Unfair Scheduling Patterns in NUMA Architectures

Naama Ben-David[*]
Carnegie Mellon University
nbendavi@cs.cmu.edu

Ziv Scully[*]
Carnegie Mellon University
zscully@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

***Abstract*—Lock-free algorithms are typically designed and analyzed with adversarial scheduling in mind. However, on real hardware, lock-free algorithms perform much better than the adversarial assumption predicts, suggesting that adversarial scheduling is unrealistic. In pursuit of more realistic analyses, recent work has studied lock-free algorithms under gentler scheduling models. This begs the question: what concurrent scheduling models are realistic? This issue is complicated by the intricacies of modern hardware, such as cache coherence protocols and non-uniform memory access (NUMA).**

In this paper, we thoroughly investigate concurrent scheduling on real hardware. To do so, we introduce Severus, a new benchmarking tool that allows the user to specify a lock-free workload in terms of the locations accessed and the cores participating. Severus measures the performance of the workload and logs enough information to reconstruct an execution trace.

We demonstrate Severus's capabilities by uncovering the scheduling details of two NUMA machines with different microarchitectures: one AMD Opteron 6278 machine, and one Intel Xeon CPU E7-8867 v4 machine. We show that the two architectures yield very different schedules, but both exhibit unfair executions that skew toward *remote* nodes in contended workloads.

I. Introduction

Creating pragmatic concurrent programs is essential for making the best use of modern multicore systems. When considering what constitutes a pragmatic program, designers often aim for high throughput, but another important feature is *fairness* among the cores participating in the algorithm. Fairness is sometimes a goal in its own right, such as in multicore web servers and other applications where each individual core's responsiveness is important. Even outside of such use cases, fairness can be important as a prerequisite for performance. Parallel programs in which work is statically assigned to cores, as is routine when using POSIX Threads[1] or OpenMP[2], often have synchronization barriers, at which point the last core to complete its work is the performance bottleneck. Such programs run faster if there is fairness among cores.

A large body of work has focused on designing algorithms that are *lock-free* or have other fairness guarantees [1], [2], [3], [4], [5]. However, lacking an understanding of memory operation scheduling on modern hardware, lock-free algorithms are typically designed with an *adversarial* scheduler in mind, meaning memory operations can happen in any order consistent with the memory model. While this guarantees correctness on any hardware, it leads to overly pessimistic predictions of performance and fairness. This observation has been made by practitioners and theoreticians alike, and has led to most lock-free algorithms being evaluated exclusively through experimentation [6], [7], [8], [9], [1]. While experimental analysis of these algorithms is important, experiments can miss practical use cases and yield misleading results [10]. Furthermore, holes in our theoretical understanding can cause practical designs to be overlooked [11], [12].

Algorithm 1 Generic lock-free algorithm (simplified)

```
1: loop
2:     parallel_work()
3:     repeat
4:         old ← read(x)
5:         new ← atomic_modify(old)
6:         success ← CAS(x, old, new)
7:     until success
8: end loop
```

A recent line of work aims to relax adversarial scheduling assumptions to better reflect reality [13], [14], [15], [16], [11], [17]. It is well-known that if the hardware schedule guarantees fairness properties, then algorithms can be faster, simpler, and more powerful [18], [11], [19]. However, it is not clear if such fairness properties or other assumptions are realistic. Thus, to understand the performance of lock-free algorithms, we must study the *scheduling of memory operations in hardware*.

Let us first consider the kinds of demands that most concurrent lock-free algorithms make on the scheduler. Many lock-free algorithms have the structure shown in Algorithm 1 [14], [16]. All cores run *parallel work* (line 2), that they do independently, and then synchronize in an *atomic modify* section (lines 3–7). In this section, a core executes a modification of location x that must not be interrupted by any other core's modification of x. Thus, the ordering, or *schedule*, of reads and CASes of x has a large impact on the fairness and performance of the algorithm. Intuitively, a good schedule has:

- *Long-term fairness:* we want each core to perform the same number of read and successful CAS instructions over any sufficiently long period of time.
- *Short-term focus:* for performance, whenever a core reads x, we want it to execute its following CAS without other cores performing any read or CAS instructions in between.

Having outlined what a good memory operation schedule looks like, we ask: what do memory operation schedules look

[*]The first two authors contributed equally to the paper.
[1]`https://ieeexplore.ieee.org/document/8277153/`
[2]`https://www.openmp.org`

like on modern hardware? Do practical schedules have the fairness and focus properties we want for lock-free algorithms?

Unfortunately, this is a difficult question to answer because the complexity of modern memory hierarchies makes scheduling patterns difficult to predict. Design decisions in aspects such as the cache coherence protocol and non-uniform memory access (NUMA) can have a drastic impact on the schedule. However, exactly how different designs correspond to scheduling patterns is unclear, especially when multiple features interact with one another.

For example, it is well known that the latency of a local-node cache hit is much lower than that of a remote-node cache hit [20]. This encourages the design of NUMA-aware algorithms [21], [22], [5], [23] that minimize remote-node memory accesses. However, recent work on arbitration policies in the processor-interconnect [24] shows that when most but not all memory accesses are local—which is exactly the situation for many NUMA-aware algorithms—hardware can unfairly bias the schedule towards *remote* nodes. Thus we see that a NUMA architecture can yield unexpected schedules.

A. Our Contributions

In this paper, we provide a way to test the schedules produced by today's machines and find patterns that can be important for fairness and performance. To do so, we introduce a benchmarking tool, called *Severus*, that allows the user to specify a workload, and tracks the execution trace produced. We show how to use Severus to understand the scheduling patterns of two modern NUMA machines, and provide a plotting library that helps visualize the results in an intuitive way.

Severus allows the user to play with several parameters of the execution, including which threads participate in a run, what locations are accessed, how much local work each thread does, and how long each thread waits between two consecutive operations. With this flexibility, Severus can simulate the workloads that are most relevant to the user's application.

In this paper, we describe Severus and use it to demonstrate the following takeaways:

- Operation schedules are not fair by default.
- Uniform random scheduling assumptions do not accurately reflect real schedules.
- The amount of local work a thread does in a lock-free algorithm, particularly the length of the *atomic modify* section, has a large but hard-to-predict impact on the algorithm's performance.
- The details of these effects are different on each platform, but these details can be revealed by tools such as Severus.

We believe that these new findings can guide both the design of new pragmatic concurrent algorithms on existing machines and the development of new memory architectures that enable faster and more fair concurrent executions.

We reach the above takeaways by studying the memory operation scheduling patterns of two NUMA machines: an AMD Opteron 6278 and an Intel Xeon CPU E7-8867 v4. These two machines exhibit different architectural designs: the Intel has four equidistant nodes and uses a hierarchical cache coherence protocol, whereas the AMD is arranged in eight nodes, with two different distances between them, and employs a flat cache coherence mechanism. We show how these design choices translate to differences in schedules. While the scheduling patterns remain mostly round-robin on AMD regardless of the cores participating in a run, on Intel, the schedule changes drastically depending on whether cores from more than one node are running. Interestingly, both machines show *higher* throughput for cores that access remote contended memory. We characterize workloads in which this phenomenon is prominent, and show how this unfairness changes as certain parameters of the program are varied.

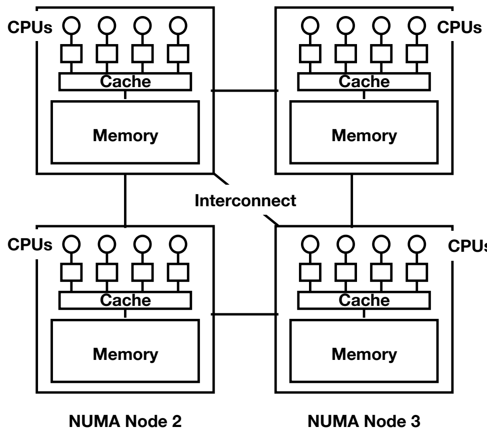


Fig. 1: NUMA architecture with 4 NUMA nodes.

II. Background and Machine Details

A. NUMA Architectures

NUMA architectures are everywhere in modern machines. Cores are organized into groups called *nodes*, and each node has cache as well as main memory (see Figure 1). Within a node, cores may have one or two levels of private cache, and a shared last level cache. Each core can often be split into two logical threads, called *hyperthreads*. All cores can access all shared caches and memory, through an interconnect network between the nodes. However, accesses to cache and memory in a core's own node (*local acceses*) are faster than accesses to the cache or memory of a different node (*remote accesses*).

B. Lock-Free Algorithms and Scheduling

Lock-free algorithms guarantee that progress is made in the algorithm regardless of the number of threads participating or their relative speeds. The correctness of lock-free algorithms is typically proved under an adversarial model, whereby a powerful adversary determines the schedule of atomic operations on each location, thus controlling who succeeds and who fails at any time. The adversarial model produces robust algorithms, but lacks predictive capabilities for performance. Usually, the best performance guarantees that can be proven under an adversarial scheduler are embarrassingly pessimistic.

Thus, recent work in lock-free algorithms proposes different scheduling models, with the goal of being able to analytically

TABLE I: Machine details.

Specs	Intel	AMD
CPU family	Xeon E7-8800	Operton 6200
Sockets	4	4
Nodes	4	8
Cores	72	32
Hyperthreading	2-way	2-way
Frequency	1200-3300 MHz	2400 MHz
L1i Cache	32k	16k
L1d Cache	32k	64k
L2 Cache	256k	2048K
L3 Cache	46080K	6144K
Coherence protocol	MESIF	MOESI

	N0	N1	N2	N3	N4	N5	N6	N7
N0	0	1	1	2	1	2	1	2
N1	1	0	2	1	1	2	2	1
N2	1	2	0	1	1	1	1	1
N3	2	1	1	0	1	1	2	2
N4	1	1	1	1	0	1	1	2
N5	2	2	1	1	1	0	2	1
N6	1	2	1	2	1	2	0	1
N7	2	1	1	2	2	1	1	0

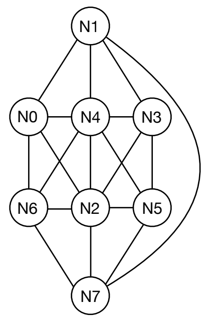


Fig. 2: AMD node layout and distance matrix.

predict performance. Common alternative models include that the scheduler picks the next thread uniformly at random [14], [17], or with some predetermined distribution [13]. The goal of our work is to test whether such assumptions are reasonable, and to understand what factors of modern architectures most affect the operation scheduling, and which most affect performance.

C. Machines Used

We test our benchmark on two different NUMA architectures; an Intel Xeon CPU E7-8867 v4 machine with 4 nodes and 72 cores with Quick Path Interconnect technology, and an AMD Opteron 6278 machine with 8 nodes and 32 cores, using HyperTransport. Throughout this paper, we refer to these machines as simply *Intel* and *AMD* respectively. Both machines have a per-core L1 and L2 cache (shared among a pair of hyperthreads), and a shared L3 cache on each node. The details of the two machines are shown in Table I. The Intel machine's interconnect layout is fully connected, and therefore all nodes are at the same distance from one another. However, this is not the case for the AMD machine, in which there are two different distances among the nodes. The AMD node layout and distance matrix is shown in Figure 2.

Both machines have an atomic compare-and-swap (CAS) instruction and an atomic fetch-and-increment (F&I) or fetch-and-add (F&A, also called xadd) instruction. A CAS instruction takes in a memory word, an old value *old*, and a new value *new*, and changes the word's value to *new* if the previous value was *old*. In this case, it returns *true*, and is said to *succeed*. Otherwise, the CAS does not change the memory word. It returns *false* and we say that it *fails*. The F&I instruction takes in a memory word and increments its value. It always returns the value of the word immediately before the increment. Both the CAS and the F&I instructions fully sequentialize accesses.

III. The Benchmark

Severus provides many settings to simulate the behavior of a large range of applications. For clarity, we begin by describing one simple setting, and then show ways to extend it.

At its core, Severus simply has all threads contend on updating a single memory location, either with a read-modify-CAS loop, or with an F&A. We measure throughput; how many changes to the memory location were made. To retain information about the execution, we also have a *logging* option, in which we have each thread record the values it observed on the shared location every time the thread accesses it. For the F&A case, simply recording these numbers allows us to reconstruct the order in which threads incremented the shared variable. For a CAS-based benchmark, we can control what values the threads write into the shared variable. To allow reconstruction of the execution order, we have each thread CAS in its own id and a timestamp. In this way, when threads record the values they observed, they are in effect recording which thread was the last one to modify the variable with a successful CAS. From this information, we obtain a total order of successful CASes, and a partial order on the reads and unsuccessful CAS attempts.

Severus provides parameters to modify the basic benchmark to reflect different workloads, including the following settings.

- The number of shared variables contended on.
- Which node each shared variable is allocated on.
- Which threads participate.
- For each thread, which shared variables it should access.
- Length of execution.
- Whether or not the threads should log execution information. Turning this option off helps optimize space usage.
- For CAS-based tests, delays can be injected between a read operation and the following CAS attempt of that thread. This simulates the time it takes in real programs to calculate the new value to be written.
- Delay can be injected between two consecutive modifications of the shared variable by the same thread. This simulates programs in which threads have other work.
- Delay can also be injected between a failed CAS attempt and the thread's next read operation. This allows simulation of backoff protocols.

A. Implementation Details

When evaluating the schedule of a concurrent application, one must be very careful not to perturb the execution. Many common instructions used for logging performance, including accesses to timers, cycle counters, or memory allocated earlier in the program, can greatly affect the concurrent execution, leading to useless measurements. Thus, we take care in ensuring that our logging mechanism minimizes such accesses.

1) NUMA memory and thread allocation: We use the Linux NUMA policy library *libnuma* to allocate memory on a specified node (both for contended locations and memory used for logging), and to specify the threads used. We pin threads to cores.

2) Logging: All information logged during the execution is *local*. We allocate a lot of space per thread for logging, and ensure that for each thread, this log space is in the memory of the NUMA node on which that thread is pinned. No two threads access the same log. This helps eliminate coherence cache misses that are not directly caused by the tested access pattern. Before beginning the real execution, we have each thread access its preallocated log, to avoid compulsory cache misses when it first accesses the log during its execution. Severus always records the total number of operations executed by each thread, and the total number of successful CASes per thread. This simply involves incrementing two counters, and thus never causes cache misses.

If the *logging* option is enabled, each thread also records which values it observed on the shared location when it accessed it. This logging takes much more space, since this information cannot be aggregated into one counter, and thus we keep a word per operation executed by each thread. Logging can also perturb the execution; more (uncontended) writing is done, and cache misses occur every once in while, when the size of the log written exceeds the cache size. However, since the memory of the log is accessed consecutively, prefetching helps mitigate the effect of log-caused cache misses. With this local method of logging, we process the results after the execution ends, and reconstruct the global trace from the per-process ones.

3) Compiler Options: To eliminate as much overhead as possible during the execution, many of the settings of a run are determined at compile time. This includes machine details, like the number of nodes and cores, and the ids of the cores on each node. The type of execution (CAS, F&A, etc.) and logging are also determined at compile time.

4) Delay: We implement atomic delay and parallel delay by iteratively incrementing a local volatile counter. The amount of delay given as a parameter for an execution translates to the number of iterations that are run. In the rest of the paper, we use 'iterations' as the unit of delay used in experiments. This is done to avoid mechanisms of waiting that are too coarse grained or can perturb the execution. Therefore, given the same delay parameter, the actual amount of time that a thread waits depends on the system on which the benchmark is run (in particular, depending on the core frequency). A single unit of delay corresponds to approximately 2.2 nanoseconds on Intel and 3.5 nanoseconds on AMD (both averaged over 10 runs). We note that measuring delay in terms of iterations of local cache accesses is reasonable for simulating algorithm workloads, since it reflects the reality that different algorithms take different amounts of time on different machines.

B. Experiments Shown

All tests shown in this paper can be broadly split into two categories.

- *Sequence Experiments.* In these experiments, we take a subset of the threads (possibly all of them), and have them repeatedly increment a single location using atomic fetch-and-increment (F&I). We call the contended location the *counter*. All threads record the return value of their fetch-and-add after each operation, using the logging option. This allows us to recreate the order in which threads incremented the counter.
- *Competition Experiments.* These experiments are similar to the sequence experiments, but differ mainly in the operation used. A subset of the threads repeatedly read a location, locally modify its value, and then compare-and-swap (CAS) their new value into the same location. We call the contended location the *target*. In competition experiments, we sometimes vary other parameters, like the local modification time (which we call *atomic delay*), and the time threads wait between a successful CAS and that thread's next operation (*parallel delay*).

The competition experiments cause different scheduling patterns than the sequence ones; the read operations mean that the cache line enters the shared coherence state in addition to the modified state. Furthermore, compare-and-swaps fail if another thread has changed the value. This means that to successfully modify the location, a thread must execute two operations in a row, possibly changing its cache line's coherence state in between. The schedules produced by sequence experiments are more regular, and thus easier to analyze to obtain a high level understanding of the scheduler.

Therefore, to learn about each machine's scheduling patterns, we use sequence experiments, with the logging option turned on (Section IV). We show how the lessons we learn from these experiments generalize to other workloads by running competition experiments (which better reflect real-world applications), without logging, and comparing the results to the predictions made based on our learned scheduling model (Section V). We also provide a script that runs the experiments described in this paper and produces the relevant plots.

IV. Inferring Scheduling Models

In this section, we show experiments that help determine scheduling models for the AMD and Intel machines. All the experiments in this section are *sequence experiments* (see Section III). To review, in a sequence experiment, multiple cores atomically fetch-and-increment (F&I) a single memory location called the *counter*. This yields a full execution trace, namely a sequence of all the F&I operations executed by all threads, which we analyze in several ways to determine a scheduling model. Across different experiments, we vary the number of threads participating, the placement of the threads, and the NUMA node on which the counter is allocated.

A sequence experiment is a hardware stress test meant to reveal details about how it schedules memory operations. It is *not* meant to model a realistic lock-free algorithm. In particular, throughput measurements of sequence experiments should be not be interpreted as a proxy for performance of a

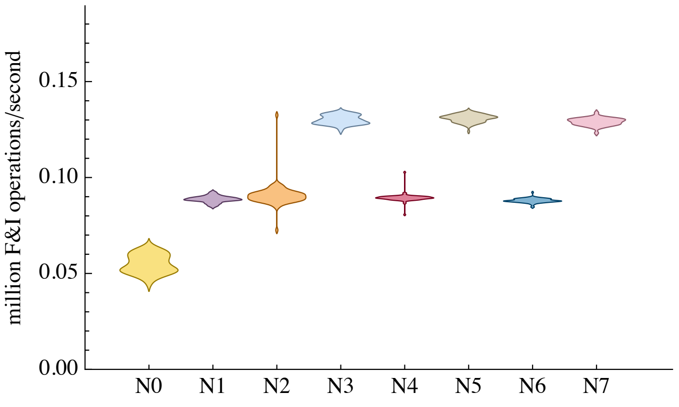


Fig. 3: Counter on Node 0

Fig. 4: AMD throughput of F&I operations with all nodes participating. Counter allocated on Node 0.

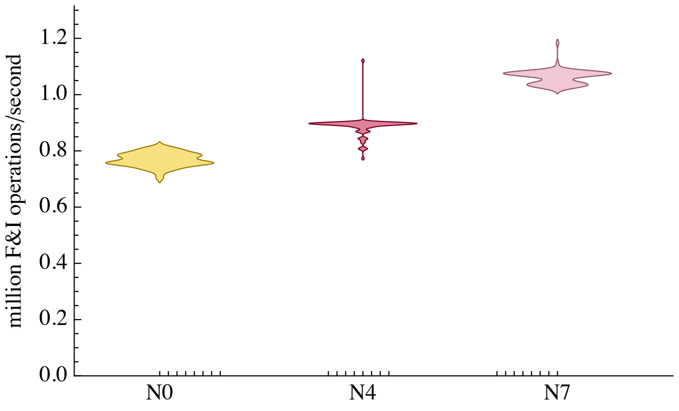


Fig. 5: AMD throughput of F&I operations with one node participating at a time. Counter allocated on Node 0. Node participating is either Node 0 (distance 0 from counter), Node 4 (distance 1), or Node 7 (distance 2).

lock-free algorithm. (In contrast, the competition experiments in Section V *are* intended to model lock-free algorithms.)

A. AMD Scheduling Model

1) AMD Throughput Measurements: We begin with a basic question: when all cores participate in a sequence experiment, do they achieve the same throughput? As we will see, the answer to this question is counterintuitive and will guide our more detailed analysis of the machine's scheduling model.

To answer this, we run a sequence experiment with the counter on Node 0 and simply count the number of F&I operations executed by each core. For each node, Figure 3 shows the distribution of throughputs among cores of that node.[3] We see that most cores within any given node have similar throughput, but different nodes have very different throughputs. We observe that the throughput is unfair:

- Node 0, which is where the counter is allocated, has the lowest throughput;
- Node 1, Node 2, Node 4, and Node 6 have intermediate throughput; and
- Node 3, Node 5, and Node 7 have the highest throughput.

What distinguishes Node 3, Node 5, and Node 7 from the other nodes? The answer lies in Figure 2: they are the *farthest* from the counter on Node 0. That is, a core's throughput tends to increase with its distance from the counter. Repeating the experiment with the counter on each node confirms this.

So far, we have seen that with all cores from all nodes participating, cores on nodes farther from the counter have a throughput advantage. We now ask: does this trend still hold when nodes participate one at a time? To answer this question, we run experiments with the counter on Node 0 with cores on just a *single node* participating. Figure 5 shows the distribution of results for each of Node 0 (distance 0), Node 4 (distance 1), and Node 7 (distance 2) participating. Unlike the previous plots, each distribution in the plot represents a *separate configuration* in which only that node is participating. The overall throughput is higher in these configurations because of reduced contention.

Remarkably, Figure 5 shows that even with only a single node participating, throughput still increases with distance from the counter. Results for other nodes at distances 1 and 2 are similar to those for Node 4 and Node 7, respectively. Similar results hold when cores from any subset of nodes participate.

We have firmly established that throughput is unfair and is skewed toward cores that are farther from the counter, even when the counter's cache line remains cached on the same node. This pattern reflects the directory coherence protocol on AMD, which seems to use the interconnect even when a cache line remains on one node, likely due to the need to update its coherence state in the directory. To understand why increased interconnect use increases throughput, we need a more detailed analysis of the execution traces.

2) AMD Execution Trace Analysis: We now thoroughly examine the execution trace of a single sequence experiment. All cores participate, and the counter is on Node 0. We examine an execution trace excerpt of 2^{20} operations, taken from the middle of the experiment to avoid edge effects. For space reasons, we show results from just one run and focus on three nodes Node 0 (distance 0 from counter), Node 4 (distance 1), and Node 7 (distance 2). We have confirmed that the results shown are robust across several trials and other nodes at distances 1 and 2 behave similarly.

The result of a sequence experiment is an execution trace, which is an ordered list of core IDs whose ith entry is the ID of the core that executed the ith F&I operation on the counter. We can think of the trace as describing how (modify-mode access to) the counter's cache line move from core to core.

To talk about the trace and its implications for throughput, we use the following vocabulary:

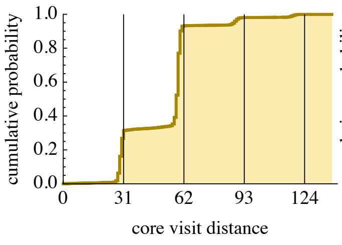
- *Core visit:* a contiguous interval during which just one core performs F&I operations.[4]

[3]Throughout this section, all throughput distribution plots show the aggregate throughput distribution of 10 separate 10-second runs.

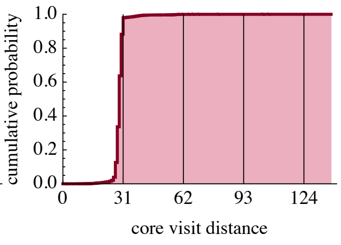
[4]When discussing core visits, we take "core" to specifically mean "physical core" and group its two threads together.

TABLE II: AMD core visit length distributions.

	LENGTH 1	LENGTH 2	LENGTH ≥ 3	MEAN
Cores on Node 0	88%	9%	3%	1.147
Cores on Node 4	93%	4%	3%	1.105
Cores on Node 7	55%	34%	11%	1.585



(a) Core 0 on Node 0, avg 51.6



(b) Core 16 on Node 4, avg 29.1

Fig. 6: AMD core visit distance distributions with all nodes participating. Counter allocated on Node 0. Showing distributions for (a) a core on Node 0 (distance 0 from counter) and (b) a core on Node 4 (distance 1). Distributions for other distance 0 cores are similar to (a), and likewise for distances 1 and 2 with (b).

- *Core visit length:* the number of F&I operations performed during a given core visit.
- *Core visit distance:* the number of core visits to other cores between two visits to a given core.

A core's throughput is

- directly proportional to its average core visit length and
- inversely proportional to its average core visit distance.

For each of Node 0, Node 4, and Node 7, Table II shows the distribution of visit lengths for cores on that node. Notably, the average core visit lengths on Node 7 is roughly 40% higher than each of Node 0 and Node 4. Recall that in Figure 3, Node 7 has roughly 40% higher throughput than Node 4, which in turn has higher throughput than Node 0. It thus appears that average core visit length explains the throughput difference between Node 4 and Node 7, but explaining the even lower throughput of Node 0 requires examining core visit distances.

We now turn to core visit distances. Figure 6 shows the CDF of visit distances aggregated over all cores for Node 0 and Node 4. Due to space limitations, we omit the plot for Node 7, but it is almost identical to that of Node 4. Remarkably, nearly all core visit distances are just below multiples of 31, which is one less than the number of physical cores on the AMD machine. This suggests that core visits occur in round-robin fashion, visiting all 31 other cores between two visits to a given core, except that cores are occasionally skipped, mainly on Node 0. Given that average core visit lengths are roughly the same for Node 0 and Node 4 (see Table II), their throughput difference is due mainly to the skipping of cores on Node 0.

B. Intel Scheduling Model

1) Intel Throughput Measurements: We begin our analysis of the Intel machine in the same way we did for AMD. We want to know whether throughput is fair among different cores, and in particular, whether the distance patterns we observed for AMD hold for Intel as well. Recall that the Intel machine has only 4 NUMA nodes, with a full interconnect that places all nodes equidistantly from one another.

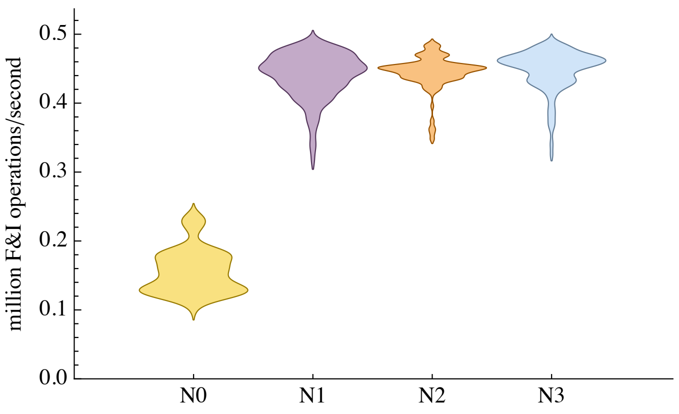


Fig. 7: Intel throughput of F&I operations with with all nodes participating. Counter allocated on Node 0.

Figure 7 shows each node's throughput distribution for a sequence experiment with all cores participating with the counter placed on Node 0. We see that, again, throughput is unfair, and cores on Node 0 have lower throughput than cores on the other three nodes. The results are analogous when the counter is allocated on Node 1, Node 2, or Node 3.

We next test whether cores close to the counter still have lower throughput when only one node participates at a time. To answer this question, we run experiments with the counter on Node 0 with cores on just a *single node* participating. Figure 8 shows the results for each of Node 0 and Node 3 participating, Unlike in the experiment with all nodes participating, we see that Node 0 and Node 3 have similar throughput distributions when only one node participates at a time. The results for Node 1 and Node 2 are similar.

We have seen that with all nodes participating, Intel and AMD both exhibit core throughput increasing with distance from the counter, but the machines differ when only one node participates. This can be explained by considering the directory coherence protocol. Each node on Intel has a shared L3 cache, and the coherence protocol does not communicate updates to other nodes so long as the cache line is not in any other node's L3 cache. This means single-node runs are virtually unaffected by where the counter is allocated.

2) Intel Execution Trace Analysis: We now investigate the Intel execution trace in detail. Figure 9 shows the execution trace produced from a sequence experiment with the counter allocated on Node 0. The y-axis shows the different thread id's color-coded by node. The x-axis shows "time", measured in number of F&I operations. The line shows the counter's migration pattern across the caches of the different cores.

To discuss the execution trace, we define the following terms:

- *Core visit:* a contiguous interval during which just one core performs F&I operations (see Section IV-A2). The *length* of a core visit is the number of F&I operations performed in it.
- *Node visit:* a contiguous interval during which cores on just one *node* perform F&I operations. The *length* of a

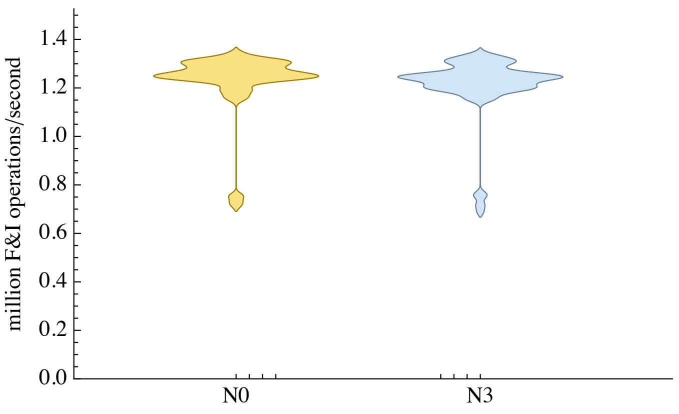


Fig. 8: Intel throughput of F&I operations with one node participating at a time. Counter allocated on Node 0. Node participating is either Node 0 (distance 0 from counter) or Node 3 (distance 1).

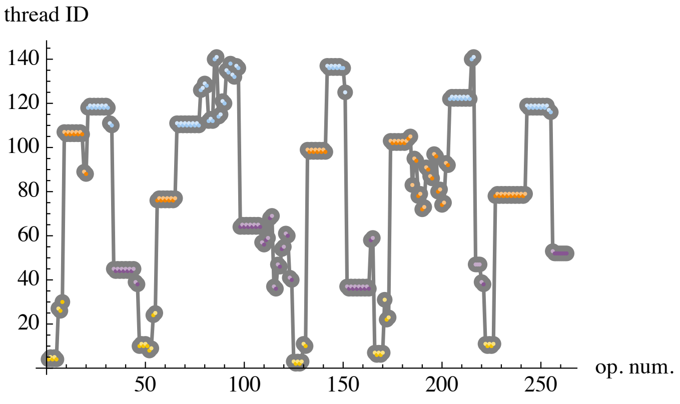


Fig. 9: Intel execution trace of F&I operations with all nodes participating. Counter allocated on Node 0. Thread IDs are clustered by node: 0–35 on Node 0 (yellow), 36–71 on Node 1 (purple), 72–107 on Node 2 (orange), and 108–143 on Node 3 (blue). Even-odd pairs of threads (0-1, 2-3, etc.) run on the same physical core. Even thread IDs are shaded darker.

node visit is the number of core visits it contains.

Figure 9 reveals unusual features of its core and node visits.

Round robin node visits: The nodes are visited in a fixed repeating order throughout Figure 9: 0, 2, 3, 1, ... We have confirmed that this pattern is consistent over the entire trace, though the order occasionally changes and Node 0 is occasionally skipped. We omit the detailed statistics for brevity.

Uneven core visit lengths: The first core visit of each node visit is usually relatively long. Moreover, these long core visits *only* occur as the first node visit: almost all other node visits are very short, having just one or two F&I operations. To confirm this observation, we show the CDF of the core visit length distribution for Node 0 (distance 0 from the counter) and Node 3 (distance 1) in Figure 10. For brevity, we omit plots for Node 1 and Node 2, which are similar to that for Node 3. The pattern is very clear for Node 3: about 70% of core visits are of length 1 or 2, but visits of length greater than 2 are likely to be at least length 10. The pattern is a bit less prominent on Node 0, where longer visits only last around 5 operations. This partially explains the difference in

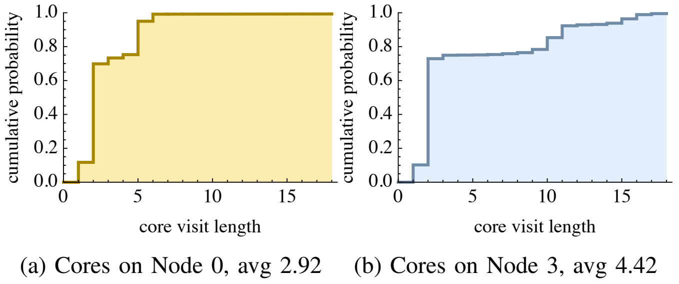


Fig. 10: Intel core visit length distributions with all nodes participating. Counter allocated on Node 0. Showing aggregate distributions for (a) cores on Node 0 (distance 0 from counter) and (b) cores on Node 4 (distance 1). Distributions for Nodes 1 and 2 are similar to (b).

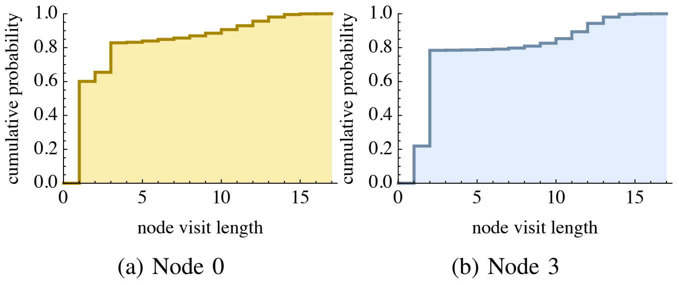


Fig. 11: Intel node visit length (measured in number of core visits) distributions with all nodes participating. Counter allocated on Node 0. Showing distributions for (a) Node 0 (distance 0 from counter) and (b) Node 4 (distance 1). Distributions for Nodes 1 and 2 are similar to (b).

throughput observed between Node 0 the other nodes.

Occasional bursts: In Figure 9, most node visits only contain a few core visits: first a long core visit, followed by 0 to 2 more core visits. However, every once in a while, a node visit ends with many short core visits in a row. We call this occurrence a "burst" of visits. A natural question is: are bursts simply the result of noise, or they a separate phenomenon? To answer this question, we plot CDF of the node visit length distribution in Figure 11, again showing only Node 0 and Node 3 for brevity. The distributions make clear that there are two distinct types of node visits: those with 3 or fewer core visits, constituting about 80% of all node visits; and those with significantly more, usually at least 8, making up the other 20% of node visits. We therefore define the following terms:

- *Burst:* a node visit of length 4 or greater. For example, Figure 9 shows bursts for each of Nodes 1, 2, and 3.
- *Cycle:* the time between the end of one burst on a given node and the end of the next burst on that node.

Interestingly, we find that in most cycles, each core is visited exactly once. This is shown in Table III. This pattern, which occurs on all nodes, suggests a possible mechanism for the bursts: requests for the counter's cache line build up in a queue in each node, and each queue occasionally "flushes" if it is too full for too long.

Finally, recall from Section IV-B1 that single-node executions produce different throughput distributions than executions that cross node boundaries. We therefore also examine the

TABLE III: Intel number of times cores are visited per cycle.

	0 Visits	1 Visit	2 Visits	$\geq$ 3 Visits
Cores on Node 1	9%	85%	5%	$< 1\%$
Cores on Node 3	10%	85%	5%	$< 1\%$

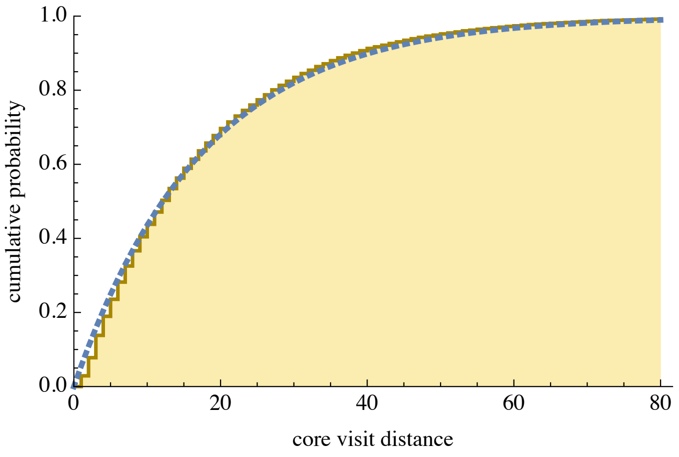


Fig. 12: Intel core visit distance distributions with only Node 0 participating. Counter allocated on Node 0. Distribution is very close to Geometric$(1/18)$ (dashed blue line).

trace of a single-node execution with the counter on Node 0 and only cores on Node 0 participating. In contrast to the multiple-node trace, the single-node trace is close to uniformly random. To confirm this, we show the CDF of the core visit distance distribution in Figure 12. The CDF is close to that of a geometric distribution, which is what the CDF would be for a truly uniformly random schedule. This means that for analyzing algorithms for single-node executions on the Intel machine, a uniformly random scheduling model is appropriate.

V. Takeaways for Fairness and Focus

Recall the desirable properties a schedule should have: in the long run, we want it to be *fair*, letting each thread make the same amount of progress, but in the short term, we want the schedule to be *focused*, allowing each thread enough time to read, locally modify, and then apply its modification on a cache line before the cache line gets invalidated.

We now go back to our original question: do memory operation schedules on modern hardware achieve long term fairness and short term focus? In the previous section, we saw some indications that the schedules might not be fair: initial throughput experiments indicated the on both machines, the node on which memory is allocated is unfairly treated, even in long runs. We saw that short-term focus might be behind this: cores on remote nodes get longer visits on average. However, recall that these experiments were *sequence* experiments, which were designed to uncover scheduling patterns but not to represent the workloads of real lock-free algorithms.

In this section, we thus test whether these initial findings carry over to more realistic workloads. More specifically, all the experiments in this section are *competition experiments* (see Section III). To review, in a competition experiment, multiple cores attempt to read from and CAS a new value into a single memory location called the *target*. Competition experiments have two *delay* parameters.

- Between a read and the following CAS is the *atomic delay*. This simulates work in the the *atomic modify* section of a lock-free operation (Line 5 of Algorithm 1).
- Between each successful CAS and the following read is the *parallel delay*. This simulates the *parallel work* of a lock-free algorithm between synchronization blocks (Line 2 of Algorithm 1).

We simulate different lock-free workloads by varying the atomic and parallel delays. To highlight the effects of the atomic delay, the experiments in this section are conducted with a high parallel delay (set to 256 iterations in all experiments. See Section III for details on how the delay is implemented). This means that long streaks of successful read-modify-CAS operations by one thread without interruption from another thread are unlikely, even when the atomic delay is small.

All plots in this section show the results over 10 repetitions of 10 second runs. Each plot point shows the median total throughput of *successful CAS instructions* over the 10 repetitions, and error bars show the 75th and 25th percentile.

A. Fairness

To test long-term fairness on lock-free workloads, we run a set of competition experiments in which all cores on all nodes are participating. We vary the atomic delay to evaluate the fairness for lock-free algorithms with differently sized atomic modify sections. We measure the throughput of successful CAS instructions exhibited by cores on each node, and compare them to the throughput on other nodes. These tests answer the following question: when all cores run the same code, how skewed is their throughput with respect to each other?

1) AMD Fairness: The results for the fairness test on the AMD machine are shown in Figure 13. It is clear that cores on distance 2 nodes (represented by Node 7 here) perform much better when atomic delay is low, outperforming other nodes by up to 31×, but this drops very quickly.[5] By the time atomic delay reaches 16 iterations (around 56 ns), distance 1 nodes start outperforming distance 2 nodes. However, recall that the throughput reported in Figure 13 shows *successful CAS* instructions. Interestingly, if we consider the number of *attempted* CAS instructions, rather than just the successful ones, the difference is less stark, with distance 2 nodes reaching a peak at an atomic delay of 5 iterations, at which point they only outperform distance 1 nodes by a factor of 2.2.[6] This indicates that at low atomic delays, distance 2 nodes succeed in a much larger fraction of their attempted CAS instructions.

The throughput reaches a steady state at around an atomic delay of 30 iterations (roughly 70 ns), but is still highly unfair. Notably, distance 1 nodes achieve the highest throughput at the steady state, outperforming the other two groups by an order of magnitude. Insight into this phenomenon can be gained by

[5]This part is truncated in the plot, to make other trends more visible.
[6]This data is not shown in the plot.

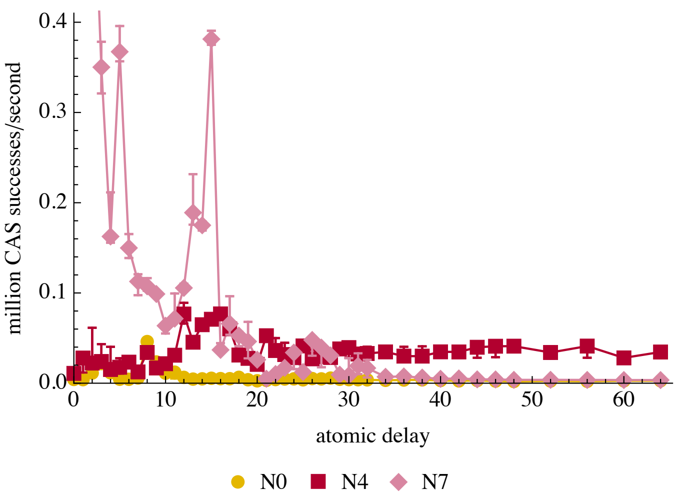


Fig. 13: AMD throughput of CAS operations for varying atomic delay with all nodes participating. Target allocated on Node 0. Showing total throughput of Node 0 (distance 0 from target), Node 4 (distance 1), and Node 7 (distance 2).

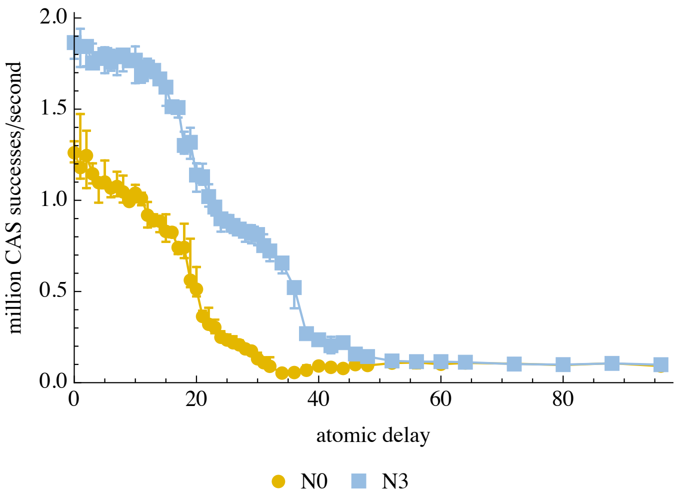


Fig. 14: Intel throughput of CAS operations for varying atomic delay with all nodes participating. Target allocated on Node 0. Showing total throughput of Node 0 (distance 0 from target) and Node 3 (distance 1).

looking at the *success ratio*, or the fraction of successful CAS instructions out of the overall number attempted. For distance 1 nodes, the success ratio is around 0.04–0.05, whereas for cores in the other two node categories, it lies at around 0.005. A failed CAS is always caused by the success of another thread's CAS. In particular, a CAS by thread p will fail if p executed its read of the target between the read and the CAS of thread whose CAS was successful. Thus, the numbers indicate that most threads align their read instructions with each other, causing repeated failures for the same set of threads. The delays inherent to the cache coherence protocol on the AMD machine thus repeatedly favor these 'mid latency' (distance 1) threads over their counterparts that are farther or closer to the memory.

2) *Intel Fairness:* The fairness test results on the Intel machine are shown in Figure 14. Only Node 0 and Node 3 are shown, as the other nodes' curves were almost exactly the same as Node 3. As could be expected, both Node 0 and Node 3 drop in throughput as the atomic delay grows, and eventually both reach approximately the same throughput.

We can see that in general, fairness here is not as skewed as on AMD; at high throughputs (corresponding to low atomic delay), Node 3 outperforms Node 0 by a factor of 1.4–1.8. Both node's performance degrades quickly, though at somewhat different speeds. At an atomic delay of 34 iterations (around 75 ns), unfairness is at its worst, with Node 3 outperforming Node 0 by a factor of 12.5. However, soon after that, starting at an atomic delay of 52 iterations, the two nodes are consistently within 10% of each other in terms of their throughput.

3) *Fairness Takeaways:* We conclude that the fairness of schedules of a lock-free algorithm is highly dependent on the algorithm itself, in particular, on the length of its atomic modify section. This observation is perhaps counterintuitive, especially for theoreticians in the field; a lot of literature on lock-free algorithms never accounts for 'local' work. However, the exact length of local operations within the atomic modify section can have a drastic effect on both fairness and performance. This is despite the fact that local work operates on the L1 cache and thus experience much lower latencies than memory instructions that access new or contended data. We thus recommend making efforts to minimize work in the atomic modify section when designing and implementing lock-free algorithms.

Furthermore, we note that despite fairness arbitration efforts within each node, fairness is not generally achieved among nodes. This is a similar observation to that made by Song et al. [24]. However, while they study workloads in which there is an uneven number of requests from competing nodes, we show unfairness even when all nodes issue the same number of requests. In general, to achieve better fairness even with relatively small atomic modify sections, it can be beneficial to design architectures to explicitly favor requests from the local node over those from remote nodes.

B. Focus

Recall the original intuition (Section I) for why focus may be useful in a hardware schedule. Ideally, to avoid wasted work, a thread should be able to keep a cache line in its private cache for long enough to execute both the read and the CAS instructions of its *atomic modify* section in a lock-free algorithm. However, this means that depending on the length of the atomic modify section of a given algorithm, the cache line must remain in one core's cache longer for sufficient focus.

Recall that when inferring the scheduling patterns of each machine in Section IV, we considered the *visit length* of a cache line at each core. That is, we measured how many memory instructions a single core can execute before the cache line leaves its private cache. Note that a schedule with better focus corresponds to a schedule with longer core visits. Thus, more focus is required from the schedule the longer the atomic delay is. We say that a hardware schedule has *meaningful focus* for a given lock-free algorithm if the entire atomic modify section of the algorithm fits in a single core visit.

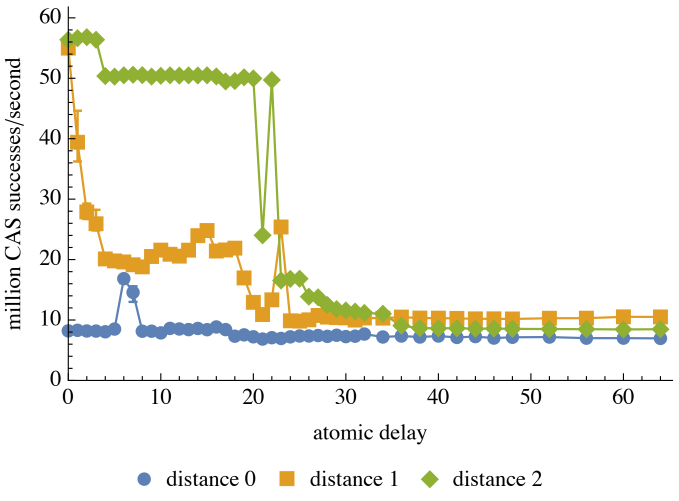


Fig. 15: AMD throughput of CAS operations for varying atomic delay with all nodes participating. A target is allocated each node. All cores accessing a given target are in the same node. In each run, all cores access targets the same distance away.

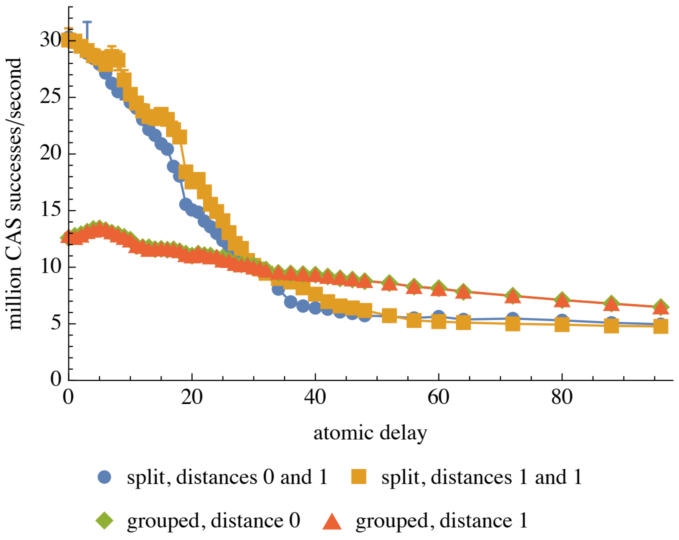


Fig. 16: Intel throughput of CAS operations for varying atomic delay with all nodes participating. A target is allocated each node. Showing results for four different target assignments. In *grouped* assignments, all cores accessing a given target are in the same node. In *split* assignments, the set of cores accessing a given target is split evenly across two nodes.

We now test how longer core visits observed in Section IV translate to meaningful focus for lock-free algorithms. Unlike previous experiments in this paper, we test focus using experiments with *multiple targets*. Specifically, we allocate one target on each node, and each core is assigned one target to access for the duration of the experiment. This means that each core is only directly contending with other cores accessing the same target. However, there may be indirect contention caused by traffic on the node interconnect. To exhibit a variety of core visit lengths, we run different types of experiments for AMD and Intel.

1) AMD Focus: Recall from Section IV-A that nodes that are 2-hops away from the memory they access have longer core visits on average. To test how these longer visits translate to meaningful focus, we conduct competition experiments with three different settings. In each setting, all cores access a target that is a fixed distance away. The results of this test are shown in Figure 15.

For small atomic delays, we observe a significant difference between the three settings. In particular, both distance 1 and distance 2 placements exhibit higher throughput than distance 0. Throughput at distance 1 drops near atomic delay 18. This indicates that at this point, a thread can no longer fit both its read and its CAS into the same visit. A similar drop happens for the distance 2 placement near atomic delay 23. In contrast, it appears that the distance 0 placement never fits a read and CAS into the same visit, even with atomic delay 0.

These findings make sense in light of the results of Section IV-A. Specifically, as shown in Table II, cores at distance 2 have longer visit lengths than those at distance 1. From the table initially appears as if distance 0 cores have visit lengths comparable to distance 1 cores. However, as shown in Figure 6, cores at distance 0 are frequently skipped in what is otherwise a mostly round-robin visit sequence. If we view these skips as "length 0" visits, then cores at distance 1 have visits longer than those at distance 0, whose visits can be so short that not even a single atomic instruction finishes executing.

All three thread placements eventually reach a steady throughput of around 7.2–10.3 million successful CAS operations per second. This happens at an atomic delay of 36 iterations, roughly corresponding to 125 ns on the AMD machine (see Section III). Distance 1 nodes display the highest throughput of the three categories in the steady state, outperforming distance 2 nodes by 20% and the Node 0 by 43%. This is consistent with the results from the fairness tests, but the difference in performance is smaller here.

There are some other phenomena that we do not yet know how to explain, such as the drops in throughput for distances 1 and 2 as atomic delay increases from 0 to 5 and the occasional throughput spikes. It is possible that some of these effects would be smoothed over by an experiment in which atomic delay was random rather than deterministic.

2) Intel Focus: Recall from Section IV-B that longer visits occur on the first core visited in a node, when the cache line travels between nodes. In particular, these long core visits happen only when cores of multiple nodes are active, rather than just one node. To test the effect of longer core visits on meaningful focus in the Intel machine, we therefore compare two types of competition experiments: the first is simply using all threads of one node, and the second uses the same number of threads, but splits them across two nodes. Just like we did for AMD, we run the experiments in parallel to create interconnect traffic. The results of this test are shown in Figure 16.

As expected given our knowledge of Intel's schedule, it is clear that for a small atomic delay, splitting the threads across two nodes produces significantly higher throughput than having them all on one node. At around an atomic delay of

30 iterations (approximately 66 ns), the runs on a single node start outperforming the split runs. This can be attributed to the lowered contention caused by such a high atomic delay. When contention is low, the dominating factor for performance becomes the latency of accessing the memory (or the L3 cache, in this case), which is known to be much lower for local accesses than for remote accesses.

3) Focus Takeaways: On both machines, we observed that for experiments with low atomic delay, higher throughput occurs on schedules that we know exhibit better focus. The higher focus seems to be *meaningful* only for an atomic delay of up to approximately 25-30 iterations, indicating that algorithms with atomic modify sections of around this length or shorter can benefit from such schedules.

However, more generally, it is clear that focus in the hardware schedule is extremely helpful for throughput; it would be desirable to achieve meaningful focus even for algorithms with a longer atomic modify section. This observation was made by Haider et al. [25]. Using simulation results, they showed that it can be very beneficial to allow each thread to *lease* a cache line for a bounded amount of time, and *release* it either when that time is up, or when it finishes its atomic modify section. Our results support those of Haider et al., but on real architectures rather than simulations. That is, even when all features of an architecture interact with each other, it can be beneficial to extend the implicit lease of a cache line that memory instruction schedules provide a thread.

VI. Related Work

Alistarh *et al.* [14] ran tests similar to our sequence experiments to verify the validity of their uniform random scheduler assumption. They ran the experiments on a single Fujitsu PRIMERGY RX600 S6 server with four Intel Xeon E7-4870 (Westmere EX) processors, but they used only one of its nodes. Our results for this setting are consistent with theirs; scheduling seems mostly uniformly random on a single Intel node. Our experiments, however, consider a much greater scope, noting when this random scheduling pattern falters.

NUMA architectures have been extensively studied. Previous works have designed benchmarks to understand the latencies and bandwidth associated with accesses to different levels of the cache and local versus remote memory on NUMA machines [20], [26], [27]. However, these papers did not consider the effect of contended workloads on NUMA access patterns.

A thorough study of synchronization primitives was conducted by David *et al.* [28]. Some of their tests are similar to ours. However, their setup is different; in all contention experiments, David *et al.* inject a large delay between consecutive operations of one thread. While we use a similar pattern for our focus and fairness experiments, we also test configurations that do not inject such delays. Thus, our work uncovers some performance phenomena that were not found by David *et al.*

Song *et al.* [24] show that NUMA architectures can have highly unfair throughput among the nodes. They also show that this unfairness does not always favor nodes that access local memory, displaying this behavior in VMs. However, they do not study lock-free algorithms or contention.

Performance prediction has been the goal of a lot of work, not only in the lock-free algorithms community [29], [30], [31], [32], [33]. Techniques range from simulation, to hand built models, to regression based models, to profiling tools. Goodman *et al.* present one such profiling tool [32]. While this produces accurate results, sometimes it is impractical to have the algorithm ready to use for profiling before performance predictions are made, since performance predictions can help develop the algorithm. Our work aims to obtain a high level performance model to guide algorithm design in its earlier stages. Furthermore, our benchmark can be used on any machine to gain an understanding of its underlying model.

VII. Conclusion

Analytical performance prediction of lock-free algorithms is a hard problem. One must consider the likely operation scheduling patterns on the machines on which the algorithm is run. Previous approaches assumed a random scheduler instead of an adversarial one, but did not show whether such an assumption is reflective of real machines.

In this work, we present a thorough study of scheduling patterns produced on two NUMA architectures, using our new benchmarking tool, Severus. Our experiments uncover several phenomena that can greatly affect the schedules of lock-free algorithms and make models based solely on uniform randomness seem inaccurate. In particular, we show that thread placement with respect to a contended memory location can be crucial, and that surprisingly, remote threads often perform better under contention than local threads.

On both tested machines, the reason for this rise in throughput seems to stem from improved *focus*, or the increased length of visits of the cache line for cores on remote nodes. This phenomenon has been largely overlooked in literature that aims to approximate the operation scheduler, other than a few exceptions [25]. Additionally, these focus benefits come at the cost of *fairness* on modern machines; not all cores on a machine experience these beneficial longer visits.

We believe that there are several takeaways and further directions from this paper. Firstly, fairness is not a given. This knowledge can affect algorithm design, as well as programming frameworks chosen; in a system with low fairness, a work-stealing scheduler may be crucial for ensuring a fair allocation of parallel tasks that leads to high throughput. Secondly, this paper casts doubt on previous works that assume requests for a cache line are simply handled in a random order, and shows that more careful modeling may be necessary. Furthermore, we've shown in our experiments that the length of the atomic delay (the delay between the read and the following CAS in a read-modify-CAS loop) has a significant—yet a priori unpredictable—effect on performance, since different platforms can behave drastically differently. Finally, we provide a tool that allows a user to test their platform and understand what assumptions are reasonable for them, and what factors might have the greatest effect on their algorithm's performance.

REFERENCES

[1] S. Timnat and E. Petrank, "A practical wait-free simulation for lock-free data structures," in *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, vol. 49, no. 8. ACM, 2014, pp. 357–368.

[2] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[3] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[4] T. E. Anderson, "The performance of spin lock alternatives for shared-money multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, 1990.

[5] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for numa-aware contention management on multicore systems," in *USENIX Annual Technical Conference (ATC)*, 2011.

[6] F. Ellen, Y. Lev, V. Luchangco, and M. Moir, "Snzi: Scalable nonzero indicators," in *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2007, pp. 13–22.

[7] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 1996, pp. 267–275.

[8] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *International Symposium on Distributed Computing (DISC)*. Springer, 2001, pp. 300–314.

[9] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *Journal of the ACM (JACM)*, vol. 53, no. 3, pp. 379–405, 2006.

[10] T. L. Harris, "Five ways not to fool yourself: designing experiments for understanding performance," https://timharris.uk/misc/five-ways.pdf, 2016.

[11] N. Ben-David and G. E. Blelloch, "Analyzing contention and backoff in asynchronous shared memory," in *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2017, pp. 53–62.

[12] N. Ben-David, G. E. Blelloch, Y. Sun, and Y. Wei, "Efficient single writer concurrency," *arXiv preprint arXiv:1803.08617*, 2018.

[13] D. Alistarh, T. Sauerwald, and M. Vojnović, "Lock-free algorithms under stochastic schedulers," in *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2015, pp. 251–260.

[14] D. Alistarh, K. Censor-Hillel, and N. Shavit, "Are lock-free concurrent algorithms practically wait-free?" *Journal of the ACM (JACM)*, vol. 63, no. 4, p. 31, 2016.

[15] A. Atalar, P. Renaud-Goud, and P. Tsigas, "Analyzing the performance of lock-free data structures: A conflict-based model," in *International Symposium on Distributed Computing (DISC)*. Springer, 2015, pp. 341–355.

[16] ——, "How lock-free data structures perform in dynamic environments: Models and analyses," *arXiv preprint arXiv:1611.05793*, 2016.

[17] M. Herlihy and N. Shavit, "On the nature of progress," in *International Conference on Principles of Distributed Systems (OPODIS)*. Springer, 2011, pp. 313–328.

[18] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[19] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.

[20] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore smp systems," in *International Symposium on Microarchitecture (MICRO)*. IEEE, 2009, pp. 413–422.

[21] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, "Black-box concurrent data structures for numa architectures," *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 51, no. 2, pp. 207–221, 2017.

[22] I. Calciu, J. Gottschlich, and M. Herlihy, "Using delegation and elimination to implement a scalable numa-friendly stack," in *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, 2013.

[23] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. M. Lohman, "Numa-aware algorithms: the case of data shuffling." in *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[24] W. Song, G. Kim, H. Jung, J. Chung, J. H. Ahn, J. W. Lee, and J. Kim, "History-based arbitration for fairness in processor-interconnect of numa servers," *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 51, no. 2, pp. 765–777, 2017.

[25] S. K. Haider, W. Hasenplaugh, and D. Alistarh, "Lease/release: Architectural support for scaling contended data structures," *ACM SIGPLAN Notices*, vol. 51, no. 8, p. 17, 2016.

[26] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 445–456.

[27] D. Molka, D. Hackenberg, and R. Schöne, "Main memory and cache performance of intel sandy bridge and amd bulldozer," in *Proceedings of the workshop on Memory Systems Performance and Correctness*. ACM, 2014, p. 4.

[28] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2013, pp. 33–48.

[29] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, "Predictive performance and scalability modeling of a large-scale application," in *Supercomputing, ACM/IEEE 2001 Conference*. IEEE, 2001, pp. 39–39.

[30] L. Carrington, A. Snavely, and N. Wolter, "A performance prediction framework for scientific applications," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 336–346, 2006.

[31] J. Zhai, W. Chen, and W. Zheng, "Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 305–314.

[32] D. Goodman, G. Varisteas, and T. L. Harris, "Pandia: comprehensive contention-sensitive thread placement," in *European Conference on Computer Systems (EUROSYS)*. ACM, 2017, pp. 254–269.

[33] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 368–377.

[34] N. Ben-David, Z. Scully, and G. Blelloch, "Severus," Aug. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3360044

Appendix

Our tool, Severus uses c++ source code that creates a specified multithreaded workload and measures various properties of its schedule, and then processes the output using gnuplot to create plots that are easy to interpret. It has been tested on Ubuntu 14.04 and 16.04, and requires the boost and libnuma libraries to be installed.

Severus is designed to be usable on many different architectures and workloads. We expect the results of the same experiments to be different on each machine. Thus, when evaluating our artifact, one should expect to see that the tool runs properly, and outputs data that is similar to what we report in this paper, but potentially showing different numbers and patterns.

We tested Severus on two different architectures: (1) an Intel Xeon CPU E7-8867 v4 machine with 4 nodes and 72 cores, each with two-way hyperthreading, and (2) an AMD Opteron 6278 machine with 8 nodes and 32 cores, each with two-way hyperthreading. If the reviewer has access to one or both of these machines, the data gathered by running our experiments should be very similar to what we report in this paper. Severus can run on different NUMA machines that use Ubuntu, but will produce different results. A machine that runs Severus must have atoimc compare-and-swap (CAS) and atomic fetch-and-add (F&A, also called xadd).

A. *Artifact check-list (meta-information)*

- **Compilation:** g++ verion 5+ (Tested on 5.3.0 and 6.4.0)
- **Run-time environment:** The tool was tested on Linux Ubuntu versions 14.04 and 16.04. Software dependencies are on boost and libnuma libraries. Root access is needed to install these dependencies if not already present, but is not needed for the tool itself.
- **Hardware:** We recommend Intel Xeon E7-886 or AMD Operton 6278 to verify results reported in this paper. Similar machines should work, and yield comparable results in some experiments, while possibly revealing new patterns for other experiments. The machine must have atomic CAS and F&A instructions and a NUMA architecture.
- **Run-time state:** For most accurate results, this program should run alone on the machine (no network or cache contention).
- **Execution:** The program should execute solo. It runs for approximately 10 mins to complete the experiments in `paper.sh`.
- **Metrics:** Number of memory accesses per thread/node, some other related measurements.
- **Output:** Data files are outout in .txt format, and then plots are created in .pdf files. Plots highlight important properties of the execution, including the execution trace, and how many memory accesses were executed by each thread, by access type (read, write, etc).
- **How much disk space required (approximately)?** Less than 100M
- **How much time is needed to prepare workflow (approximately)?** Less than 5 mins
- **How much time is needed to complete experiments (approximately)?** Less than 10 mins
- **Publicly available?** Yes, on Github: https://github.com/cmuparlay/severus
- **Code licenses (if publicly available)?** Apache License 2.0
- **Archived (provide DOI)?** Yes, on Zenodo: https://doi.org/10.5281/zenodo.3360044 [34]

B. *Description*

1) How delivered: Available on GitHub: https://github.com/cmuparlay/severus.

2) Hardware dependencies: We recommend Intel Xeon E7-886 or AMD Operton 6278 to verify results reported in this paper. Similar machines should work, and yield comparable results in some experiments, while possibly revealing new patterns for other experiments. The machine must have atomic CAS and F&A[7], and a NUMA architecture.

3) Software dependencies: For compiling and running the source code:

- **GCC.** Available as `gcc` from most package managers.
- **Boost: Program_options.** Available as `libnuma-dev` from most package managers, or download from https://www.boost.org/doc/libs/1_66_0/more/getting_started/unix-variants.html.
- **libnuma.** Available as `libnuma-dev` from most package managers.
- **gnuplot.** Only required for producing plots. Available as `gnuplot` from most package managers, or download from https://sourceforge.net/projects/gnuplot/files/gnuplot/5.2.7/.

C. *Installation*

a) Downloading the code.: Clone or download the repository from GitHub at https://github.com/cmuparlay/severus. All the artifact code is contained in a single directory, and all the scripts are intended to be run from that directory.

b) Installing dependencies.: Follow the instructions in Section B3 to download and install the software dependencies if they are not already installed on your machine.

D. *Experiment workflow*

The easiest way to use the tool is to run `./paper.sh`. This works on any machine and reproduces the experiments that were run and presented in this paper. The script has four modes:

- `./paper.sh easy` replicates the experiments from Sections IV and V-A on any machine.
- `./paper.sh amd` replicates the experiments on the AMD machine in Sections IV-A, V-A1, and V-B1. This mode requires exactly 8 NUMA nodes.

[7]The benchmarks use F&A instructions but only ever add 1, so the code could be easily adapted to a machine with only an atomic fetch-and-increment (F&I) instruction.

- `./paper.sh intel` replicates the experiments in Sections IV-B, V-A2, and V-B2. This mode requires exactly 4 NUMA nodes.
- `./paper.sh mapping MAP` replicates the experiments from Sections IV and V-A and adapts the experiments from Section V-B on any machine. See below for the details of the argument `MAP`.

By default, all results are output to `./output/` and its subdirectories. The output directory can be overriden with a command line argument. Run `./paper.sh --help` for full details.

The `./paper.sh mapping MAP` option, with help from the user in the form of the argument `MAP`, recreates a version of the focus experiments from Section V-B. `MAP` is list that maps each node to the node it will access during the experiment. For example, on a 4-node system, `./paper.sh mapping 1 2 3 0` has Node 0 access Node 1, Node 1 access Node 2, etc. The intention is that the list is a permutation such that no node is accesses itself, but other configurations may also be interesting.

A lower-level interface is the `go.sh` script, which runs single experiments. (The `paper.sh` script is mostly a series of calls to `go.sh`.) Run `./go.sh --help` for a short manual on how to use it. The script produces a text file with the output data from the execution it ran, and by default creates plots in pdf format using gnuplot. Furthermore, a short summary of the execution, including the total number of successful and unsuccessful accesses to memory, is output to the commandline. If for any reason the plots are not desired, they can be disabled with the `--no-plot` option.

Both `paper.sh` and `go.sh` avoid rerunning experiments, which can be lengthy, if output files already exist. To force rerunning an experiment, delete its output file. (Deleting only)

E. Evaluation and expected result

It is difficult to compare the results across different machines. In general, it is not the exact throughput reported, but the scheduling patterns observed that are the main take-away that should be considered. Overall, when running on a new machine, the desired result is the ability to understand the scheduling patterns produced. The code should compile, run, and produce informative plots.

Some patterns should remain fairly constant across machines. We expect that in the sequence experiments, for example, the node on which the memory is allocated (node 0 as default) should perform worse than others on most NUMA machines.

If running experiments on Intel Xeon E7-886 or AMD Operton 6278, the scheduling patterns produced should be similar to the ones reported in this paper, for all experiments.

On any machine, it is important to ensure that this program is the only one running on the machine when gathering data, since the schedule is easily skewed by other things happening in the system.

F. Experiment customization

Severus is parametrized, and allows the user to control the threads participating, node(s) on which memory is allocated, and the amount of delay threads should wait between memory accesses. For details, run `./go --help`.

G. Notes

In this paper we produced the plots using a Mathematica library. However, Mathematica is proprietary, so not all users have access to it. Furthermore, our Mathematica library is currently not configurable to handle machines other than the Intel and AMD machines used in this paper. In the interest of open access and portability, our artifact uses gnuplot to generate versions of nearly all of the figures in this paper.