

# An Almost Non-Blocking Stack

Hans-J. Boehm  
HP Laboratories  
1501 Page Mill Rd.  
Palo Alto, CA 94304  
Hans.Boehm@hp.com

## ABSTRACT

Non-blocking data structure implementations can be useful for performance and fault-tolerance reasons. And they are far easier to use correctly in a signal- or interrupt-handler context.

We describe a weaker class of “almost non-blocking” data structures, which block only if more than some number  $N$  of threads attempt to simultaneously access the same data structure. We argue that this gives much of the benefit of fully non-blocking data structures, particularly for signal or interrupt handlers.

We present an almost non-blocking linked stack implementation which is efficiently implementable even on hardware providing a single word compare-and-swap operation, while potentially providing the same interface as a well-known fully non-blocking solution, which relies on a double-width compare-and-swap instruction. By making a platform-dependent choice between these, we can implement a signal-handler-safe stack or free-list abstraction that is both portable and exhibits uniformly high performance with any flavor of compare-and-swap instruction.

## Categories and Subject Descriptors

D.4.1 [Operating Systems]: Process Management — *Concurrency*; D.3.3 [Programming Languages]: Language Constructs and Features — *Concurrent programming structures*

## General Terms

Algorithms, Measurement, Performance.

## Keywords

Stack, non-blocking, lock-free, linked list, signal handler, interrupt handler, compare-and-swap, memory allocation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*PODC'04*, July 25–28, 2004, St. Johns, Newfoundland, Canada.  
Copyright 2004 ACM 1-58113-802-4/04/0007 ...\$5.00.

## 1. INTRODUCTION

Non-blocking data structures have received a large amount of attention (cf. [22, 7, 8, 20, 6]), primarily as a mechanism for avoiding unnecessary waits for stalled or faulty threads. Their use in the context of operating system kernels has been repeatedly explored (cf. [21, 5, 12]). Primitives to support them are being added to the Java Programming Language[17].

We encountered the need for non-blocking data structures in a slightly different context. We are interested in building sample-based profilers and similar tools[3] for Linux and re-discovered that it can be useful to update shared data structures from asynchronously invoked signal handlers.<sup>1</sup> This is usually impossible to do efficiently if lock-based approaches are used for concurrency control, since a signal handler may run as, and hence block, a thread owning a lock needed by the handler.<sup>2</sup> Non-blocking data structures can be a solution to this problem. We argue in the next section that a slightly weaker variant of “almost non-blocking” data structures are also sufficient, and for some architectures, more practical.

In this paper we focus on a particularly simple data structure, namely a linked LIFO stack. It is of interest both because it serves as a simple illustrative case of a technique that we expect to be generalizable, but also because it is almost immediately needed and, for our purposes, is often sufficient. In particular, it gives us a way to represent a set of memory objects. For example, it can be used to implement a free list of available memory, and thus allows memory allocation from a signal handler.

The problem is nontrivial on standard hardware since the underlying synchronization primitives are often too weak to support a direct non-blocking implementation. Most modern hardware provides the following, which we will assume for the rest of this paper:

- Byte addressability.
- Atomic read and write operations on address-sized, properly aligned data.
- An atomic, non-blocking compare-and-swap (*CAS*) operation on a single address-sized word.<sup>3</sup>

<sup>1</sup>This is closely related to the use of non-blocking primitives in kernel interrupt handlers, as in [21].

<sup>2</sup>The Posix standard[13] imposes severe restrictions on the actions that may be performed in a signal handler, largely motivated by this issue.

<sup>3</sup>Many architectures provide LL/SC operations which disal-

- Objects containing pointers may be aligned on word boundaries. (Usually this is in fact a requirement.) Thus pointers to list nodes will always have a few low order zero bits.

These facilities are provided, for example, by all recent X86, Itanium, Alpha, SPARC, MIPS, and PowerPC variants.

Some, but not all, of these processors also provide a double width compare-and-swap *CASW* operation that operates on two adjacent address-sized values simultaneously.<sup>4</sup>

In the presence of a *CASW* instruction, there are relatively old and well-known techniques for implementing efficient non-blocking linked stacks[22]. Our goal is to approximate this on an architecture with only single-width *CAS*, such that we can provide the same abstraction in either case, and encounter only moderate slowdown where *CASW* is not supported. This appears to be a minimal requirement for use of these techniques in portable user-level code.

With only a single address *CAS* operation, using C notation, we may write the *push* operation as follows, where *list* points to a list header pointing to the top of the stack, and *element* is the new node to be inserted:

```
void push(node **list, node *element)
{
    node *first;
    do {
        first = *list;
        element -> next = first;
    } while (!CAS(list, first, element));
}
```

Analogously, we might attempt to write a *pop* operation, which returns the former top of the stack, as follows:

```
/* WRONG !! */
node *pop(node **list)
{
    node *result, *second;
    do {
        result = *list;
        node *second = result -> next;
    } while (!CAS(list, result, second));
    return result;
}
```

Unfortunately, this may produce incorrect results if list nodes can be removed and then reinserted into a stack. Another process may concurrently *pop result* and another node, and then reinsert *result*. In this case the compare-and-swap operation succeeds even though *second* is no longer the second element in the stack.

low other intervening memory accesses[20]. These can efficiently emulate such a *CAS* operation, but do not support a more direct solution of our problem. Hence we view them as equivalent.

<sup>4</sup>*CASW* is available on Pentium or later X86 processors, and on most 64-bit processors when running 32-bit applications. But 128-bit *CAS* instructions are currently rare on 64-bit processors. It is part of the Intel 64-bit extension and Itanium architectures, but not implemented on current Itanium or 64-bit AMD processors. Some M68K processors have a *DCAS* instruction which can simultaneously operate on two discontinuous words in memory.

The underlying problem is that the initial value *A* of *\*list* changed to *B* and then back to *A* before the compare-and-swap was executed. This is commonly referred to as the “ABA” problem, which is unfortunately difficult to avoid without significant costs.

Here we present a practical solution to this problem, which relaxes the “non-blocking” requirement in a way that sacrifices some nice theoretical properties, but preserves most of the practical advantages of fully non-blocking algorithms.

## 2. RELATED WORK

The literature contains many examples of fully non-blocking, and even wait-free implementations of linked stacks. They fall into three categories:

- They rely on the presence of at least *CASW*. A non-blocking linked stack implementation using this approach was given in [22]. In the presence of *CASW*, our measurements confirm that this is at least competitive with any form of locking, and is clearly the preferred approach if locking is not viable.

- They rely on fairly complex algorithms with at least moderate time and/or space overhead to provide a fully non-blocking or wait-free implementation. Some recent work has done this directly usually for more general linked list structures (cf. [6, 23]).

A simple linked stack can also be built directly from an implementation of a fully general LL/SC primitive[20, 15], which is not subject to this kind of ABA problem.<sup>5</sup>

Probably the most practical such construction is the algorithm presented in section 2 of [15]. But it still requires order *NL* additional memory, where *N* is the number of processes and *L* is the number of lists in the system. And this must be in process-indexed arrays, which is difficult to arrange in most user-level environments, which support dynamic process creation. It also makes it nontrivial to make an approach based on [15] interface-compatible with a *CASW*-based solution, which is highly desirable for portable code.

- They disallow immediate reinsertion of previously removed list elements (cf. [19]). This is a very reasonable solution if every node pushed onto the stack is newly allocated. In a garbage-collected environment, it is even automatic. However, in many contexts, we would like to be able to move preallocated nodes between stacks, and have to tolerate immediate reinsertion of a given node. For example, this is true if we want to use the algorithm in a memory allocator, which was much of our original motivation.

The *hazard pointers* introduced by Michael in [19] are probably closest to the technique we present below. But the solution as a whole still does not allow arbitrary movement of list nodes between lists, and again requires a thread-indexed array, which is nontrivial to generalize to environments without a priori bounds on thread creation.

<sup>5</sup>The LL/SC primitives provided directly by many hardware architectures are more restricted, and do not avoid the problem, since it is not safe to load another memory location between the LL and SC[20].

A similar idea is explored in detail in [9], but they again resort to a double width (pointer plus version number) CAS instruction in their memory allocator.

### 3. ALMOST NON-BLOCKING DATA STRUCTURES

We propose a data structure that is *N-non-blocking* in the following sense:

- The data structure supports concurrent access by any number of processes.
- If at most  $N$  processes are both trying to update the data structure and are *inactive*<sup>6</sup>, and at least one active process is trying to access or update the data structure, then some process will succeed in accessing or updating the data structure in a bounded amount of time.

We say that a data structure is *almost non-blocking* or *almost lock-free* if it is *N-non-blocking* for some  $N > 0$ .<sup>7</sup>

This condition is strictly weaker than the standard definition of a non-blocking data structure, which imposes no limit on the number of inactive threads that are in the middle of a data structure update.

The weaker definition is usually sufficient for our motivating problem of avoiding deadlocks in signal- or interrupt-handlers.

In the single-threaded case, it is clearly sufficient if we prevent nested invocation of handlers; the only “inactive” thread accessing the data structure is the one thread interrupted by the handler. This is the normal behavior of Posix[13] signal handlers. The current signal is blocked in the handler by default, preventing reentry of the handler.

If  $N$  distinct signal handlers, as well as the main program, access the data structure, then an *N-non-blocking* data structure will normally avoid deadlock. Assuming each handler blocks its own type of signal (the default), then only  $N$  nested invocations of relevant handlers are possible. Only the main program and the “bottom”  $N - 1$  signal handlers can block as a result of handler invocation; thus we can have at most  $N$  simultaneous blocked accesses to the data structure.

For a multithreaded main program, we insist that signal handlers accessing our data structure not be recursively invoked. This can be ensured by accessing the data only from the handler for a single signal, or by blocking other signals that might cause data structure access in each such handler.

For a multithreaded program, we further require that all accesses to a data structure outside a handler be confined to a single thread. If this is not naturally the case, a conventional locking scheme can be safely used to ensure this, provided only accesses to the data structure from the main program, *but not from signal handlers*, acquire the lock.

<sup>6</sup>Such a process is inactive if it fails to execute instructions at a minimum rate while trying to update the data structure. This may happen, for example, if it blocks while waiting for the completion of a signal handler, or if it was running on a processor that has stopped as the result of a hardware failure.

<sup>7</sup>Note that this notion is unrelated to the use of the term *almost wait-free* in [4], since their data structure is in fact entirely lock-free. It is almost complementary to the notion of *obstruction-freedom* in [10].

In this setting, at most a single updater to the data structure, namely the single thread currently holding the data structure lock, can become inactive due to interruption by a handler. Hence a 1-non-blocking data structure is sufficient to prevent deadlock.

If our goal is not to accommodate signal handlers, but to tolerate thread pauses, such as during a page fault, an *N-non-blocking* solution guarantees behavior similar to a non-blocking solution in the presence of up to  $N$  simultaneously paused threads.

Our primary goal is deadlock-avoidance, not performance or scalability. Nonetheless we expect a mostly non-blocking data structure to perform almost as a non-blocking data structure at low or modest levels of contention. At very high levels of contention, a mostly non-blocking data structure performs more like a lock-based algorithm, and many of the same considerations apply. It is similarly easy to insert a back-off algorithm in the contention path without affecting the fast path.

### 4. AN ALMOST NON-BLOCKING STACK

Our *N-non-blocking* stack combines two old ideas in a novel way:

- We can solve the ABA problem by including a version number with each pointer. The problem is that since we need to atomically update pointers, and we assume only a single-word CAS instruction, we have very few bits available for a version number, and hence cannot avoid wrap-around issues. However, on all modern architectures, list nodes will be naturally word-aligned, but addresses are capable of referring to bytes. Hence it is possible to include at least a two bit “version number”.<sup>8</sup> We present a way to use these two bits to good advantage, such that version wrap-around does not introduce correctness issues. Unlike earlier work, we allow ourselves to block before we run out of distinct version numbers.
- We use a technique reminiscent of Michael’s “hazard pointers” to limit the need for version numbers to cases of potential conflict. Unlike a per-thread collection of “hazard pointers” we use a global “black-list” associated with the list header.

At any point, our global black-list contains pointers to nodes that are currently being popped from the stack by some thread; hence they could give rise to the ABA problem if they were reinserted with the same version number. If we encounter a black-listed `element` as an argument to a `push` operation, we increment the version number.

In presenting the algorithm we treat pointers with their short version numbers as a single entity. We assume that there is a function `perturb` such that `perturb(p)` yields a pointer identical to `p` but with a different version number, and such that repeated application of `perturb` cycles through all version numbers. In practice, `perturb` is usually implemented as an add instruction followed by a zero (overflow) test on the least significant bits.

<sup>8</sup>On a 32-bit machine, we assume that list nodes are at least 32-bit, or 4-byte, aligned. This ensures that the least significant two bits of every pointer are zero. On 64-bit machines, pointers will generally be at least 8-byte aligned, giving us at least three unused low order bits in each pointer.

```

void push(node *perturbed * list,
         node * element,
         node *perturbed bl[])
{
    node *perturbed my_element = element;

retry:
    for (int i = 0; i <= N; ++i) {
        if (bl[i] == my_element) {
            my_element = perturb(my_element);
            goto retry;
        }
    }
    do {
        node *perturbed first = *list;
        element -> next = first;
    } while (!CAS(list, first, my_element));
}

```

Figure 1: push implementation

```

node * pop(node *perturbed * list,
         node *perturbed bl[])
{
    unsigned bl_index;
retry:
    node *perturbed result = *list;
    for (bl_index = 0; ; ) {
        if (CAS(&(bl[bl_index]), 0, result))
            break;
        if (++bl_index > N) bl_index = 0;
    }
    if (result != *list) {
        bl[bl_index] = 0;
        goto retry;
    }
    node *perturbed second =
        strip(result) -> next;
    if (!CAS(list, result, second)) {
        bl[bl_index] = 0;
        goto retry;
    }
    bl[bl_index] = 0;
    return strip(result);
}

```

Figure 2: pop implementation

We also need a function *strip* which removes the version number from a pointer and yields a pointer that can be dereferenced directly.

We present the *push* and *pop* functions with an extra explicit black-list argument. The black-list is an  $N + 1$ -element array in order to tolerate  $N$  inactive threads. We require that  $N + 1$  be strictly smaller than the number of distinct version numbers we can represent.<sup>9</sup>

We use C notation throughout (with C++-style embedded declarations), except that we use a new type qualifier “perturbed” to indicate that a pointer may include a version number and may not be directly dereferenced. We assume that list nodes include a field “list \*perturbed next;” which we use as a link field.

A list itself is represented as a pointer to a header field, which is itself a perturbed pointer to a node.

Unlike our actual implementation, we assume a sequentially consistent[16] machine.

The *push* function implementation is given in figure 1.

As others have observed, the push routine is not subject to the ABA problem. It does not matter whether the head of the list changed between the initial read and the CAS instruction, so long as its final value matches the one we read. Thus this is essentially the same as the naive version above, except that

- If we find the node being inserted on the black-list, because another thread is still attempting to remove it, we *perturb* the pointer.
- We verify that each slot in the black-list contained a value other than the final perturbed node pointer we use for insertion sometime during the execution of the initial loop.

The *pop* routine (see figure 2) is also a variation on the naive one, but we need to be a bit cleverer to ensure that the element being removed correctly appears on the black-list at the right time:

Essentially we look for an unused black-list slot, and atomically replace it with the list entry we are planning to remove. After adding the black-list entry *but before looking up the second list entry*, we verify that the list head is unchanged. We then proceed as in the naive algorithm.

The algorithm may block, i.e. loop for a reason other than a successful list update by another thread, only if there is no available slot in the black-list.

In practice it is beneficial to insert back-off code at the point at which *bl\_index* is reset to zero to reduce contention on the black-list.<sup>10</sup>

Note that there is no problem if the list node being removed is moved to another list or deallocated in the meantime, so long as it remains addressable. We may access

<sup>9</sup>On a 32-bit machine, pointers and hence list nodes are usually 32-bit-aligned, and therefore we have two unused bits for the version number. We can thus represent four distinct version numbers, yielding  $N \leq 2$ . On a 64-bit machine we usually get eight distinct version numbers and  $N \leq 6$ . Some platforms allow stricter alignment restrictions and hence larger  $N$ . Often  $N = 1$  is actually the most interesting case, since it usually suffices for data structure accesses from signal handlers.

<sup>10</sup>Our back-off code first spins, and then sleeps for short periods of time. Thus we do not require any form of notification when a black-list entry is cleared, though that may be a better approach in high contention cases.

the `next` field of a node that was concurrently removed and deallocated by another thread. But the value read cannot matter in that case, since the following CAS operation will fail.

The list header and black-list on the other hand, must remain valid as long as any `push` or `pop` operation is in progress, since the CAS operations might otherwise modify memory that has since been reallocated for a different use.

## 4.1 Correctness

We assume that stacks are used correctly by client code. In particular:

- A `push` operation is only performed on a new `element` i.e. one that was never previously passed to `push`, or on one that was returned by `pop` after the last time it was pushed onto the stack.
- No client code modifies a list header, a black-list, or the `next` field in any stack node except by calling `push` and `pop`. (The `next` field in a list node may be arbitrarily modified before it is passed to `push` for the first time or after it was returned from `pop` and before it is again passed to `push`.)
- List headers and black-lists are permanent (or garbage collected, or managed in some other safe way). List nodes always remain accessible for reading, but they may otherwise be reused arbitrarily after they are returned from `pop` for the last time.

We focus first on safety by arguing that our implementation is linearizable in the sense of [11].

### 4.1.1 Atomicity of push

It is easy to show that the push operation behaves as though the update to the list header and next field occurred atomically at the successful CAS instruction. By our assumptions about client behavior, the `next` field cannot be concurrently modified or read by client code. Neither is a concurrent `push` operation on the same element allowed. Hence there is no way to observe that the `next` field was updated before the CAS instruction.

### 4.1.2 Atomicity of pop

Clearly the `pop` operation is far more interesting. We argue that it behaves as though it occurred at the point of the final CAS. This is true if we can guarantee that the CAS succeeds only if there were no intervening changes to `strip(result)->next` between the `result != *list` test and the success of the CAS operation.

Assume that the `next` field in fact did change in the interim during a `pop` operation, but the CAS succeeds. Let `pop_main` be the first such operation. The `next` field could only have legitimately changed after a successful `pop` operation `pop_intervening` which also returned the same node, i.e. `strip(result)`. For the CAS operation to succeed, this node must have later been reinserted by another `push` operation `push_intervening` which resulted in the same perturbed element pointer, and hence the `next` value we see must have been written by `push_intervening`. (If there were multiple intervening `pop` and `push` operations removing and reinserting the node, we let `pop_intervening` and `push_intervening` refer to the last ones.)

Consider two cases:

1. The `push_intervening` operation started after (the read operation in) the last `result != *list` test. This is impossible, since `pop_main`'s `result` pointer would have been in the black-list for the entire execution of `push_intervening`, thus it would have refused to reinsert it into the list.
2. The `push_intervening` operation started before the last `result != *list` test. In this case `pop_intervening` must have finished before the test. Since we assumed earlier that `push_intervening` interfered with `pop_main`, i.e. the CAS did not complete until after the last test, this is also impossible, since the node referenced by `result` could not possibly have been at the top of the stack. Hence the `result != *list` would have been true, we would have retried the `pop` operation, and this could not have been the last execution of the test.

This argument explains the need for the seemingly redundant test after the black-list insertion in `pop`.

### 4.1.3 Liveness

Both `push` and `pop` operations may loop if another thread succeeds in updating `*list` while they are executing. Here we address other possible reasons they may loop and thus fail to make progress.

As we mentioned earlier, the `pop` operation may loop looking for a black-list entry in which to insert the current top-of-stack pointer `result`. It may make more than one pass over the entire black-list only if either black-list entries were concurrently removed, or if the array remained full for the entire duration. The former case does not violate lock-freedom, since other threads clearly progressed.

The second case can result in blocking only if all  $N + 1$  entries in the black-list remain unchanged, which can occur only if  $N + 1$  threads are performing `pop` operations on the list and have associated black-list entries. Since the `pop` operation always clears its black-list entry after a fixed number of instructions, this can occur only with  $N + 1$  inactive threads. Hence the `pop` operation is  $N$ -non-blocking.

The body of the initial loop in the `push` operation may execute at most  $V(N + 1)(P + 1)$  times, where  $P$  is the number of processes executing `pop` when the `push` operation is entered, and  $V$  is the number of distinct version numbers we can represent. Hence the `push` operation is completely non-blocking.<sup>11</sup>

To see this, observe that the node currently being pushed cannot be in the stack until `push` completes. Hence its perturbations can appear in the black-list at any time during the current `push` only if they were added to the black-list by `pop` operations that had already been previously started and had already read the list header before `push` was called. There are at most  $P$  such in progress `pop` operations.

Since there are more possible perturbations than black-list entries, we can cycle through all possible perturbations without finding an unused one only if a new perturbation of the current node appeared in the black-list while the loop was executing. A complete cycle through all perturbations and black-list entries requires  $V(N + 1)$  loop iterations. A

<sup>11</sup>This bound could be easily tightened, especially for larger values of  $V$ . In practice, we expect that the loop almost always executes exactly  $N + 1$  times.

new perturbation of the current node can only be added once by each of the  $P$  in-progress `pop` operations. We can be forced to redo this process at most  $P$  times, for a total of no more than  $V(N + 1)(P + 1)$  total loop iterations.

## 4.2 Enhancements

In practice it is beneficial to insert back-off code at the point at which `bl_index` is reset to zero in the `pop` implementation to reduce contention on the black-list. We do so in our implementation.

As with Michael’s “hazard pointers”, it is possible to share a black-list between multiple stack data structures. This increases the probability of black-list overflow and hence blocking. But it affects the correctness argument only in that the liveness argument must now include threads performing the `pop` operation on any of the lists sharing the black-list. If any of these threads are not progressing, they must be included among the inactive threads. The result remains suitable for use in signal handlers, if we are careful to ensure that signal handlers cannot be reentered, and no more than  $N$  threads can be simultaneously interrupted by handlers accessing any of the  $N$ -non-blocking stacks with a shared black-list.<sup>12</sup>

Sharing black-lists can reduce the space overhead for list headers to near zero, but it makes it impossible to simply include the black-list as part of the list header. Hence it makes it harder to present a common interface for both our algorithm and the CASW algorithm.

The following “optimization” attempts were found to be less than completely successful:

- Modifying the `pop` procedure to first read the black-list and check that it is zero, before attempting the CAS. On an Itanium machine, this resulted in a slowdown of up to 50%. We conjecture that at high contention levels, a black-list entry is frequently updated between the initial read and the compare-and-swap, resulting in more cache coherence traffic and less back-off. Interestingly, on X86 machines, it apparently results in a less dramatic performance *win*, not loss. As a result, we currently perform this optimization only for X86.
- Increasing the number of black-list slots above the default of two. Any increase resulted in an appreciable slowdown, presumably due to the need of the `push` operation to scan more slots, and again the increased coherence traffic due to the increased probability of a processor losing a cache line with a black-list as it is being scanned. It may be of benefit to store only version numbers and selected pointer bits in the black-list, to compress it to a single word, or to use multi-word load instructions.
- “Randomizing” the starting position of the black-list search in the `pop` procedure. This has minimal effect, and slows down the important case of a single thread.
- Separating the black-list and list header into separate cache lines seemed to have minor and inconsistent effect. Since this eliminates the option of including the black-list as part of the list header, something which

<sup>12</sup>This is guaranteed if either the application is single-threaded, or of all lists are protected by a single lock when accessed outside a signal handler.

can significantly simplify the interface, this option was not pursued.

## 5. PERFORMANCE

We measured the performance of our mostly non-blocking stack on a few different small multiprocessors. In each case, we timed a total of approximately one million push plus one million pop operations executed by between one and 20 concurrently executing threads. We report the total execution time in milliseconds for these 2 million operations.

Our test consists of a simple microbenchmark designed such that

- Not all threads execute exactly the same pattern of stack operations. We felt this would have been unnecessarily prone to anomalies introduced only by the regular behavior, though we did not observe such behavior.
- It would be likely to fail in an easily detectable way in the event of an implementation error.<sup>13</sup>

More precisely, for  $n$  threads, we start with a stack containing  $\frac{n(n+1)}{2}$  elements. The  $i$ th thread then alternately pops  $i$  elements, and then pushes all  $i$  elements back onto the stack. It terminates at the end of the first such cycle which results in more than a million push and pop operations having been performed.<sup>14</sup> Once we finish timing these operations, we check that the final result is a permutation of the original stack.

For this experiment, all threads are performing `push` and `pop` operations as rapidly as possible. Any number of threads greater than one results in significant contention. The performance with exactly one thread is important, since it is likely to reflect the performance on a uniprocessor, and to be indicative of contention-free performance.

Early on we observed some systematic variation in execution times; runs could proceed in faster or slower modes for extended periods of time. The addition of exponential back-off appears to have remedied that. But we nonetheless average three time-separated runs for each data point.

We compare the following linked stack implementations:

**Mutex** Each `push` and `pop` operation acquires and releases a `Pthreads[13]` mutex.

**Spin-backoff** Each operation acquires and releases a “spin”-lock. After each attempt to acquire a test-and-set lock, we back off roughly exponentially, first by spinning in a tight loop, then by attempting to yield the processor, and finally by sleeping for short periods. The details are very similar to what has long been used in [2].

**MostlyNB** The algorithm we presented here. Every time the `pop` operation fails to find an empty black-list slot in the entire array, we back off using a slightly different, but signal-safe algorithm. We either spin or sleep,

<sup>13</sup>We did exercise this feature with a subtle error in an early version of our CASW-based implementation.

<sup>14</sup>A counter is atomically updated and tested once per cycle. All versions of the program perform identical counter updates.

we do not yield. (This is our best signal-safe approximation to the above.) The measured implementation has a black-list of size 2, and hence is 1-non-blocking. As we pointed out above, this is usually sufficient if signal-safety is the goal.

Our implementation uses a moderately portable atomic operations package [1] to implement compare-and-swap and the necessary memory ordering constraints.<sup>15</sup>

**CASW** This is an implementation of the fully non-blocking algorithm using a double-pointer-sized compare-and-swap operation, and a pointer-sized (in this case 32-bit) version number. For us, this was only possible on 32-bit X86 hardware. It may fail in the *extremely* unlikely case that a `pop` operation stalls sufficiently long for the version number to be incremented exactly a multiple of  $2^{32}$  times. We implemented the optimization suggested in [22]. In this variant a single-width CAS operation is used in the `push` routine, and the version number is not updated.

We present results for four different machines, two of which are obsolete. We present all results since the variation was interesting to us, and is probably representative of more widespread differences across architectures.

Each of these machines shares a single bus between all processors.

Note that both the RedHat 9 and Itanium machines used the NPTL threads package, while the other machines used `linuxthreads`. We believe the `linuxthreads` mutex implementation enforces a strict FIFO discipline, and immediately yields the processor when it has to wait, while NPTL spins to avoid some of the resulting context switches.

Larger machines appear to encounter larger overhead as a result of contention, presumably because more movement of cache lines is necessary.

## 6. CONCLUSIONS

Although the precise performance relationship among the four implementations is surprisingly variable, at least on NPTL, all seem comparable.

It is apparent from the preceding section that we can construct a very performance competitive implementation of a mostly non-blocking stack by selecting either the CASW implementation (when possible) or our mostly non-blocking implementation (when necessary). The result can be used portably across all architectures supporting a CAS-like primitive, and thus significantly extends the kinds of operation that can be safely performed in contexts such as user-level signal handlers, where a lock-based solution is not safe.

<sup>15</sup>In particular, the CAS operation in the `push` function must have *release* semantics, forcing all prior memory operations to become visible before it. The initial CAS operation in `pop` has *acquire* semantics, enforcing ordering with respect to the later reread of `*list`. The final CAS and the clearing of the black-list entry have *release* semantics. As appears to be standard for X86 processors, this package assumes that loads are ordered, in spite of the official documentation [14] for that processor. On X86 processors the ordering constrains only the compiler. See [18] for some details. On Itanium, we assume only the documented rules. The ordering is specified explicitly as part of the store and CAS instructions.

## 7. ACKNOWLEDGEMENT

We are grateful to Doug Lea and the anonymous reviewers for exceptionally detailed lists of suggestions. Unfortunately, we did not yet have the time to fully explore some of the deeper observations.

## 8. REFERENCES

- [1] H.-J. Boehm. The atomic\_ops atomic operations package. Included in [3].
- [2] H.-J. Boehm. A garbage collector for C and C++. <http://www.hpl.hp.com/personal/Hans.Boehm/gc/>.
- [3] H.-J. Boehm. The qprof project. <http://www.hpl.hp.com/research/linux/qprof>.
- [4] H. Gao, J. Groote, and W. Hesselink. Efficient almost wait-free parallel accessible dynamic hashtables. <http://www.archiv.org/abs/cs.DC/0303011>.
- [5] M. B. Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999.
- [6] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300–314, 2001.
- [7] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, 1991.
- [8] M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.
- [9] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lockfree data structures. Technical Report TR-2002-110, Sun Microsystems, 2002.
- [10] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003.
- [11] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [12] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001.
- [13] IEEE and The Open Group. *IEEE Standard 1003.1-2001*. IEEE, 2001.
- [14] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual - Volume 3: System Programming Guide*. Intel Corporation, 2004.
- [15] P. Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the 22nd annual ACM Symposium on Principles of Distributed Computing*, pages 285–294, August 2003.
- [16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- [17] D. Lea. Concurrency jsr-166 interest site. <http://gee.cs.oswego.edu/dl/concurrency-interest>.
- [18] D. Lea. The JSR-133 cookbook for compiler writers.

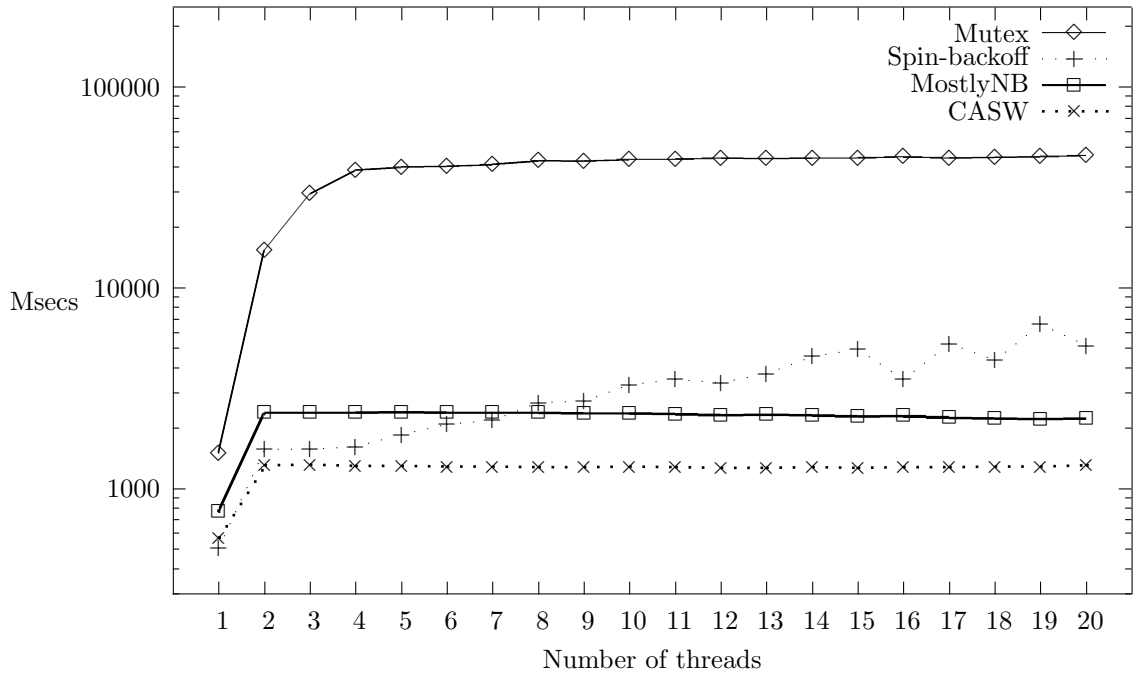


Figure 3: 2x Pentium II, 266MHz, RedHat 8

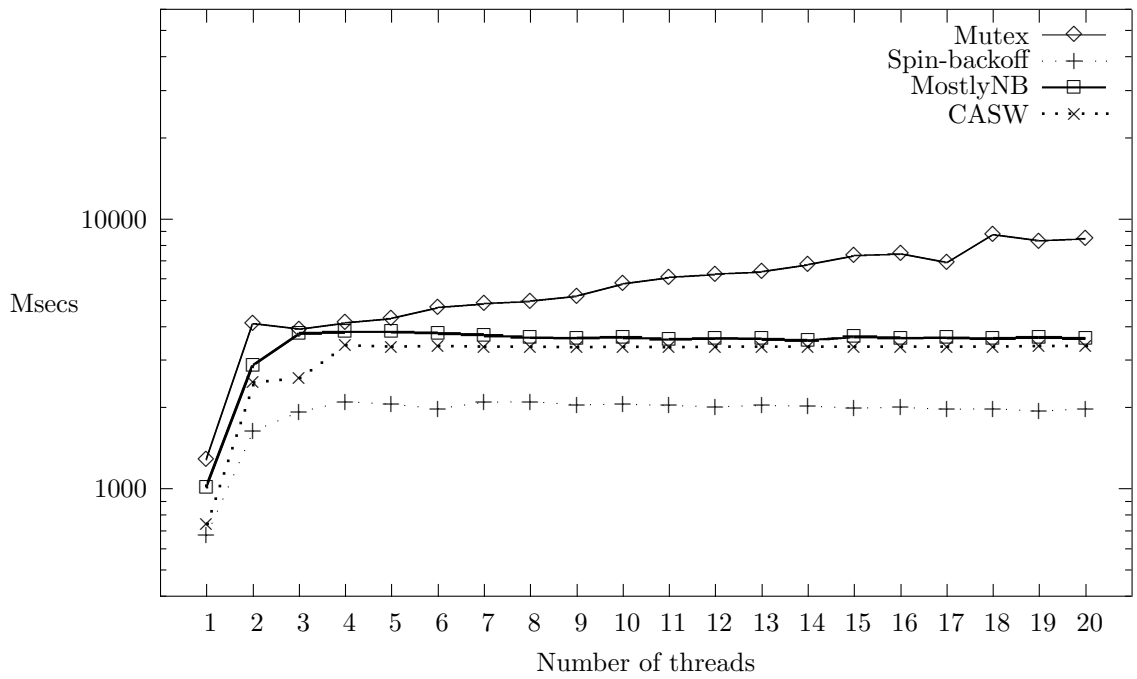


Figure 4: 4x PentiumPro, 200MHz, RedHat 9

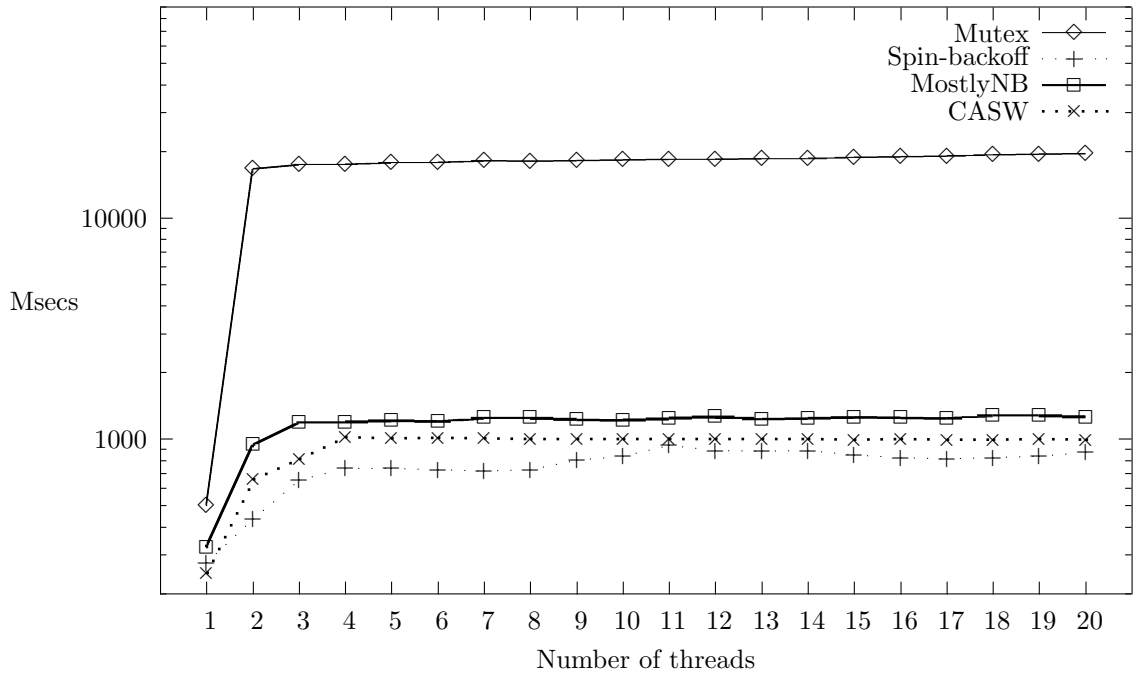


Figure 5: 2x Pentium 4 Xeon, 2.0GHz, “hyperthreaded”, RedHat 7.2, kernel 2.4.18

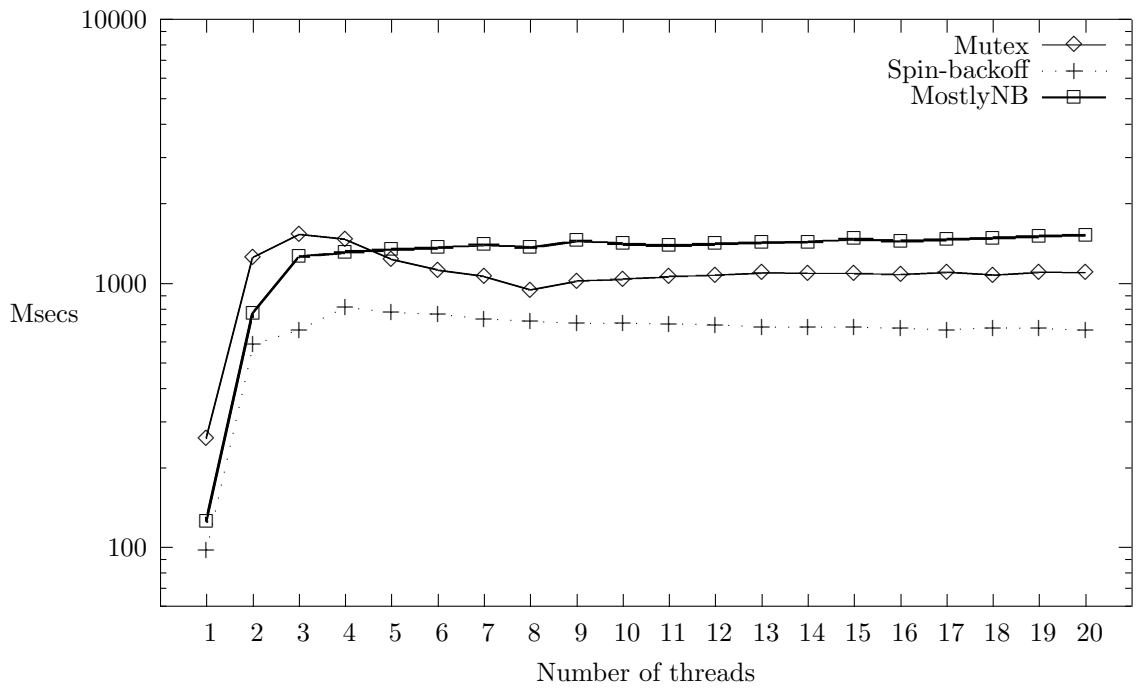


Figure 6: 4x Itanium 2, 1.0GHz, recent Debian Linux, NPTEL, kernel 2.6.2-rc1

- <http://gee.cs.oswego.edu/dl/jmm/cookbook.html>.
- [19] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, August 2002.
  - [20] M. Moir. Practical implementations of nonblocking synchronization primitives. In *Proceedings of the 16th annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, August 1997.
  - [21] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proceedings of First Symposium on Operating System Design and Implementation*, pages 179–193, November 1994.
  - [22] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 1986.
  - [23] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.