Spread-Spectrum Organization for Concurrent Pools

Kavon Farvardin
University of Chicago
kavon@cs.uchicago.edu

John Reppy
University of Chicago
jhr@cs.uchicago.edu

ABSTRACT

A lock-free concurrent pool (bag) data structure that is simple to implement, yet has performance competitive with the much of the prior work, is presented. The pool offers a flexible *pseudo-priority* functionality that takes advantage of the unordered semantics of a pool. Consumers can define a set of item categories for which they have a ranked affinity for, and these items are discovered using a new temporal organization technique called *spread-spectrum organization*, which is inspired by solutions for signal multiplexing in radio communications.

1. INTRODUCTION

A pool (bag) is a data structure with two operations: `put`, which adds an element to the pool, and `get`, which removes and returns an arbitrary element from the pool. Lock-free pools are widely used in parallel programs, often serving the rôle of a central work queue or resource cache for heavy duty computations in machine learning, data mining, and NP-hard optimization problems [32]. While many data structures satisfy the interface of a pool, an underlying implementation with strict ordering, such as first-in first-out (FIFO) and priority queues, quickly becomes a sequential bottleneck. Contention in an ordered concurrent data structure is very difficult to eliminate [6], thus the focus across much of the literature has been on relaxing the semantics of the operations for use in algorithms where strict adherence is not required [4, 5, 15, 26, 25]. Thus, many specially designed lock-free pools have been developed to to reduce contention by distributing concurrent operations across many smaller data structures [13, 29, 7, 2, 14, 10].

Prior work in lock-free pools has mainly focused on minimizing latency by keeping the load among all locations evenly distributed using approaches such as round-robin [10, 14] or random [7, 2, 14] item insertions and removals. Among the most scalable pools, such as those by Gidron et al (SALSA) [13] and Sundell et al (CB) [29], the design is such that a fixed number of threads normally operate on their own sub-regions of the pool, and resort to work-stealing in case of imbalance. Work-stealing schedulers have proven to be effective at reducing synchronization and improving memory locality, but they were not designed with respect to the semantics of a *linearizable* [16] pool data structure that must observe total emptiness. Thus, in order to take advantage of work-stealing's benefits and still be linearizable, both the SALSA and CB pool `get` operations require *considerable* effort to implement correctly, which impedes their adoption. For example, in higher-level languages with concurrency support such as Manticore [12], Haskell, or Java, the runtime system typically lacks proper support for the forced CPU scheduling suspension technique, originally proposed by Dice et al [9], that is used in SALSA to elide synchronization for the steal-free path[1].

Contribution We propose a *simple*, scalable lock-free concurrent pool that achieves performance competitive with more complex and hard-to-implement designs. Our pool can be implemented using lock-free data structures already available in many standard libraries. Additionally, our pool exposes an item prioritization functionality directly to the user. The user can define a dynamic set of categories into which certain types of items can be placed and later sought out according to a priority distribution. We do this without priority queue sub-structures, as they would undermine the performance expected of a pool.

Our motivating use case for a *pseudo-priority pool* is to allow items in the pool to be organized and retrieved based on their memory locality relative to each retriever. This is particularly important for non-uniform memory access (NUMA) architectures, where memory performance has costs according to a distance function.

The key design difference with respect to prior work is that we do not ascribe to a *spatial* binning and search which organizes items in the pool according to their type, such as its location in the memory hierarchy or coming from a particular thread, to elide the need to do work-stealing for balancing. Instead, we use a novel temporal organization technique called *spread-spectrum organization* that achieves item binning and selection in a reliable and efficient way. Spread-spectrum organization is inspired by a solution to an analogous signal multiplexing problem in radio communications.

Under consideration for SPAA '16

[1] A newer technique has emerged but is not general [23].

2. BACKGROUND

2.1 Related Work

Data structures that do not relax ordering requirements when used as a pool have poor scalability. Dedicated pool designs such as the Café pool by Basin et al [7], which is based on a list of trees, and the ED pool by Afek et al [2], which applies an elimination technique to cancel out simultaneous `put` and `get` operations, also have limited scalability.

More recent pools, such as MultiLane by Dice and Otenko [10] and Distributed Queues (DQ) by Haas et al [14] both use a work-sharing principle similar to ours, though they lack the pseudo-prioritization capability proposed in this work, which can be used to improve memory locality. MultiLane uses `fetch-and-add` atomic operations to update cursors, and while we also have the concept of a cursor, ours are private and we use many of them.

The highly scalable pools proposed by Sundell et al (CB) [29] and Gidron et al (SALSA) [13] both use work-stealing with independent sub-pools, and as mentioned in the introduction these are also quite complicated designs to implement in practice. Our goal in this work is to have performance competitive with these designs *without* complexity; our entire data structure spans less than 100 lines of very straightforward C++ code.

While priority pools are not a new concept [17], nor are relaxed priority queues [5, 25], our work differs in that we do not offer a `delete-min` operation in the traditional sense. In the relaxed priority queue MultiQueue, while items are inserted at random, `delete-min` locks two queues and compares the head elements, removing and returning the one of lower cost. In our design, we *always* take the first item touched in the pool, making a simple decision to improve our later guesses *after* the item has been removed. Extra hesitation may put scalability with respect to state-of-the-art pure pools in jeopardy. Our pseudo-prioritization comes from the ability to improve these first-guesses by tracking down where desirable items live, as we will see in the next section.

2.2 Spread-Spectrum Organization

In radio communication a *frequency spectrum* needs to be shared among multiple users in such a way that simultaneous broadcasts avoid causing electromagnetic interference. Two techniques used to effectively share a frequency spectrum are *Frequency Division Multiple Access* (FDMA) and *Code Division Multiple Access* (CDMA).

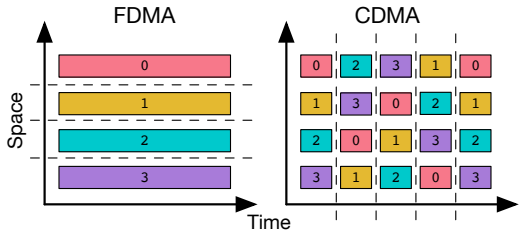


Figure 1: Two multiplexing techniques used in radio systems. CDMA uses jump sequences to track signals.

As its name suggests, in an FDMA system the frequency spectrum is divided up and each piece is used for separate conversations. CDMA systems on the other hand mix the conversations across the entire spectrum, using unique mixing patterns to represent each conversation. Each pattern is essentially a pseudorandom closed walk that visits each point in the spectrum, but because the walk is known to all parties in the conversation, *listeners can follow speakers* in the same direction as they jump around [28, 30]. To an observer outside of the conversation, this random jumping is indistinguishable from uniform random noise, which is why spread-spectrum systems are resilient in the presence of a signal jammer, whether benign or malicious.

The analogy to lock-free pool designs is that work-stealing pools are similar to an FDMA system: threads effectively own one spatial location, or bucket, in a *fixed partitioning* of the pool, and in the case where items are not found, the threads look elsewhere, which is like fiddling with the radio dial. In this work we propose a pool that based on the idea of CDMA systems where we have a fixed number of *virtual buckets* users can assign their own meaning to. These virtual buckets are represented using a pseudorandom insertion and removal sequence that producers and consumers follow to track each other with good probability, which we show empirically in Section 4.0.2. In terms of load balancing, when a producer is overactive, it is effectively contributing *extra noise* uniformly across the data structure to all consumers, even those trying to listen for other items.

3. THE CDMA POOL

In this section we apply the ideas of spread-spectrum organization to develop a simple lock-free pool.

3.1 Data Types

3.1.1 *The Pool*

The primary component of the pool is an array, `Q`, of length `period`, that consists of initially empty non-blocking (lock-free) LIFO or FIFO queues. The main requirement of these individual queues is that they preserve the order in which items are inserted and removed, as we use this timing to synchronize producers and consumers in spread-spectrum organization technique. We will consider Treiber's lock-free stack [31] for the LIFO queue and Michael and Scott's lock-free FIFO queue [21] in this paper because they are ubiquitous and easy to implement. In Section 4 we will see that these less sophisticated queues work well because contention is effectively eliminated.

```
type CDMA_Pool<QueueType> {
 QueueType Q[period];
 int mult; // LCG multiplier
 int buckt[numBkts]; // LCG increments
 int maxDwell; // max time at any location
}
```

The *period* is a power of 2 greater than 4 and $numBkts \leq \frac{period}{2}$ for reasons described in Section 3.1.2. In order to implement the signal tracking scheme for our buckets, we need the ability to distinguish items belonging to each bucket. Thus, we mark items as they are inserted into the pool with a bucket ID by marking the queue node it is placed in. The marking can be done by simply adding another integer field to each node or wrapping the item's type to include the extra value.

3.1.2 The Buckets

While spread-spectrum systems typically use a linear-feedback shift register to generate a pseudorandom number sequence, we will be using a *linear congruential generator* (LCG), which is given by the recurrence

$$n_{i+1} = (a\, n_i + c) \bmod p$$

where a is called the multiplier, c the increment, and n_0 the seed; all of which are strictly less than p, the period. We use LCGs in particular because they have useful properties[2] that we take advantage of in order to quickly compute the paths taken by each producer instead of taking up memory.

We say that a pseudorandom number generator has a *full period* if every integer in the interval $[0, p)$ is generated exactly once after p iterations of the generator, given any initial value. A full period is needed by our algorithm so that our `get` operation is linearizable. An LCG has a full period if p is a power of two that is at least 4, $a \equiv 1 \pmod{m}$, and c is any odd number such that $0 < c < p$ [18].

OBSERVATION 3.1. *Let the family $F_{p,a}$ be the set of LCGs generated by choices of c when we fix p and a. Then for all sequences r, s in $F_{p,a}$ where $r \neq s$, for every i, if $r_i = s_i$ then $r_{i+1} \neq s_{i+1}$.*

PROOF. Let c and d be the constants which define r and s and WLOG assume $c < d$. For any x, $ax + c \equiv j \not\equiv ax + d \pmod{p}$ because $c, d < p$. □

The significance of Observation 3.1 is that, even if we pick pathological constants for an LCG or significantly reduce its period length and thus its apparent randomness, the sequences within any family $F_{p,a}$ *relative to each other* are still unique and have the contention spreading qualities for which random number generators are used in concurrent data structures. Namely, these sequences exhibit behavior very similar to the balancers in diffracting trees [27] where, after two threads have arrived at a common place, they will leave towards two opposite parts of the data structure. In our case, when different bucket users meet at a common location, their sequences dictate that their next locations will always be different.

3.1.3 The Users

Each user of the pool maintains its own context, whether the user is a consumer or producer. These contexts act as cursors that are continuously adjusted to keep in sync with the bucket items being tracked *without extra communication*, and are kept in thread-local storage since they are private[3]. For example, if a thread is self-communicating, or simultaneously sending items to two buckets, then two contexts would be used by that thread.

```
type PoolContext {
 int bktID; // subscribed bucket
 int loc; // current location
 float dwell; // time left at loc
 float penalty[numBkts]; // time penalties
}
```

[2]Notwithstanding their "gross" lattice structure [19].

[3]Conceivably contexts could be shared in order to build a dynamic bucket allocation system. We have not yet pursued this idea.

3.2 Pool Operations

3.2.1 Put

The `put` algorithm is shown in Figure 2. It should be thought of as part of an iterated process to place items in the pool such that its pattern is distinguishable and avoids contention. Using the machinery from Section 3.1.2 the `put` operation's unique pattern given a context is easy to compute. The `next` function on line 3 of Figure 2 computes the next value in the LCG sequence defined by the multiplier and period for the particular pool operated on, given the subscribed bucket's unique increment and the current location.

A key concept we borrow from CDMA systems is *dwelling* at each location for a short period of time, which we think of as the number of operations performed before jumping to another place, as the group of similar items being spread about the pool act as a signal synchronization point for tracking. An illustration of this process is depicted in Figure 4a, where the dwell time was set to 4 and the producer's next `put` operation will cause a jump to the next location before inserting the item.

Input: The user's context (cxt) and the item.
1: **if** cxt.dwell ≤ 0 **then**
2: cxt.dwell ← maxDwell
3: cxt.loc ← next(cxt.loc, cxt.incr)
4: **end if**
5: Q[cxt.loc].insert(item, cxt.bktID)
6: cxt.dwell ← cxt.dwell - 1
7: **return**

Figure 2: Place an item into a CDMA pool. Dwelling is used to create pockets of items from the same bucket.

Note that we tag the items as they are placed in the queues in order to guide consumers as they retrieve items *without* further thread communication. In Figure 4b we see that the producer's remaining dwell time is 2, yet its two prior items placed at that location as well as one item tagged with a different bucket ID, were picked up by the consumer that has now passed by.

3.2.2 Get

The `get` operation is where item tracking and prioritization takes place, and it is shown in Figure 3. The algorithm can be broken up conceptually into two parts: *searching* and *tuning*.

The search loop on lines 7-19 starts off with an initial guess that is determined by the user's context on line 5. On each iteration of the loop we attempt to remove an item from the queue. In the case of the queue being empty, the version of the empty queue as it was observed during the removal operation is saved locally. The queue version values serve to prevent the ABA problem when it comes to the emptiness verification loop on lines 20-26. For an MS queue or Treiber's stack, the queue version values are just the ABA counter values their standard implementation uses. This local versioning technique was adapted from Haas et al's pool [14] for linearizability, which we discuss in Section 3.2.3.

The tuning takes place on lines 1-4 and 9-16. On line 14 we inspect the removed item's bucket ID to determine what time deduction we should make for the current location. The time deduction for an item of the same bucket as the user's

```
Input: The user's context (cxt).
Output: Some item, or None if it is empty.
 1: if cxt.dwell ≤ 0 then
 2:     cxt.dwell ← maxDwell
 3:     cxt.loc ← next(cxt.loc, cxt.incr)
 4: end if
 5: idx ← cxt.loc                                  ▷ initial guess
 6: start:
 7: for i = 0 . . . (period - 1) do
 8:     res ← Q[idx].remove()
 9:     if res.foundAnything then
10:         if cxt.loc ≠ idx then                 ▷ somewhere new
11:             cxt.dwell ← maxDwell
12:             cxt.loc ← idx
13:         end if
14:         cxt.dwell ← cxt.dwell - cxt.penalty[res.bktID]
15:         return res.item
16:     end if
17:     chk[i] ← res.version                      ▷ remember ABA counter
18:     idx ← next(idx, cxt.incr)
19: end for
20: for i = 0 . . . (period - 1) do               ▷ emptiness check loop
21:     ver ← Q[idx].version()
22:     if chk[i] ≠ ver then
23:         goto start                            ▷ Q[idx] changed, try a removal!
24:     end if
25:     idx ← next(idx, cxt.incr)
26: end for
27: return None
```

Figure 3: Remove an item from a CDMA pool. Tracking of a bucket is done through the adjustment of the dwell time and prioritization through custom time penalties.

context is always set to 1 to match the producers. The time deductions for the other items is where we are afforded prioritization flexibility because we can assign meanings to the items in the buckets and then corresponding penalties. The get operation is depicted in Figure 4 where in part (a) the consumer is correctly tracking the producer because its dwell time is 1 and there is one item from bucket 2 left for it to pick up.

In part (b) of Figure 4, during the time the consumer spent performing `get` operations, the producer performed only two more `put` operations, leaving the producer with a remaining dwell time of 2. Observing only 2 of the expected 4 items, the consumer unexpectedly picks up an item from bucket 1. The penalty for doing so was equal to `maxDwell`, which represents an infinite penalty, and caused an early jump. The consumer will perform an early jump again after the consumer's next `get` operation.

The early jumping by the consumer after losing sync with a producer that was too slow in comparison achieves load balancing in a CDMA pool. More active producers for different buckets will have spread their items in such a way that it appears as a uniform mix to others, and early jumping becomes a means of sampling from this distribution until the desired items are found again.

Compressing ABA Counters.

Locally saving ABA counter values during the `get` operation might become a performance issue if the number of queues to check becomes large. At the highest levels of parallelism, optimal performance is seen using a larger number of queues, such as 1024 (Section 4.0.2). In a real-world application where the pool happens to be mostly empty, saving 8-byte ABA values can mean writing to 8KB of stack space[4], potentially causing extra L1 cache pressure. Our solution is to use a *cyclic-redundancy check* (CRC) [24] that is available on conventional x86 processors supporting SSE 4.2 and on the ARMv8. For the x86, the CRC instruction implements CRC-32C and has a latency of 3 clock cycles on Intel's processors since Sandy Bridge [1] so it is quite cheap.

Instead of storing each individual ABA value in the `chk` array on line 17 of Figure 3, we can fold two or more values at a time using a CRC and just store the one value, cutting memory usage by at least one half. The emptiness check loop would be augmented to recompute the current CRC values, rewinding to the last check point before the part of the subsequence where the CRC value mismatched. The rewinding can be performed by simply remembering the last checkpoint, or in the case of high compression where a

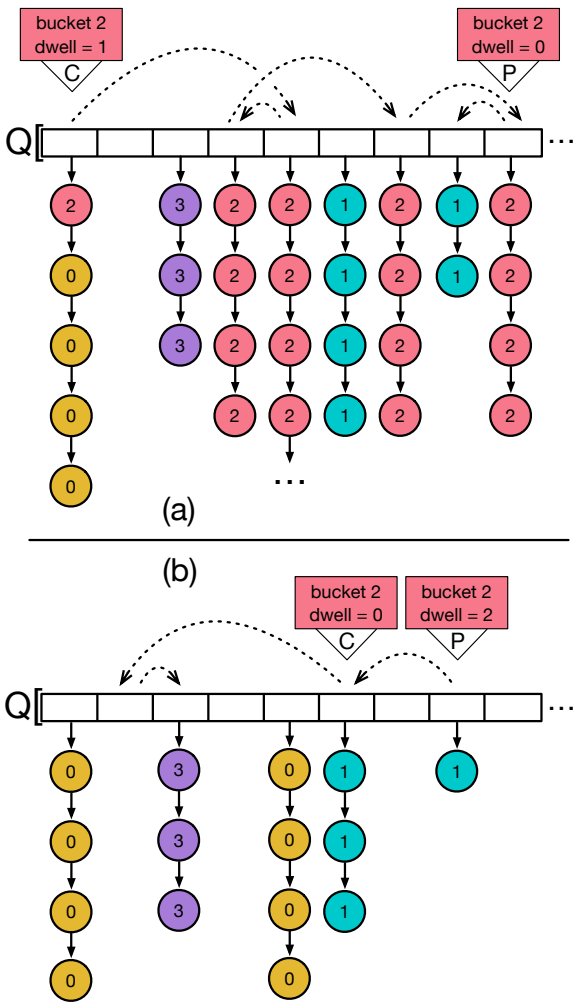


Figure 4: CDMA pool operation using LIFO queues. In (a) the producer has left a trail of values and in (b) the consumer catches up and passes the slower producer.

[4]Our specific implementation uses 2-byte ABA counters stuffed in the upper part of each pointer value, so it is not an issue for us.

rewind may go too far, a random position in a sub-interval of an LCG sequence can be computed in $O(log(period))$ iterations using the jump-ahead algorithm by Brown [8].

3.2.3 Pool Correctness

We claim that our pool is correct with respect to the semantics of a lock-free pool. Our pool is lock-free because (1) the underlying queues are lock-free and (2) the `get` operation only gets stuck in a loop if an ABA counter changed on the verification pass on lines 20-26 in Figure 3, but such a counter changes if and only if an enqueue operation succeeded, thus at least one thread made progress. Linearizability follows from Haas et al's pool [14], where we confirm that the pool was empty at some point by checking the ABA counters. The intuition behind the proof is similar to checking if a city street is empty before crossing it by looking left-right-left for cars. If no cars were seen approaching in all three looks then the street was empty at some point.

4. EVALUATION

This section details experiments we carried out to help understand the performance tradeoffs and characteristics of the pool proposed in this work. Our experiments were conducted on a Linux machine equipped with 60GB of memory and two Intel Xeon E5-2697's, both of which have 12 cores that support 2 threads each, yielding 48 hardware threads total. All code[5] was written in C++ and compiled with GCC 4.8.1 using the same aggressive optimization flags. We also replaced `glibc`'s memory allocator with the more efficient `jemalloc` [11] and all efforts were made to minimize memory allocator overhead for all algorithms tested in order to emphasize their design choices. The pools we evaluated are

- **CDMA & CDMA (FIFO)** The pool presented in this work, which we default to using LIFO queues, but we also test a version using FIFO queues. The period size was fixed at 2048, dwell time at 64, and the dwell penalties set to 1 for the same bucket and infinity for the rest so that we match one producer up with one consumer. We separately investigate the CDMA pool parameter space in Section 4.0.2.
- **SALSA** The pool by Gidron et al [13]. We use their implementation along with its optimal parameters of 200 preallocated *chunks*, which are array-like queue nodes, and 1000 items per chunk.
- **CB** The pool by Sundell et al [29]. Our implementation is based on the version used by Gidron et al in their evaluation of SALSA, but we fixed a bug in the linearizable emptiness condition. We use a chunk size of 128 and preallocate 200 chunks, which gave the best performance.
- **DQ** The most performant pool by Haas et al [14] that uses a random load balancer (1-RA), using the their implementation. Setting the number of queues to 80 showed the best performance.
- **LCRQ** The FIFO queue by Morrison and Afek [22] using their implementation and a ring size of 2^{17}. We include LCRQ as our baseline for peak performance expected of a single FIFO queue when used as a pool.

[5]Available at https://bitbucket.org/kavon/bonnenuit

For the implementations of DQ, CDMA, and LCRQ, queue nodes are bump allocated in large (32MB) blocks of memory to reduce frequent `malloc` calls, and the blocks are not reused. The SALSA pool relies on its own memory recycling system using hazard pointers so that producer threads gain information about load-balancing, since consumers maintain their own pools of free chunks and a lack of available chunks indicates an overloaded consumer. Thus we could not preallocate too many of these chunks without causing a spike in work stealing operations. The CB pool also uses hazard pointers, but removing memory reclamation degrades performance due to poor memory locality.

We did not consider other pools such as Café [7] or ED-trees [3] because their performance in prior evaluations did not scale as well as those tested [13, 29, 14] and their implementations were not on hand.

4.0.1 Producer-Consumer Benchmark

In this experiment, every thread is assigned to be either a producer or consumer of fake items, but between each operation a variable amount of work is performed to simulate the work needed to produce and consume such items. The work performed after each operation is a loop that generates one random number per iteration using the Xorshift128 algorithm [20], thus each loop iteration adds a handful of shift and xor instructions to be performed on variables local to each thread. Throughput is measured by counting the total number of items removed from the pool within in a 2 second wall-clock window, which starts counting down immediately before the `put`/`get` loop once all threads have met at a rendezvous point. All reported values are averages of 10 runs. We consider the following situations using this benchmark.

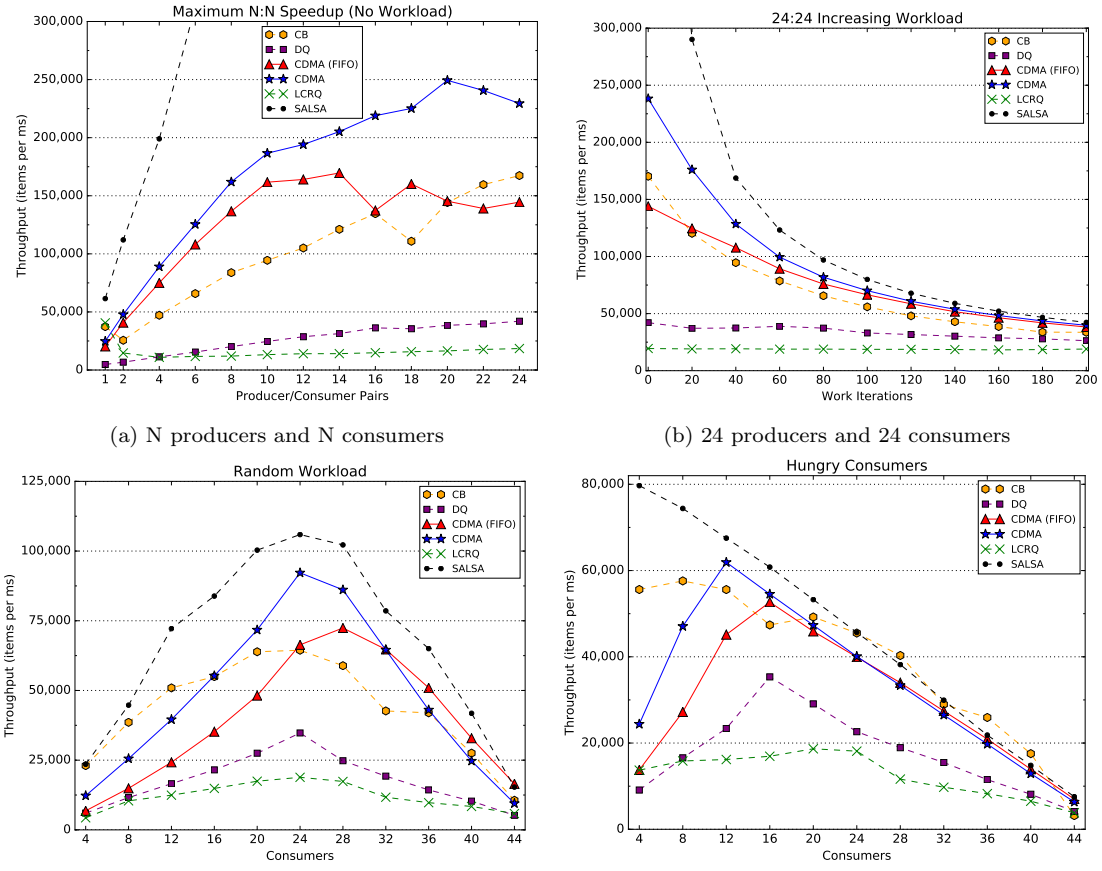
N:N Speedup.

In order to test the maximum scalability of the pools when the ratio of producers to consumers is even, we set the number of work iterations to zero, and the number of producer-consumer pairs N is varied from 1 through 24. As seen in Figure 5a the CDMA pool is roughly 1.43-2x better throughput than the CB pool, and roughly 4x better throughput than DQs, at higher levels of parallelism.

While the CDMA pool does not perform better than SALSA, we also did not expect it to: our goal was to create a pool that easy to implement correctly. In SALSA, expensive synchronous instructions such as `CAS` or memory fences are avoided in its no-stealing path, and the tradeoff is that stealing is expensive and tricky, however, stealing almost never happens in this balanced scenario.

24:24 Workload.

To show the performance in a real-world scenario where other code is executed between operations, we fix the number of threads to 24 producers and 24 consumers, and vary the number of iterations of work between each operation. In Figure 5b we see that DQ and LCRQ are mostly invariant with respect to the amount of work done between iterations. For LCRQ this can be explained by the fact that there is still a sequential bottleneck and many threads. SALSA is very sensitive to additional work, dropping from roughly 700 to 300 thousand items per millisecond by 20 iterations. The



(a) N producers and N consumers

(b) 24 producers and 24 consumers

(c) N consumers and 48-N producers

(d) N consumers and 48-N producers

Figure 5: Performance measurements of various pools in the producer-consumer benchmark, higher throughput is better. The CDMA pool's performance is competitive with prior designs.

CDMA pool shows better performance than the CB pool throughout the work iterations.

Random Workload.

Using all 48 threads, we vary the ratio of producers to consumers, but this time *all* threads pick a random number between 0 and 200 at the start of each 2 second run, and that value is the number of iterations of work performed for the entire run. This situation simulates unbalanced workloads and the results are shown in Figure 5c. The throughput curves of CB and CDMA initially cross in approximately the same horizontally shifted places as in Figure 5d presmably due to the consumer pickiness described in the next scenario.

Hungry Consumers.

Using all 48 threads, we vary the ratios of producers to consumers, where the producers perform 200 iterations of work while consumers perform zero. This situation stresses the ability for consumers to find items in a totally starved situation. In Figure 5d we see that for the CDMA pool, the performance is lower when there are few consumers because those consumers quickly eat up the items in the bucket they are after because their corresponding producer is slow. Then, the consumers begin jumping around the pool, taking items from the other buckets, which have an abundant number of items because there is no corresponding consumer tracking them. In these producer-consumer benchmarks, we set the priority distribution (dwell penalties) of each consumer to be very picky: an infinite penalty (the max dwell time) for items from all other buckets, and a penalty of 1 for the same bucket. If we make consumers less picky about the buckets from which the items are taken from, such as setting the other-bucket penalty to half of the dwell time, the throughput improves by 3x, to roughly 72 thousand, when there are 4 consumers in this scenario.

4.0.2 CDMA Pool Characteristics

In this section, we investigate the behavior of a CDMA pool with respect to its parameters: dwell penalties, maximum dwell time, and period size.

Item Prioritization.

One of the flexibilities afforded by the CDMA pool is the dwell penalties, which controls which types of items a consumer is willing to dwell for. In Section 4.0.1 we considered a producer-consumer benchmark and the CDMA pool's buckets had arbitrary meaning and used picky penalties: consumers dwell for the full dwell time when an item of the same bucket is found, but jump immediately otherwise. If it is known that certain items are equivalent in cost for the consumer, such as items being within the same NUMA node, we can change the consumer priorities without recreating or re-

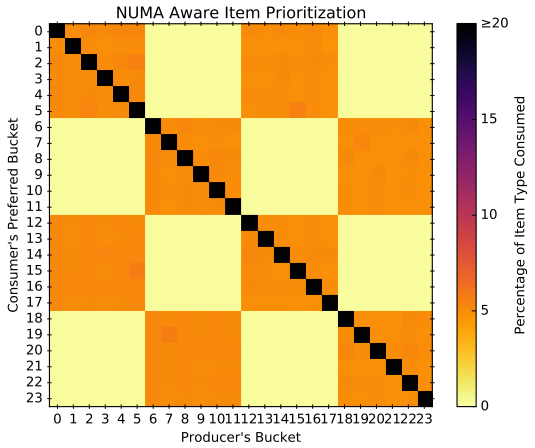


Figure 6: The CDMA pool allows consumers to choose which types items to avoid, such as those far away according to NUMA distance. Dwell penalties for a bucket allocated to any producer within the same NUMA node relative to each consumer was set to 1.5 and for consumer's most preferred bucket it was set to 1. Darker is better and the black parts in the heat map are exactly 40% for all consumers, giving a total of 95% signal quality.

designing the data structure. In Figure 6 we see that bucket tracking can be easily modified to suit more complicated notions of the memory locality. We used the same period and maximum dwell times, 2048 and 64 respectively, as in our benchmarks earlier and took an average of 10 runs while looking at the types of items retrieved. The checkerboard pattern is an artifact of the underlying CPU's pattern of assigning core IDs to NUMA nodes. The 95% signal quality normally seen with picky priorities becomes spread out, and each consumer now has 40% of all items removed coming from their preferred bucket (one producer was matched up with one consumer), plus 5% of items coming from eleven other producer's buckets, but only those producers on the same NUMA node. There were 12 producers and 12 consumers assigned to each NUMA node and two NUMA nodes total. The advantage of relaxing consumer pickiness is to see better throughput in some situations, as discussed in the Hungry Consumer scenario in Section 4.0.1.

Contention and Throughput.

We varied the maximum dwell time and period size in order to get a sense of what effect this has on the contention and throughput. In particular, as a control in this experiment we changed the `next` function in the CDMA pool algorithm to generate the same linear sequence, $s_{i+1} = s_i + 1 \pmod{period}$ instead of using the specially designed LCG sequences. The linear sequence combined with a dwell time of 1 gives a traditional round-robin approach.

We measure contention and throughput using the producer-consumer benchmark from Section 4.0.1 with zero work iterations. Contention is estimated by measuring the failed compare-and-swap (CAS) instructions per `put` or `get` operation performed by any thread and then taking the average of 10 runs. Throughput is measured just as in the regular producer-consumer benchmark.

In Figure 7 the relationship between period size and dwell time in various configurations is shown. It is clear from the figure that for the LCG sequences, contention is significantly lower. In Figure 8 we see the same data as in the previous figure but its throughput instead of contention. There is not a significant difference in throughput between the two types of sequences, though linear sequences are somtimes better due to cache effects. While the significantly lower contention typically suggests better throughput, the CDMA pool appears to have hit a limit due to its use of expensive CAS operations, regardless of whether they are contended. Thus, we believe a more optimistic synchronization technique for the underlying queues, such as transactional memory may lead to even better throughput in the CDMA pool.

5. CONCLUSIONS

We presented a lock-free concurrent pool (bag) data structure using a novel spread-spectrum organization technique, which is easy to implement for real applications and reuses ubiquitous lock-free stacks or queues. The pool's throughput performance under various producer-consumer scenarios is also competitive with respect to much of the prior state-of-the-art.

We showed that the pseudo-prioritization capability can be applied to problems such as NUMA locality. In our evaluation we also show that the spread-spectrum technique could be applied to future designs with optimistic synchronization primitives because the technique is highly distributed and very effective at reducing contention without degrading performance.

6. ACKNOWLEDGMENTS

Thanks to Elad Gidron, Dmitri Perelman and Idit Keidar for cordially providing the SALSA code for evaluation. Thanks also to the Scal group and the TAU Multicore Computing group for making their respective implementations easily accessible and Joseph Wingerter for comments on an earlier draft.

This material is based upon work supported by the National Science Foundation under Grant CCF-1010568. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

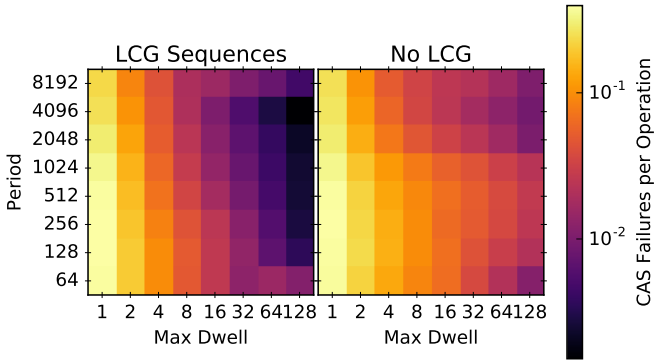
7. REFERENCES

[1] Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/, Accessed Jan. 2016.

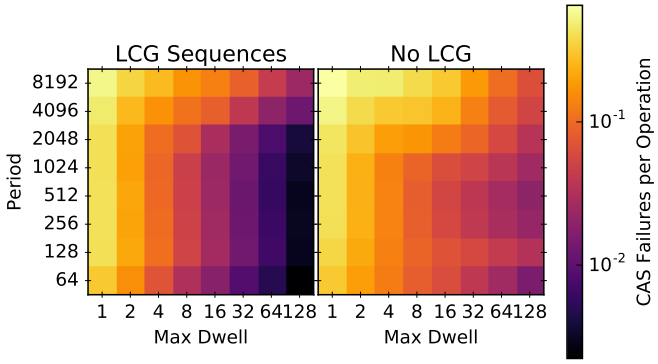
[2] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In P. D'Ambra, M. Guarracino, and D. Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 151–162. Springer Berlin Heidelberg, 2010.

[3] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, Euro-Par'10, pages 151–162, Berlin, Heidelberg, 2010. Springer-Verlag.

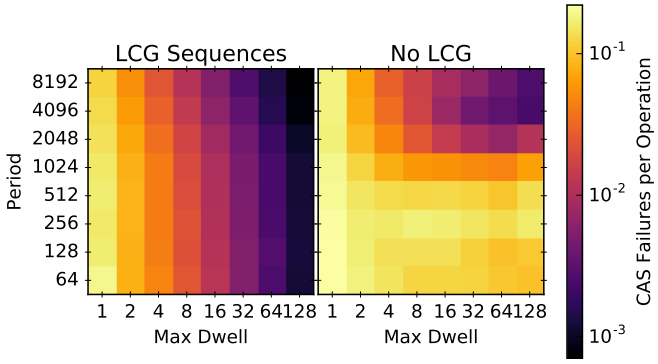
[4] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for



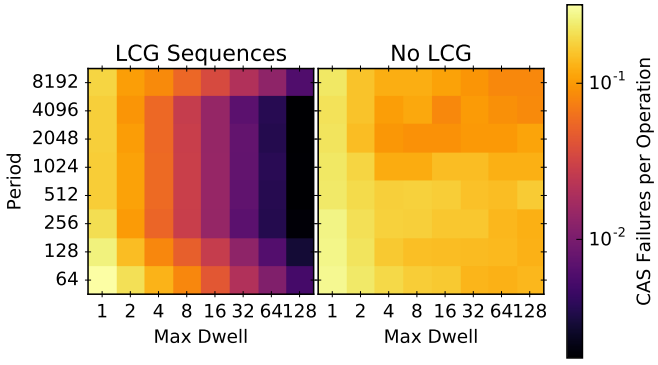
(a) 24:24 workload, LIFO queues



(b) 16:32 workload, LIFO queues

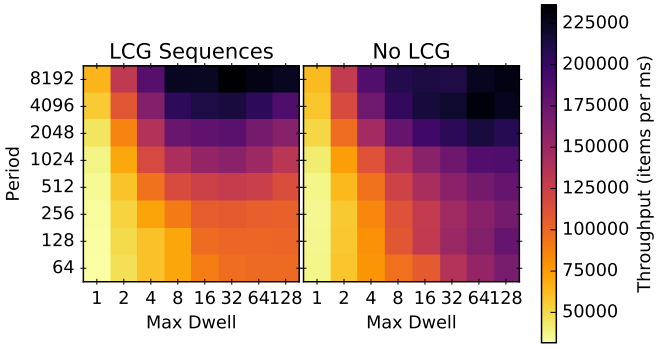


(c) 24:24 workload, FIFO queues

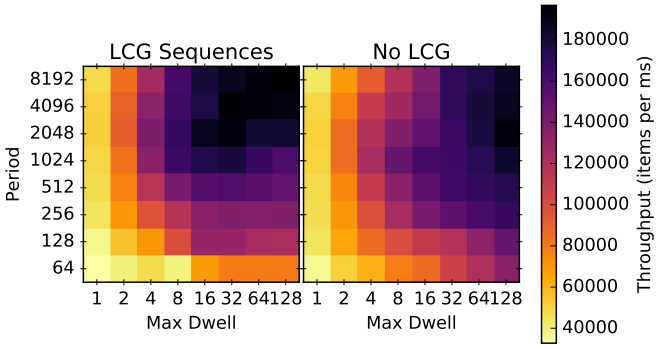


(d) 16:32 workload, FIFO queues

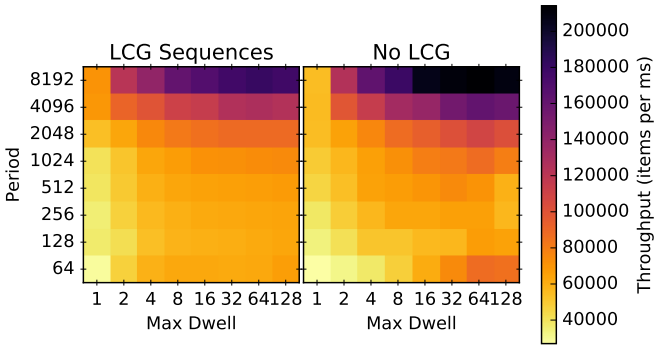
Figure 7: Contention in a CDMA pool. Darker is better.



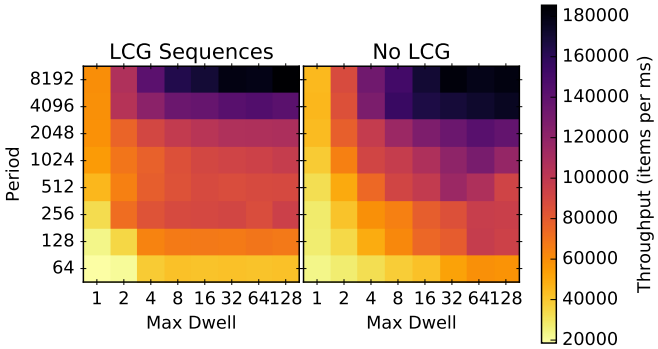
(a) 24:24 workload, LIFO queues



(b) 16:32 workload, LIFO queues



(c) 24:24 workload, FIFO queues



(d) 16:32 workload, FIFO queues

Figure 8: Throughput in a CDMA pool. Darker is better.

improved concurrency. In *Principles of Distributed Systems*, pages 395–410. Springer, 2010.

[5] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 11–20, New York, NY, USA, 2015. ACM.

[6] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[7] D. Basin, R. Fan, I. Keidar, O. Kiselov, and D. Perelman. Café: Scalable task pools with adjustable fairness and contention. In D. Peleg, editor, *Distributed Computing*, volume 6950 of *Lecture Notes in Computer Science*, pages 475–488. Springer Berlin Heidelberg, 2011.

[8] F. B. Brown. Random number generation with arbitrary strides. *Transactions of the American Nuclear Society*, 71:202–203, November 1994.

[9] D. Dice, H. Huang, and M. Yang. Asymmetric dekker synchronization. https://blogs.oracle.com/dave/resource/Asymmetric-Dekker-Synchronization-140215.txt, July 2001.

[10] D. Dice and O. Otenko. Brief announcement: Multilane - a concurrent blocking multiset. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 313–314, New York, NY, USA, 2011. ACM.

[11] J. Evans. Scalable memory allocation using jemalloc. www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919, 2011.

[12] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 37–44, New York, NY, USA, 2007. ACM.

[13] E. Gidron, I. Keidar, D. Perelman, and Y. Perez. Salsa: Scalable and low synchronization numa-aware algorithm for producer-consumer pools. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 151–160, New York, NY, USA, 2012. ACM.

[14] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and A. Sezgin. Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 17:1–17:9, New York, NY, USA, 2013. ACM.

[15] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 317–328, New York, NY, USA, 2013. ACM.

[16] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[17] Q. Huang and W. E. Weihl. An evaluation of concurrent priority queue algorithms. In *Parallel and Distributed Processing, 1991. Proceedings of the Third IEEE Symposium on*, pages 518–525. IEEE, 1991.

[18] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[19] G. Marsaglia. The structure of linear congruential sequences. *In: S.K. Zaremba (Ed.): Applications of Number Theory to Numerical Analysis*, pages 249–285, 1972.

[20] G. Marsaglia et al. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[21] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[22] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.

[23] A. Morrison and Y. Afek. Fence-free work stealing on bounded tso processors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 413–426, New York, NY, USA, 2014. ACM.

[24] W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[25] H. Rihani, P. Sanders, and R. Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 80–82, New York, NY, USA, 2015. ACM.

[26] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, Mar. 2011.

[27] N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, Nov. 1996.

[28] M. K. Simon, J. K. Omura, R. A. Scholtz, and B. K. Levitt. *Spread Spectrum Communications Handbook (Revised Ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1994.

[29] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 335–344, New York, NY, USA, 2011. ACM.

[30] D. Torrieri. *Principles of Spread-Spectrum Communication Systems, Second Edition*. Springer-Verlag, New York, NY, USA, 2011.

[31] R. K. Treiber. *Systems programming: Coping with parallelism*. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[32] G. I. Webb. Opus: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research*, pages 431–465, 1995.