Fine Grained Analysis of Read-Modify-Write Performance in NUMA Architectures

Naama Ben-David Ziv Scully

{nbendavi, zscully}@cs.cmu.edu

1 Overview

Read-Modify-Write is a widely used synchronization primitive. Many lock-free data structures base their design on a read-modify-write loop, in which each thread reads the current state of the data structure, locally modifies it, and then compare-and-swaps its new value. However, despite their ubiquity, the performance of such read-modify-write loops under varying degrees of contention is poorly understood.

In this project, we conduct a thorough experimental study of compare-and-swap based read-modify-write workloads. We design fine-grained benchmarks that allow collecting detailed information about the throughput and data movement patterns of the execution, while introducing minimal perturbation. Through our experiments, we uncover several unintuitive characteristics of such workloads in NUMA architectures. In particular, we show that it can be beneficial for threads to contend on their read-modify-write operations across node boundaries rather than keeping contention within their own node. Furthermore, we show how this effect changes as the amount of local modification work is varied. We also plot more detailed information, tracking the movement of the contended cache line among the different cores. We compare the behavior of an Intel machine and an AMD machine. Our goal for the project is to further understand what causes the throughput patterns that we observe, and be able to use this knowledge to gain insight into algorithm design.

2 Benchmarks and Architectures

2.1 Tested Architectures

We run our benchmarks on two different NUMA architectures. The first is an Intel Xeon machine with four NUMA nodes, each on a different socket. Each node has 18 cores, with two-way hyperthreading. The other machine is an AMD Bulldozer with four sockets, and two NUMA nodes per socket. Each node has 4 cores, again with two-way hyperthreading. The details of the two machines are summarized in Table 2.1.

	Sockets	Nodes	Cores	Coherence Protocol
Intel	4	4	72	MESIF
AMD	4	8	32	MOESI

Table 2.1: Machine Details

2.2 Benchmark Details

The goal of our benchmarks is to simulate various contended workloads that may appear in practice. At its core, our benchmark simply has all threads execute a CAS-based fetch-and-add on a single memory location for a given amount of time. We measure throughput; how many successful changes to the memory location were made.

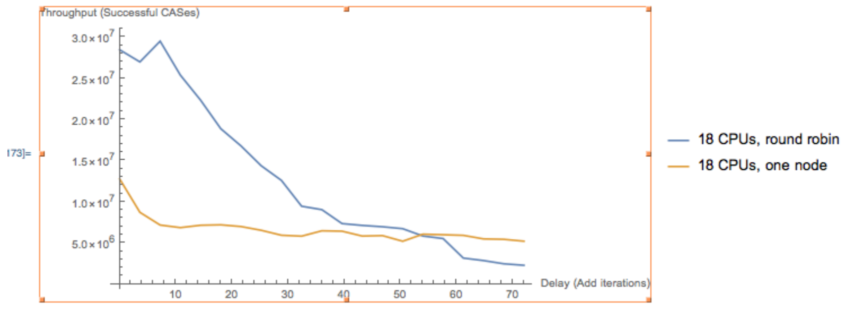
Our benchmarks allow varying several parameters: the number of threads contending and their location in the NUMA architecture, as well as number of memory locations accessed and on which nodes this memory is allocated. Furthermore, we allow injecting a given amount of delay between a thread's read operation and its subsequent CAS (called 'read-cas', or 'rc' delay), or between its CAS and its next read (called 'cr' delay). Using these parameters, we can simulate many different workloads.

In order to retain information about the execution, we have each thread CAS in its id and a timestamp. When a thread reads or executes a CAS on the contended location, it records in a locally allocated log the data that was stored in the shared location. This data is always the id and timestamp of the thread that executed the most recent successful CAS. This logging has low overhead - the execution with logging has only about 5% lower throughput than when running without logging. However, the resulting logs (one per thread) allow us to extract a lot of information about the run, as we discuss in Section 3. One downside of this detailed logging is that it takes a lot of space; every single shared memory operation by every single thread is recorded. Since each thread can get in as many as 300, 000 attempts per second, memory can run out fast. Even if we allocate a large amount for each thread to use, longer executions can cause segfaulting. Therefore, we have a ‘logging’ flag, which we always keep off for long runs. Thus, the logging information that we present was all created from relatively short (few seconds) runs. One possible future direction is to solve this problem by having threads turn their logging on in the middle of a longer run, after they reach ”steady state”, or periodically turn on logging for a little bit. However, this is not as easy as it sounds, since the logging information is most useful when it is synchronized across all threads, thus allowing us to reconstruct a part of the execution. If we have threads begin their logging in the middle of their execution, having them synchronize to turn on the log could throw off the trace, yielding useless results.

We also made options for our benchmark to vary the default read-modify-CAS pattern in order to test out other approaches and heuristics. For example, we have an option for the benchmark to back off when it reads certain values, or even to skip its reads altogether, and just use the CAS’s return value as the ‘expected value’ for its next CAS. These heuristics and their results are discussed in Section 5.

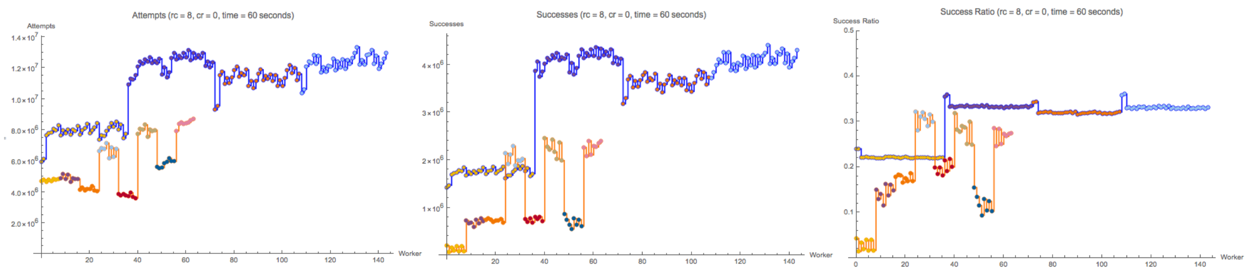
3 Initial Results

The first surprising result we encountered is the following: with low delay between memory operations, throughput (number of successful CAS attempts per second) is highest when workers are *distributed across different nodes* instead of clustered in a single node.



Above, we show the throughput on the Intel machine for increasing delay between the read and CAS for two different worker layouts: one with 18 workers on a single NUMA node (orange), and one with 18 workers distributed near-evenly over all four NUMA nodes (blue). Clustering on a single NUMA node is better for throughput only for large delays between the read and CAS.

To try to get a better understanding of why this is the case, we measured the number of attempts and successes of individual workers. Below we show results for benchmarks on the Intel and AMD machines with all workers active and minimal delay. From left to right, we show for each core (1) the number of CAS attempts made, (2) the number of successful attempts, and (3) the success ratio. Blue is Intel; orange is AMD. Each NUMA node has a different dot color. In both cases, the location being read and CASed is allocated in the memory of NUMA node 0.

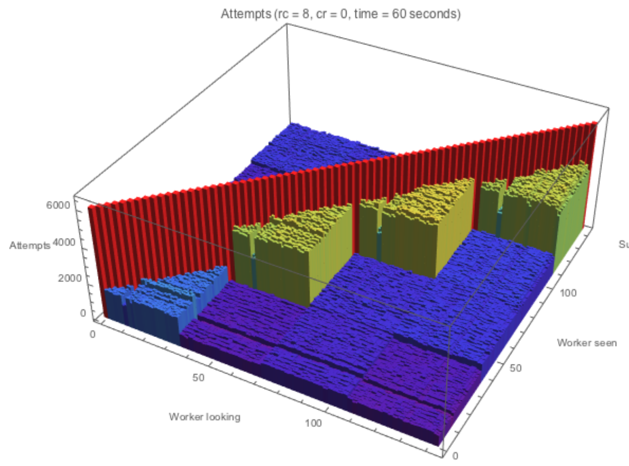


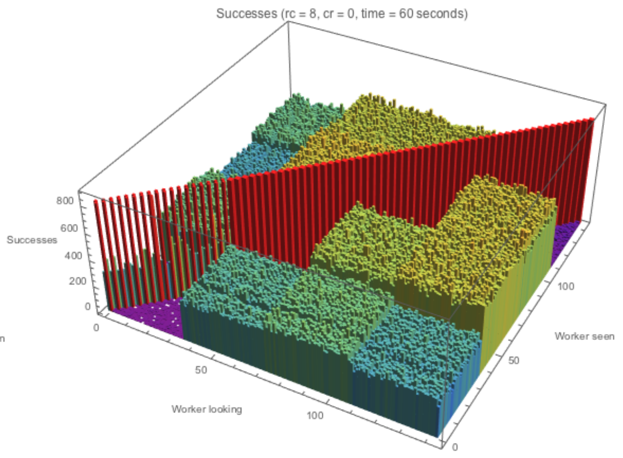
On both machines, workers on node 0 seem to have slightly fewer read attempts than those on other nodes, but they have substantially fewer CAS successes. This is a phenomenon that we found hard to explain. We know that in cases with low delay with a single read/CAS location, the node with fewer successes is the one where the location is allocated. This was unexpected because we expect that the location is always in cache. A possible explanation is that cores on node 0 attempt to load the location from main memory. If the cache line were clean, it is in some systems the case that loading from main memory within a NUMA node is faster than reading from a different node's L3 cache. Of course, in our benchmark, the cache line is always dirty, so reading from main memory never helps but might instead create additional overhead.

The AMD machine gets in fewer attempts and successes, which might be expected given that the Intel machine is newer. However, it also has lower success ratio, indicating that some of Intel's success-throughput advantage are due to architectural differences, not just improved hardware. Strangely, the AMD machine also seems to behave very asymmetrically with respect to the nodes other than node 0, in contrast to the Intel machine, which treats the other nodes similarly. Moreover, the exact asymmetry of the the AMD machine's performance is qualitatively different depending on which NUMA node the location is allocated on (not shown).

One fun feature of the Intel attempts plot (left, blue) is the downwards ridges visible for the first worker of each NUMA node. We believe these indicate the cores where the location's cache line lives (each core keeps part of a shared L3). This is because from run to run, the ridges' locations change, but there always is one per NUMA node, and the "in-node offset" of the ridges is always consistent accross the four nodes, suggesting that they in-node offset is being decided by the same hash function of the locations' address. It seems that the traffic at the L3 directory means the corresponding core has fewer changes to read. Similarly, in the Intel success plot (right, blue), there are upward ridges at the same locations. This suggests that although the traffic limits the number of the core's attempts, being nearby gives it a higher success rate.

So far, we have just seen data summarizing the statistics for each worker. The plots below, which are for the Intel machine, show how often each worker sees *each other worker* during a read, counting both the total number of sightings (left) and the number of sightings that precede a CAS success (right). The diagonal, representing nodes seeing themselves or the hyperthread on the same core, is truncated in both plots.

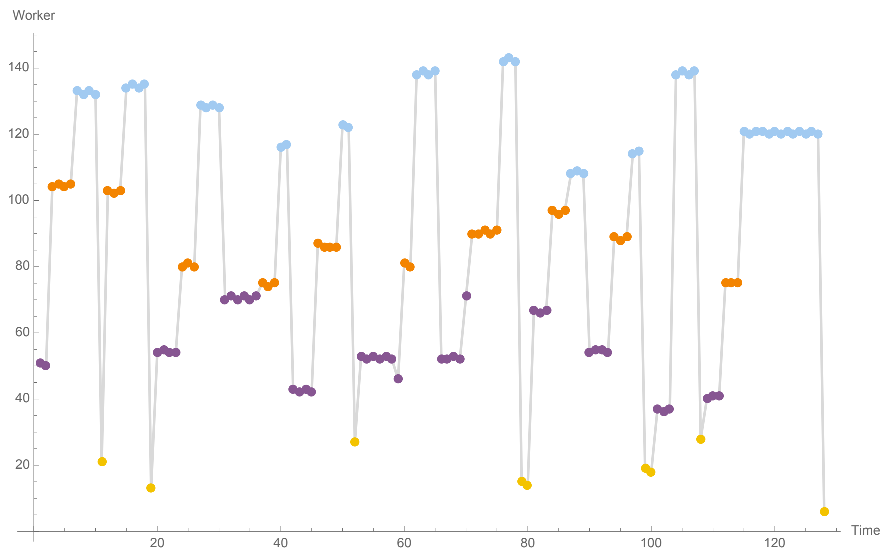




One thing we notice from the attempts plot (left) is that workers are far more likely to see the last success from a worker on their own NUMA node. However, we see from the successs plot (right) that workers are very *unlikely* to succeed if they last saw someone from their own NUMA node that was not themselves or the hyperthread on their core. In the next section, we take a closer look at exactly how the benchmarks are playing out to try to figure out why this is.

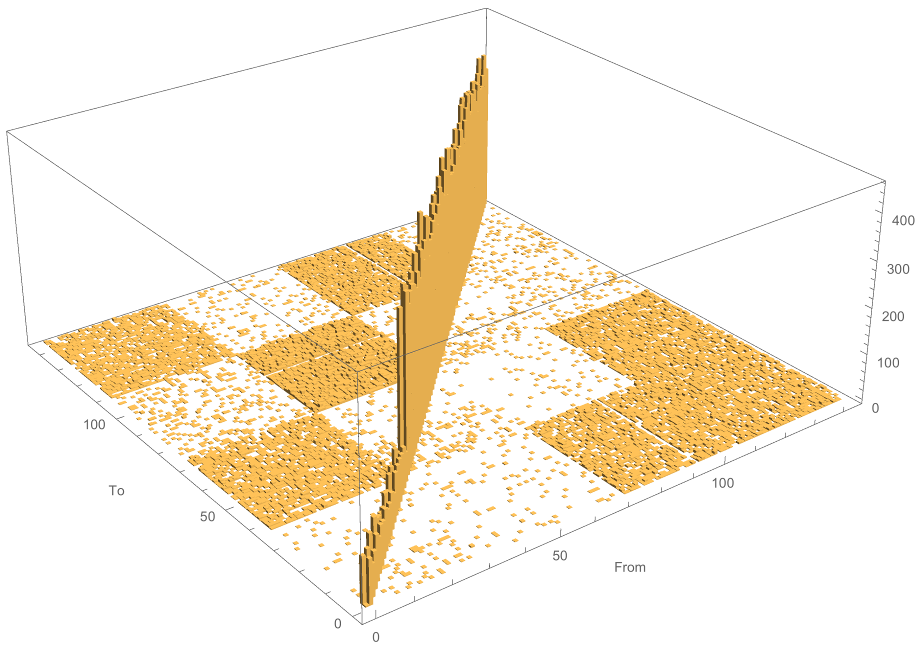
4 Fine-Grained Traces

Our fine-grained logging allows us to see in detail what happens during the benchmarks. For example, the following plot shows an excerpt of the trace of successful CAS attempts on the Intel machine.



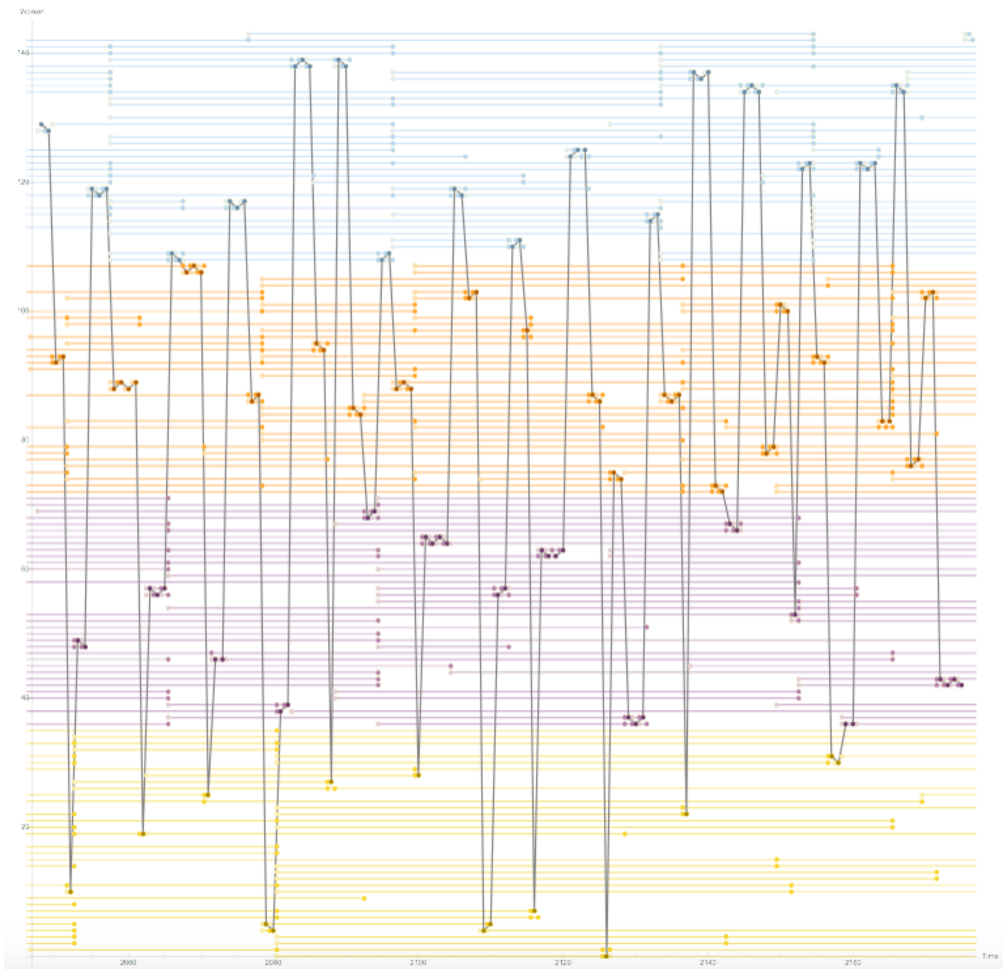
Above, different colors correspond to workers on different nodes. The x-axis is the index of the success; that is, it can be thought of as time, but rather than real-time or clock cycles, it is measured in successful CASes. Thus, it is quite possible (and even likely) that more time passed between some pair of successes than between some other pair, but this is not represented in the plot.

One striking feature of the plot is that the trail of successes by and large rotates between the nodes in a fixed order: yellow, purple, orange, blue, then back to yellow, etc., with only occasional skips. So far, we know that the fixed order changes occasionally over time, but we do not yet know how long streaks of a single order last. The followng plot, which summarizes jumps in the above trace over a region of 65536 successful CAS attempts, shows that two different orderings occurred during the region.



Above, the height of a yellow bar at (x, y) indicates the number of times a CAS attempt by worker x was followed by a successful attempt by worker y. The ridge in the middle indicates that hyperthreads on the same core (which have consecutive worker numbers) often succeed after each other.

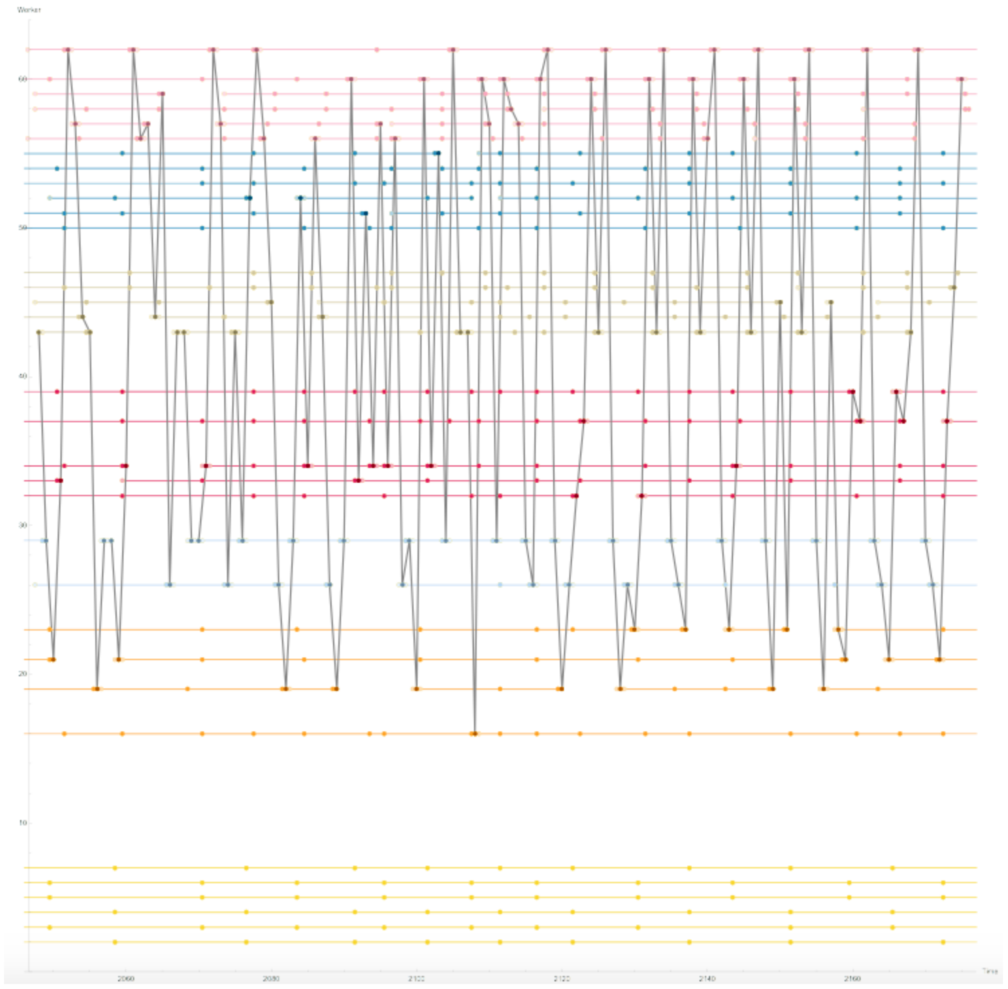
We were interested in understanding whether the 'skipped' nodes in the almost-round-robin trace pattern were indeed skipped altogether, or whether perhaps the cache line made its way to those nodes, but the cores were simply not given enough time to execute any successful CAS. With some careful analysis, we were able to reconstruct not only the trace of successes, but also where failures occurred during a given trace interval. Below, we show the Intel trace plot, but with the failed attempts presented as well. Here, a read is represented with a light dot, and a CAS with a dark dot. Lines connect a read and its corresponding failed CAS. Just like before, points on the line represent successful CASes.



From the above plot, we can see that indeed, in most cases where the cache line seemed to have ‘skipped’ a node, it actually did go there, but no core on the node was able to successfully modify it before it was taken away. Examples of this can be seen towards the end of the trace shown, where the grey line does not reach the yellow node for a few rounds, but points off the line (failed attempts) can be seen in the yellow node during that time.

A few more interesting phenomena emerge from observing this plot. One in particular is that it seems like failed CASes and reads seem to often happen in batches; the line travels to a certain node several times, during which there are barely any failures, and then, once every 4 or 5 visits to that node, there is suddenly a large amount of action going on, and many threads fail their attempts, and possibly start new ones all at once. We believe that this is some heuristic that Intel implements to ensure some form of ‘fairness’, whereby every once in a while, it allows all pending threads on a node to get the cache line.

Below, we show the same kind of plot, but for the AMD machine. We note that the access pattern is not round robin, but rather seems to be mostly uniform among the 3 nodes that get in many attempts. The 5 other nodes get the cache line much less often. Furthermore, the batching phenomenon observed for the Intel machine is not present.



In this plot, there are many 'blank' y-values. This indicates that the threads corresponding to those y-values did not get in a full reads or CASes during the interval we consider.

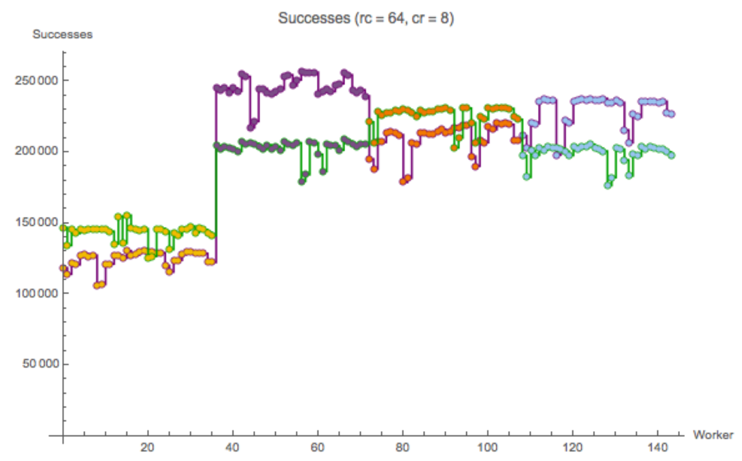
Interestingly, for the threads that *did* get in at least two operations, there are very few spaces between the horizontal lines. This means that upon failing a CAS attempt, it is very common for the threads to immediately restart another attempt, and get a response for their read operation before the cache line gets taken away from them. In particular, it means that read operations go through quickly (more so than on the Intel machine, on which there are more spaces between the horizontal lines of the same thread).

5 Different Protocols

We attempted to use the information we learned from the detailed traces on the Intel machine to create a smarter backoff protocol that, based on which worker it saw upon reading, would back off a different amount of time in an attempt to reduce contention. Specifically, we used the following protocol for each worker:

> If the last success was from another worker on your NUMA node *other than* you or your hyperthread, *wait then read again* instead of trying to do the work (that is, wait the RC delay) and CAS straight away.

We put a bound on the maximum amount of backoff. The results are summarized in the plot below, which shows the number of successes per worker. Purple is default; green is the "smart" backoff strategy described above.



Unfortunately, our strategy had slightly *lower* throughput than the default. Though the above plot has a large RC delay, representing a parallel algorithm with a small but nonzero workload, we got similar results for a wide array of parameters. In future work, we hope to either

- find a hardware-aware backoff protocol that significantly increases throughput or
- identify from the benchmarks concrete limitations in the hardware, which might show that no backoff protocol can significantly increase throughput.