Quantifying the cost of a context switch

. 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.

1. Cache Basics:
- CPUs have multiple levels of cache (L1, L2, L3) that store recently used memory data
- These caches are shared resources between different processes
- When a context switch occurs, the new process may invalidate or push out existing cached data from the previous process
2. Performance Impact:
- When a new process starts running, it may need to reload data into the cache
- This can cause "cache thrashing" where cached data is constantly being replaced
- The new process might need to fetch data from slower main memory instead of the faster cache
- This leads to increased memory access latency and reduced overall system performance

We first measure the direct time cost per context switch (c1) using Ousterhout's method where processes make no data access. Then we measure the total time cost per context switch (c2) when each process allocates and accesses an array of floating-point numbers. Note that the total cost (c2) includes the indirect cost of restoring cache state. The indirect cost is estimated as c2 − c1.

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 direct cost per context switch (c1):

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 (t1).

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 (t2), which include

no context switch cost. We get the direct time cost per context switch as c1 = t1/20000 – t2/10000.

The subtraction logic:

- t1/20,000 represents the average time per operation with context switches
- t2/10,000 represents the base time for system calls without context switches
- By subtracting t2/10,000 from t1/20,000, you isolate the additional overhead caused by context switches

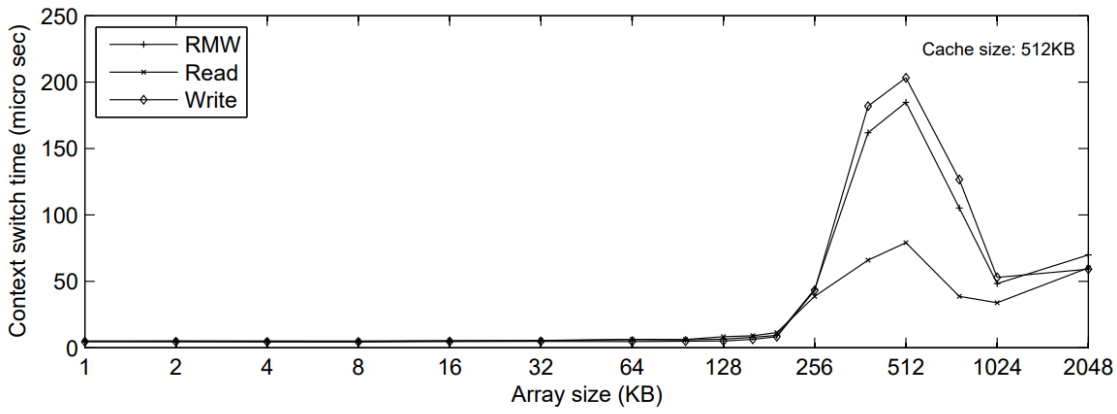
2.2 The total cost per context switch (c2)

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 writes a message to the other process and then blocks on the next read operation.

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.

2.4 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.

3.1 Effect of data size



. 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-modifywrite (RMW) operation on each array element.

As the array size increases, the context switch performance shows three distinct regions:

1. First Region (1KB to 200KB):
- Flat performance curve
- Dataset fits entirely in L2 cache
- Minimal context switch overhead (4.2-8.7µs)
- No significant cache interference
2. Second Region (256KB to 512KB):
- Dramatic increase in context switch cost (38.6µs to 203.2µs)
- Simulation process fits in L2 cache
- Combined communicating processes do not fit in L2 cache
- Requires frequent cache "warm-ups" during context switches

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.

Logic Behind Bandwidth Limitation:

- When a program repeatedly accesses contiguous memory locations
- If these accesses are larger than the cache can handle
- The program becomes "bandwidth-bound" rather than "compute-bound"

In the third region (array size >= 512KB), datasets exceed L2 cache size, leading to cache misses even without context switches. While context switch cost and cache interference remain high, the performance curves don't increase monotonously because the primary cause of cache misses is now the large dataset size, not solely context switching.

3.2 Effect of access stride

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.

Each process does a read-modify-write operation on each element of the array.

For small datasets that fit in the cache (32KB, 128KB), the access stride has little impact on context switch cost. However, for larger datasets that exceed cache capacity, increasing the access stride significantly raises the context switch cost. This is because a larger stride hinders hardware prefetching and increases the cost of cache warm-ups during context switches, similar to its effect on program runtime. Contiguous access (small stride) benefits from prefetching and thus has a lower context switch cost compared to strided access.

2. Strided Access:

- Memory is accessed with a fixed but non-sequential jump between accesses
- Hardware prefetchers struggle to predict the pattern
- Higher cache miss rates
- More significant context switch overhead
- Example: Accessing every 4th or 8th element in an array

Impact on Context Switch:

- Contiguous access:
    - Quick cache warm-up
    - Hardware can predict and pre-load cache lines
    - Minimal performance penalty during context switch
- Strided access:
    - Unpredictable memory access pattern
    - Hardware prefetchers less effective
    - More cache misses during context switch
    - Higher latency in restoring cache state
    - Increased cache warm-up time

Because contention to the cache resource also happens when multiple processors share the same cache, regardless whether they incur context switches or not.

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.

Time-Shared Cache:

- Processes take turns using the cache
- Only one process is actively using the cache at a given time
- When a context switch occurs, the new process may need to reload cache blocks
- Lower contention, as processes are isolated in time
- Overhead is primarily the cache block reloading during context switches
- Less complex synchronization requirements

Concurrently Shared Cache:

- Multiple processes are actively using the cache simultaneously
- Processes are running in parallel on different cores
- Simultaneous access to the same cache resources
- High potential for contention and interference
- Requires complex cache coherence protocols (like MESI)
- Parallel access can cause:
    - Cache line thrashing
    - Increased synchronization overhead
    - More frequent cache invalidations
    - Bandwidth contention between cores