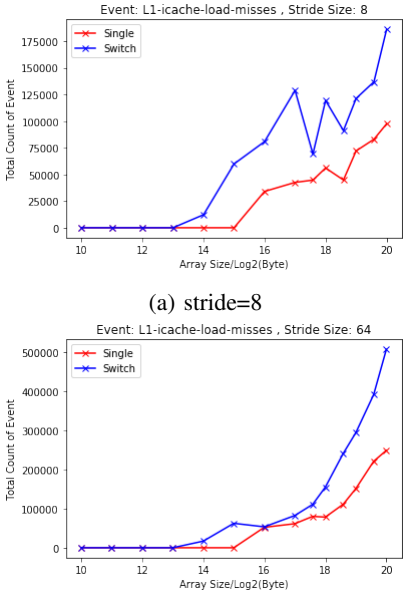


Fig. 5: User level context switch time

likely to be evicted out of L2 cache. In order to maintain the inclusiveness, the same piece of data would also be evicted from L1 iCache. Thus, it results in an increasing difference in L1 iCache misses.

Figure 7 shows the user-level iCache load miss patterns. Similarly, we can find that the context-switch case is still a little more than single-thread case. Again this might be due to the inclusive L2 cache. Since there is not much difference among different stride sizes, only stride size of 8 is present in this paper.

The reason for the gap being larger in the kernel-level context switch than in the user-level context switch may be ascribed to the fact that kernel-level context switch runs a more complex scheduling algorithm thus leaving a larger memory footprint in both dcache and icache.



(a) stride=8

(b) stride=64

Fig. 6: Kernel level L1-iCache load misses

2) L1 dCache Load Miss: Figure 8 and 9 illustrate the L1 dCache load misses in the kernel-level and user-level respectively. Since the access to dCache is limited by the

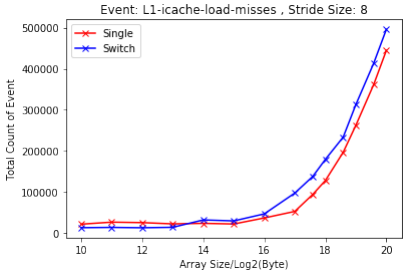
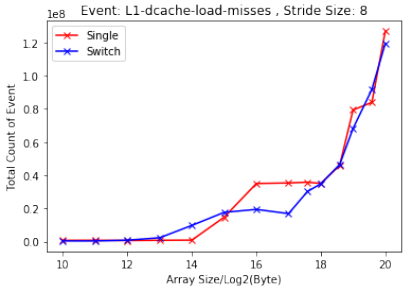
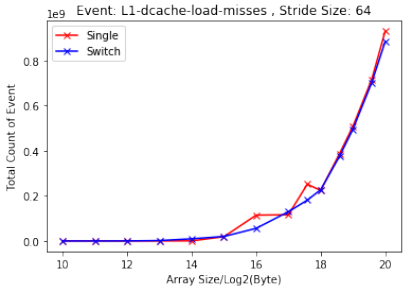


Fig. 7: User level L1-iCache load misses



(a) stride=8



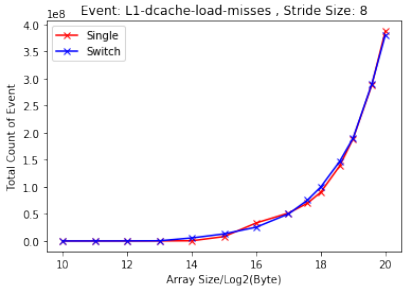
(b) stride=64

Fig. 8: Kernel level L1-dCache load misses

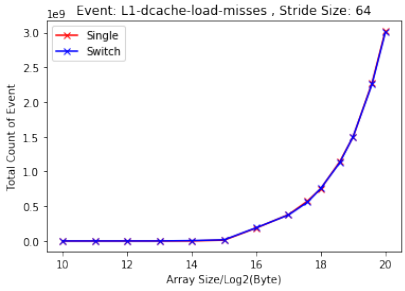
dCache cache line size, which is 64B, the dCache load misses for stride size 64 or more are approximately 10 times more than that of stride size of 8 (here only stride size of 64 is present). Furthermore, because the size of L1 dCache is 32KB, when the array size is smaller the capacity of L1 dCache, there is zero load misses in the single-thread case since the entire working set can be fit into the L1 dCache. This also applies to the context-switch case, but the boundary capacity is 16KB, half of the capacity of dCache due to the doubled overall working set. As the working set increases and goes beyond the boundary capacity (i.e., 32KB for single-thread case and 16KB for context-switch case), capacity conflict becomes an issue and the load misses increases for both cases.

B. Impact of dTLB Load Miss

Figure 10 shows the dTLB load misses in kernel-level case. The number of dTLB load miss for context switch case becomes non-zero when the working set is larger than 32KB. When the array size is smaller 32KB, the address space can perfectly fit into the dTLB. Thus, there is no need to load new



(a) stride=8



(b) stride=64

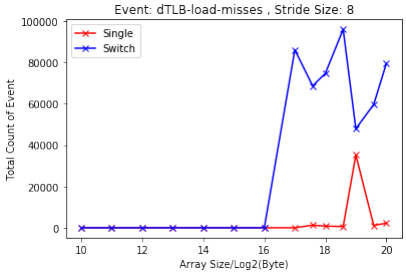
Fig. 9: User level L1-dCache load misses

physical address bases to map those virtual addresses when the array size is smaller than 32KB.

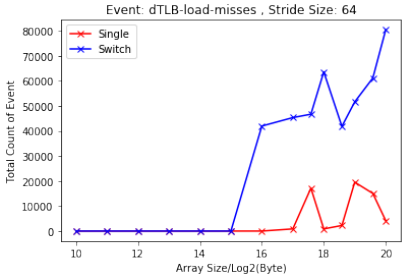
When the stride size is larger than 64 bytes, the number of dTLB load miss for context switch increases along with the stride size. When stride size is smaller than 64B, there is not much variation among different stride sizes. When the array size is larger than 32KB and fixed, the number of memory access during the array traversing is also fixed. The smaller the stride size (especially smaller than 64B), the more memory access can be supported by a single page mapping because of the spatial locality. Under this circumstance, dTLB does not need to load new physical address bases, because there is no new virtual address mapping request from dCache. It reduces the number of dTLB load miss as well.

The number of dTLB load misses for context-switch case is apparently larger than the single-thread case under kernel mode. Since we use `fork()` to create a child process to run the second thread, two threads are running in two different address space. Therefore there are increasing TLB contentions when the array size is large and thus consumes more pages. Moreover, when using the kernel-level context switch, we need to switch to the kernel mode, thus new TLB entries would be brought into the TLB and the TLB entries of the user-level program may be evicted due to conflict. It's even possible for the OS or the processor to flush the TLB entirely when switching between kernel mode and user mode depending on the implementation. However, this is not the case for user-level context switch.

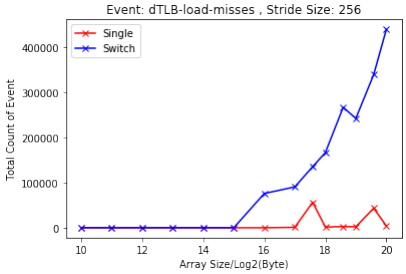
In the user-level experiment, we can find that there is only a slight difference between single-thread and context-switch cases. The patterns in figure 11 overlap a lot. There is more



(a) stride=8

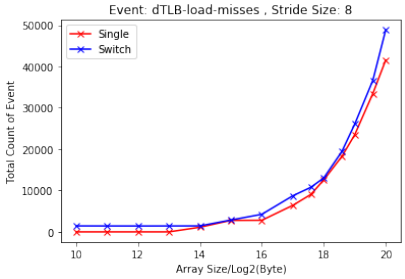


(b) stride=64

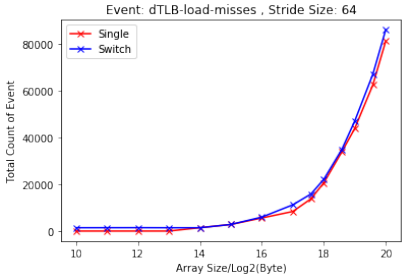


(c) stride=256

Fig. 10: Kernel level dTLB load misses

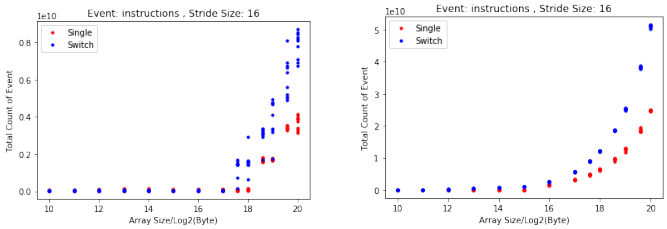


(a) stride=8



(b) stride=64

Fig. 11: User level dTLB load misses



(a) Kernel-level (b) User-level

Fig. 12: Instruction count

dTLB load miss in bigger stride size compared to smaller stride size, because the data might distribute in different physical pages. Traversing with a larger stride size leaps over to a different page more frequently than with a smaller stride size. The patterns of stride size bigger than 64B is similar to that of 64B.

To reduce the dTLB load misses, either a larger page size or more TLB entries can be possible solutions. A larger page size might reduce the probability of page fault under large stride size, while having more TLB entries enlarges the capacity of holding more mappings.

C. User-level v.s. Kernel-level context switch

The direct cost (i.e. no working set accessing) of user-level context switch of our experiment is about 1.18μs, while that of the kernel-level context switch is around 0.62μs, which is much faster. However, although our user-level implementation shows higher context switch overhead with small working set size, it has smaller variance than that of kernel-level cases as the working set size increases. As the `perf` results in Figure 12a indicate, the instruction counts of kernel-level process are different in each iteration even if the working set size and access stride size are fixed. In contrast, there is relatively less variance in the instruction counts of user-level context switch. A possible reason is that kernel-level threads might be stuck inside loops while waiting for a system call to return, and the duration of waiting depends on the implementation of kernel functions and is non-deterministic. This characteristic of kernel-level context switch makes it unfriendly to some tail-latency-sensitive applications that require good quality-of-service, while the latency of the context switch in user-level is more predictable and thus more suitable for this kind of applications.

Another observation from the instruction counts is that the user-level process executes more instructions than kernel-level one. It is because that we directly made use of the functions in `gtthread` library without adding any optimization, while the well-accepted opinion that user-level context switch occurs less overhead than kernel-level context switch is based on the fact that programmers have more knowledge about the behavior of their processes than OS so that they usually take advantage of it to hide the latency. And the thread scheduling algorithm adopted by the OS is usually very aggressive, which

also contributes to the difference of overhead between kernel-level and user-level cases shown in the Figure 12. Therefore, the main takeaway from this observation is that, in order to gain improvement in performance by implementing user-level context switching, non-trivial efforts is needed to add finer-grained scheduling optimization.

V. Related Work

In Cho's *Taming the killer microsecond*, they mention existing systems cannot hide microsecond scale latencies effectively. There exists some mechanisms which can solve the problem by replacing on-demand memory accesses with prefetch-based device access, which is followed by fast user-mode context switches. Increasing hardware queues, which tracks in-flight accesses, outperforms the application-managed software queue.

In Lis *Quantifying the cost of context switch*, the author collected the overhead of context switch with regard to the different size of working dataset and stride. However, this study has the following the limitations: 1) The study was conducted back in 2007, which may not sufficiently reflect the present context switch overhead. 2) The study is conducted with a fixed cache size and frequency, which may not reflect the best configuration for context switch workload. 3) The study only considered two threads. More threads would introduction more interference and maybe different overhead characteristics.

Multiple solutions to enable fast context switch have been proposed overtime, on both hardware level and software level. However, each of them has drawbacks which make them have limited appeal in warehouse-scale computing environment.

Smith proposed a pipelined, shared resource MIMD architecture in Denelcor HEP [4]. In a HEP processor, two queues are used to time-multiplex the process states. In this type of organization, skeleton processors compete for execution resources in either space or time. Such design gives up the locality advantage and single thread performance due to its nature of contention.

R. Alverson et al. proposed an architecture with 128 program counters and instruction streams in the Tera Computer System [5]. For each cycle, the scheduler would choose one instruction stream to execute. While it enables negligible context switch overhead, the cost of such system can be much greater than the conventional architecture. Moreover, the number of threads supported is limited to the number of hardware. Since, ideally, we want to support thousands of threads in a warehouse-scale computing environment, such design may not be appealing.

A lot of effort has been put into implementing light-weight threads (i.e. Go [7] and Erlang [6]). Light weight threads are usually light in two ways: 1) Light in context switching cost by avoiding switching to kernel level library so that the program can bypass TLB flush, memory paging etc. This approach is similar to the user-level multithreading library we used in this project. 2) Light in memory footprint. However, such approach has two major drawbacks: first, it usually falls back to conventional context switch approach when dealing with I/O events; second, bypassing system-level scheduling may stall other programs and lead to inefficiency.

VI. Conclusion

In this work, we measure the context switch overhead under different working set sizes and memory access stride sizes. In the kernel level experiment, OS controls the context switch work. It uses several system call functions such as `read()` and `write()` to ping-pong message between two threads. In the user level experiment, we go through the same flow as kernel level experiment. From the results, the context switch time of user level is larger than that of kernel level. Although the user level threads are supposed to be lightweight without calling OS functions, the scheduling mechanism is not optimized in our implementation because we directly adopted the functions in the `gtthread` library. However, since usually the programmers can have better knowledge about the events in the user level programs than in kernel level ones, the overhead in the user-level programs can be easily reduced. In addition, a more aggressive scheduling mechanism can be adopted to achieve further speedup.

There are two peaks demonstrated in the cost of kernel-level context switch plot. The peak at array size 32KB is due to L1-dCache load miss; the peak at array size 256KB is due to dTLB load miss, which results in even more latency penalty than L1-dCache load miss does. The cache inclusion policy also impacts on the context switch latency when the array size is at the edge of L1 cache size. There is a peak at array size 256KB in user-level context switch graph as well. We showed that the cache and TLB misses contribute to the most part of the indirect cost of context switch. By implementing user level threads, the percentage of cache load misses can be significantly reduced, since the TLB flush due to the switch of virtual address space can be avoided. Hardware solutions could be enlarging TLB or page size to reduce the context switch time.

VII. Contributions

The workloads in this project include:

- Background research
- Research and adapt kernel-level and user-level context switch codes
- Set up script and run simulations on Xeon server
- Gather and analyze the experiment statics

And we partitioned the workload equally.

Shibo Chen	25%
Yu Wu	25%
Xinyun Jiang	25%
Wen-Jye Hu	25%

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.

[2] LancelotGT. "GTThread–A User Level Thread Library", v1.0. (2019). Available: https://github.com/LancelotGT/gtthread

[3] John D. McCalpin. ”Notes on the mystery of hardware cache performance counters.” https://sites.utexas.edu/jdm4372/2013/07/14/notes-on-the-mystery-of-hardware-cache-performance-counters/. 2013.
[4] Smith, B. A pipelined shared-resource MIMD computer. Chapter in Advanced Computer Architecture. D.P. Agrawal, Ed. IEEE Computer Society Press, Los Alamitos, CA, 1986, 3941.
[5] Alverson, R. et al. The Tera computer system. In Proceedings of the Fourth International Conference on Supercomputing (Amsterdam, The Netherlands, June 1115). ACM Press, New York, 1990, 16.
[6] Erlang. Erlang User's Guide Version 8.0. Processes; http://erlang.org/doc/efficiency_guide/processes.html
[7] Golang.org. Effective Go. Goroutines; https://golang.org/doc/effective_go.html#goroutines