Scalable Statistics Counters

Dave Dice
Oracle Labs
dave.dice@oracle.com

Yossi Lev
Oracle Labs
yossi.lev@oracle.com

Mark Moir
Oracle Labs
mark.moir@oracle.com

ABSTRACT

Statistics counters are important for purposes such as detecting excessively high rates of various system events, or for mechanisms that adapt based on event frequency. As systems grow and become increasingly NUMA, commonly used naive counters impose scalability bottlenecks and/or such inaccuracy that they are not useful. We present both precise and statistical (probabilistic) counters that are nonblocking and provide dramatically better scalability *and* accuracy properties. Crucially, these counters are competitive with the naive ones even when contention is low.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; E.1 [**Data Structures**]; E.2 [**Data Storage Representations**]; G.3 [**Probability and Statistics**]

Keywords

Statistical counters, performance, scalability, accuracy

1. INTRODUCTION

Most large software systems use statistics counters for performance monitoring and diagnostics. While single-threaded statistics counters are trivial, commonly-used naive concurrent implementations quickly become problematic, especially as thread counts grow, and as they are used in increasingly Non-Uniform Memory Access (NUMA) systems. Designers face difficult tradeoffs involving latency of lightly contended counters, scalability—and in some cases accuracy—of heavily contended ones, and probe effects.

Although counters are almost as simple a data structure as one can imagine, there is a wide variety of use cases, assumptions and constraints that can significantly impact the range of implementations that are acceptable and/or effective. In this paper, we mostly concentrate on statistics counters that are used to count events that may occur with high frequency, while the value of the counter is read infrequently, as is common for performance monitoring and diagnostics. Another interesting use case is data structures that employ counters to monitor access patterns, allowing the data structure to be reorganized to improve performance (see [1], for example).

In discussing shared counters with both theoretical and practical colleagues, we find good understanding of some issues, but confusion and misinformation about others. It is widely understood that simply incrementing a shared counter without synchronization does not work for multiple threads, because one thread's update can overwrite another's, thereby losing the effects of one or more increments. It is also well known that such counters can be made thread-safe by protecting them with a lock, but that in most modern shared memory multiprocessors, it is better to increment the counter using an atomic instruction such as compare-and-swap (CAS). CAS increments the counter—and indicates success—only if the counter has the value the incrementing thread "expects". Otherwise, it is retried, perhaps after some backoff.

This solution is simple, correct, and nonblocking but does not scale to large, NUMA systems. A common "solution" is to use the single-threaded version, eliminating the overhead of using CAS, as the precise value of the counter is not important, so losing "occasional" updates is acceptable.

This approach is flawed in two crucial ways. First, using separate load and store instructions to update the counter may *slightly* reduce the latency, as compared to using CAS, but it does not avoid the dominant cost of resolving the remote cache miss that is likely to occur when a variable is modified by many threads in a NUMA system. But most importantly, when contention on the counter increases, it is not just "occasional" updates that are lost: *many* increments can be lost due to a single delayed update. In our experiments, such counters consistently lost over 90% of increments when shared by 32 or more threads. Ironically, this problem becomes worse as contention increases: exactly the scenario many counters are intended to detect.

Some minor variations can mitigate the poor scalability and/or inaccuracy of naive counters to some degree. Examples include prefetching to avoid unnecessary cache state upgrades, ignoring CAS failures or using atomic add instructions to avoid retry loops and branch mispredicts, etc. Most of these are architecture-specific and—more importantly—none of them changes the fact that updating a single variable on every increment of a statistics counter will not scale to large, NUMA systems. Experiments confirm our intuition that they do not adequately addresses the shortcomings of the commonly used naive counters.

SPAA'13, June 23–25, 2013, Montréal, Québec, Canada.

This paper explores scalable implementations of two kinds of counters: precise and statistical. Although statistical counters may not provide an exact count, to be useful, they should have reasonable statistical properties such that, with high probability, they do not deviate from the precise count by "too much", as explained in more detail later. In both categories, we present implementations that are substantially better than those in common use.

We focus on counters that are incremented only by one and are never decremented. We exploit these assumptions to improve our algorithms, though many of the techniques we describe can be generalized to weaken or avoid them.

We cannot explore all possible implementations. In this paper, we investigate a variety of existing and new counter implementations, concentrating on those we believe are most likely to be used in practice in large software systems. We aim to highlight shortfalls of some counters that are widely used, and to offer practical alternatives that overcome them.

We present precise counters in Section 2 and statistical counters in Sections 3 and 4. In Section 5, we present experimental results showing that our algorithms can provide dramatically better throughput than the naive counters discussed above. Furthermore, our statistical counter algorithms are very accurate, even under heavy contention. We conclude in Section 6.

2. PRECISE COUNTERS

A simple and well-known approach to making counters scalable is to split them into per-thread components, with each thread incrementing its own component without synchronization. This increases space requirements by a factor of the number of threads that use it. While this space overhead may be acceptable in many modern systems, and per-thread components may be acceptable in monolithic applications with fixed sets of threads, modern application structures are often more dynamic, executing tasks via a dynamic pool of worker threads, for example. In such caoses, use of per-thread components is inconvenient at best. In this section, we present several algorithms that mitigate these disadvantages to varying degrees.

2.1 Minimal space overhead

If additional space overhead is unacceptable, and counters must be precise, well-known randomized backoff techniques [2] can at least avoid a complete meltdown under heavy contention; we call this version `RBO`.

We have also explored some approaches that are inspired by previous NUMA lock algorithms, such as the HBO algorithm due to Radovic and Hagersten [10] and the cohort locks of Dice et al. [5]. These locks significantly improve performance and scalability under contention by handing off the lock multiple times within a NUMA node before it is acquired on another node. One simple approach based on this idea is to use a cohort lock to improve contention management for `RBO`: when a thread fails an attempt to increment the counter using CAS, it retries after acquiring a cohort lock, thereby encouraging multiple updates on one NUMA node before an update on another when contention arises. We have implemented this algorithm and found it to be effective in improving performance over `RBO`. However, because of the space overhead of the cohort lock, this algorithm has no particular advantage over others we introduce later (particularly `MultilineAdapt`).

Another approach that is similar in spirit to the above-mentioned NUMA locks, but does not add significant space overhead, is to use a few bits of the counter's value to identify which node currently has "priority": threads on other nodes delay retrying after a failed update, making it more likely that threads on the priority node can perform consecutive updates. Of course, the bits used to identify the priority node should not include the lowest-order bits that change frequently, but should be chosen so that the priority changes often enough to avoid unreasonable delays. This approach is simple, adds no space overhead, and performs well when increment operations are evenly spread across all nodes. But it does not adapt well to less uniform workloads.

This shortcoming is addressed by `RBONuma`, which augments the counter with a few bits (or alternatively steals bits from the counter, thereby restricting its range). It requires only enough additional bits to store the ID of a NUMA node plus one more bit. Thus, it can accommodate a counter that ranges from 0 to $2^{(N-1)-\lceil \log_2(\#NODES) \rceil}$-1 using N bits. Our implementation—for a 4-node system—uses 32 bits, thus supporting counting up to $2^{29} - 1$.

`RBONuma` stores the ID of the node on which the counter was last incremented with the counter, allowing threads on other nodes to hold off in order to encourage consecutive increments on that node. A thread that waits too long can become "impatient", at which point it stores its node ID into an *anti-starvation variable*.[1] This tells threads on other nodes to wait before attempting to update the counter, thus enabling threads on the node with the impatient thread to fetch the cache line and increment the counter. Unlike the HBO lock [10], we do not *prevent* other threads from incrementing the counter before the impatient thread because we did not want to sacrifice the non-blocking property of the counter, and the heuristic approach described above avoids starvation in practice, even under heavy contention.

In `RBONuma`, threads on the same node as an impatient thread abort their "slow" backoff, and attempt to increment the counter immediately. Regardless of which thread on a node increments the counter, this has the effect of bringing the relevant cache line to that node, which gives all threads on that node a better chance to increment the counter. Thus, we do not attempt to ensure that the thread that becomes impatient is the next to increment the counter, but rather allow nearby threads whose increments will help the impatient thread to increment the counter before it. We found that this approach gave the best performance.

Although `RBONuma` can yield an order of magnitude better throughput than `RBO` under heavy contention (see Section 5), it imposes significant overhead in low-contention scenarios. Part of the reason is the need to test the anti-starvation flag before even trying to increment the counter. We therefore implemented `RBONumaAdapt`, which begins as a regular counter that does not record the node of the most recent increment; while there is no node recorded, there is no need to check the anti-starvation variable. If a thread retries more than a certain number of times (our implementation tries three times quickly, followed by 16 times with randomized backoff) before successfully incrementing the counter, then it records its node ID in the counter. Thereafter, the slower but more scalable algorithm described above is used. Al-

[1]Each counter must be associated with an anti-starvation variable, but it is not necessary to have one per counter. Our implementation uses a single, global anti-starvation variable.

though we did not do so, it would be straightforward to reset the counter to an ordinary counter occasionally, so that the effects of occasional contention do not persist forever. The results in Section 5 show that `RBONumaAdapt` is competitive with the best of `RBO` and `RBONuma` at all contention levels.

The counters described so far achieve good single-thread performance and scalability under heavy contention. However, their advantage over simple `RBO` is reduced under moderate load, because there is less opportunity to perform consecutive increments on the same node. Furthermore, these algorithms tend to be sensitive to system-specific tuning, making them less stable. Next, we explore counters that overcome these issues by using a little more space.

2.2 Using a little more space

To avoid expensive cross-node communication without suffering the above-mentioned disadvantages of per-thread counter components, our `Multiline` algorithm uses a counter component *per NUMA node*; each component occupies a separate cache line to avoid false sharing. Synchronization on per-node components is again via CAS increments with randomized backoff. CAS synchronization is not as problematic as is often assumed, as there is no cross-node contention. (Somewhat analogously, Dice and Garthwaite [4] used per-processor allocation buffers in their memory allocator, achieving good scalability while avoiding the pitfalls of using per-thread allocation buffers.)

`GetVal` reads each component in turn, with no synchronization, and returns the sum of the values read. (The correctness of this approach depends on our assumption that increments only add one; other techniques are available for more general cases [7].)

Although the space blow-up is limited by the number of nodes, we prefer to avoid it for counters that are incremented rarely. Our `MultilineAdapt` implementation begins with a regular counter, and "inflates" it to use the above-described technique only if more than a certain number (four in our implementation) of attempts to increment it fail. Other policies are possible, such as inflating the counter if it frequently causes remote cache misses. To inflate a counter, we allocate a structure with one counter per node and replace the counter with a pointer to that structure (we reserve one bit to distinguish such pointers from counter values).

With this solution, low-contention counters use just the reserved bit (in practice cutting the range of the counter by half), and we only pay higher space overhead for contended counters. `MultilineAdapt` introduces an extra level of indirection for contended counters. The resulting slowdown to the increment operation is not a problem if the counter is contended (on the contrary, reduces the rate of CAS attempts on the counter, hence reducing contention).

The result is a counter that is competitive in both space overhead and throughput with the basic `RBO` counter at low levels of contention, scales well with increasing contention, and yields more than 700x higher throughput than `RBO` under high contention (see Section 5). Despite this encouraging data, our results show that `Multiline` and `MultilineAdapt` suffer under high contention levels because of contention between threads on the same node using a single component.

This contention can be alleviated by using more components per node. While per-node components must be in separate cache lines to avoid false sharing between nodes, if we use more than one component per node, it may not be unreasonable to locate multiple components for a single node in the same cache line. While false sharing may still arise in this case, it will be only within one NUMA node, and there is still benefit from using multiple components, as fewer CAS failures will occur. Thus, it may be possible to improve performance without increasing space usage.

The additional space overhead used by `Multiline` may be problematic in systems with large numbers of statistics counters, most of which are not heavily contended. While `MultilineAdapt` allows us to incur the space overhead only for contended counters, if different counters are contended at different times, this may result in excessive overhead over time. Furthermore, these algorithms increase `GetVal` latency (see Section 5.5), and in some contexts, they may be unacceptable due to their use of dynamically allocated memory. In Section 3, we show how we can avoid all of these issues if counters are not required to be precise.

3. STATISTICAL COUNTERS

Simple *unsynchronized* counters lose significant fractions of counter updates, even at moderate levels of contention (see Section 5). Because counters are often used to detect excessive rates of various system events, these naive implementations are ironically least effective exactly when the data they should provide is most important. Nonetheless, precise counts are often not required. In this section, we explore counters that exploit this flexibility while still aiming to maintain a prescribed level of accuracy that is not achieved by the naive implementations.

We begin with background on a counter algorithm published by Morris [9] in 1978, from which we borrow some ideas. Morris's primary motivation was to maintain a large number of counters, each of which could represent large numbers using only 8 bits. He did not address concurrent counters, much less scalability in the NUMA systems of the (then) distant future. Our key observation, however, is that Morris's technique has the effect of substantially reducing the frequency of updates to the counter, which lends itself perfectly to our purposes.

3.1 Morris's algorithm

Morris's counter [9] represents a larger range than 8 bits usually does by storing a probabilistic approximation of the following value, where n is the precise count (i.e., number of times increment has been invoked):

$$v(n) = \log(1 + n/a)/\log(1 + 1/a),$$

where a is a parameter that controls the accuracy of the counter, as explained below. Adding one to n/a ensures that the function is well defined and equals zero when $n = 0$, and dividing by $\log(1+1/a)$ ensures that the function is one when $n = 1$. As we will see, this ensures that the counter contains accurate values at least for values zero and one. It follows from this definition that, when the value stored in the counter is v, the counter value it represents is:

$$n(v) = a((1 + 1/a)^v - 1).$$

We call the value v that is physically stored in the counter the *stored value*, and the value $n(v)$ that it represents the *projected value*.

The algorithm stores a *probabilistic approximation* of $v(n)$, where n is the precise count, because the stored value must be an integer, as Morris assumed only 8 bits. As a result, the

precise count cannot be determined from the stored value. Therefore, there is no deterministic way to know when to increment the value stored in the counter to reflect that enough increments have occurred that the counter's value should now be represented by a higher stored value.

To address these issues, Morris's algorithm increments the stored value with probability

$$p(v) = 1/(n(v+1) - n(v))$$

when it contains the value v. Intuitively, this means that on average the value stored in the counter will be incremented once out of the $n(v+1) - n(v)$ increment operations after value v is stored. This ensures that the value projected by the stored value is a random variable whose expected value is equal to the precise count.

To avoid computing probabilities on each increment, Morris's algorithm precomputes all 256 probabilities for a given value of a and stores them in a lookup table; this table does not need to be replicated for each counter, only for each accuracy class (i.e., each choice of a).

The parameter a determines both the range that the counter can represent and the expected error between the projected and actual counts (measured as the ratio between the STDV of the projected value and the actual count, a.k.a. relative STDV, or RSTDV). The variance of the projected value when the precise count is n is given by $\sigma^2 = n(n-1)/2a$ [6, 9] from which it follows that the RSDTV is bounded by $1/\sqrt{2a}$. Thus, taking an example from [9], choosing $a = 30$ yields an RSTDV of about 1/8. Furthermore, this choice of a allows the counter to represent $n(255)$, which is about 130,000. While impressive for using only 8 bits, this is not satisfactory—either in terms of range or accuracy—for counters to be used in today's systems. We next discuss how we have adapted Morris's ideas to implement scalable counters with much larger ranges and higher accuracy.

3.2 Our statistical counter algorithms

Because $n(v)$ is exponential in v, counter updates become less frequent as the precise count grows. This is the key property we borrow from Morris in order to reduce contention on frequently updated shared counters, while bounding expected error. However, we wish to implement counters with a higher range and with higher accuracy than is possible using Morris's algorithm with 8 bits per counter. While the basic principles behind Morris's algorithm extend to counters that use more bits, it becomes significantly less desirable to precompute update probabilities for all possible stored values as more bits are used. In this section, we explain how we have extended Morris's ideas to avoid this requirement.

First, we observe that the probability to increment the stored count from v to $v+1$ is a geometric series in v with a factor of $a/(a+1)$. To see this, we observe that

$$\begin{aligned} n(v+1) - n(v) &= a((1+1/a)^{v+1} - (1+1/a)^v) \\ &= a((1+1/a)^v(1+1/a-1)) \\ &= (1+1/a)^v \\ \Rightarrow p(v) &= 1/(1+1/a)^v = (a/(a+1))^v \end{aligned}$$

Therefore, given $p(v)$ we can compute $p(v+1)$ simply by multiplying by $a/(a+1)$. We can precompute this constant to avoid performing the floating point division repeatedly.

We further observe from above that $n(v) = a(1/p(v) - 1)$. Therefore, we can compute the projected value $n(v)$ of the stored counter value v directly from $p(v)$, without knowing v (in fact, it turns out that doing so is about 5 times faster than computing $n(v)$ directly from v). Therefore, we decided not to store v in the counter, as Morris's algorithm does, but instead store the floating point value $p(v)$. (Our implementation uses 32-bit floats, but the range and/or accuracy could be extended further using 64-bit doubles.)

On each increment, we read the value p stored in the counter, and with probability p replace it with $p * a/(a+1)$. The advantage of this approach, besides faster evaluation of the projected counter value, is that we avoid precomputing and storing values for all 2^b bits when using a b bits to represent a counter. Instead, we need only precompute the value of a for the desired RSTDV and $a/(a+1)$.

Practical implementation details.

During each increment operation, we update the stored value with probability p, which is determined by the stored value. To do so, we use a thread-local Marsaglia xor shift pseudorandom number generator (PRNG) [8] using parameters $(6, 21, 7)$, which returns an integer i between 1 and `MaxInt` $(= 2^{32} - 1)$, and we update the stored value if i/`MaxInt` $\leq p$. As a practical matter, our implementation stores `MaxInt` $* p$ (as a float), so that we only need to compare i to the stored value to make this decision. We call this stored value a "threshold". The initial threshold $T_0 =$ `MaxInt`. When we update the stored value, we replace the current value T_i with T_{i+1} $(= T_i * a/(a+1))$ if and only if the number returned by the PRNG is at most T_i. Pseudocode for this algorithm is presented in Figure 1.

With this implementation, care is needed to avoid updating T_i when it becomes too small, as the properties of the counter may be lost. In particular, we note that, because we use an integer PRNG, if an update does not reduce the integer part of the stored threshold, this does not actually affect the probability of update. For our implementation, we have observed that $T_i - T_{i+1} \geq 1$ at least while $T_i \geq a + 1$. We therefore reset the counter when this is no longer true (it is straightforward to instead raise an error if this is preferable). For our choice of $a = 5000$ to achieve a 1% RSTDV, and using a 32 bit counter, we cross this threshold when the projected value is about 0.02% below `MaxInt`. Thus, we achieve low relative error and much better scalability (see Section 5) without significantly reducing the range of the implemented counter as compared to naive 32-bit counters.

Variations.

The algorithm as described thus far performs very well when the counter becomes contended and reaches higher values, but it is significantly slower than a regular CAS-based counter when contention is low and the projected counter value is low. We therefore developed a hybrid version—`MorrisAdapt`—that starts with the implementation of a regular concurrent counter, using CAS, but if the CAS fails multiple times it switches to the probabilistic counting scheme described above. Our implementation stores a regular counter in one half of a 64-bit word, and a probabilistic counter in the other half. When contention is encountered, we switch from updating the regular counter to updating the probabilistic one; to read the counter, we add the value projected by the probabilistic counter to the value stored by the regular one. This is especially important when the application has thousands of counters, only a few of which are contended.

```
1  template <int Accuracy> // RSTDV as percentage
2  class MorrisCounter {
3  private:
4    float threshold;

6    // Static (global per accuracy class) info
7    //
8    static float s_a;
9    static float s_probFactor;    // a/(a+1)

11 public:
12   static StaticInit () {
13     // a = 1/(2*err^2)
14     //
15     float tmp = ((float)Accuracy/100.0);
16     s_a = 1/(2*tmp*tmp);
17     s_probFactor = s_a/(s_a+1.0);
18   }

20   MorrisCounter() {
21     threshold = (double)MaxInt;
22   }

24   unsigned int GetVal() {
25     float pr = threshold/MaxInt;
26     float val = (1.0/pr - 1.0)*s_a;
27     return lroundf(val);
28   }

30   void Inc() {
31     unsigned int r = rand();
32     float seenT = threshold;

34     while(true) {
35       if (r > (unsigned int)seenT) return;

37       bool overflow = (seenT < s_a + 1.0);
38       float newT = seenT * s_probFactor;
39       if (overflow) newT = (float)MaxInt;

41       float expected = seenT;
42       seenT = CAS(&threshold, seenT, newT));
43       if (seenT == expected) return;
44     }
45   }
46 }
```

Figure 1: Pseudocode for the counter based on Morris's algorithm.

```
1  template <int Accuracy> // RSTDV as percentage
2  class BFPCounter {
3  private:
4    // BFP Counter type: 4 bits for the exponent,
5    //                   28 bits for the mantissa.
6    //
7    struct Counter {
8      int mantissa : 28;
9      int exp: 4;
10     enum {MaxExp = (1<<4) - 1 , MaxMant = (1<<28) - 1};
11   };

13   Counter bfpData;

15   enum {
16     MantThreshold = 2*((30000/(Accuracy*Accuracy) + 3)/8)
17   };

19 public:
20   BFPCounter() {
21     bfpData = {0,0};
22   }

24   // Note: represented value could be larger than MaxInt,
25   // so we use 64bit return value
26   //
27   unsigned long long GetVal() {
28     Counter data = bfpData;
29     return (unsigned long long)(data.mantissa << data.exp);
30   }

32   void Inc() {
33     int r = rand();
34     int numFailures = 0;
35     while (true) {
36       ExpBackoff(numFailures);
37       Counter oldData = bfpData;
38       int e = oldData.exp, m = oldData.mantissa;

40       // Choose to update the counter with probability 1/2^e
41       //
42       if ((r & ((1<<e)-1)) != 0) return;

44       // We assume that the mantissa field can hold
45       // MantThreshold-1, so we do not check for mantissa
46       // overflow unless the exponent is saturated.
47       //
48       bool overflow = (e == Counter::MaxExp &&
49                        m == Counter::MaxMant);
50       Counter newData = {0,0};
51       if (!overflow) {
52         if ((m == MantThreshold - 1) &&
53             (e < Counter::MaxExp)) {
54           newData = {e+1, (m+1)>>1};
55         } else {
56           newData = {e, m+1};
57         }
58       }
59       if (CAS(&bfpData, oldData, newData) == oldDdata)
60         return;
61       numFailures++;
62     }
63 }
```

Figure 2: Pseudocode for the Deterministic Update Policy BFP counter (`BFP-DUP`).

4. AVOIDING FLOATING POINT

The counters presented above are appealing in terms of accuracy, performance under low contention, scalability under higher contention, and space usage. However, we received feedback from a group that would like to use such counters, but floating point operations cannot be used in their context. Even when floating point operations can be used, it is useful to avoid them so that, for example, they can be powered down if not in use by the application itself. We were therefore motivated to explore counters that provide similar properties without using floating point operations.

The key idea behind the counters discussed in this section is to constrain update probabilities to non-positive powers of two. We have developed two such algorithms, which are instances of a more general approach. We describe the general approach, and then present two specific instances of it.

4.1 The general approach

Using non-positive powers of two for update probabilities means that we can decide whether to update the counter with probability $1/2^k$ simply by checking whether the low-order k bits of an integer random number are all zero, thus avoiding any floating point computation. This approach requires us to use coarser-grained update probabilities: we can only halve the update probability, in contrast to reducing it by a factor of $a/(a+1)$ in the previous section. Reducing the update probability is important for performance and scalability (up to a point, as discussed later). However, if we halve the update probability after every update, it becomes small too quickly, harming accuracy. Thus, we must use the same update probability repeatedly before eventually reducing it, so there is a tradeoff to manage; we discuss this later.

The algorithms in this section represent counter values

using *binary floating point* (BFP): we store a pair (m, e), which represents a projected value $m \cdot 2^e$ (m is the mantissa and e is the exponent). We use bitfields in the counter variable to store m and e. For example, if we use four bits for e and 28 bits for m, we can represent a counter value of up to $(2^{28} - 1) \cdot 2^{15}$, about 2K times `MaxInt`.

When the exponent is e, we update the counter with probability 2^{-e}. As in the previous section, in order to keep the expected projected value of the counter equal to the total number of increments, we add 2^e to the projected value when incrementing the counter with probability 2^{-e}.

Observe that we can add 2^e to the projected value of a counter represented by (m, e) in two ways. The first is to update it to $(m + 1, e)$. The second way, which applies only when m is odd and the exponent field is not saturated, is to update the counter to $((m + 1)/2, e + 1)$. The first way leaves the update probability unchanged, and the second way halves it. The two algorithms we have implemented based on this general approach differ in the policy that controls which method to use when updating the counter.

4.2 Deterministic update policy

For our first BFP counter, `BFP-DUP`, we aimed for similar properties as the counters presented in the previous section, namely that we could specify a desired bound on the RSTDV, and reduce update probabilities as quickly as possible in order to improve scalability, while ensuring the desired RSTDV bound. Ensuring this bound requires that we do not reduce the update probability too quickly.

The policy we chose increments the mantissa by default, but if it would become `MantissaThreshold`, which is required to be even, we instead halve the mantissa (after incrementing it) and increment the exponent. This way, the first `MantissaThreshold` increments update the counter with probability $2^0 = 1$, thus ensuring that the counter reaches `MantissaThreshold` without introducing any error. Thereafter, the exponent is incremented (and the mantissa halved) every `MantissaThreshold`/2 counter updates. The choice for `MantissaThreshold` determines how quickly the exponent grows (and thus the update probability reduces); we explain how its value is chosen later.

`BFP-DUP` is shown in Figure 2. The `BFPCounter` class accepts as a template argument the desired bound on RSTDV as a percentage (e.g., if `Accuracy` is 1, then a 1% bound is desired). The value of `MantissaThreshold` is determined based on the desired `Accuracy`, as explained below. The `Inc` operation decides with probability $1 - 1/2^e$ not to update the counter, where e is the exponent currently stored in the counter (lines 33–42). If the decision is to update the counter, we first check if the counter has reached its maximum value (line 48), in which case we attempt to update the counter to zero (it is easy to signal an error if that is preferable). Otherwise, a new pair is determined based on the current pair, as described above (lines 52–57). Finally, the algorithm attempts to store the new pair to the counter, using CAS to confirm that the counter has not changed (line 59). If the CAS fails, the operation is retried.

Details and optimizations.

Figure 2 elides several optimizations that are included in the implementation used for the experiments in the next section. For example, our implementation code inlines the common update case in which the CAS succeeds, and uses the return value of a failed CAS to avoid the need to reread `bfpData` before retrying. Furthermore, when a CAS fails due to a concurrent update, the test to determine whether an update should be applied based on the new value is performed before backing off, as this will almost never be the case. We also note that all calculations are performed using bit shifting and masking operations.

Accuracy analysis and choice of parameters.

While exploring methods for analyzing the accuracy properties of our `BFP-DUP` algorithm, we found closely related work by Csűrös [3], who presents a sequential approximate counting algorithm. This algorithm does not support concurrent updates, and is less flexible than ours, but the results Csűrös developed and used to analyze the accuracy properties of the counter enabled us to analyze our algorithm easily. The algorithm in [3] is similar in spirit to ours but, rather than explicitly updating the mantissa and exponent, whenever it updates the counter, it does so simply by incrementing the stored value. When the mantissa part of the counter is incremented past its maximum value, the overflow naturally increments the exponent field (which is placed appropriately to ensure this).

As a result of this choice, the update function used in Csűrös's algorithm is slightly simpler than ours, but this has little performance impact because the counter is updated less and less frequently over time. Another implication is that the frequency with which an update increments the exponent (and thus reduces the update probability for subsequent operations) is required to be a power of two. Finally, a slightly different way of computing the projected value from the counter's stored data is needed, because the mantissa part becomes zero when the exponent is incremented.

Nonetheless, the Markov chain used in Csűrös's analysis is easily modified to model our `BFP-DUP` algorithm. The only substantive difference is due to `BFP-DUP` performing twice as many increments to the mantissa before incrementing the exponent for the first time as it does between subsequent increments of the exponent The Markov chain used to model our algorithm thus has a deterministic chain of length `MantissaThreshold`/2 before a chain that is otherwise identical to the one used by Csűrös [3]. This does not change the result in the limit, because these deterministic increments of the mantissa occur with probability 1, and therefore do not increase the inaccuracy of the counter. Thus we can use similar techniques as Csűrös did to apply his theorems to in order to determine a bound on RSTDV for our algorithm.

In contrast to the algorithms presented in the previous section, this analysis does not yield a bound on RSTDV that is independent of the number of increment operations performed. Rather, using the techniques presented in [3], we establish that the algorithm provides a bound on expected RSTDV in the limit as the number of increments n approaches infinity. More precisely, we have

$$\limsup_{n \to \infty} A_n \leq \sqrt{\frac{3}{8M - 3}}$$

where A_n is the expected RSTDV after n `Inc` operations, and M is the number of increments of the mantissa between increments of the exponent (i.e., `MantissaThreshold`/ 2).

This formula yields a bound on RSTDV only in the limit, unlike the one in the previous section, which provides a bound that holds for all counter values. Nonetheless, we

can use it heuristically to guide our choice of M for a given desired bound. Because the `BFPCounter` class accepts its accuracy argument as a percentage (e.g., if we desire a 1% bound, the `Accuracy` argument is 1), this equation implies

$$M \leq ((30,000/\texttt{Accuracy}^2) + 3)/8$$

hence the formula for `MantissaThreshold` at line 16 (recall that `MantissaThreshold`$=2M$).

Because `BFP-DUP` does not constrain the number of increments to the mantissa between increments of the exponent to be a power of two, as Csűrös's algorithm does, we have the flexibility to choose `MantissaThreshold` based on this calculation, giving us finer-grained control over the accuracy-performance tradeoff. For the experiments presented in the next section, we used a 1% bound, resulting in `MantissaThreshold` being set to 7500.

4.3 Contention-sensitive update policy

The deterministic update policy used by `BFP-DUP` is attractive because it lends itself to the analysis described above. However, while it is important for scalability and performance to reduce the update probability as the counter grows, at some point for a given system and workload, contention on the counter variable will be insignificant, and the overhead of updating the counter occasionally will be unnoticeable. Thereafter, reducing the update probability further only increases the inaccuracy of the counter. Therefore, we explored a different policy, whereby we choose to update the exponent (thereby reducing the update probability) only in response to contention. In particular, we always try to increment the mantissa once using a CAS (unless it would overflow), and only if that CAS fails, we decide whether to update the exponent and halve the mantissa using a similar policy to that used in `BFP-DUP`. We call this algorithm Contention-Sensitive Update Policy (`BFP-CSUP`). As the experiments presented in the next section show, `BFP-CSUP` yields similar performance to the `BFP-DUP` algorithm, while achieving higher accuracy in practice.

5. EVALUATION

We have evaluated both throughput and relative error of our counters under a variety of workloads via experiments conducted on an Oracle T5440 series machine. The T5440 consists of 4 Niagara T2+ SPARC chips, each containing 8 cores, with each core containing 2 pipelines with 4 hardware thread contexts per pipeline, for a total of 256 hardware thread contexts, running at a 1.4 GHz. Each chip has a 4MB L2 cache, and each core has a shared 8KB L1 data cache. Each Niagara T2+ chip is a NUMA node, and the nodes are connected via a central coherence hub.

Thread distribution is controlled solely by the operating system (Solaris 10) scheduler, which typically spreads threads out across nodes, then across cores, and finally across hardware threads within each core. Each run consists of a two-second warmup period to allow thread placement to settle, after which we reset the counter(s), and then measure throughput during a further eight seconds of execution. Each data point is the median throughput from five runs.

In all of our experiments, each thread alternates between performing an increment operation and doing some "external work" (modeling other work of the application). External work is implemented by executing a short `Pause` routine some number of times that is chosen uniformly at random from a range controlled by a parameter. To give some intuitive meaning to this parameter, we reflect it as the fraction of time spent executing external work in a single-thread run using the `CAS` counter. For example, 99% external work means that the single-thread throughput is 1/100 of the throughput achieved when run with no external work.

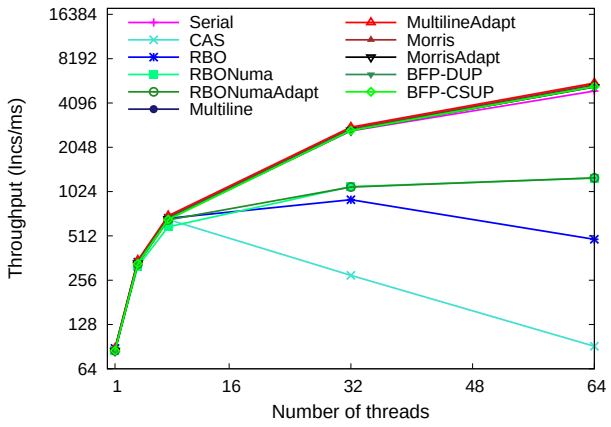

Figure 3: Single counter, low contention (99.02% external work) up to 64 threads.

5.1 Single counter, low contention

Figure 3 shows throughput (increment operations per millisecond) on the log-scale y axis, varying the number of threads on the x axis. This experiment has 99.02% external work, resulting in low contention, at least for low thread counts. The counters perform similarly up to 8 threads, but with higher numbers of threads, contention starts to arise, and the behavior of the algorithms becomes more important.

`RBO`'s backoff mitigates the poor scalability of `CAS` considerably, but it still shows negative scalability. Both `RBONuma` and `RBONumaAdapt` provide positive scalability, but are not competitive with the remaining algorithms, which all perform similarly and show significantly better scalability. These algorithms are: `Serial`, `Multiline`, `MultilineAdapt`, `Morris`, `MorrisAdapt`, `BFP-DUP` and `BFP-CSUP`.

`Serial`, which uses simple load-increment-store sequences to avoid the overhead of `CAS`, provides *slightly* lower throughput than the other algorithms that show similar performance in Figure 3. However, the others all entail significantly more algorithmic complexity, and in some cases significant space overhead too. One might therefore conclude that `Serial` is the best choice on balance.

However, we believe this conclusion is seriously flawed. First, Figure 3 shows data only up to 64 threads. As we will see, the scalability of `Serial` falls off at higher thread counts because every operation updates the single counter, so it becomes a bottleneck.

More important, however, is that `Serial` does *not* lose only "occasional" updates. In fact, the relative error exhibited by `Serial` can be dramatic. Under heavy contention (e.g., 0% external work, 256 threads), we consistently observe relative error of more than 99% meaning that fewer than 1% of increments on the counter are actually reflected in the count. The effect is not limited to such extreme cases: even with low contention (99.01% external work), we observe over 75% relative error for 64 threads and above, and over

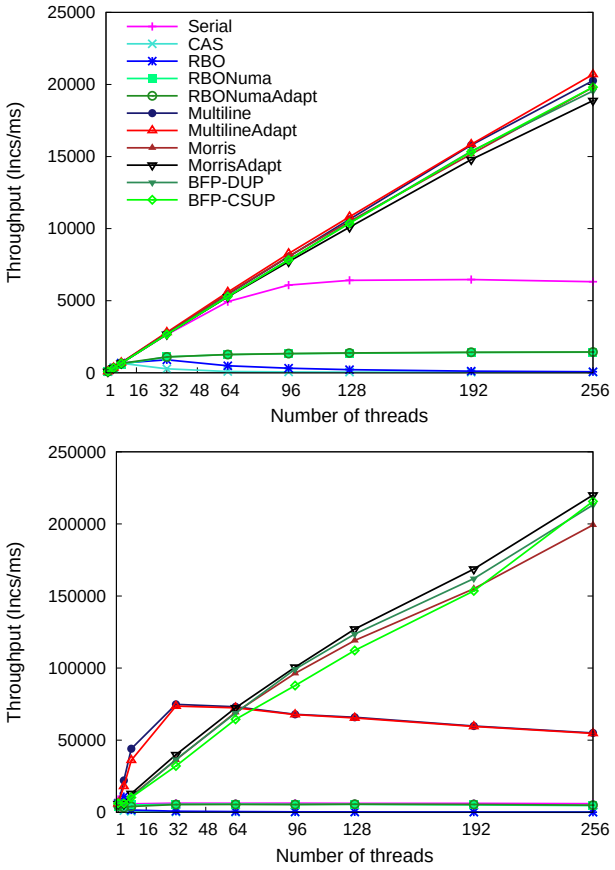

Figure 4: Single counter throughput, full thread range. Top: low contention (99% external work), bottom: high contention (0% external work).

10% error for 8 threads and above. The reason is that, from the time a thread reads the counter until the time that it stores its new value, it is guaranteed that the counter will increase by only one, regardless of how many other increment operations are performed by other threads in the interim.

Such counters are worse than useless: they impede scalability *and* fail to fulfil their purpose exactly when they are needed most, i.e., when the counter is being incremented more frequently. We also note that, with `Serial`, the counter regularly "goes backwards", which may be very undesirable in some cases, even when an accurate count is not required. Thus, we believe `Serial` is simply unacceptable as a statistics counter. Nonetheless, it is widely used in practice. We hope our paper brings this pitfall to the attention of practitioners, and helps them to choose another counter implementation that provides much better scalability *and* accuracy.

Figure 4 (top) shows the same data as Figure 3 up to 256 threads, with a linear y axis. None of `Serial`, `CAS`, `RBO`, `RBONuma`, and `RBONumaAdapt` is scalable. Nonetheless, we observe that, at 256 threads, `RBONuma` and `RBONumaAdapt` both provide about 20x the throughput of `RBO`, the most commonly used implementation, while requiring minimal space overhead. Although `Serial` provides higher throughput, it is not useful due to its egregious error properties, as discussed above. On the other hand, `Multiline`, `MultilineAdapt`, `Morris`, `MorrisAdapt`, `BFP-DUP` and `BFP-CSUP` all exhibit good scalability, achieving at least 3x the throughput of `Serial`, more than 13x the throughput of any of the other precise counters at 256 threads, and more than 260x the throughput of `RBO`. Amongst these good performers, the precise counters (`Multiline` and `MultilineAdapt`) have a slight edge in throughput over the probabilistic ones (`Morris` and `MorrisAdapt`), although they use 256 bytes of additional space (one cache line for each of four nodes) to do so.

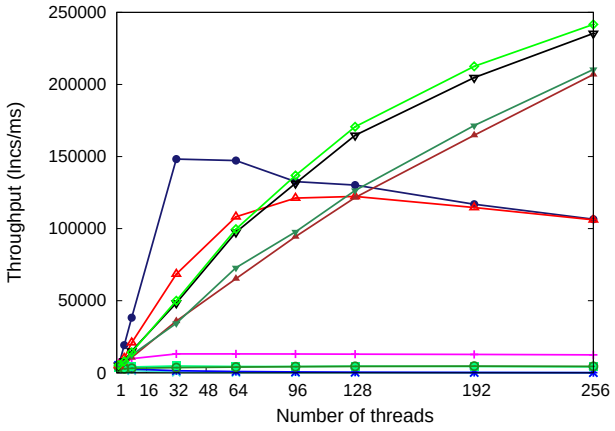

Figure 5: 7500 counters, one highly contended, 0% external work.

5.2 Single counter, high contention

We also performed a similar experiment with no external work. Although this may not be a realistic workload, it is important to understand the behavior of these algorithms under extreme conditions. Furthermore, as the number of cores per chip and the number of chips per system grows, realistic workloads may move closer to the levels of contention exhibited by this workload on this machine. Results are shown in Figure 4 (bottom).

We first observe that, although `Serial` yields two orders of magnitude higher throughput with one thread than before (because it is no longer spending 99% of its time doing "external work"), it again fails to scale, this time flattening out at about 32 threads. The other nonscalable counters all perform similarly poorly. `Morris`, `MorrisAdapt`, `BFP-DUP` and `BFP-CSUP` again scale well, and yield over 33x throughput compared to `Serial` at 256 threads. However, in this scenario, `Multiline` and `MultilineAdapt` peak around 32 threads, and scale negatively thereafter, due to contention on the per-node counter components. As discussed in Section 2, we could further partition these components to avoid this problem at the cost of additional space overhead; this cost would apply only to counters that experience heavy contention in the case of `MultilineAdapt`.

All algorithms except `Morris`, `MorrisAdapt`, `BFP-DUP` and `BFP-CSUP` either use significantly more space, or exhibit poor scalability in the above results, or both. Therefore, these algorithms appear to be the best choice, provided their error properties are acceptable for the application.

5.3 Many counters, mixed contention

It is common for an application to have many counters, of

which only a small number are contended. We performed an experiment in which threads alternate between incrementing a counter chosen uniformly at random from an array of 7500 counters, and incrementing one specific counter.

In the low contention case (e.g., 98.89% external work), the results (not shown) were qualitatively similar to the single-counter experiment except that `Serial` performed significantly better in this scenario because there was little contention on counters other than the central one. Results with no external work were more interesting (see Figure 5). As before, the probabilistic counters (`Morris`, `MorrisAdapt`, `BFP-DUP` and `BFP-CSUP`) all scaled well and produced low relative error (see Section 5.4). However, whereas `Multiline` and `MultilineAdapt` performed about the same in the single-counter experiment, in the array experiment, `Multiline` provided significantly higher throughput than `MultilineAdapt` for low thread counts (more than 2x at 32 threads), though this gap mostly disappeared with higher thread counts. The reason is that, with `Multiline`, counter components can be accessed directly, without a level of indirection or associated checks to determine whether a counter is inflated. On the other hand, `Multiline` pays a significant space overhead for all counters, while `MultilineAdapt` inflates only the contended counters, so the space overhead associated with the additional level of indirection is paid only by the counters that benefit from it.

The adaptive probabilistic counters (`MorrisAdapt` and `BFP-CSUP`) both outperformed their non-adaptive counterparts significantly because most of the counters are not heavily contended. For `MorrisAdapt`, the half of the operations that access non-contended counters avoid the expense of generating a random number and deciding whether to update the counter. Similarly, `BFP-CSUP` need not generate a random number when the update probability is 1, which is usually the case for the uncontended counters. `BFP-DUP` and `BFP-CSUP` outperform `Morris` and `MorrisAdapt`, respectively, probably due to their avoidance of floating point operations. (Despite our efforts to eliminate floating point operations from the common case, `Morris` performs a floating point conversion in every `Inc` operation in order to compare the integer random number to the floating point update probability; see line 35 in Figure 1.)

For a conservative comparison, we padded `RBO` counters to eliminate false sharing and packed the `Morris` counters as tightly as possible (16 32-bit counters per cache line). Despite these conservative choices, `Morris` dramatically outperformed `RBO`, as discussed above. `Morris` avoids the pitfalls of false sharing by reducing the frequency of updates to counters as the number of increments to them increases.

The components of `Multiline`, and of `MultilineAdapt` when inflated, are padded to avoid false sharing. However, in our experiments, the initial counters used by `MultilineAdapt` were packed with multiple counters per cache line; false sharing is avoided by inflating (only) contended counters. Because counters become read-only pointers when inflated, `MultilineAdapt` will perform well even if *multiple* highly contended counters are in the same cache line; this scenario is a certain performance disaster for the more naive, nonscalable counters.

5.4 Accuracy in practice

Next we examine the accuracy of the probabilistic counters in practice. Figure 6 shows the absolute value of the relative error for low-contention (99% external work) and high-contention (0% external work) workloads. These graphs show the *worst* relative error from five runs for each data point. No relative error higher than 3% is observed, and in most cases, the worst relative error is within 1.5%.

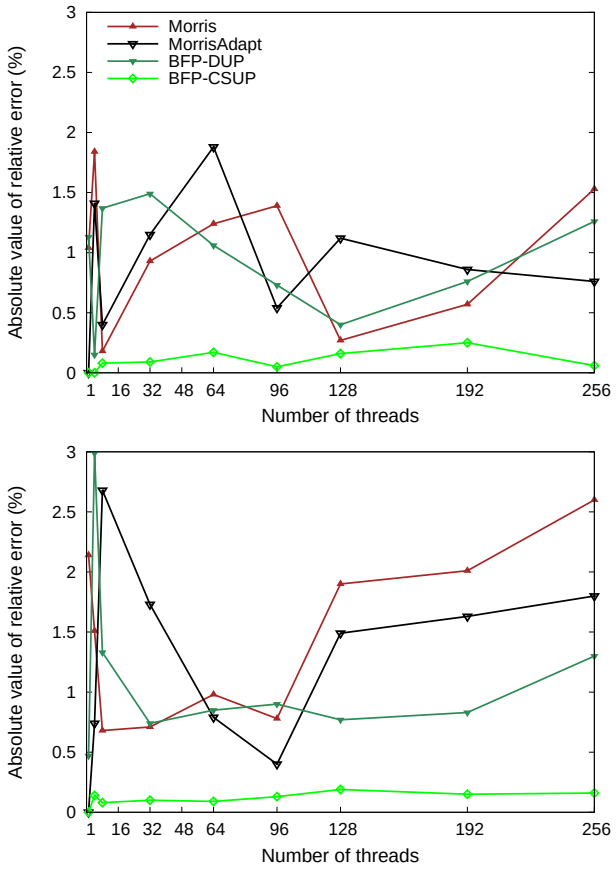

Figure 6: Relative error (absolute value). Top: low contention (99% external work), bottom: high contention (0% external work).

The maximum relative error exhibited by `BFP-CSUP` was often close to zero, and was never higher than 0.25% in these tests. In contrast, all of the other probabilistic counters typically exhibit relative error in the 0.75-2% range, and sometimes up to 3%. Together with performance data already presented, this shows that `BFP-CSUP`'s approach of not increasing the exponent unnecessarily achieves similar performance while reducing maximum error in practice.

We note that, in the high-contention case, most of the probabilistic counters exhibit somewhat higher relative error. This is because they count to higher values in this case (they provide higher throughput during our 8-second measurement period; see Figure 4). Therefore, they reduce the update probability more in the high-contention experiment. In contrast, `BFP-CSUP` increments the exponent only when contention is observed (or the mantissa is saturated). Thus its relative error does not increase simply due to executing more `Inc` operations, as is the case with all of the others.

5.5 `GetVal` overhead

We have targeted scenarios in which there are many counters, some of which may be incremented frequently. Our main goals have thus been low space overhead, low overhead in the absence of contention, and good scalability under heavy contention. We have not optimized for `GetVal` performance, but we believe that the costs discussed below are reasonably low for most of the algorithms.

The cost of `GetVal` comprises reading the necessary data and computing a return value from it. In the scenarios that have motivated our work, the cost of reading the data is likely to dominate, because it is likely not to be in cache for the thread executing `GetVal`: it may need to be fetched from memory, or from another—likely remote—cache.

`GetVal` for `Serial`, `CAS`, and `RBO` simply reads the underlying data and returns the value read, costing a single cache miss. `RBONuma`, `RBONumaAdapt`, `BFP-DUP`, and `BFP-CSUP` similarly require a single cache miss, but also entail some simple masking and/or shifting to determine the counter's value. `Multiline` requires multiple cache lines to be read. We note, however, that these are independent reads, so the cache misses will be resolved largely in parallel on most modern architectures. Read operations with `MultilineAdapt` are like the simple counters unless the counter is inflated, in which case `GetVal` must read not only the multiple cache lines for the counter, but also the pointer that determines where they are. The reads of the allocated cache lines *do* depend on the value of the pointer, and therefore the latency of `GetVal` likely includes at least two cache misses in series, even if all of the allocated lines are read in parallel.

For `Morris` and `MorrisAdapt`, `GetVal` entails mutliple floating point operations that will add noticeable overhead if executed often, in which case BFP-based counters may be preferable. Alternatively, projected values calculated from recent stored value(s) could be cached, as (eventually) the stored value of a counter changes infrequently.

6. CONCLUDING REMARKS

We have shown that nonatomic, centralized counters scale poorly *and* yield highly inaccurate counts. We have explored a variety of counters, some of which provide a precise count, while others aim for reasonable relative error, such that they are still useful for the purpose of detecting counters that are incremented many times. We present and evaluate several counters that dramatically outperform commonly used counters in terms of both throughput and accuracy, especially in NUMA systems, while keeping space overhead low.

The algorithms in this paper are easily seen to be lock-free. Furthermore, the probabilistic counters retry less often over time because the update probability becomes smaller over time (particularly in the case of contention in the case of `BFP-CSUP`). Modifying the counters in this paper to be wait-free while preserving their accuracy properties would likely add significant overhead and complexity. It would also introduce constraints such as needing to know the maximum number of threads in advance, or more overhead and complexity to avoid such constraints. In practice, lock-freedom is usually a strong enough progress property, provided backoff is used in case of contention.

Finally, we note the value of scalable statistics counters when used with transactional memory (hardware, software, or hybrid). Counters are used for many purposes, such as recording the number of entries in a hash table, or maintaining code execution frequencies, etc. The use of counters within transactions often causes all pairs of transactions to conflict because they all update the counter. The key to making the counters in this paper scalable has been to reduce contention on them, either by splitting them up so that multiple updates can occur in parallel (as in `Multiline`) or reducing the frequency of updates (as in the probabilistic counters). These techniques therefore reduce conflicts between transactions relative to using the naive nonscalable counters. We therefore expect them to be particularly valuable when used in transactions.

Acknowledgments.

Sumanta Chatterjee, Paul Loewenstein, and Steve Sistare gave useful input that motivated this work.

7. REFERENCES

[1] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. Cbtree: a practical concurrent self-adjusting search tree. In *Proceedings of the 26th international conference on Distributed Computing*, DISC'12, pages 1–15, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th annual international symposium on Computer architecture*, ISCA '89, pages 396–406, New York, NY, USA, 1989. ACM.

[3] Miklós Csűrös. Approximate counting with a floating-point counter. In *Proceedings of the 16th annual international conference on Computing and combinatorics*, COCOON'10, pages 358–367, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proceedings of the 3rd international symposium on Memory management*, ISMM '02, pages 163–174, New York, NY, USA, 2002. ACM.

[5] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: a general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM.

[6] Philippe Flajolet. Approximate counting: a detailed analysis. *BIT*, 25(1):113–134, June 1985.

[7] Yossi Lev and Mark Moir. Lightweight parallel accumulators using c++ templates. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 33–40, New York, NY, USA, 2011. ACM.

[8] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 7 2003.

[9] Robert Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, October 1978.

[10] Zoran Radovic and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 241–, Washington, DC, USA, 2003. IEEE Computer Society.