Synchronization with SpinLocks, Atomic Update Primitives, and Transactional Memory

18-344: The Hardware / Software Interface

Carnegie Mellon University

In this lab you will learn how to use the advanced synchronization primitives provided by your machine. You will learn to go beyond using simply mutex locks and reader/writer locks for concurrency. Instead, you will learn how to use spinlocks, atomic update primitives, and hardware transactional memory.

There are two parts to the assignment. The first part is to implement and evaluate the performance of several variants of the "Giga-Updates Per Second" (GUPS) kernel, each of which uses different synchronization primitives. The second part is to implement and evalauate the performance of the Swap Within Array Partition Set (SWAPS) kernel.

GUPS

The GUPS kernel is a multi-threaded program that increments elements in a key-value store. Each thread loops for a pre-determined number of iterations, selecting a random element, and incrementing that element by one. The starter code that we have provided to you is *incorrect* because it does not use synchronization to increment the key-value store element. Instead, the increment is part of a data-race. The practical consequence of this data-race is that sometimes different threads' increments of the same key-value store element interleave, losing the effects of some increments.

Your task in this part is to implement the threads' updates using pthread_spinlock_t, __sync_fetch_and_add, __sync_bool_compare_and_swap, and Intel RTM hardware transactional memory. The provided starter code has some rudimentary argument processing code that allows you to select which variant of synchronization to use. You should implement the function body of each of the functions process_mut, process_fetchandadd, process_compexchg, and process_tm, which should synchronize the key-value store increments in the way indicated by the function name.

The command line interface to GUPS should be left alone. The arguments are positional:

$./gups

where entries is the size of the key-value store, threads is the number of threads to spawn, iters is the number of iterations each thread should execute, and variant is the synchronization strategy that the program should use:

0 -> unsynchronized (buggy!) 1 -> spinlocks 2 -> fetch_and_add 3 -> compare_and_swap 4 -> tm

GUPS evaluation: After implementing each of the variants of GUPS, you should run tests to collect data showing the relative performance of each of the variants. A good performance metric for this lab is run time, collected by using the `time` utility (The manpage for `time` explains its options and interpretation).

Plan and run an experiment that shows how performance of a variant scales with varying thread count. Run the experiment for a fixed number of iterations (a large enough number that your program runs for a few seconds at least), and for a fixed key-value store size (1000 is reasonable). Choose at least 4 different thread counts, varying from 2 up to 2x the number of hardware threads available on the machine. Be sure to collect multiple experimental trials for each thread/variant pair. You should present data that illustrate a performance scaling trend and explain the trend in your write up.

SWAPS

The SWAPS kernel is a multi-threaded irregular memory access kernel that transforms sub-spaces of an array. The program first selects a list of array elements to include in the sub-space to transform. The program then swaps each element in the sub-space with the element in the sub-space with the next lowest index in the array, modulo the array length. After shuffling an element down, SWAPS increments the element by one, marking its involvement in a transformation.

All of the swaps should be performed atomically together! If there are 10 swaps to perform, all 10 of them should happen atomically together. The goal is *not* to make each individual swap of one pair of elements atomic. All of the swaps should be atomic.

The key variables in a sub-space transformation are:

*array size *sub-space size (i.e., number of swaps) *number of threads *number of swaps per thread

The SWAPS starter code provided includes an unsynchronized variant that will produce incorrect results when run with multiple threads. Your task in this lab is to write two variants of SWAPS, one that uses pthread_spinlock_t synchronization and on that uses Intel RTM transactional memory synchronization.

The command line interface to SWAPS is very similar to GUPS

$./swaps <# threads>

Variant is which sync strategy to use.

0 -> unsynchronized (buggy!) 1 -> spinlocks 2 -> tm

SWAPS evaluation: After implementing each of the variants evaluate the performance of your SWAPS variants. You should plan and execute an experiment that measures how performance of each variant changes with thread count, sub-space size, and array size.

You can scale contention by increasing thread count, or by increasing the odds that threads will interact when accessing randomly chosen elements from your array . To increase those odds, try decreasing the vector size, or increasing the number of elements that each iteration accesses. Eventually, contention for elements in the array should be a significant part of run time, which might be interesting to study.

When experimenting for a particular parameter, hold other parameters constant (and reasonable -- e.g., array size=1000, thread count=#CPUs, sub-space size=10) to isolate the effect of the varied parameter. Show using data what happens to a variant's run time on average as you change different parameter's value. You should characterize your performance results by adding code to measure transaction commits, aborts, and any cases of interest (e.g., fallbacks). For each performance experiment, you should do these characterization measurements. Plots of the characterization data should support your conclusions about the performance scaling of your system.

Short Answers:

Your writeup should answer these questions and any others that you think will be interesting in explaining the behavior of your synchronization implementations.

*Why are we asking you to write a version of GUPS, but not SWAPS, that uses __sync_fetch_and_add or __sync_bool_compare_and_swap?

*Which variant of SWAPS scales best with thread count? sub-space size? array size? For each, why?

*How does the rate of transactional commits and aborts change as you vary array size, sub-space size, and thread count?

*How does your SWAPS evaluation measure the effect of contention?

*How does your SWAPS evaluation measure the amortization of synchronization overheads?

Evaluation notes:

Getting "clean" measurements of real time on a real machine may be difficult because of performance interference (e.g., between your experiments and other peoples' experiments). You will have to run repeated trials of each of your test conditions (e.g., each different sync primitive). You may want to plot averages of multiple trials and include error bars on your results that show a confidence interval, or just the standard error of your data.

If you are unable to collect meaningful timing data because of performance noise, you should do as thorough an evaluation, show the noisy data you are able to collect, report characterization data (e.g., number of aborts), and speculate about what the interference-free performance results would be.

The following sections are informative.

Spinlocks

Spinlocks are a variant of mutex lock. To acquire a spinlock (using `pthread_spin_lock`), a thread runs in a loop, checking whether a memory location that represents a lock shows that the lock is unheld. When the check reports that the lock is unheld, the thread atomically updates the location to record that the lock is again held (by the acquiring thread). To release a spinlock (using `pthread_spin_unlock`) A GNU glibc spinlock (`pthread_spinlock_t`) can be cast to an `int` and its state be referred to directly as the operand of a memory access. If two such direct accesses are concurrent, include at least one write, and are not otherwise synchronized, those direct accesses may constitute a data-race error; otherwise, such accesses are safe.

Lock Ordering Matters! You should be sure to always acquire locks in lexicographic/sorted order, if you have multiple locks to acquire. If you do not adhere to a locking discipline, you will eventually reach a deadlock.

__sync_fetch_and_add(type *val, int addand)

This intrinsic invokes a fetch-and-add using a locked exchange-add, i.e.,

__sync_fetch_and_add(val, 1)

corresponds to

lock addq $0x1,(%rax,%rdx,8)

(You can prove this to your self by disassembling your code using objdump -d ./gups. Use this intrinsic, compile, and disassemble; what do you see?)

The instruction adds 1 to the variable `var`, stored in memory at (%rax,%rdx,8). The lock prefix ensures that the fetch, addition, and store into val happen atomically and are well-ordered (i.e., are not a data-race).

__sync_bool_compare_and_swap(...)

This intrinsic invokes an atomic compare-and-swap operation using a locked compare-exchange instruction

The instruction compares the value in `mem` to `oldval` and if they are equal, replaces the value in mem with `newval`. The intrinsic uses the cmpxchg instruction to implement this behavior atomically, and in a well-ordered way (i.e., not a data-race).

__sync_bool_compare_and_swap(mem, oldval, newval)

corresponds to

lock cmpxchg %rcx,(%rdx)

(You can prove this to your self by disassembling your code using objdump -d ./gups. Use this intrinsic, compile, and disassemble; what do you see?)

Transactional Memory

Intel's "Reduced Transactional Memory" (RTM) is an atomicity primitive that allows defining a transactional region of your program that executes strongly atomically with respect to memory locations accessed. The RTM ISA extensions are wrapped in C functions in the tm.h header file, which you should read and understand. Here are the basics:

To start a transaction, you should execute:

_xbegin();

The _xbegin() wrapper is defined in tm.h and directly inlines the assembly for the xbegin ISA extension. _xbegin() stores the *transactional status* in %eax before it returns (i.e., the function returns this value). Our wrapper for _xbegin() also sets the *fallback instruction* to be the instruction after the _xbegin().

If _xbegin() returns zero (i.e., _XBEGIN_STARTED), a transaction has successfully begun and the transactional status is 0.

To end a transaction, you should execute:

_xend();

The _xend() wrapper is defined in tm.h and, like _xbegin(), directly inlines the assembly for the xend ISA extension.

_xend() does not return a value. If an _xend() is called from an executing transaction and the instruction completes its execution, then the transaction commits successfully and the effects of the transaction are made non-speculative and globally architecturally visible.

To manually abort an ongoing transaction, during that ongoing transaction you should execute:

_xabort(char code);

_xabort(code) aborts an ongoing transaction, and adds `code` to the transactional status.

One good reason to abort a transaction is that another thread is executing the lock-based fallback code for the transaction that you are running. You can detect this condition in a transaction by checking the value of the spinlock that you use to implement your lock-based fallback.

The glibc spinlock implementation (e.g., https://code.woboq.org/userspace/glibc/sysdeps/x86_64/nptl/pthread_spin_unlock.S.html) writes a constant 1 into the lock word to release a lock. If, in a transaction, you read the value of a spinlock and see anything except for a 1, you should probably abort the transaction because another thread is manipulating the data protected by the lock.

Aborted Transactions:

If a transaction aborts several things happen. First, the system restores all architectural register state to the state at the _xbegin(). Second, the system discards all memory updates performed in the aborting transaction. Third, the system updates %eax with a status code that may indicate the reason that the transaction is aborting. Fourth, the system sets the instruction pointer to the fallback instruction saved by the instruction's _xbegin().

If your code compares the result of an _xbegin() to _XBEGIN_STARTED at the start of the transaction, then the fallback path that executes on abort will be that same comparison. On the fallback path of an aborting transaction, a comparison between the transactional status returned by _xbegin() to _XBEGIN_STARTED will evaluate to false. If a transaction was explicitly aborted, the transactional status contains the constant argument passed to _xabort(). If a transaction aborts for another reason, the transactional status will be set to a condition-specific status code encoding several conditions; these statuses are defined at the top of tm.h.

_XABORT_EXPLICIT indicates an explicitly aborted transaction.

_XABORT_RETRY indicates that it is worth retrying your transaction (i.e., it may commit if you try again).

_XABORT_CONFLICT indicates that the abort is the result of an access conflict.

_XABORT_CAPACITY indicates that the abort is the result of the transaction exceeding the capacity of the system to buffer speculatively accessed, transactional state.

int _XABORT_CODE(status) extracts the status code passed into an explicitly aborted transaction's _xabort(char code) and returns that code.

_XABORT_NESTED indicates that a transaction (aborted or ongoing) was nested inside of another transaction. (You are unlikely to encounter this situation in the lab.)

Dealing with aborted transactions:

You should consider what your program should do in the event of different transactional abort situations. If a transaction repeatedly aborts, or is very unlikely ever to commit, the logic in your fallback path should fallback to a non-transactional alternative implementation (perhaps using locks). Be careful: you should be sure that the (potentially lock-based) code on your fallback path will correctly interact with the code on your transactional path. It is not uncommon to have to jump to a fallback path (or to jump past a fallback path) using a goto statement (because control flow in transactional code can get a little bit messy. For instance, in pseudocode:

for(max transaction attempts){ /*xbegin to start Tx*/ /*if status is _XBEGIN_STARTED*/ /*transaction body*/ /*xend to end transaction*/ /*got past xend: committed! goto: success*/ } /*fallback path -- you get here if you fall through the xend w/o goto: success*/

/*success: jump here if tx committed to skip fallback*/

If you are having a difficult time understanding your program's performance, you should consider tracking the number of successfully committed transactions and any interesting cases of aborted transactions that you think may help you understand.

Counters & Statistics Collection

You need to be careful with how you collect statistics. If you are keeping event counters for things like aborts and fallback executions (for Transactional Memory), you need to be sure to use some form of synchronization. You might want to use spin locks or __sync_fetch_and_add for your counters.

Synchronization on counters will affect the parallel performance of runs that you are timing. The performance effect of synchronization can be significant and you should probably not collect event counters during executions that you are using for comparative timing analysis.

Transactional Memory Support

Some machines support and some do not support transactional memory. The lab includes the has-tsx program that reads a CPU register containing configuration information and reports (as best as possible) whether your system supports the Intel TSX Transactional Memory extensions.

Support for TSX on the number cluster machines is not 100%, but many of these machines support TSX. We recommend that you find one or two machines that support TSX and try to use them consistently, especially when it comes to timing experiments. ece029-ece031 work for TSX at the time of writing.