Contention resistant non-blocking priority queues

Lars Frydendal Bonnichsen

Kongens Lyngby 2012
IMM-MSC-2012-21

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Table of contents

1 Abstract

This thesis primarily deals with the design and implementation of concurrent data structures, as well as related facilities. Any concurrent data structure may have strictly limited scalability, unless care is taken in their access patterns.

This thesis seeks to investigate ways to reduce these issues, for the specific context of priority queues used for picking tasks in operating systems.

The thesis makes improvements upon a state of the art locking mechanism, to provide up 27 times faster locking, for small data structures. This is in part achieved, by improving a leading backoff scheme, and applying it in a novel fashion. We have designed and implemented a priority queue based on a balanced search tree. The new data structure is based on a new lock-free data structure based on B-trees. To the best of our knowledge, this is the first lock-free B-tree, that does not depend on the presence of a garbage collector.

2 Acknowledgments

First and foremost I would like to thank Anders Handler. You have been the best possible sparring partner during the past couple of months, where we have discussed theses on an almost daily basis.

I would also like to thank all the other guys down at the lab. I want to thank all of you for the good atmosphere, and the helpful, and enlightening discussions we have had during the writing of this thesis.

I would like to thank my long time study partners Thoai Lam Nguyen and Nawar Al-Mubaraki, for helping me proof read.

I am also very grateful to my brother Jesper Frydendal Bonnichsen, for proof reading the thesis, even though I came to you at the very last moment. I would also like to thank the rest of my family, for being very supportive during the writing of this thesis.

I would like to thank my supervisor Sven Karlsson. I have really appreciated your advice, interest and enthusiasm during this project. Finally I would also like to thank Christian Probst, who has acted as my supervisor, and managed the project since Sven fell ill.

3 Introduction

This thesis deals with data structures suitable for controlling the order in which tasks run, on computers that can run tasks in parallel. Specifically the thesis deals with the case where tasks are given priority levels, where tasks with the highest priority are run first. The general data structure for solving this issue is called a priority queue[CLRS09]. The priority queue is to be implemented into the AMD64 branch of FenixOS, a research operating system developed at DTU.

Picking the task with the highest priority takes computation time. Most solutions tend to significantly increase the computation time, when more tasks are picked concurrently, due to contention of resources. As computer systems grow in complexity, they tend to get more concurrent. With this change, it is increasingly important to be able to deal with high contention efficiently.

3.1 Contributions

This thesis presents three primary contributions:

1. Refinement of ways to keep the computation time low at high contention.
2. Refinement of contention resistant stacks and counters.
3. Introduction of a new priority queue.

Managing contention

The improved ways of keeping computation time low at high contention, are focused on ways of reducing the contention. We present three significant contributions:

1. We provide an efficient way of giving each task a unique access pattern.
2. We provide an improvement to truncated exponential backoff, which is a state of the art backoff scheme, ie scheme for reducing contention.

On the tested setups the improved backoff scheme gets up to 15 % higher throughput, in highly contended test cases. The new scheme does have the drawback, that it has a slightly higher memory consumption.

3. We show how to apply the improved truncated exponential backoff scheme to MCS locks. MCS locks is a state of the art locking mechanism, ie a mechanism for ensuring exclusive access.

On the tested setups the improved locking scheme was able to provide a shared counter up to 2700 % higher throughput. The scheme was also able to give a shared priority queue 150 % higher throughput. In general the the improved locking mechanism provides significantly better performance, when operating on contended data. The only drawback is a slightly higher memory consumption.

New priority queue

The new priority queue has 4 attractive features:

1. Finding the highest priority task has a worst case amortized running time of $O(\log n)$, in the uncontended case.
2. It supports priorities in the range $[1; 2^{31}-1]$.
3. The data structure is lock-free, ie as long as operations are being performed, at least one operation is making progress.
4. The data structure is concurrently accessible, without requiring a traditional garbage collector.

The new priority queue is built from a new dictionary data structure, with the same attractive features. Dictionary data structures provide remove and add operations, for key-value pairs. The underlying data structure is a balanced k-ary search tree, where each node can have up to k children. For instance a binary tree has k=2. To the best of our knowledge this is the first k-ary search tree that can reclaim unused resources without the use of a traditional garbage collector, or reference counting.

Evaluation of contention resistant data structures

We have also designed, implemented and evaluated the performance of contention resistant of counters and stacks. Such data structures can be used to implement priority queues for more limited priority ranges. Some of the counters and stacks use our improved backoff scheme. Some of the counters and stacks use mechanisms to reduce the number of operations on the data structures, rather than just spacing out the operations. Our evaluation show that the mechanisms to reduce the number of operations, provide at most a 4 % speedup.

3.2 Outline

This section describes the structure of the remainder of the thesis.

Chapter 4 covers the terminology, theory, and prior work that our contributions build on.

Chapter 5 covers the design and implementation of the concurrent primitives used through the rest of the thesis. The concurrent primitives include the improved backoff scheme, and the improved locking mechanism.

Chapter 6 covers the design, implementation, and evaluation of contention resistant stacks and counters, for use in bounded priority queues.

Chapter 7 covers the design, implementation, and evaluation of the new dictionary and priority queue.

Chapter 8 concludes the thesis, by bringing the findings together, and suggesting future research.

Chapter 9 contains the risk analysis, and time-line used for this project, and evaluates how the project has progressed.

4 Background

This chapter describes the terminology and theoretical background for the thesis, and prior work in the same area. The subjects covered are mainly related concurrency, non-blocking data structures, priority queues, and the hardware support for synchronization. The purpose of this chapter is to make the issues encountered in the following chapters relatable, and show how similar issues have been addressed previously.

4.1 Terminology

The kind of system studied in this thesis, is called a cache coherent shared-memory multiprocessor system.

To explicitly define what this means, we introduce the following 10 commonly used terms. Be aware that the first 3 terms are often used with meanings that differ from the ones used in this thesis:

1. Microprocessor, or processor, is a chip that is responsible for executing general purpose code.
2. A thread is a running task, that may share some of its state with other threads.
3. CPU is the hardware unit in a processor that can execute a single thread. Every thread is at most running on one CPU at any given time.

Consider a system with 4 Xeon E7-8870 processors. Each processor contains 10 “cores”, and each core is capable of simultaneously executing 2 threads, therefore the system would be said to have $4 \cdot 10 \cdot 2 = 80$ CPUs.

4. A shared multiprocessor is a system with multiple CPUs, that can access the same shared-memory. On such a system, threads can run on any CPU.
5. Scheduling is the process of picking a thread for a given CPU to run.
6. A cache is a system for speeding up access to recently accessed memory locations.
7. A cache line is a continuous fixed size memory location, that is stored in the cache.
8. Cache coherency protocols are systems for keeping shared-memory coherent across CPUs, often through cache line invalidation.
9. Cache line invalidation, is when a cache line is removed from the cache.
10. Serial bottlenecks, are when the performance of an algorithm, is limited by access to a single resource. For instance several CPUs writing to the same memory location, would typically be a serial bottleneck.

The following sections deal with the theoretical properties of tasks running on such systems.

4.1.1 Blocking data structures

When multiple threads access a data structure concurrently, it must be protected, in order to maintain a correct state. Depending on the guarantees provided by the operations, the data structure is either blocking, obstruction-free, lock-free, or wait-free. The guarantees provided are summarized in in table 1. The table considers 5 guarantees [Andrews00]:

1. Independence. Delaying or stopping threads performing operations on the data structure, does not affect other threads.
2. Fairness. Any operation is guaranteed to make progress.
3. Deadlock-free. The data structure is guaranteed to return to a usable state.
4. Livelock-free. At any time at least one operations is guaranteed to make progress.
5. Avoiding priority inversion. High priority threads never yield indefinitely to lower priority threads.

	Independence	Fairness	Deadlock-free	Livelock-free	Avoids priority inversion
Blocking	No	Maybe*	Maybe**	Maybe**	Maybe*
Obstruction-free	Yes	No	Yes	No	Yes
Lock-free	Yes	No	Yes	Yes	Yes
Wait-free	Yes	Yes	Yes	Yes	Yes

Table 1: The properties provided by different types of data structures
** depends on scheduling and synchronization*
*** depends on use*

The most common solution to accessing a data structure concurrently, is to have threads acquire exclusive access to the data structure before use. After use the thread has to release the exclusive access. The thread can safely operate on the data structure, when it holds exclusive access, because there is no concurrency. Data structures using such a solution are called blocking data structures. Blocking data structures, do not provide independence, making them unreliable in systems where threads are spuriously delayed or killed. Blocking data structures can avoid priority inversion, if the synchronization is handled in the scheduling, with schemes such as priority inheritance. If any thread may acquiring exclusive access to multiple regions, then blocking data might not be livelock-free or deadlock-free. This makes it difficult to compose blocking data structures, and using them inside operating systems.

Obstruction-free, lock-free, and wait-free data structures avoid some of the issues of blocking data structures. They do so, by never acquiring exclusive access in multi CPU systems. In practice most wait-free data structures are fairly inefficient, because they have to provide a strong fairness guarantee to all operations. Meanwhile lock-free data

structures often provide performance that is competitive with lock based data structures. As a result most work on practical non-blocking data structures has focused on creating lock-free variants. Recently a scheme has been suggested, for creating wait-free data structures with performance that resembles lock-free data structures[KP11].

4.1.2 Atomic primitives

In non-blocking algorithms, threads are not allowed to indefinitely prevent the progress of other threads. In other words, such algorithms may not use any kind of exclusivity guarantee such as a regular lock, mutex, semaphore, monitor, barrier or signal primitives.

To provide safe concurrent updates, such algorithms instead use atomic operations, such as read-modify-write operations. An example of such an operation is a fetchAndAdd operation. fetchAndAdds behavior is shown in listing 1.

```
fetchAndAdd(*a, b) {
  c = *a;
  *a = c + b;
  return c;
}
```

Listing 1: Atomic behavior of fetchAndAdd operations

The atomicity basically provides the same functionality as acquiring a lock for `a` when entering the function, and releasing it upon leaving. Read-modify-write instructions basically provide locks at an instruction level. In concurrent settings read-modify-write instructions are typically used to implement locks. The advantage of having the lock inside an instruction, is that instructions are either executed or not. In other words they first show an impact when they have been completed. Therefore the “lock” acquired when performing the instruction, is guaranteed to eventually be released. If the lock had been implemented in software, then the thread that acquired the lock could be indefinitely delayed while holding the lock.

There are many different kinds of read-modify-write instructions, with different strengths and weaknesses. The different read-modify-write instructions can be used to implement non-blocking versions of different data structures. One of the more powerful read-modify-write instructions is compareAndSwap (CAS). The behavior of CAS is shown in listing 2:

```
compareAndSwap(*adr, oldVal, newVal) {
  if(*adr == oldVal) {
    *adr = newVal;
    return true;
  }
  return false;
}
```

Listing 2: Atomic behavior of compareAndSwap operations

CAS can be used to implement any operation on any wait-free data structure, under some mild conditions[Herlihy91]. Some other read-modify-write write instructions cannot. The proof can be summarized in two parts as:

1. A solution to the consensus problem, can be used to implement any data structure with wait-free guarantees.

The consensus problem, is basically the problem of getting n entities to agree on a decision. Using a solution to the consensus problem to implement wait-free guarantees is possible, but not necessarily efficient.

2. CAS can solve the consensus problem for any number of entities.

This step assumes that CAS can set enough data to identify the decision. On most systems the decision could be identified by a memory location.

Some RISC architectures support loadLinked/storeConditional (LL/SC) instructions instead of CAS operations. The LL instruction loads from a memory location, and SC writes a value back to the location, if the data at the memory location has not changed. LL/SC can be used to implement CAS. Unfortunately the use of LL/SC on modern systems, may fail infinitely often. If LL/SC can fail infinitely often, then algorithms using it cannot provide more than obstruction-free guarantees[PMS09].

4.1.3 The ABA problem

CAS operations are frequently used to write to fields in data structures, with invocations on the form `CAS(&field, oldValueOfField, newValueOfField)`. Such usage of the CAS operation may suffer from the ABA problem, if the correct new value cannot be described as a function of the old value. Listing 3 shows a simple stack that suffer from the ABA problem.

The type stored in the stack is represented with `T`, and for the sake of simplicity the code assumes that `T` contains a next pointer. The code suffers from the ABA problem. For instance if the stack contains two elements A and B, two threads can perform the following actions:

Step	Stack contents	Actions
1	AB	2 starts popping A, 1 pops A
2	A	1 pops B
3		1 pushes A
4	A	2 finishes popping A
5	AB	

Illustration 1: The simple stack can fail in these 5 steps

If thread 2 reads `oldVal` as A and `newVal` as B, then thread 1 pops A and B, and pushes A again, thread 2 can still succeed in its pop leading to a stack containing B. In the correct case thread 2 should fail its pop, and the stack should be empty, and 2 should have failed. The problem is basically that the operation depends on the contents of the head element, and the element is not guaranteed to be constant when interleaving push and pop operations.

```
class Stack {
  T* top = nullptr; // Head pointer of the stack

  void push(T* elem) {
    T* oldVal;
    do {
      oldVal = top;
      elem->next = oldVal;
    } while(!CAS(&top, oldVal, elem));
  }

  T* pop() {
    Element<T>* oldVal, *newVal;
    do {
      oldVal = top;
      newVal = oldVal->next;
    } while(!CAS(&top, oldVal, newVal));
    return oldVal;
  }
}
```

Listing 3: Link based stack that does not handle the ABA problem.
The basic loop structure of writing do - read/calculate - while(CAS), is fairly common, especially for simple updates of data structures.

One way to alleviate the ABA problem is to include a tag in the field being updated [IBM83]. The tag is typically implemented as a counter field in the head, that is incremented whenever the head is updated. Listing 4 shows the code for such a stack:

```
class Stack {
  typedef struct {
    T* ptr;
    uintptr_t counter;
  } StackPointer;
  StackPointer head = {nullptr, 0}; // Head pointer of the stack

  void push(T* elem) {
    StackPointer oldVal, newVal;
    do {
      oldVal = head;
      elem->next = &oldVal;
      newVal = {elem, oldVal.counter + 1}
    } while(!CAS(&head, oldVal, newVal));
  }

  T* pop() {
    StackPointer oldVal, newVal;
    do {
      oldVal = head;
      newVal = {oldVal->ptr->next, oldVal.counter + 1}
    } while(!CAS(&head, oldVal, newVal));
    return oldVal;
  }
}
```

Listing 4: Link based stack using a counter tag to deal with the ABA problem

The purpose of the counter is to ensure that for any successful CAS operation, the head has not changed since it was read. If the head has not changed, the operation will be correct, since there will be no opportunity for an ABA problem. Using tags is computationally cheap, but it has 2 disadvantages:

1. It requires being able to perform a CAS operation on a field that contains the counter, as well as the original field.

In many cases, this can be accomplished by reducing the size of the field, or the counter. For the head of link based stacks, one could store the next field as index instead of a pointer, or use a smaller address space.

2. It is not a completely safe solution, since the counter can get the same value twice through overflow.

This would lead to CAS operations that write an incorrect result. In practice it is quite unlikely to happen when using large counters.

An alternative solution is to ensure that the data pointed, is constant while visible to any other thread. This can for instance be achieved with garbage collectors, that is never reusing objects.

4.2 Prior work

This section describes work relevant to the creation of concurrent priority queues, non-blocking data structures, and ways of reducing contention. The descriptions focus on the problems that have been solved, and the concepts used to solve them.

4.2.1 Dynamic non-blocking data structures

Concurrently readable, and non-blocking data structures face an interesting issue, when dynamically deallocating elements. Their elements can be read by any thread, but it unsafe to deallocate data that other threads may read from. If it sufficient to reuse elements, then it is not necessary to explicitly deallocate them, avoiding the issue. Deallocation of such elements can be handled by using a more or less reduced form of garbage collection[Michael04] [MS98] [HMBW07] [GPST05] [KPS09] [Valois95].

In order for the data structure to be non-blocking, the utilities used for operations should also be non-blocking, including the memory allocator. There are a number of fairly efficient non-blocking memory allocators in litterature, most notably nbmalloc[GPT05] and McRT-Malloc[HSAH06].

4.2.2 Garbage collection

As previously mentioned there are quite a few garbage collection schemes that can be used for dynamic non-blocking data structures. This section covers reference counting, QSBR, Hazard pointer, generic garbage collectors, and related schemes. The properties of the different schemes are summarized in table 2.

Reclamation scheme	Memory overhead	Performance overhead	General
Reference counting	$O(p+n)$	Very high	No
QSBR	Unbounded	Low for read Medium for updates	No
Hazard pointers	$O(s \cdot k \cdot p^2)$	Medium	No
Beware&Cleanup	$O(s \cdot (k+l) \cdot p^2+n)$	Higher than hazard pointers	Yes

Table 2: Properties of reclamation schemes.
The symbols correspond to the number of: p threads, n objects, k maximum active reference from a thread, l references in an object. s is the average object size.

Reference counting

Reference counting can be the simplest form of garbage collection. It is typically implemented by having a counter maintain the number of references to the object[Valois95]. When the counter reaches 0, the object can be deallocated. Reference counting has the advantage that inaccessible objects can be deallocated immediately. A significant issue with reference counting, is finding a place to store the counter. For instance, in a tree data structures it is possible to store the counter inside the globally visible pointer. Another issue is that every time a thread accesses an object, it must first write to the counter. Updating the counter can lead to contention, serial bottlenecks, and cache invalidation.

QSBR

Quiscent-state-based reclamation (QSBR) is another relatively simple garbage collection scheme. It is related to Epoch based reclamation[MS98], and the use of limbo lists/delete lists[ML84]. To safely deallocate objects, they are put on a list. When no threads are accessing the data structure, the elements on the list can be deallocated. QSBR has a relatively low performance overhead [HMBW07]. The main issue with QSBR is that nothing is deallocated, as long as one thread is accessing the data structure. Since it is generally not possible to tell if a thread is currently active, preempted threads may also prevent deallocation. QSBR is typically implemented in a locking fashion, but it is possible to implement with lock-free guarantees. A study[HMBW07] has found that the lock-free QSBR implementations tend to be slower than the lock based versions.

Hazard pointers

Hazard pointers is a scheme where each thread publicly declares that it is going to access objects before accessing it [Michael04]. If the object is still globally accessible after the declaration, the thread can safely access the object, and otherwise the thread should find another object to operate on. By globally accessible, we mean whether or not the object is reachable from the data structures current state. Illustration 2 shows an example of our definition.

To safely deallocate objects, the object is first made globally inaccessible, by removing it from the data structure. The thread that makes the object inaccessible then adds the object to a thread local list of objects that are pending deallocation. This process is called retiring. Once the list of retired objects exceed a given size, the thread deallocates

any object in the list, that is not are currently used by other threads. The cost of deallocation can be kept low, by allowing long lists. Specifically the amortized cost of deallocation is constant, if the list is allowed to grow to a size $s \propto n$, for n threads. Such a case will also permit a memory overhead of $O(n^2)$.

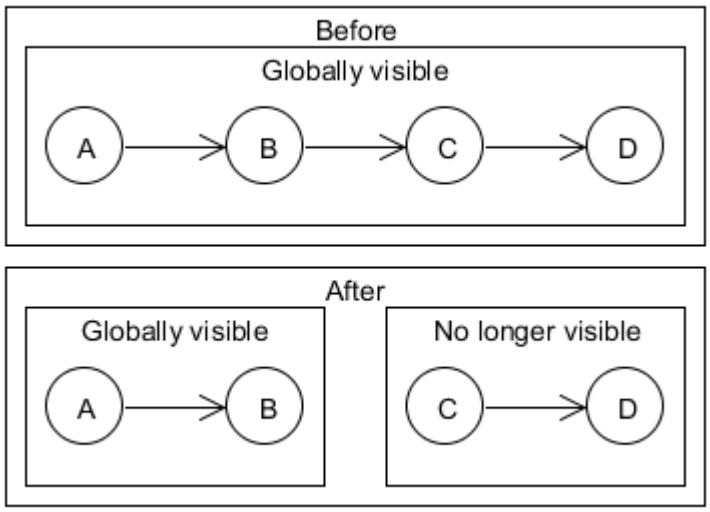


Illustration 2: A linked list before and after culling the elements after B. Afterwards C and D are not globally visible, although some threads may have references to them

Hazard pointers require strict ordering on some operations. Specifically the public declaration must happen strictly before accessing the object, leading to overhead on out-of-order machines. The efficiency of hazard pointers also depends on how difficult it is to tell whether or not objects are globally accessible. Using hazard pointers for reclaiming elements in data structures, tends to be slower than using QSBR [HMBW07]. Whether hazard pointers or QSBR has the lowest overhead, depends on the ratio of reads and replacements of elements. If elements are rarely replaced, the overhead for publicly declaring what is being read is significant.

As an alternative to testing for global accessibility, one can use the reclamation scheme Beware&Cleanup. The scheme uses a combination of reference counting and hazard pointers. By keeping reference counters, it is significantly simpler to test global accessibility for arbitrary data structures [GPST05].

Generic garbage collectors

Most generic garbage collectors are lock based, including those used in runtime environments. Popular examples include the garbage collectors of the java virtual machine (JVM), and the common language runtime (CLR). Recently a lock-free generic garbage collector has been proposed [KPS09]. Unfortunately implementing such a garbage collector would seem like a significant effort. Additionally we could not find any evaluation of its performance or memory overhead of the garbage collector.

Summary

There are several garbage collection schemes, and none of them are optimal for every case. Generic lock-free garbage collectors may be coming, but they will likely require significant implementation effort, and might not have comparable performance. QSBR usually has the lowest performance overhead, but it provides no guarantees about its memory overhead. Reference counting usually has the lowest memory overhead, but it is not generally applicable, and it can cause significant contention. The memory overhead of hazard pointers increases dramatically with the number of threads, and it requires implementing an accessibility test. The test may be expensive, or even impossible for arbitrary data structures. Beware&Cleanup trades the overhead of the accessibility test of hazard implementation with occasional use of reference counting.

QSBR tends to be fastest. If it is possible to argue about how often all threads stay away from the data structure, then QSBR is probably the best solution. Hazard pointers are likely to have the second lowest performance overhead, if testing whether objects are globally accessible is simple. If the test is expensive Beware&Cleanup is likely to have the second best performance overhead. Reference counting can be extremely slow, and it is not generally applicable. Reference counting does however tend have the lowest possible number of unreclaimed objects.

4.2.3 Providing non-blocking algorithms

There are a few general strategies, that are typically used to implement non-blocking data structures.

One strategy is to follow the following three steps:

1. Create a partial copy of the data structure.
2. Perform the operation on the copy.
3. Write the partial data structure back into global storage, unless the data structure has changed.

In such a setup, any thread may start an operation at any time. If the operation fails, then it must be because some other operation made progress.

Many non-blocking data structures have been implemented efficiently in ways that are similar to this strategy. Specifically stacks [IBM83], skip-lists [Sundell04], queues [Michael04] and counters have been implemented in similar fashions.

Another strategy, referred to as help locking works as follows:

1. Write the operations being performed directly into the data structure, to avoid conflicting operations.
2. Perform the operations written into the data structure.

The name “help locking” derives from, the strategy avoiding conflicts in a manner similar to locking, while avoiding blocking. It avoids blocking, by allowing other threads to complete pending operations. A pseudo code version of the general strategy can be seen in listing 5.

The operations is first written with CAS or a similar read-modify-write instruction. If writing the operation fails, then help the operation preventing the write, and retry writing. After successfully writing, perform the operation, and remove the description.

```
doOperation(op, dataStructure) {
  do {
    status = dataStructure.tryToAdd(op);
    if(status != SUCCEEDED) {
      status.helpPreventingOperation();
    }
  } while(status != SUCCEEDED);
  datastructure.do(op);
  datastructure.remove(op);
}
```

Listing 5: General form used in help locking

The actual operations execute in a very similar fashion to the other strategy. The advantages of this strategy are that the operations can have multiple steps, there is less need for local copies, and it may simplify checking for success. The main disadvantage is the expense of making more writes to global memory. Some of the more complex non-blocking data structures have been implemented in this way [Fomitchev03] [BH11] [EFRB10].

There are also 3 ways to create non-blocking versions of data structures, that require fewer changes:

1. Use of a lock-free software transactional memory (STM) scheme [Fraser04].
2. Use a solution to the consensus problem to orchestrate operations on objects [Herlihy91].
3. Conversion of explicit locks to lock-free operations using the scheme from the paper “Locking without blocking” [TSP92].

All three solutions are applications of the help locking strategy, applied in ways that work for any data structure. These general solutions do however have significant disadvantages. Using the consensus problem to implement wait-free data structures tends to be extremely slow, and has a high memory overhead. The scheme from “Locking without blocking”, was primarily a proof of concept, for there being other general solutions with lower overhead. From what we can tell the details of the scheme was never fully published, or used outside the paper. STM is a more realistic solution. It depends on having a specialized framework, and/or compiler support. Data structures based on STM tend to perform worse than data structures that have been adapted to be lock-free by hand [Fraser04].

4.2.4 Non-blocking priority queues

There are several existing lock-free priority queues. They generally use some kind of search structure to find a location to insert or extract elements from.

One overall strategy for implementing priority queues is only allowing a fixed range of priorities and keeping a container for each priority level. This scheme is often called static search structures or quantatized priority queues. Such schemes has the advantage that it does not really require much in the way of maintenance, since there is a 1:1 mapping from priority level to container. The message passing system Tempo[BER07], implements such a priority queue. That queue is based on the SimpleTree data structure described in the paper "Scalable Concurrent Priority Queue Algorithms" [SZ99], only using lock-free counters and stacks.

Alternatively one can use a dynamic structure, allowing for a wider range of priorities, at the expense of additional maintenance. Prior work has been done on using non-blocking heaps[IR93]. Unfortunately most lock-free heaps depend on exotic read-modify-write instructions, that are not generally available. Some of the instructions can be simulated on current hardware, either using STM or special algorithms, but it tends to come at a significant cost. By comparison some lock based queues perform rather well, especially under low contention [DB08].

Priority queues based on lock-free and lock based skip-lists have also been brought forth [ST05] [SL00]. Skip-lists normally supports add remove operations, of key-value pairs. It is possible to adapt skip-lists to work as priority queues, with the limitation that the elements in the queue must have unique keys. Providing unique keys can be accomplished in a number of ways. For instance using key-values pairs as keys, having redundancy bits in the keys, or by using references to containers as values. Storing a container in each element, may require using more complicated lock-free objects, to ensure that they can be composed properly. Recently non-blocking binary search trees[EFRB10], and k-ary search trees[BH11] have been proposed. A k-ary search tree is a search tree where each node can have up to k children. It might be possible to use those data structures as priority queues, with similar schemes to those for skip-lists.

4.2.5 Backoff schemes

Backoff schemes are measures to keep contention low. One way of doing this is to make threads wait before or after performing operations, or after failing to perform operations. One of the more influential papers dealing with backoff schemes for concurrent data structures is "The performance of spin lock alternatives for shared-money multiprocessors" [Anderson90]. The paper presented a number of backoff schemes, and applies them to a lock implementation. The more successful schemes are simulating queuing, using constant individual spin times to each thread, and truncated exponential backoff.

Simulating queuing works by having the processors get in a queue, and perform their operation when its their turn. There have been proposed a number of different kinds of queue based locks, such as CLH locks [Craig93] [MLH94]and MCS locks[MCS91], providing different tradeoffs. The schemes can all dramatically reduce contention.

Unfortunately they are not directly applicable for non-blocking data structures, since it forces threads to wait for each other.

Giving each thread a constant individual spin time, can keep contention low, and ensures that some processors can operate with fairly high throughput. Unfortunately good results require highly tuned spin times. Poor choices for the individual spin times can lead to redundant or insignificant spinning. This results in overhead from spinning or contention. In short, the solution is not very flexible, and might give very poor results if not properly tuned.

Truncated exponential backoff works by having the threads spin for a number of time slots before attempting to perform operations. The number of slots to spin for is sampled from a discrete uniform random distribution. The longest possible duration is stored individually for each thread. If the threads detect significant contention when performing an operation, they double the upper bound on the spin duration. If it detects low contention it halves its spin duration. The longest possible spin duration has an upper bound (truncation), proportional to the number of CPUs. The truncation reduces the overhead for changing levels of contention. Additionally it ensures that exponential backoff is always competitive to assigning each processor separate spin times. The scheme assumes you can find some way of detecting contention. For CAS operations, one hint would be failing operations. For other read-modify-write instructions that provide the old value there are the following 4 possibilities:

1. Reading the value being updated, before changing it, and check if someone else changed the value in between.
2. Remembering the fields last value, and assume contention, if it has changed.
3. The value might indirectly be a tell of whether or not there is significant contention.
4. The level of contention on one object might be proportional to the contention of another object.

4.2.6 Elimination and combination of operations

For some data structures, their operations can be eliminated and combined with one another, to reduce contention. Doing so reduces the number of operations on the contended data. Combination and elimination can be implemented through a synchronization scheme called combining funnels [SZ00]. Combining funnels have been applied to counters, bounded counter, and stacks by the authors of the original paper. They used the bounded counters and stacks to implement a priority queues [SZ99]. The priority queues showed significantly better performance than competing priority queues, such as the queue by Hunt et al [HMPS96]. The priority queues were compared under high contention, running on simulated hardware with hundreds of processors. Combining funnels have not been extensively studied lately, but some of its concepts have seen recent interest. For instance, elimination has been applied to the lock-free stack by Treiber et al[IBM83] [HSY10]. The stack with elimination is still lock-free, because elimination of two opposing operations is fairly simple to implement in a lock-free fashion. They were able to show a significant performance improvement over a regular stack, on a specific hardware setup.

4.3 Summary

This chapter introduced the theoretical background for rest of the thesis, and prior work in the same area. The theoretical background covered features of concurrent data structures, and the concepts necessary to argue about them. The prior work included concurrent data structures, and high level ways of ensuring properties of concurrent data structures. The remainder of this thesis is primarily about concurrent priority queues, and concurrent primitives. Therefore the atomic primitives, concurrent priority queues, and backoff schemes were covered in greater detail.

5 Concurrent building blocks

5.1 Introduction

This chapter describes basic building blocks used to for the data structures presented in the remainder of this thesis. The chapter covers methods for accessing hardware synchronization primitives, reducing context switches, and reducing the impact of contention.

Reducing the impact of contention includes ways to randomize access patterns, and ways to implement backoff schemes. We will also apply those concepts to lock implementations, so we can perform fair performance evaluation, when comparing lock based and lock-free data structures.

5.2 Random number generation

Generating a uniformly distributed pseudo random number is a fairly standard feature in many standard libraries and toolkits. Pseudo random numbers are typically generated by initializing a pseudo random number generator (PRNG) to a certain state, and picking new numbers by evolving that state. The evolution is deterministic, unlike true random number generators.

On SMP machines the PRNGs internal state is usually protected with locks. Locking in order to get a random number may by quite expensive, and it completely ruins the advantages of non-blocking data structures. To avoid such issues we have each thread store a PRNG, and ensure that threads only access their own PRNG.

There are many different PRNGs for uniform distributions, and picking a suitable PRNG depends on what it is used for. The typical performance metrics of an RNG are its period length, internal state size, the time it takes to evolve its state, how closely the output follows a uniform distribution, and how random the output is. There are a number tests to determine how random the output of a PRNG is, but the criteria they measure are beyond the scope of this thesis. The length of the period is how many numbers it can generate before reaching a previously reached internal state.

We primarily use PRNGs to randomize access patterns to contended resources. For instance randomizing the amount of time a thread back off before accessing contended fields, or picking a random order to access fields in. The primary reason for using PRNGs in randomized access patterns, is to reduce the chance of resources being contested. We opt to use a linear congruential generator (LCG). An LCG has very small state, and fast evolutions, at the expense of poor randomness and period length. Poorer randomness, and a shorter period are not significant issues for our uses, as described in the following section.

5.2.1 LCGs for randomizing access patterns

An LCG that generates m-bit integers stores an m-bit internal state x. LCGs evolve their state as $x_{i+1} = (x_i \cdot a + c) \bmod m$, where m is typically 2^{16}, 2^{32} or 2^{64}. Listing 6 shows what such a RNG might look like:

```
template<class IntType, IntType a, IntType c>
class LCG {
  IntType x;
  IntType rand() {
    x = x * a + c;
    return x;
  }
}
```

Listing 6:A templated LCG, where a, c, and the type of integer generated is passed as template parameters

The quality of the random numbers depend highly on the values used for a, c, and m. If an LCG can achieve all internal states possible, it is said to have a full period. LCGs have full periods if the following conditions are met [Knuth97]:

1. c and m are relatively prime
2. a-1 is divisible by all prime factors of m
3. a-1 is a multiple of 4, if m is a multiple of 4

When $m = 2^n, n \in \mathbb{N}$, the conditions translate to $(c \bmod 2) = 1 \wedge ((a-1) \bmod 4) = 0$, since the only prime factor of such m is 2.

LCGs with full periods do not necessarily produce good representations of random numbers. However an LCG without a full period produce a poor representation of uniformly distributed numbers, because it cannot generally produce all numbers. Our LCG has $a = 2531011$, $c = 2531011 + 2 \cdot t_{id}$, where t_{id} is the threads id. Using a different c value for each thread ensures that the each thread evolves the internal state of in a unique pattern, assuming $t_{id} < 2^m - 1$.

LCGs are often considered to be poor PRNGs, because they have relatively short periods, and the numbers produced from a single LCG tends to be highly correlated. In other words it is easy to predict the next value generated based on a few outputs, even if a and c are unknown. LCGs do however have some redeeming qualities. If you sample 2^k random numbers from an LCG with $m = 2^n, n \in \mathbb{N} \wedge n \geq k$ then there will be no numbers with the same k least significant bits. This property is very attractive when randomizing access patterns, if the number of elements to chose from is a power of two, and each LCG evolves its state in a unique fashion. The property can also be used generate 2^k different values relatively quickly, without having to resort to shuffling.

Proof:

1. The result of updating the k lowest bits of x only depends on a, c and the previous k lowest bits of x. This is true because the update of x can be written as a series of additions $x_{i+1} = \left(\left(\sum_{j=1}^{a} x_i \right) + c \right) mod\, m$, and the k lowest bits of the additions only depend on the k lowest bits of x.
2. The k lowest bits of the samples from the LCG correspond to the entire result of an LCG B. B has the parameters $a_B = a, c_B = c, m_B = 2^k$, and an initial state that corresponds to the k lowest bits of the original LCGs initial state.
3. Since the LCG B uses the same variables uses the same a and c parameters, it also satisfies the constraint $(c\, mod\, 2) = 1 \wedge ((a-1)\, mod\, 4) = 0$. Therefore B has a full period.
4. The LCG B would therefore produce 2^k different variables, meaning the lowest k-bits of the original LCGs samples are different from each other.

The property is also one of LCGs primary weaknesses. The lowest k bits of the samples produced by the LCG have a period of 2^k. This implies that the lowest bit has a period of 2, the two lowest bits have a period of 4, and so on. This property puts an upper bound on the periods of the individual bits of the samples. The bound primarily affects the lowest bits. For this reason one should generally avoid depending on the randomness of the lower bits in LCGs. This also means that when generating samples from the Bernoulli distribution based on a sample from an LCG, one should generally use code that looks like: `lcgSample < successCriteria`.

A similar property exists for some forms of XOR shift PRNGs. We chose to use LCGs because the space of useable XOR shift PRNGs is smaller, and they tend to be slower than LCGs on AMD64 hardware. By comparison, using a “better” PRNGs, such as Mersenne Twisters, or multiply-with-carry PRNGs, may produce more random output, but they do not have the property.

5.3 Avoiding context switches

Using threads to implement task states requires storing the state of the thread, whenever it is preempted and restored. In some cases it may be possible to avoid restoring the entire state of the thread. For instance when restoring a thread that previously ran, registers that the operating system does not touch do not need to be restored. Even with such optimizations switching threads inside and outside of the kernel can still be expensive.

Instead of using threads, one can describe tasks with continuations. Continuations are typically identified as a function, and the parameters to the function. Continuations are frequently used inside functional programming languages. Uses include lazy evaluation of variables, where the evaluation is postponed, by storing a continuation to the computation of the variable, rather than the actual variable. For scheduling purposes, the advantage of continuations is that they may be run within the context of any thread, with the same privileges, avoiding the need for context switches. To replace the use of

threads with continuations, the task should never block, but instead create a continuation that finishes the task after blocking.

Continuations have been used to avoid using threads inside the L4::Pistachio kernel [Warton05], by instead using continuations whenever code blocks inside the kernel. By avoiding the use of kernel threads, each processor only needs to have one stack for use inside the kernel. The L4::Pistachio already applied other tricks to avoid most full context switches, but continuations seemed to improve performance, due to better cache locality when keeping the same stack. Prior to L4::Pistachio the Mach kernel also used continuations to reduce the need for threads inside the kernel [Draves94] [DBRD91]. In addition the Mach 3.0 kernel used what they referred to as continuation recognition, to implement fast paths for certain types of operations. For instance continuation recognition was used for the send/receive operations in inter processes communication. If a message is sent to a thread that is waiting for a message, then the kernel can in some cases process the message inside the sender kernel context. Doing so avoids running the continuation for the receiving thread.

Describing tasks as continuations has also been used in various user level threading libraries, including the C-Threads library for Mach 3.0 [Dean93]. The library used the same basic framework as the Mach kernel to implement continuations, and also supported continuation recognition.

More recently C++11 lambda expressions with capture, can also model continuations. Listing 7 shows how to create a continuation `c` in C++11. The continuation can be called at a later time to evaluate calculation(0, 1).

```
int a = 0, b = 1;
std::function<int()> c = [=]() {
  return calculation(a, b);
};
```

Listing 7:Creates a continuation to `calculation(a,b)`.
`[=]` defines how to store the parameters `a` and `b`.

One significant problem with the use of continuations for describing tasks, is that the code must be split into functions whenever a blocking call is executed, in order to describe the remainder of the operation. This requires splitting code blocks that are logically connected, leading to a more fragmented view of the operations, and it may require significant amounts of refactoring. Parallel Patterns Library is a library by Microsoft that can wrap C++11 lambda expressions into “tasks”, in attempts to reduce the issue. They do so by having “tasks” store a field that can contain the continuation to call after performing blocking calls. This means that they can define the continuations in a way that resembles a normal structured flow[PPL11].

5.4 Interfacing to synchronization primitives

Synchronization primitives, such as read-modify-write instructions, are used to provide guarantees, to the executing threads. In order to use the read-modify-write instructions of a given platform, in a non-assembly language, it is necessary to interface to them in some way. This section investigates the performance and code impact of various interfaces to such instructions. The purpose of this comparison is to find interfaces to

the atomic instructions of the AMD64 instruction set, that leads GCC to produce the best results. Whether or not the results are good, is determined based on the throughput, and the quality of the assembly code generated for selected test cases. The comparison primarily focuses on the `CMPXCHG` instruction, corresponding to CAS. We chose to focus on that instruction, because it is one of the more difficult instructions to interface to, and because it is frequently used in non-blocking data structures.

This section starts out describing the guarantees provided by the compiler and processor, used tested interfaces, and it ends with a discussion of the results.

5.4.1 Analysis

This section covers the theoretical background for implementing and using interfaces to read-modify-write instructions. It covers memory ordering guarantees provided by the GCC, and AMD64, the read-modify-write instructions of AMD64, and ways to interface to them.

5.4.1.1 Memory ordering

Strict ordering of memory accesses is required for the correctness of some algorithms, such as Dekkers algorithm, Petersons algorithm, and the Hazard Pointers. One typically use abstractions referred to as memory barriers. Memory barriers ensure that all the CPUs memory accesses made before the barrier are completed before leaving the barrier. Implementing memory barriers requires interacting with both the compiler and underlying CPU architecture.

At a compiler level, the volatile qualifier can be applied to variables in C++, to ensure that the compiler does not break the memory ordering. Specifically access to volatile variables guarantees that the compiler will not reorder any accesses to the variable. It does not work as a memory barrier, since accesses to non-volatile variables can be reordered across volatile accesses. In addition the compiler can still remove code that accesses volatile variables, if it can guarantee that the code is never reached. The volatile qualifier does not guarantee anything about how the processor will execute the code, it only makes guarantees about the produced assembly code. In other words whether or not the memory ordering is satisfied, depends on the processor that it is running on.

The AMD64 instruction set provides the following guarantees[AMD10]:

- The weakening of the ordering is never visible to a single CPU system.
- Stores to regular memory (write-back memory) is executed in order
- If n CPUs write to a memory location, and m other CPUs observe the memory location, then the m CPUs will see the writes in the same order.
- Loads cannot be executed out of order, with respect to loads and stores to the same memory location.

To implement memory barriers, one can use one of the following instructions, ordered by how fast they tend to be:

- `SFENCE` guarantees that all the CPUs stores to memory locations are terminated before the next store to a memory location.
- `LFENCE` guarantees that all the CPUs loads from memory locations are terminated before the next load from a memory location.
- `MFENCE` guarantees that all the CPUs access to memory locations are terminated before the next access to a memory location.
- Instructions with the `LOCK` prefix also serve as mfences.

In general the instructions that provide the weakest guarantees are the fastest. When accessing regular write-back memory in AMD64, the stores are already in order, so sfence is only necessary when using other types of memory.

5.4.1.2 Available primitives

The AMD64 instruction set supports a number of commonly used read-modify write operations[AMD09], that return the read value in a register, as summarized in table 3. The AMD64 instruction set supports additional read-modify-write instructions, that only indirectly refer to the read value through condition codes as summarized in table 4. Condition codes are flags that are set after the execution of instructions. They can be used to branch, and make predicated assignments. The condition codes can also be read explicitly.

All of the instructions can be performed on 8, 16, 32, and 64 bit memory locations, and CAS can additionally be performed on 128 bit memory locations. Any memory location being written to must be aligned on its own base, for instance 32 bit memory locations must be aligned on 32 bits. The instructions must be prefixed with `LOCK,` when used in contexts with multiple CPUs, except for `XCHG` which is “always locked”.

Instruction (mnemonic)	Description
extended CAS (`CMPXCHG`)	Like regular CAS but it returns both whether it succeeded or not, and the previous value.
fetchAndAdd (`XADD`)	Regular fetchAndAdd
fetchAndStore (`XCHG`)	Swaps a value in memory with a register value

Table 3:The conventional read-modify-write instructions of the AMD64 instruction set

Instruction mnemonic	Description
`INC`, `DEC`	Increment and decrement
`ADD`, `SUB`, `ADC`, `SBB`	Add and subtract, with and without a carry or borrow bit
`OR`, `AND`, `XOR`, `NOT`	Bitwise operations
`NEG`	Subtracts the value of an integer from 0
`BTC`, `BTR`, `BTS`	testAndComplement, testAndClear, testAndSet

Table 4:Atomic instructions that only provide output through the condition codes

5.4.1.3 Interfacing to read-modify-write instructions

There are a number of different ways to interface to the read-modify-write instructions of any given platform. FenixOS is targeted towards GCC, so the most obvious solutions is using the compiler builtins, using GCC inline assembly syntax, or the atomic primitives introduced in C++11 described in proposal N2427[BC07].

Unfortunately using the C++11 features is not an option, since the primitives depend on library functions. Such functions are not available inside FenixOS, unless we implement them ourselves. A feature in the proposal is that it allows programmers to describe the minimum memory ordering that should be enforced for each atomic operation. Doing so can affect the overhead of the code produced by the compiler.

GCC inline assembly syntax allows the programmer to run assembly code in other programming languages. The read-modify-write instructions can be performed in assembly code, wrapped in functions or macros, to provide a simple API. By comparison GCC's atomic builtins are wrapper functions to atomic operations, that the compiler creates read-modify-write instructions from. The atomic builtins supported by GCC are basically the same as ICC's atomic builtins [GNU11]. The ICC builtins are in turn inspired by the Itanium processor architecture.

5.4.2 Implementation

5.4.2.1 The interfaces

GCC provides the builtins `__sync_bool_compare_and_swap` and `__sync_val_compare_and_swap`, as an interface CAS operations. The builtins return whether or not the operation succeeded and the previous value at the memory location respectively. Additionally GCC provides the builtins `__sync_fetch_and_add` and `__sync_test_and_set`, to interface to the fetchAndAdd and fetchAndStore operations, respectively.

Most of the calls to the atomic builtins work as memory barriers for both hardware and the compiler. This means that that all loads and stores before the call are completed before the call, and no loads or stores are moved from after the call to before the call.

GCC inline assembly syntax allows the programmer to specify constraints for placement of variables. This may give the compiler more freedom regarding register windowing, leading to less overhead from using inline assembly. Each assembly block contains a list of outputs, inputs, and a clobber list. Each element in the input and output lists specify constraints on where they are placed. The clobber list specifies the variables or memory locations that are invalidated.

```
template <class T>
T casVal(volatile T * const adr, T oldVal, T newVal) {
    asm volatile("LOCK CMPXCHG %2, %0" // assembly code
    : "=m"(*adr), "=a" (oldVal) // output
    : "r"(newVal), "1"(oldVal)); // input
    return oldVal;
}
```

Listing 8:`casVal`, a templated interface to CAS that returns the previous value.

Listing 8 shows the interface `casVal` that returns the value previously stored at `adr`. The function loads `oldVal` into the `a` register, and the `newVal` into any register. Then it performs a `LOCK CMPXCHG` instruction, and returns the value stored in the `a` register.

The interface `casBool` is presented in listing 9. `casBool` also returns a boolean value that specifies whether or not the operation succeeded, by returning the zero bit of the condition codes.

```
template <class T>
bool casBool(volatile T * const adr, T* oldVal, T newVal) {
    uint8_t success;
    asm volatile(" LOCK CMPXCHG %3, %0; setz %2"
    : "=m"(*adr), "+a"(*oldVal), "=r"(success)
    : "r"(newVal));
    return success;
}
```

Listing 9:`casBool`, a templated interface to CAS that returns whether the operation succeeded, in addition to the previous value through oldVal

In most cases the purpose of getting a boolean success value, is to branch based on whether or not the operation succeeds. Doing so based on the previous interface might be inefficient, since the compiler generally cannot optimize the assembly blocks, or interpret the condition codes after an assembly block. One way to reduce this problem is to use assembly goto blocks. Asm goto blocks are supported by GCC 4.5 and newer. Such blocks allows the assembly code to jump to regular code, through the use of labels, but such blocks cannot currently specify outputs.

The interface `casGoto` in listing 10 avoids this problem by performing the operation in an assembly block, and branch in a separate assembly goto block. It branches based on the `zf` condition code, similar to how `casBool` returns `zf`. Although the compiler cannot optimize the assembly code, it can optimize the code that the assembly block jumps into, by inlining the function.

```
template <class T>
bool casGoto(volatile T* adr, T* oldVal, T newVal) {
    T o = *oldVal;
    asm volatile(" LOCK CMPXCHG %2,%0"
    : "+m"(*adr), "+a"(o)
    : "r"(newVal));
    asm volatile goto("JNZ %l[failed]"::::failed);
    *oldVal = o;
    return true;
failed:
    *oldVal = o;
    return false;
}
```

Listing 10:`casGoto`, a templated interface to CAS that returns the previous value through oldVal, and branches based on whether or not the operation succeeded

All of the inline assembly blocks allow the compiler to keep temporary copies of variables, as long as they do not overlap with `adr`. On the other hand the hardware is not allowed to speculate any loads or stores across the `LOCK CMPXCHG` instruction, because of the semantics of the lock prefix. In some cases it might be desirable to ensure rereads the variables after a read-modify-write instruction. In such cases one can add assembly blocks after the atomic operation, that specify that the memory may have changed, forcing the compiler discard any temporary copies.

For other atomic instructions, one can use inline assembly similar to the `casVal` to interface to the registers returned, and blocks similar to `casBool` or `casGoto`, to interface to the condition codes returned. To use the 128 bit CAS operations, we use code that is very similar to the previous interfaces, but it stores the value of `oldVal`, and `newVal` in two 64 bit values. For instance the `casGoto` code for 128 bit values is:

```
bool casGoto(volatile uint128_t* adr, uint128_t* oldVal, uint128_t
newVal) {
    uint64_t o1 = *oldVal, o2 = (*oldVal) >> 64;
    asm volatile("LOCK CMPXCHG 16B %0"
    : "+m"(*adr), "+a"(o1), "+d"(o2)
    : "b"((uint64_t)(newVal)), "c"((uint64_t)((newVal)>>64)));
    asm volatile goto("JNZ %l[failed]"::::failed);
    *oldVal = (((uint128_t)o2) << 64) | o1;
    return true;
failed:
    *oldVal = (((uint128_t)o2) << 64) | o1;
    return false;
}
```

Listing 11:The 128 bit version of casGoto. The primary difference is that all parameters are stored in specific registers

5.4.3 Evaluation

This section describes how the interfaces were applied, how the interfaces performed, and describe the characteristics of the generated code. The results are abbreviated to the most relevalt pieces.

5.4.3.1 Setup

To evaluate the performance we have used the different interfaces to implement increment, xor, and swap operations, using CAS. We also implemented interfaces to the read-modify-write instructions `add`, `inc`, `swap`, and `xor`, to compare their respective performance. The interface based on `inc` does not have the same functionality as the other adding interfaces, and instead it always adds 1. All the instructions, aside from CAS, only support up to 64 bit fields. We also implemented a 128 bit `add` operation using locked `add` and `adc` instructions, and a 128 bit `xor` with two locked `xor` operations. Performing the operation with two read-modify-write instructions means that the value stored in the memory location might not be valid at all times, but the end result will be correct, because add and xor operations are associative. The operations are implemented in slightly different fashions for the various interfaces, as shown in the following listings 12 through 14.

```
T add(T* field, T increment) {
  T old;
  do {
      old = *field;
  } while(!CAS(field, old, old + increment));
  return old;
}

T swap(T* field, T newVal) {
  T old;
  do {
    old = *field;
  } while(!CAS(field, old, newVal));
  return old;
}

T xor(T* field, T bits) {
  T old;
  do {
      old = *field;
  } while(!CAS(field, old, old ^ bits));
  return old;
}
```

Listing 12:The operations implementation, when using __sync_bool_compare_and_swap

```
T inc(T* field, T increment) {
  T old = *field, tmp;
  while(old != (tmp = CAS(field, old, old + increment))) {
    old = tmp;
  }
  return old;
}

T swap(T* field, T newVal) {
  T old = *field, tmp;
  while(old != (tmp = CAS(field, old, newVal))) {
    old = tmp;
  }
  return old;
}

T xor(T* field, T bits) {
  T old = *field, tmp;
  while(old != (tmp = CAS(field, old, old ^ bits))) {
    old = tmp;
  }
  return old;
}
```

Listing 13:The operations implementation when using __sync_val_compare_and_swap or casVal

```
T inc(T* field, T increment) {
  T old = *field;
  while(old != CAS(field, &old, old + increment));
  return old;
}

T swap(T* field, T newVal) {
  T old = *field;
  while(old != CAS(field, &old, newVal));
  return old;
}

T xor(T* field, T bits) {
  T old = *field;
  while(old != CAS(field, &old, old ^ bits));
  return old;
}
```

Listing 14:The operations implementation, when using casBool or casGoto

5.4.3.2 Evaluated performance

This section evaluates the performance of the interfaces to the atomic operations. The tests were performed by doing the 300,000 operations on a shared field, as seen in listing 15. The tests were performed using up to 16 threads, where each was bound to a specific CPU. The tests started all the threads simultaneously, and each thread measured the wall-clock time when they start and finished their tests. Each of the presented results are based on 160 measurements. To get 160 measurements, when testing with p threads, the tests is run 160 / p times.

```
field

opTest() {
  … measure start time
  for(i = 0 .. 300000) {
    useInterface(&field, id)
  }
  … measure completion time
}
```

Listing 15:The test run to evaluate performance of the interfaces

Based on the measurements when run on the system described in table 5, we present the graph 1 through 3.

Graph 1 shows the throughput of the interfaces in the single threaded case, as a function of the size of the operations. Graph 2 shows the thread total throughput of the interfaces, averaged over 8, 16, 32 and 64 bit operations, as a function of the number of threads. Graph 3 shows the thread total throughput of the interfaces as a function of the number of threads.

The graphs name the individual tests as the concatenation of the operation and interface of the test. The naming scheme used for the interfaces is given in table 6.

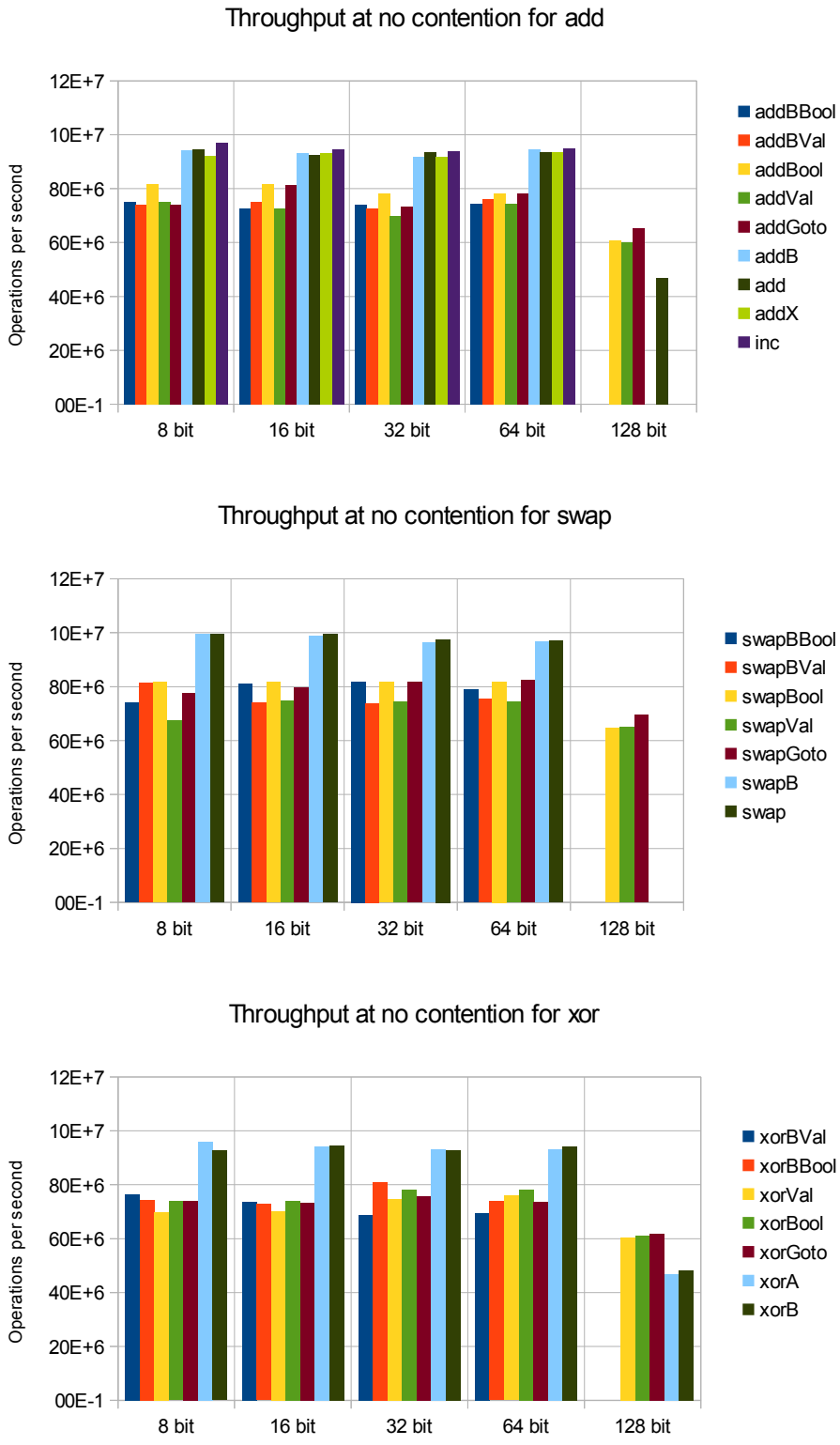
By thread total throughput, we refer to the sum of the threads throughput. The term is different from system throughput. The difference between the two terms, is that the thread total throughput calculates each threads throughput over the based on the threads start and completion time. The system throughput and thread total throughput are only the same, if all of the threads completed the tests in the same time. Typically the system throughput can be no more than constant, when increasing the number of threads running the operations on highly contended data. In the same setting the thread total throughput can double, if the threads completion times are uniformly distributed.

System name	HP ProLiant SL165z G7 server
Ram	64 gb
OS	Scientific Linux 6.1
Processors	2 x AMD Opteron 6168 (24 CPUs)
Compiler	GCC (Ubuntu/Linaro 4.6.1-9ubuntu3)
Compiler flags	`-m64 -std=gnu++0x -fopenmp -DNDEBUG -fopenmp -Ofast -fwhole-program -static -flto -fno-align-functions -fno-align-labels -fno-align-loops -fno-align-jumps -s`

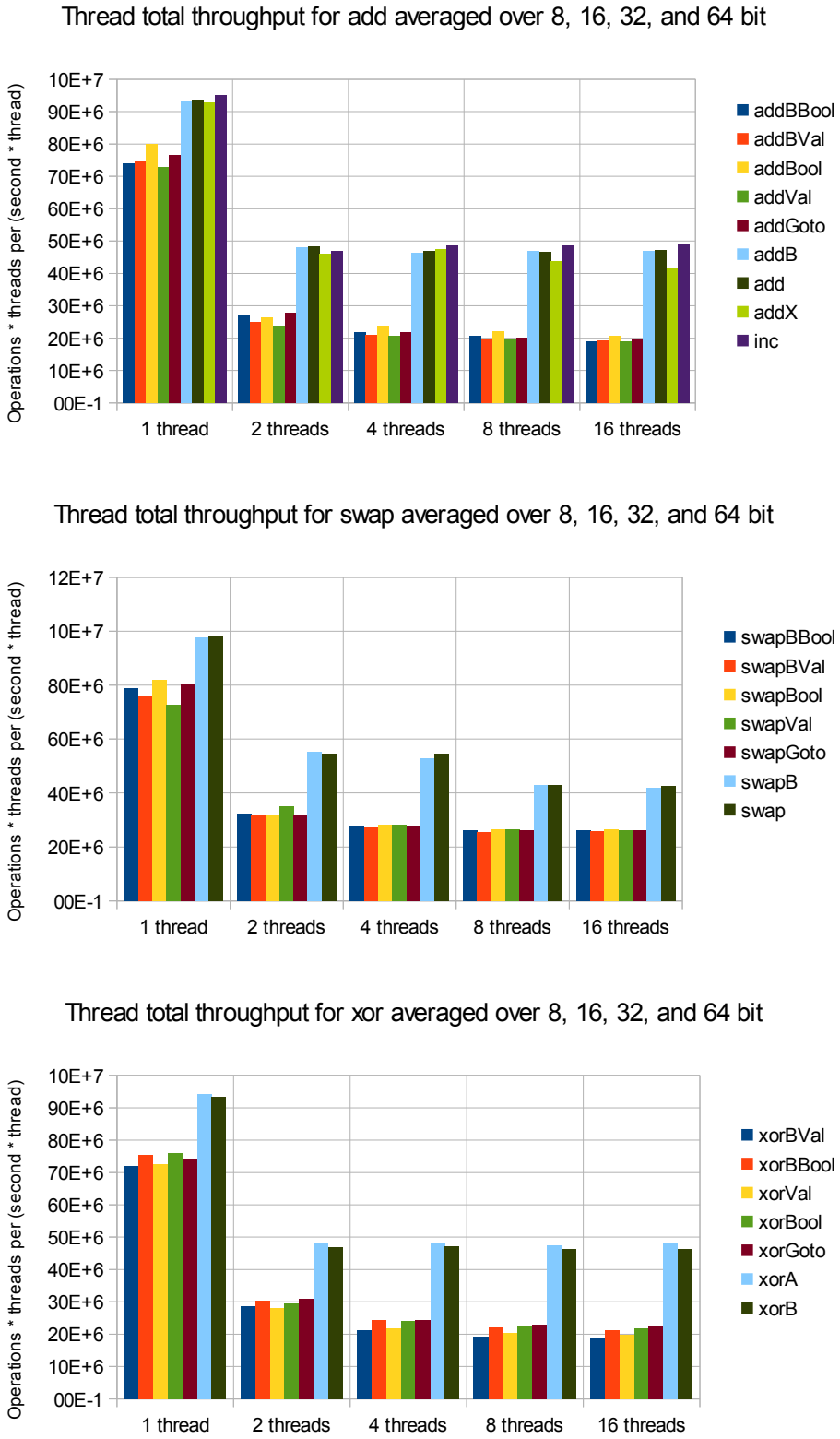
Table 5:A description of the platform for running the tests

Ending	**Interface**
BVal	`__sync_val_compare_and_swap`
Val	`casVal`
BBool	`__sync_bool_compare_and_swap`
Bool	`casBool`
Goto	`casGoto`
A	Inline assembly to read-modify-write version of operation
B	Builtin to __sync_`fetch_and_` version of operation

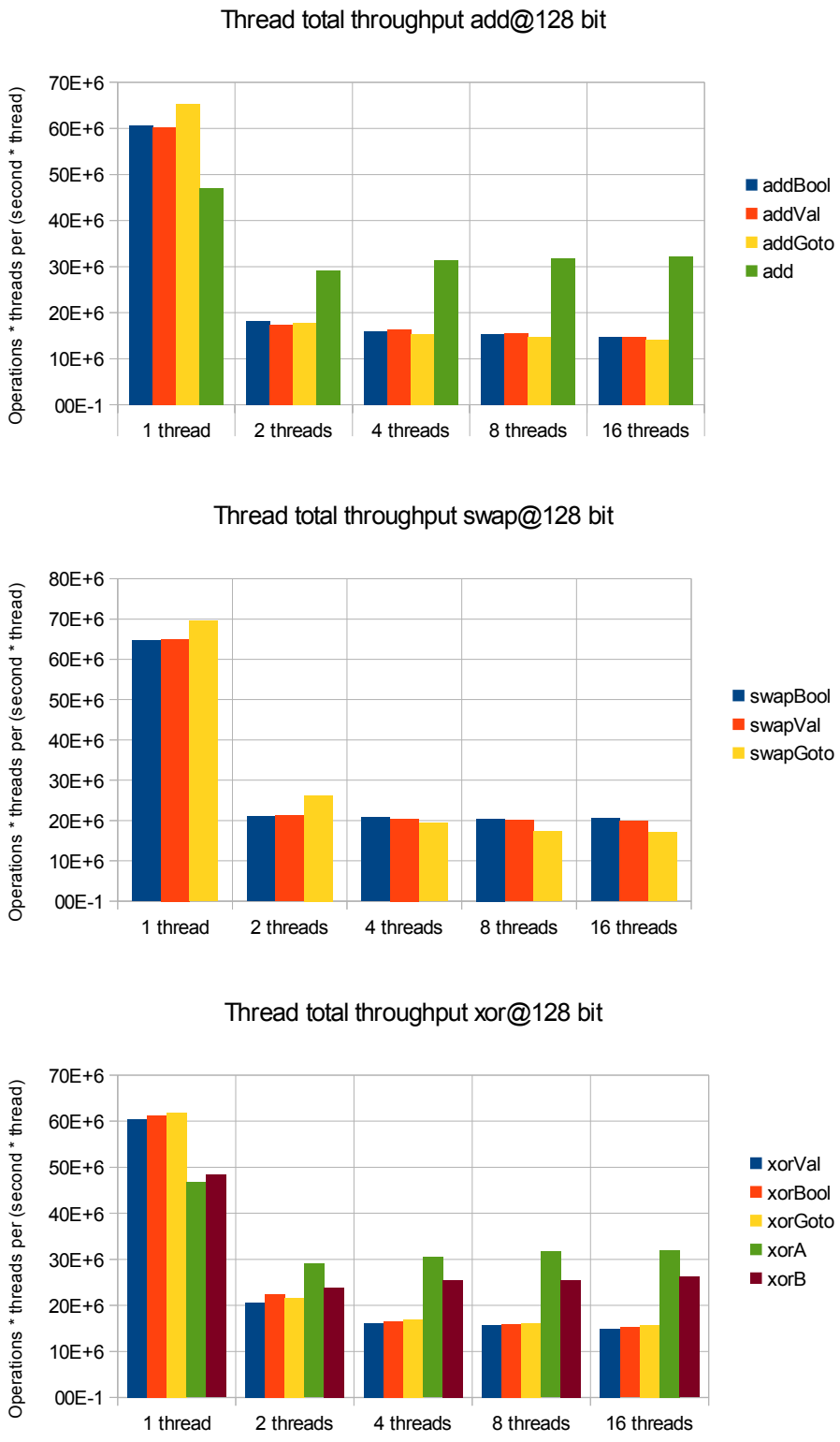
Table 6:Shorthand names for atomic instruction interface



Graph 1: Throughput for the various interfaces in the single threaded case



Graph 2: Thread total throughput for the various interfaces



Graph 3: Thread total throughput for the interfaces to 128 bit operations

The following 6 observations are immediately obvious from the graphs:

1. In graph 2 and 3 we see that the thread total throughput drops, when increasing the number of threads. The throughput of a system with 16 threads, is less than half that of a single thread in all cases
2. In graph 2 the direct read-modify-write versions of the operations are faster than the CAS based operations.
3. In graph 3 the CAS based operations are faster than the direct read-modify-write operations, in the single threaded case, and slower at high contention.
4. In graph 1 and 2 the locked `xchg`, `xor`, `xadd`, `add`, and `inc` instructions have very similar throughput, when under the same level of contention.
5. In graph 1 and 2 the various interfaces to CAS have very similar performance.
6. In graph 3 the casGoto interface consistently outperforms the other interfaces in the single threaded case. There is no significant difference in the other cases.

The first observation basically shows that there is significant overhead to contention for the test cases. The observation is in no way surprising, and it illustrates why backoff schemes are useful for contended resources

The second and third observations show characteristics about the locked instructions. CAS based updates tend to be slower than more direct instructions, because CAS is more complex, and it because the updates can fail. In graph 3 the updates can be performed by a single CAS instruction, while it takes two of the direct instructions, explaining why CAS is faster in the single threaded case. At higher contention levels of graph 3 the direct interfaces are faster, because their updates always succeeds, unlike the CAS instruction.

The forth and fifth observations indicate that ensuring atomicity of the operations, is more expensive then actually performing them, for these tests.

5.4.3.3 Generated code

This section reviews the quality of the assembly code generated for the update loops, of the different CAS based operations, when using the different interfaces.

The full update loops can be seen in appendix 10.1Read-modify-write update loops. We evaluate the quality of the code based on:

- **Min instructions**: The number of instructions executed per update, if the operation succeeds.
- **Retry instructions**: The number of instructions executed each time an update fails.
- **Total size**: The total size of the code in bytes.

The quality is evaluated individually for 8, 16, 32, 64, and 128 bit operations.

Table 7 through 9 show the characteristics of the generated code, for the different implemented operations.

Interface	**Min instructions**					**Retry instructions**					**Total size**				
	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128
Val	8	7	7	7	12	5	5	5	5	9	25	25	21	25	44
BVal	7	7	7	7		5	5	5	5		29	31	26	31	
Bool	7	6	6	6	12	4	4	4	4	8	23	23	20	21	40
BBool	6	6	5	5		4	4	3	3		19	20	14	17	
Goto	5	4	4	4	8	2	2	2	2	7	16	18	13	14	35

Table 7:Performance metrics of CAS based swap operations

Interface	**Min instructions**					**Retry instructions**					**Total size**				
	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128
Val	9	8	8	8	19	6	6	6	6	16	28	28	24	29	66
BVal	10	10	10	10		7	7	7	7		36	38	29	35	
Bool	8	7	7	7	18	5	5	5	5	14	24	24	21	23	56
BBool	8	8	6	6		6	6	4	4		24	25	17	21	
Goto	6	5	5	5	14	3	3	3	3	13*	19	19	16	18	53

Table 8:Performance metrics of CAS based add operations

Interface	**Min instructions**					**Retry instructions**					**Total size**				
	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128
Val	11	9	9	9	17	8	7	7	7	14	31	29	25	31	59
BVal	11	11	11	11		8	8	8	8		37	39	30	37	
Bool	9	8	8	8	16	6	6	6	6	12	25	25	22	25	50
BBool	8	7	7	7		6	5	5	5		24	25	18	23	
Goto	7	6	6	6	12	4	4	4	4	11*	20	20	17	20	48

Table 9:Performance metrics of CAS based xor operations

The code generated by all of the interfaces in all of the test cases, except the 128 bit versions of `casGoto`, store the update loops in one continuous block. The 128 bit versions of `casGoto` store some of the retry instructions slightly after the update loop.

The 32 bit operations appear to be very space conserving. In fact all interfaces produced the smallest update loops for all of the test cases, when they used 32 bit operands. The update loops for the 128 bit operations are by far the largest. This is because the 128 bit values require two registers, and it generated code for the update loops with a lot of redundant `mov` instructions. The problem is most pronounced for the `add` operations. For instance the 128 bit `add` update loop using `casGoto`, has 6 redundant `mov` instructions. By comparison the 128 bit `xchg` loop for `casGoto` only has 2 redundant `mov` instructions. The poor results indicate that GCC is less efficient at optimizing the 128 bit types. This is most likely because the extensions see far less use than the primitive types

of C++, so optimizing operations on such variables is not a high priority. Additionally GCC might not be able to satisfy the assembly constraints for 128 bit values as efficiently.

The `casGoto` interface produces the best results in all the test cases, in the sense that it uses the fewest and smallest instructions. None of the interfaces introduce any expensive instructions, aside from the locked instructions.

Table 10 shows the characteristics of the interfaces to atomic instructions, that directly implement their operation.

Interface	**Instructions**					**Total size**				
	8	16	32	64	128	8	16	32	64	128
xorB	3	3	3	3	5	12	12	11	11	19
xorA	3	2	2	2	5	10	9	7	8	22
addA	3	2	2	2	5	10	9	7	8	22
xaddB	3	3	3	3		12	14	11	11	
xaddA	3	2	2	2		12	10	8	9	
incA	2	1	1	1		6	4	3	4	
swapA	3	2	2	2		11	9	7	8	
swapB	3	3	3	3		11	12	10	10	

Table 10:Performance metrics of operations implemented without loops

In general the inline assembly blocks tend to produce better code than the compiler builtins. The builtins do however have some significant advantages, that are not visible from the above table. For one the compiler may be able to use the condition codes stored after the locked instruction. Another advantage is that the compiler may chose to pick simpler instructions, if they provide the same functionality. For instance the `__sync_fetch_and_add` compiled to a `LOCK ADD` instruction, instead of a `LOCK XADD` instruction, because the test case does not use the output from the builtin. Both advantages make it easier to write efficient code using the builtins, but it seems that better results can be achieved with inline assembly. Another thing worth noticing is that the `incA` code is smaller than the `addA` code. Furthermore `addA` in turn is smaller than the `xaddA` code.

5.4.4 Summary

This section described issues related to ordering constraints, and low level synchronization primitives. It also empirically tested the performance and code quality of a number of interfaces to read-modify-write instructions. The findings from the tests can be summed up as:

- The the systems throughput decreases, when increasing the number of threads actively modifying a field.
- CAS is generally the slowest read-modify-write instruction, especially at high contention.
- There is no significant performance difference between fetchAndAdd, `ADD` and `INC` instructions. However `INC` instructions are shorter than `ADD`, and `ADD` are shorter than fetchAndAdd.
- The performance of all the read-modify-write instructions, is practically independent of whether they operate on 8, 16, 32, and 64 bit integers. The size of the instructions tend to be lowest for the 32 bit operations.
- 128 bit CAS operations are slower than their 64 bit counterparts, but faster than two 64 bit operations at low contention levels.
- The most efficient interface to the read-modify-write instructions can be derived with gcc's extended asm blocks, or asm goto blocks, but compiler builtins may similar results.

From a performance view the best practice is:

- Try to reduce contention.
- Use GCCs extended asm blocks or asm goto blocks to interface to the instructions.
- Prefer to use any other operation over CAS, whenever possible without introducing new overhead.
- Prefer `DEC`/`INC` over `ADD`/`SUB`, and `ADD`/`SUB` over fetchAndAdd, whenever possible without introducing new overhead.

Based on the results, we have decided to use an interface based on asm goto blocks when the code branches based on whether or not CAS operations succeed. For all other uses of CAS, we used an interface based on inline assembly. The interfaces for the implementations in the rest of the thesis are similar to `casGoto` and `casVal`, as described in section 5.4.2.1 The interfaces.

5.5 Truncated exponential backoff

The basic design of truncated exponential backoff is covered in section 4.2.5 Backoff schemes on page 18. Unfortunately truncated exponential backoff has several often overlooked implementation details. The following sections describe those details, the variations of exponential backoff that have been implemented, and ends with an evaluation of those variations.

5.5.1 Implementing truncated exponential backoff

Exponential backoff can be applied in different ways, by changing when threads backoff, and the conditions for when the delay duration may increase or decrease. Listing 16 shows how we represent these actions in code form, and listing 17 shows an example of applying truncated exponential backoff:

```
class Backoff {
  mask;

  void spin() {
    if(mask != 0) {
      spinTime = (rand() & mask) * slotSize;
      delayFor(spinTime);
    }
  }

  void success() {
    mask = mask / 2;
  }

  void failure() {
    mask = (mask * 2 + 1) & largestMask;
  }
};
```

Listing 16:Basic interface used to apply truncated exponential backoff

```
while(1) {
  b.spin();
  if(tryOperation()) {
    b.success();
    return;
  }
  b.failure();
}
```

Listing 17:A possible application of truncated exponential backoff, to an operation that can fail. The backoff scheme is used through a `Backoff` object `b`,

In listing 16 `mask` is a variable that defines the upper bound for the number of slots that the current thread can delay for. `largestMask` is an upper bound for `mask`, that must satisfy $largestMask = 2^n - 1, n \in \mathbb{N}$, $largestMask \propto p$, where p is the number of CPUs.

In order to back off for an appropriate amount of time, each thread should store a `Backoff` object, for each object with a different level of contention.

A problem with truncated exponential backoff, is that it requires tuning `slotSize` and `largestMask`, in order to get good results. Unfortunately different systems may require different settings to get good performance. `slotSize` needs to be tuned such that the smallest backoff period is sufficient to provide some impact on the level of contention, without backing off too much. `largestMask` needs to be large enough to handle the highest levels of contention, but not so high that changing levels of contention, or overestimates of contention, harm the throughput. Another problem with truncated exponential backoff is that just adding the code for backing off, might harm performance, especially if the operation being attempted is fairly simple. This issue can however be reduced by ensuring that the compiler optimizes the code for the case where no backoff is necessary.

The following paragraphs describe ways meet the requirements of `largestMask`, and implement ways to implement the `delayFor` spin loop.

The constraints for `largestMask` can be satisfied by setting:

$largestMask = 2^{round(k + \log_2 p)} - 1$, or $largestMask = nearestPowerOfTwo(k \cdot p) - 1$

Where p is the number of CPUs, and k is a constant for tuning the variable.

Calculating $\lfloor \log_2 \rfloor$ of positive integers is equivalent to finding the most significant set bit. This is in turn equivalent to counting the number of leading zeros in the integer, and subtracting it from the bit length of the integer. Rounding to the nearest power of two, and counting the number of leading zeros, can be implemented fairly efficiently with regular code [Anderson09]. Such functions are also available as builtins for some compilers. The implementations covered in this section use the GCC compiler builtins for counting the number of leading zeroes. Using the builtin function has certain advantages. The result can be calculated at compile-time for constants, and for variables it ends up compiling to a single AMD64's bitScanReverse (`BSR`) instruction. `BSR` returns the most significant bit set in a register, and sets the zero flag, if the register is 0.

Delaying for a number of slots can be implemented with a simple spinning loop using a counter, assuming the loop is not optimized away. Using a simple spinning loop is fairly easy, and it is possible to have loops that delay for very short periods of time. A disadvantage of such a loop is that it might produce wildly different spinning times, because the spinning thread may be preempted while spinning, causing extra delay. Instead of delaying with a regular counter, one can use architecture specific performance counters, to tell how long the thread has waited for, and spin until the loop has waited for sufficiently long time. AMD64 has such a time-stamp counter, that can be read with the `RDTSC` instruction.

While a thread is delaying, it could also execute the `PAUSE` instruction. The `PAUSE` instruction, was introduced to reduce power consumption of threads in spin loops, by having the CPU wait for number of cycles. While the CPU waits it may reduce power consumption, and possibly free resources for CPUs that are physically close to it.

5.5.2 Modifications to truncated exponential backoff

Truncated exponential backoff is fairly simple, and tends to provide decent protection from contention. However at very high levels of contention, even the best truncated exponential backoff schemes tend to degrade in performance. The following sections look at various modifications that might improve the performance at high contention levels.

One of causes of the performance degradation at high contention levels, is that threads might have a fairly high success ratio, even at high contention levels. As long as threads have a success ratio of 50 % or less, they will not increase their backoff duration.

For some data structures, one can load the variable, that is about to be changed, before the backoff duration. This is basically applying backoff in a way that gives a more pessimistic view of the level of contention. In addition, such a scheme would give the threads with longer backoff periods a lower probability of succeeding. Therefore it would favor keeping the backoff period short for threads that already have short backoff periods. Doing so will lead to fever cache line invalidations, due to less false sharing, and it might also lead to less time spent where all threads are backing off.

Another possibility is to use an aging variable to decide whether or not to decrease the mask variable when succeeding, as shown in listing 18. Using an aging variable, evens out fluctuations of contention, and makes it easier to bias towards more or less pessimistic views of the contention.

```
class Backoff {
  mask, failureRate;

  void spin() {
    if(mask != 0) {
      spinTime = (rand() & mask) * slotSize;
      delayFor(spinTime);
    }
  }

  void success() {
    failureRate = failureRate * 0.75;
    if(failureRate < 0.28) {
      mask = mask / 2;
    }
  }

  void failure() {
    failureRate = failureRate * 0.75 + 0.25;
    mask = (mask * 2 + 1) & largestMask;
  }
};
```

Listing 18:The modified Backoff data structure, that uses an aging variable

The aging of the variable `failureRate` can, and should, be implemented with integer operations.

5.5.3 Evaluation of truncated exponential backoff

This section evaluates the performance of the traditional and our modified truncated exponential backoff implementations, when applied to a shared CAS based counter. All of the backoff schemes use k = 6 and a slot size of 256 cycles. We applied each of the backoff schemes in two ways, as seen in listing 19. The first application loads the counter before backing off. The second application backs off before loading the counter. As a blind test we also tested the performance of a CAS based counter without any backoff scheme. CAS based counters were chosen as the basic test case, because the operations are fast, leading to high contention. The test case gives the backoff schemes optimal conditions to make an impact on performance.

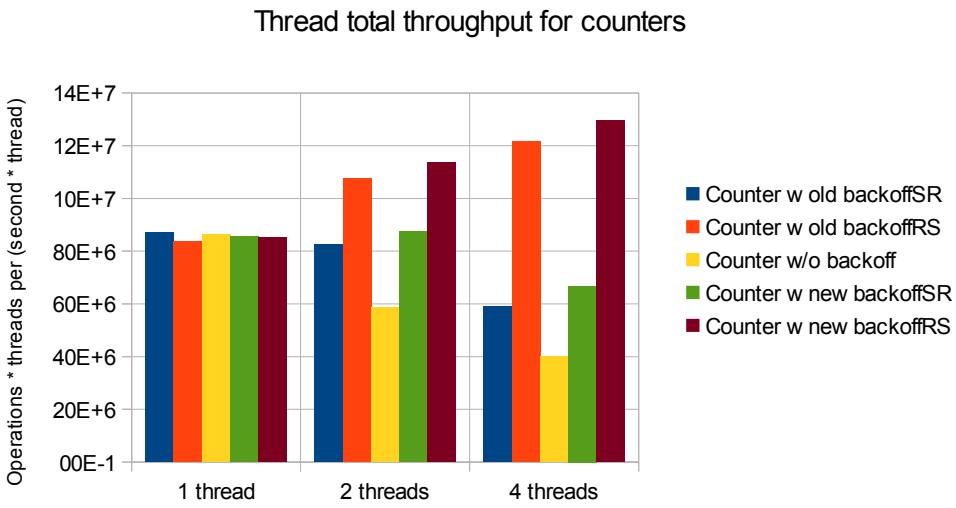
```
counter

backoffSR() {
  while(1) {
    b.spin()
    old = counter
    if(CAS(&counter, old, old + id)) {
      b.success()
      break
    }
    b.failure()
  }
}

backoffRS() {
  old = counter
  while(1) {
    b.spin()
    if(CAS(&counter, &old, old + id)) {
      b.success()
      break
    }
    b.failure()
  }
}
```

Listing 19:Application of backoff on a counter, that either spins before reading (SR), or reads before spinning (RS)

Each test was performed using up to 16 threads, where each was bound to a specific CPU. In the tests each thread incremented the counter 3.000.000 times. The tests started all the threads simultaneously, and each thread is measured the wall-clock time when they start and finished their tests. Each of the presented results are based on 160 measurements. To get 160 measurements, when testing with p threads, the tests is run 160 / p times.

System name	Acer Aspire 4820TG
Ram	4 gb
OS	Windows 7 64 bit
Processors	Intel Core 5 M450@2.4GHz (4 CPUs)
Compiler	GCC (TDM-64)
Compiler flags	`-m64 -std=gnu++0x -fopenmp -DNDEBUG -fopenmp -Ofast -fwhole-program -static -flto -fno-align-functions -fno-align-labels -fno-align-loops -fno-align-jumps -s`

Table 11:A description of the platform for running the tests



Graph 4: Thread total throughput with different backoff schemes

Graph 4 shows the results of running the tests on the system described in table 11. The following 4 observations are immediately obvious from the results:

1. Not using any backoff scheme when accessing a contended resource, yields a much lower throughput, than using a backoff scheme.
2. The RS solutions are up to twice as fast as the corresponding SR solutions.
3. The RS solutions have decreasing average latency.
4. Using the modified backoff scheme provides up 15 % higher throughput, than the original.

The first observation should not come as a surprise. It shows that having the threads back off, if the resource is contended, can reduce the average latency of operations.

The size of the gap between the average latency of the RS and SR solutions, is somewhat surprising. As previously mentioned, the RS applications have a more pessimistic view at the contention. RS applications also and tend to favor short backoff periods for threads that already have short backoff periods. Doing so leads to lower average latency for two reasons, the actual contention is reduced more aggressively, and some of the threads might finish their operations earlier than others. The observation that the RS solutions have decreasing average latencies can also largely be explained by some of the threads finishing their operations significantly earlier.

The last observations shows, as we argued, that using an aging variable, to decide when to reduce the backoff duration, can lead to lower contention.

5.6 MCS locks

This section describes the locks used to compare performance of lock based and lock-free data structures, throughout the rest of this thesis. The locks are basically MCS spin locks, that use the modified truncated exponential backoff scheme, when acquiring locks. In this section describe how MCS locks work, our modifications, and the performance impact of using the different kinds of MCS locks.

5.6.1 Design

MCS locks support acquiring and releasing locks, and they guarantee a queue like first-in-first-out (FIFO) ordering. This ordering is achieved by storing the order of pending acquires as a link based queue. Each element in the queue corresponds to a spinning thread, waiting to acquire the lock. Each spinning thread only needs to look at the element in front of it in the queue, reducing the locks contention. Illustration 3 shows an example of a system with three MCS locks, and four threads.

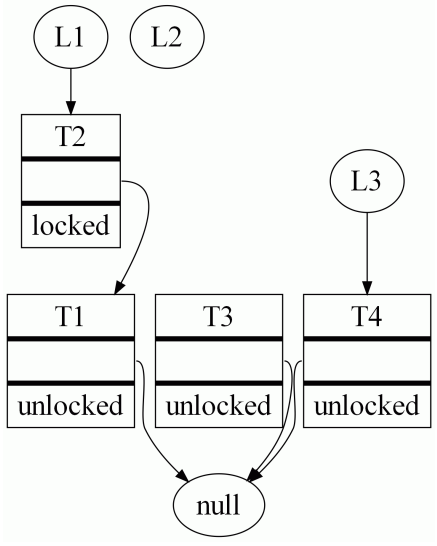


Illustration 3:3 locks L1, L2, L3, and 4 threads T1, T2, T3, T4.
T4 has acquired L4. T1 has acquired L1, and T2 is waiting for T1 to release L1

The memory overhead of storing a queue is sometimes used as a reason for avoiding MCS locks. However, each thread can only wait on a single lock at a time, so the memory overhead for using n MCS locks in a p threaded program is $O(n+p)$.

5.6.2 Implementation

The MCS can be implemented like as shown in listing 20. In the listing each thread stores their own `QNode locksState`, that is inserted into the queue when acquiring, and removed from the queue upon release.

```
class MCSLock {
public:
  class QNode {
    QNode* volatile next;
    volatile bool locked;
  };

  QNode* volatile lock = nullptr;

  void acquire(QNode* lockState) {
    lockState->next = nullptr;
    QNode* prev = fetchAndStore(&lock, lockState);
    if(prev != nullptr) {
      lockState->locked = true;
      prev->next = lockState;
      while(lockState->locked);
    }
  }

  void release(QNode* lockState) {
    if(lockState->next == nullptr) {
      if(compareAndSwap(&lock, lockState, nullptr)) {
        return;
      }
      while(lockState->next == nullptr);
    }
    lockState->next->locked = false;
  }
};
```

Listing 20:A traditional MCS lock implementation

We propose using a backoff scheme to reduce contention on the lock, when acquiring the lock. Specifically we propose using the modified exponential backoff scheme described in section 5.5.2. The backoff scheme should be applied in a way where threads spin before acquiring the lock. Threads increase the spin duration when they do not immediately get exclusive access, and they may decrease it when they do get immediate exclusive access, as seen in listing 21. We advice against using a backoff scheme to reduce contention when releasing the lock, because the lock should be released as quickly as possible. Using the backoff scheme increases the amortized memory consumption to: $O(n+p\cdot c)$, where n is the number of threads, p the number of threads, and c is the number of backoff objects each thread has to maintain.

```
  void acquire(QNode* lockState, Backoff* b) {
    lockState->next = nullptr;
    b->spin();
    QNode* prev = fetchAndStore(&lock, lockState);
    if(prev != nullptr) {
      lockState->locked = true;
      prev->next = lockState;
      b.failure();
      while(lockState->locked);
    } else {
      b.success();
    }
  }
```

Listing 21:Acquiring MCS locks with a backoff scheme

5.6.3 Evaluation of MCS locks

This section evaluates the performance of the traditional and our modified MCS lock implementations, when applied to a shared counter. The tests lock the counter, increment it, and release the lock, as seen in listing 22. As a blind test we also tested the performance of the corresponding OpenMP locks. A counter was chosen as the basic test case, because the operations are fast, leading to high contention. By locking a counter, we get to see how big the overhead for locking is, when under extreme contention.

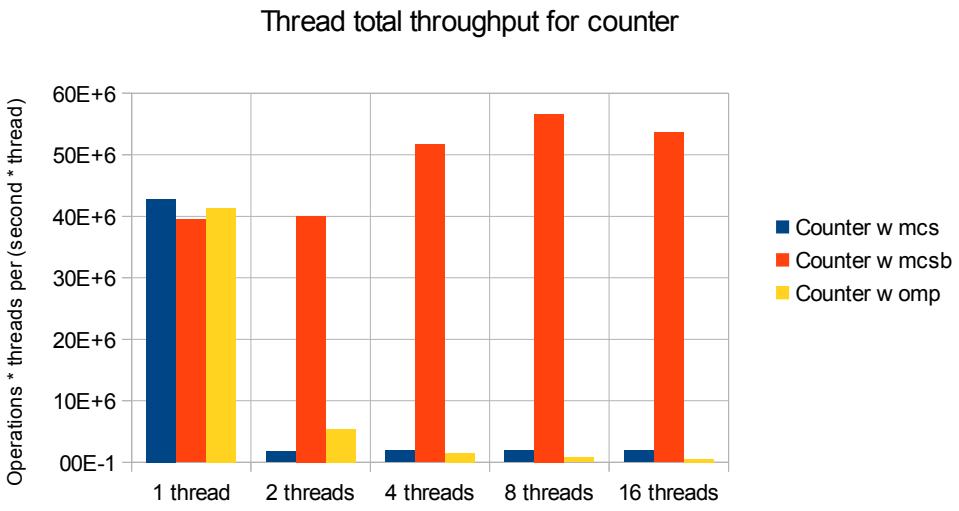
```
counter
counterLock

lockTest() {
  counterLock.acquire()
  counter = counter + id
  counterLock.release()
}
```

Listing 22:The test run to evaluate performance of locks

Each test was performed using up to 16 threads, where each was bound to a specific CPU. In the tests each thread incremented the counter 3.000.000 times. The tests started all the threads simultaneously, and each thread is measured the wall-clock time when they start and finished their tests. The each of the presented results are based on 160 measurements. To get 160 measurements, when testing with p threads, the tests is run 160 / p times. Based on the measurements we present the average latency per operation.

System name	HP ProLiant SL165z G7 server
Ram	64 gb
OS	Scientific Linux 6.1
Processors	2 x AMD Opteron 6168 (24 CPUs)
Compiler	GCC (Ubuntu/Linaro 4.6.1-9ubuntu3)
Compiler flags	`-m64 -std=gnu++0x -fopenmp -DNDEBUG -fopenmp -Ofast -fwhole-program -static -flto -fno-align-functions -fno-align-labels -fno-align-loops -fno-align-jumps -s`

Table 12:A description of the platform for running the tests



Graph 5: Thread total throughput of a shared counter, protected by different locks. mcs are user space MCS locks. mcsb are user space MCS locks with the modified backoff scheme. omp are the default OpenMP locks for the tested system.

Graph 5 shows the results of running the tests on the system described in table 12.

In the test case with 16 threads, the threads finish 27 times faster on average, by using the backoff scheme when locking MCS locks. In the same case the MCS locks without backoff, are on average 2.5 times faster than the default OpenMP locks. The following 2 observations are immediately obvious from the results:

1. The MCS locks with the new backoff scheme are much faster in the presence of contention.
2. The MCS locks with the new backoff scheme increase the thread total throughput, when increasing the number of threads.
3. The MCS locks without backoff, are slower than the OpenMP locks at low levels of contention, and faster at high levels.

The first observation shows that using the proposed backoff scheme when acquiring MCS locks dramatically reduces contention for locking. Part of the reason why the backoff scheme produces such good results, is that has been tuned to reduce contention for single atomic operations. This makes it ideal for reducing contention on locks with very short lock durations.

The second observation can be explained by the fact that we are measuring thread total throughput, and not system throughput. The increased thread total throughput, is due to some of the threads finishing the test much earlier than other threads. This phenomenon is largely due to the use of a pessimistic backoff scheme.

All of the observations also highlight the importance of selecting the right lock for the task at hand.

5.7 Summary

In this chapter we have presented the basic building blocks used to implement the data structures presented in the remainder of this thesis. Specifically we presented efficient ways to access hardware synchronization primitives, ways to reduce context switches, and ways to reduce the impact of contention.

We found that using linear congruential generators with specific parameters, can guarantee that each thread has a unique access pattern. We found that GCC inline assembly and goto assembly blocks provide the most efficient interface to synchronization primitives, but compiler builtins are almost as fast. We presented a modification to truncated exponential backoff, leading to a 15 % higher throughput. We presented a lock based on MCS locks, that use the modified backoff scheme to reduce the overhead of locking contended resources. The modified locking is highly promising, showing throughput improvements of up to 2700 % for highly contended data structures.

6 Static search structure based priority queues

6.1 Introduction

This chapter covers the research we have done into how one can create a lock-free quantatized priority queue. We have made the groundwork to create a solution that is related to the FunnelTree data structure described in “Combining Funnels: A Dynamic Approach to Software Combining” [SZ00]. The FunnelTree uses concurrent stacks and counters, with fairly complex locking schemes. As an alternative we have also implemented stacks and counters that are similar to the data structure described in “A Scalable Lock-free Stack Algorithm”, truncated exponential backoff, and a combination of the two concepts.

This chapter starts with a description of the concepts we have inherited from, before moving on to the specifics of the various concurrent stacks and counters. Throughout the chapter we evaluate the performance of the various data structures for current AMD64 hardware.

6.2 A static tree structure for priority queues

The paper “Scalable Concurrent Priority Queue Algorithms”[SZ99] proposed using a binary search tree with a static structure, to store a priority queue, as shown in illustration 4. The search tree has counters at each internal node, and containers for the prioritized objects at each leaf node. The counter of a node indicate the number of objects stored in the left sub tree of the node. The leftmost leafs are the represents the objects that should be extracted first.

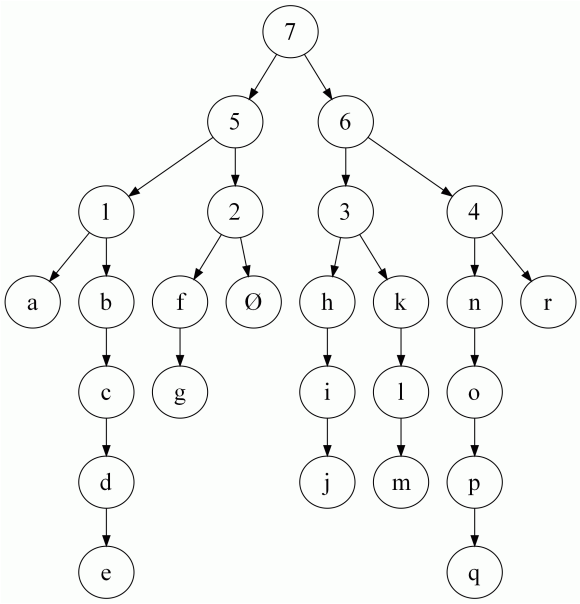


Illustration 4:An example of a tree structure supporting 8 priorities. Nodes with numbers are internal nodes, and nodes with letters represent the contents of leaf nodes. Ø is the empty stack.

```
Internal* root;
int height;

void insert(Value value, Key key) {
  Node* n = getLeaf(key);
  getLeaf(key)->push(value);
  for(int h = 0; h < height; h++) {
    Internal* p = n->getParent();
    if(n == p->getLeftChild()) {
      p->increment();
    }
    n = p;
  }
}

Value extract() {
  Internal* n = root;
  for(int h = height; h > 0; h--) {
    uint i = n->boundedFetchAndDecrement();//Never decrements below 0
    n = (i > 0) ? n->getLeftChild() : n->getRightChild();
  }
  return ((Leaf*)n)->pop();
}
```

Listing 23:Operations on a static tree structured priority queue

Inserts proceed from the leaf node representing the stack to insert into, and traverse up the tree to the root. Whenever it goes up from a left child it increments the parents counter by one, since the insertion into the leaf node added one object to the parents left sub tree.

Extractions proceed from root to the leaf it extracts from. To decide the child to descend into, it attempts to decrement the counter of the parent by 1, without the counter decreasing below 0. If it was possible it descends into the left child, otherwise it descends into the rightmost child. Upon reaching the leaf it pops an element and returns.

The updating of the counters in the internal nodes is used to keep track of the number of elements with a given priority. Insertions proceed bottom up to ensure that the count is never incremented before an object is inserted to the leaf. Extractions proceed top down to ensure that the count is always decremented before an object is removed. Doing so ensures that the counters are always equal to, or less than, the number of objects in the leafs of the left sub tree.

The nodes can be stored in an array, where node node at index i has the $i \cdot 2$, $i \cdot 2+1$ and $i/2$, as left child, right child, and parent respectively. Such kind of storage is also often used for binary heaps, and avoids storing references to the other nodes.

To reduce contention for the stacks and counters, they are implemented with combining funnels. Combining funnels allow operations to be combined, if they are similar, or eliminated if they are opposite. For instance two push operations can be combined into one push operation that push two values, and a push and pop operation can be eliminated so that the pusher sends the value being pushed to the popper.

During elimination of `increment` and `boundedFetchAndDecrement` for this data structure, it is not strictly necessarily have to ensure linearizability. Specifically it is not required that `decrementing` threads receive the correct values. If a `decrement` operation eliminates with an `increment` operation, then it has prevented an `increment` operation, implying that there is at least one object in the left sub tree.

The paper "Scalable Concurrent Priority Queue Algorithms"[SZ99] found that implementing a priority queue with combining funnels for the counters and stacks gave better performance MCS lock at high contention. Specifically they tested the performance of combining funnels on a simulated MIT Alewife architecture, with up to 256 processors. They found that when more than 16 processors are consistently working on the priority queue, combining funnels significantly outperform MCS locks.

6.3 Combining funnels

In order to determine if combining funnels are fit for current AMD64 processors, this section explores an their design, implementation, and evaluates their performance. Be aware that combining funnels are lock based. The implementation is primarily made for the purpose of determining if the basic structure of combining funnels, can be applied to a highly concurrent priority queues, for current hardware.

6.3.1 Design of combining funnels

Combining funnels allow operations to be combined or eliminated. After eliminating operations they exchange data and return without modifying the data structure behind the combining funnel. If a pair of operations combine, one of the threads assume responsibility for starting the operation. Before starting the operation, the responsible thread will try to combine or eliminate the pair of operations with other another pair of operations. Combined pairs of operations can continue to try to combine or eliminate with other operations. For instance in a system with 64 processors operations can combine into groups of 2, 4, 8, 16, 32, or 64 operations, and groups of 1, 2, 4, 8, 16, and 32 operations can eliminate with similarly sized groups of opposite operations.

Combining funnels can generally be applied to data structures whose operations can be efficiently combined or eliminated. The only applications of combining funnels that we have found described in prior work, are for array based stacks, and possibly bounded counters.

Operations find other operations to combine or eliminate with by attempting to collide with them, as shown in listing 24.

```
width = getWidth(depth)
r = rand() % width
q = swap(&layer[depth][r], id)
if(q == NOBODY) {
  return;
}
if(CAS(&location[id], desc, NOTHING)) {
  if(CAS(&location[q], desc, NOTHING)) {
    // Collided with q
    … decide to combine or eliminate with q
  } else {
    location[id] = desc
    … spin and give other a chance to collide with you
  }
} else {
  // Someone collided or eliminated with this thread
  … wait for the other thread to get access, then do your part
}
```

Listing 24:Pseudo-code for attempting collision

The following is a description of the used variables, and functions:

`depth` is $\log_2$ of the number of operations in this group. `id` is this threads id. `layer` is a per data structure globally visible two-dimensional array, used to find other groups. Groups at layer i describe groups of 2^i combined operations. `location` is an array that maps to a per thread description of the object and depth that it currently operates on. `desc` is a description of the object and depth that it currently operates on. `getWidth(depth)` returns the width of the layer at the given depth multiplied by a factor that represents the threads guess at the amount of contention for the data structure. `random(0, width)` returns a uniformly distributed random number in the range $[0;width[$.

Before attempting to combine, the thread should set its `location` field, to specify its current operation, so other threads can collide with it. In the uncontended case, threads should access the data structure, without affecting the layers of the combining funnel. As the contention estimate increases, threads should attempt to collide more times, and use a larger fraction of the layers for attempting collisions. After each failed collision attempt, threads enter a backoff loop, giving other threads a chance to collide with them. The duration of the backoff loops should be tuned for the given machine. The number of attempted collisions is maintained in a manner similar to truncated exponential backoff.

After a thread has finished attempting to collide, it removes the description of its operation, and attempts to access the underlying data structure, as per listing 25. For counters the operation is performed with a regular fetchAndAdd operation, or CAS operation for the bounded counters. After the operation, the thread sends results to the group. Accessing the stacks is more complicated.

Each stack has a stack pointer, and a counter for queuing up the pending operations, in addition to the stack itself. The queue is maintained through the fields `stack.ticket`, and `stack.serving`, forming a ticket lock [RK79]. Listing 26 shows how threads handle access to the stack, Once a thread has access to the stack it does the following:

1. Inform its group
2. Wait for its turn in the queue
3. Tell the group to start
4. Perform its part of the operation
5. Wait for the group to finish
6. Let the next group start.

Communication in the group is handled through simulated message passing, where each operation has a responsibility to send messages to the operations that it combined with.

```
location[id] = desc;
while(1) {
  for(uint attempt = 0..attempts[id]) {
    … try to collide
  }
  if(&CAS(location[id], desc, NOTHING)) {
    if(tryOperation()) {
      … possibly decrease attempts[id] and accessed layer width
    } else {
      … possibly increase attempts[id] and accessed layer width
    }
  } else {
    … wait for the other thread to get access, then do your part
  }
}
```

Listing 25:The basic structure of combining funnels

```
if(!tryLock(stack.lock)) {
  … go back to colliding
}
sp = stack.pointer
t = stack.ticket
stack.pointer += operations_in_this_group
stack.ticket++
releaseLock(stack.lock)
… pass a stack pointer to the operation this thread combined with
based on sp
while(stack.serving != t);
go[id] = operations_in_this_group
… perform the stack operation
while(go[id] != 0);
stack.serving++            // let the next group of operations start
```

Listing 26:Getting access to a stack behind a combining funnel

6.3.2 Implementation of combining funnels

There are a some issues to consider when implementing combining funnels. The issues addressed in this section are mainly related to the layout of the data, and the synchronization.

The layout of the data has a significant effect on performance, due to the performance implications of cache misses and false sharing. False sharing is especially problematic, because the threads communicate, by sending messages, and attempting to collide with one another. This communication is performed by writing to memory locations, that are likely to be in other threads caches, causing cache line invalidations. To reduce such issues, the following 4 measures have been taken:

1. All data that is only accessed by one thread, is stored on the stack.
2. All threads have their own data structure used for communication. The data structure is aligned to a cache line.
3. Other concurrent data structures, such as the layers of the funnel, and the data structure behind the combining funnel, are stored on separate cache lines.
4. The layer is stored as a 1 dimensional array. The maximum width of the layer at height i is $w_i = 2^{h-i}$ where h is the height of the funnel. A look up at (i, j) in the 2 dimensional layer, corresponds to a look up at: $2n - w_i + j$.

All AMD64 processors have L1 and L2 data cache line sizes of 64 bytes, according to the AMD64 instruction set [AMD09]. This guarantee makes it fairly easy to ensure that the data is aligned at compile-time.

The synchronization used in the combining funnels is fairly complex, and it may have significant overhead in some cases. To improve on this, one could do the following:

1. Do not have threads publicly declare the operation they want to perform, in cases where contention is low. In such cases the cost of retracting the operation may be significant compared to accessing the underlying data structure.
2. It is possible to avoid checking if collision partners exist. This can be done by having communication fields for a dummy thread, that never performs operations, and initialize all fields in the layers to point to the dummy thread.
3. When combining or eliminating with another group of operations, do so by setting their operation field to a representation of the current group operations. That way the other group might not have to wait as much.
4. Do not use an explicit lock for the stack pointer/ticket counter, use a CAS operation to update them instead.
5. Allow several groups to start their operations on stacks early, if the group ahead of them in the queue is of the also allowed to start, and is performing a similar operation. That is several groups of pushes can operate concurrently, or several groups of pops can operate concurrently, but pushes and pops cannot operate concurrently.

```
attempt = 0
while(1) {
  if(attempt == attempts[id]) {
    if(tryOperation) {
      … possibly decrease attempts[id] and accessed width
    } else {
      … possibly increase attempts[id] and accessed width
    }
  }
  location[id] = desc
  if(attempt != 0) {
    … spin and give others a chance to collide with you
  }
  if(CAS(&location[id], desc, scramble(id))) {
    while(1) {
      width = getWidth(depth)
      r = random(0, width)
      q = swap(&layer[depth][r], id)
      if(CAS(&location[q], desc, scramble(id))) {
        // Collided with q
        … decide to combine or eliminate with q
      } else {
        break
      }
    }
  } else {
    // Someone collided or eliminated with this thread
    … do your part, possibly wait for the other thread
  }
}
```

Listing 27:Combining funnel code structure with the first three improvements

The 5th improvement can be implemented by keeping an additional counter on the stack, that counts the number of similar operations that are enqueued after each other. The counter is updated together with the stack pointer and ticket counter. For instance if a pushing thread can tell that n other groups of pushes are ahead of it in the queue, then it knows that it can start its operation when $stack.serving - ticket - n \leq 0$. This change also means that the serving variable has to be updated with read-modify-write updates, in order to ensure its consistency.

When implementing the spin loop for testing if group has access to the stack, one should be aware that in C and C++ the result of signed integers overflowing is undefined. The spin loop is particularly problematic to express properly after the 5th improvement.

6.3.3 Evaluation of combining funnels

This section evaluates the performance of the combining funnels for stacks and bounded counters. The combining funnels are separated into by whether or not they have synchronization improvement 1. All other improvements are implemented for all of the combining funnels. The tests are performed as per listing 28. Stacks alternate between pushing and popping elements. Counters alternate between incrementing and decrementing. For reference we also tested the performance incrementing and

decrementing a counter using CAS operations, and fetchAndAdd operations. The tests alternate between operations, because it gives optimal conditions for elimination. The combining funnels were run with spin durations ranging from 0 to 2^{20}, to find the best conditions. These optimal conditions were chosen, because we want to determine if the combining funnel mechanism is viable for the tested setup.

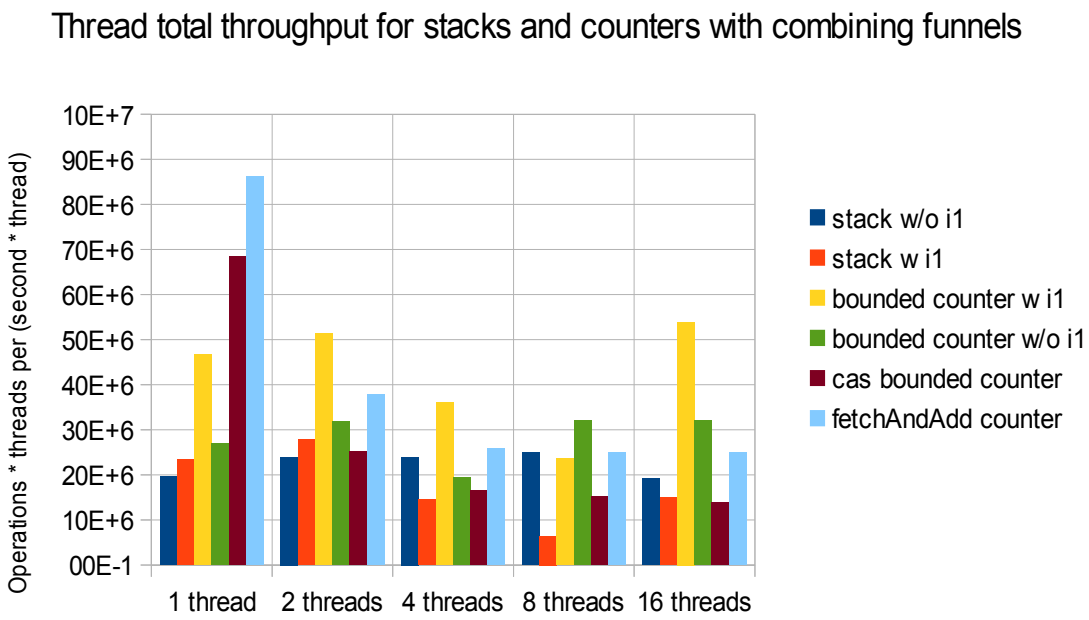
```
counterFunnel
stackFunnel

counter() {
  counterFunnel.inc()
  counterFunnel.dec()
}

stack() {
  stackFunnel.push(val)
  val = stackFunnel.pop()
}
```

Listing 28: The test case performed for the combining funnels.

Each test was performed using up to 16 threads, where each was bound to a specific CPU. Each thread measured the wall-clock time, ran the tested code 300.000 times, and measured the wall-clock time again. The tests started all the threads simultaneously. Each of the presented results are based on 160 measurements. To get 160 measurements, when testing with p threads, the tests are run 160 / p times. Based on the measurements we present the average time that threads spend to complete the test.

System name	HP ProLiant SL165z G7 server
Ram	64 gb
OS	Scientific Linux 6.1
Processors	2 x AMD Opteron 6168 (24 CPUs)
Compiler	GCC (Ubuntu/Linaro 4.6.1-9ubuntu3)
Compiler flags	`-m64 -std=gnu++0x -fopenmp -DNDEBUG -fopenmp -Ofast -fwhole-program -static -flto -fno-align-functions -fno-align-labels -fno-align-loops -fno-align-jumps -s`

Table 13:A description of the platform for running the tests



Graph 6: Evaluation of counters, and stacks using combining funnels

Graph 6 shows the results of running the tests on the system described in table 13. The following 2 observations are immediately obvious from the results:

1. The performance of the same kind of data structure, can vary greatly depending on contention.
2. The use of synchronization optimization 1 improves performance significantly in the single threaded case.

One of the things that are not immediately obvious from the results, is that the spin durations have a large impact on performance. For instance the bounded counter w i1 was 10 times faster when using a delay loop of 2^{15}, then with a delay loop of 0 or 2^{18}.

We never see an increase in the systems throughput when increasing the number of threads for any of the combining funnels. With a lot of tuning of the spin duration, the thread total throughput is nearly constant. Based on these results we conclude that combining funnels do not produce particularly good results on current hardware. The poor results are most likely due to the complexity of the operations.

6.4 Stacks with elimination

The paper “A Scalable Lock-free Stack Algorithm” presented a lock-free stack using elimination to reduce contention and increase throughput [HSY10]. The stack uses similar principles to combining funnels, but it is generally simpler and avoids locking, so such a data structure might be more suitable for current hardware. Additionally the stack is a link based stack, making it more attractive for practical use.

The following sections describe the design, implementation and evaluation of such stacks. Again the implementation is primarily made for the purpose of determining if such a scheme is suitable for current hardware.

6.4.1 Design of stacks with elimination

Operations on the stack alternate between trying to do an operation on the stack, and trying to eliminate with an opposing operation, until either succeeds, as per listing 29.

In general elimination is fairly simple to apply to lock-free data structures, without harming the lock-free property. The only requirement is that both parties in the elimination can determine the operation that was eliminated, without waiting for the other party.

The operations can be made ABA safe by using tags or any kind of memory reclamation scheme, similar to what is described in the section 4.1.3 The ABA problem. Finding elimination partners works in a fashion that is similar to combining funnels, as per listing 30.

```
while(1) {
  if(tryToPerformStackOperation) {
    … return the result if any
  }
  if(tryElimination()) {
    … return the result if any
  }
  … spin for a while
}
```

Listing 29:Basic structures of the stacks

There are three main differences from combining funnels:

1. By not using combining, there is no need for message passing and having a separate layers for each group size.
2. The description of the operations are stored in `ThreadInfo` objects. These object stores a thread id, and the operation, rather than the object and the group size.
3. Threads only try to eliminate with threads performing the opposite operations.

One should be aware that the version of the paper from 2004 has an issue in the description of the elimination, that is fixed in the 2010 version. The descriptions of operations are stored and passed to other threads by reference. If the `ThreadInfo` object can be reused, then a popping operation that eliminates a pushing operation, may read another operation after the elimination. The possible race condition is prevented in the 2010 version by allocating new `ThreadInfo` objects per operation. The problem could also have been solved by storing the description of the operation by value instead.

```
location[id] = desc
r = random(0, width);
q = swap(&collision[r], id);
ThreadInfo* qdesc = location[q]
if(qdest->id == q && qdesc->operationType != desc->operationType) {
  if(CAS(&location[id], &desc, NOTHING)) {
    if(CAS(&location[q], qdesc, id)) {
      // Eliminated with q
      … q's operation is in qdesc
    } else {
      … possibly increase width, try to operate on the stack again
    }
  } else {
    // Someone eliminated with this thread
    … the eliminators operation is in desc
  }
}
… spin for a while and give others a chance for colliding with you
if(!CAS(&location[id], &desc, NOTHING)) {
  // Someone eliminated with this thread
  … the eliminators operation is in desc
}
```

Listing 30:Attempting elimination for stack operations

6.4.2 Implementing stacks with elimination

There are a some issues to consider when implementing stacks with elimination. The issues addressed in this section are mainly related to memory management, the layout of the data, and access patterns of the data structure.

Our implementation uses tags on the stack top pointer, rather than hazard pointers to avoid ABA problems. Nodes are allocated from thread local free lists, and every thread reuses the same `ThreadInfo` objects, to keep memory reclamation simple and fast. To avoid the race condition mentioned in 6.4.1Design of stacks with elimination, the description of the operation is passed by value during elimination.

The layout of the data structure has been optimized to reduce false sharing, in order to deal with high contention efficiently. 4 measures have been taken to reduce false sharing:

1. Every element of the collision and location arrays are aligned to cache lines.
2. Every node for the stack is aligned to cache lines.
3. The pointer to the top of the stack is aligned to a cache line.
4. All data that is only accessed by one thread is stored on the stack.

It might be possible to reduce the contention more significantly by starting out trying to eliminate, instead of starting out trying to access the stack. Another way of reducing contention could be remembering the stack top pointer in between attempted operations. Doing so would reduce the number of loads, at the expense of the chance of successfully updating the stack. Both of these suggestions do not guarantee performance improvements, so their effect is documented in the evaluation section. Depending on the

level of contention, it might also make sense to move the delay loops from the places described in the original paper, to before trying to perform operations on the stack.

6.4.3 Evaluation of stacks with elimination

This section evaluates the performance of the stacks with elimination. The tests are performed as per listing 31. The test code alternates between pushing and popping elements, because it gives optimal conditions for elimination. The stacks were run with spin durations ranging from 0 to 2^{22}, to find the best conditions. These optimal conditions were chosen, because we want to determine if the stacks are viable for the tested setup.

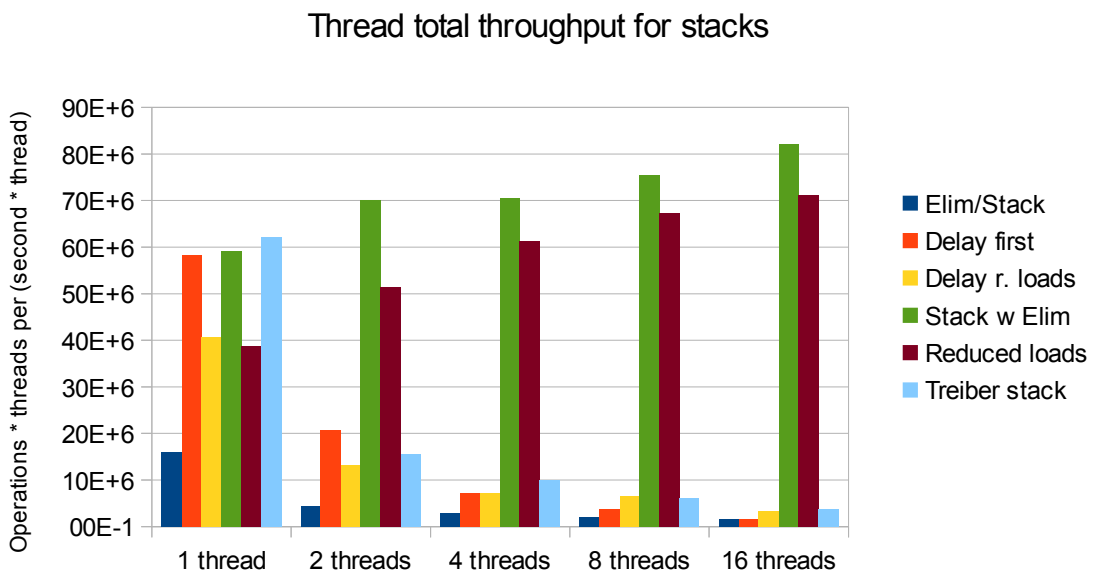
```
stack

test() {
  stack.push(val)
  val = stack.pop()
}
```

Listing 31: The test case performed for the stacks with elimination

Each test was performed using up to 16 threads, where each was bound to a specific CPU. Each thread measured the wall-clock time, ran the tested code 300.000 times, and measured the wall-clock time again. The tests started all the threads simultaneously. Each of the presented results are based on 160 measurements. To get 160 measurements, when testing with p threads, the tests are run 160 / p times.

System name	HP ProLiant SL165z G7 server
Ram	64 gb
OS	Scientific Linux 6.1
Processors	2 x AMD Opteron 6168 (24 CPUs)
Compiler	GCC (Ubuntu/Linaro 4.6.1-9ubuntu3)
Compiler flags	`-m64 -std=gnu++0x -fopenmp -DNDEBUG -fopenmp -Ofast -fwhole-program -static -flto -fno-align-functions -fno-align-labels -fno-align-loops -fno-align-jumps -s`

Table 14:A description of the platform for running the tests



Graph 7: The thread total throughput of the various stack implementations

Graph 7 shows the results of running the tests on the system described in table 14. The following 3 observations are immediately obvious from the results:

1. The fastest stack in most cases, is the one most similar to the original stack with elimination. This particular stack achieved its best result when using a delay loop of 2^{20} iterations.
2. The stack with reduced loads scales the best, but it performs poorly at low contention levels. This stack also achieved its best results when using a delay loop of 2^{20} iterations.
3. The stacks that try to eliminate, or enter a delay loop prior to accessing the stack have very poor performance. This is especially true when increasing the size of the delay loop.

Observations indicate that the placement of the elimination, and backoff loops in the original data structure, give the best performance. The observations also indicate that the most scalable stacks, are the stacks with use the longest delay periods.

The results indicate that the stacks with elimination require very long backoff periods, to get good scalability. In a more realistic setup, it might not be acceptable to have threads spin for a million iterations whenever they fail a CAS operation. In addition, when delaying for such long periods, the threads rarely get an opportunity to eliminate.

6.5 Truncated exponential backoff with elimination

This section presents stack implementations that use a combination of truncated exponential backoff and elimination. Such a scheme should be better at dealing with different levels of contention, and it can give better possibilities for elimination. This section presents changes to the modified truncated exponential backoff scheme from section 5.5.2. The changes allow threads to eliminate, rather than spin, while backing off.

This section primarily focuses on the issues raised by using elimination in a backoff scheme, and efficient implementation of eliminations. The inner workings of exponential backoff are covered in section 5.5Truncated exponential backoff.

6.5.1 Implementing elimination

To perform an elimination, in general, threads need to find an elimination partner, and the two threads must transfer a value between one another. Depending on the operation, the thread will either send or receive a value. The elimination partners always has one sending and one receiving thread.

The following basic structure is used for an attempting elimination:

1. Declare operation.
2. Publish that you are looking to eliminate.
3. Pick a partner to eliminate with.
4. Attempt to retract the operation declaration, if this fails the operation has been eliminated by some other thread.
5. Attempt to exchange operation declaration with the partner, if this succeeds, the operation has be eliminated.
6. Failed to eliminate.

The actual implementation of the elimination is described in listing 32. The threads use a collision array is for finding elimination partners, and an elimination array is used for declaring and transferring operations. In a program with `n` threads, the `n / 2` first elements of the collision array is used to store sending thread ids, and the next `n / 2` elements store the receiving thread ids. Threads attempt to find elimination partners, by writing their id to a random element in the collision array, and reading a partner id from the other part of the collision array.

The elimination array has an element per thread. In the elimination array, threads describe their operation in the least significant bits, and a counter in the most significant bits. The counter is used to ensure that at most one thread can receive a value being sent.

```
n;
collision[n];
elimination[n];

send(id, val) {
  while(isBackingOff()) {
    elimination[id] = val;
    r = rand() % (n/2);
    collision[r] = id;
    partner = collision[r+n/2];
    if(fetchAndStore(&elimination[id], NO_OP) != val)) {
      // Someone took the message
      return true;
    } else if(compareAndSwap(&elimination[partner].ls, RECIEVING_OP,
val | RECIEVED_OP)) {
      // Sent the message to partner
      return true;
    }
  }
  return false;
}

recieve(id) {
  while(isBackingOff()) {
    elimination[id] = RECIEVING_OP;
    r = rand() % (n/2);
    collision[r+n/2] = id;
    partner = collision[r];
    data = fetchAndStore(&elimination[id], NO_OP);
    if(data != RECIEVING_OP) {
      // Recieved a message from someone
      return data & ~RECIEVED_OP;
    }
    data = fetchAndAdd(&elimination[partner], COUNT_MASK);
    if((data & COUNT_MASK) == 0) {
      // Took a message from partner
      return data & ~COUNT_MASK;
    }
  }
  return DID_NOT_RECIEVE;
}
```

Listing 32:Pseudo-code for elimination of sending and receiving threads

This form of elimination has a 3 of advantages:

1. Threads have a higher probability of finding partners with the opposite type of operation. This is achieved because the collision array is separated into elements for sending and receiving threads.
2. The eliminations largely avoids using CAS operations.
3. The eliminations do not read the contents of memory locations before performing read-modify-write operations on them. This reduces the chance of cache misses.

6.5.2 Backoff with elimination

When combining elimination with the modified exponential backoff, there are a couple issues that needs to be addressed.

The first issue is that when a thread successfully eliminates, it can finish an operation before backing off completely. This might cause threads to access the stack earlier than normal, since they can get multiple chances of picking a low delay period.

The second issue is that threads should not read anything from the data structure, when the threads are backing off. Specifically if threads eliminating on the counters read the current value of the counter, it may impact performance.

The first issue is resolved by storing when the thread would be allowed to access the stack, according to the regular backoff scheme. The time that the thread is supposed to back off until, is updated whenever the thread attempts to perform an operation, unless the `mask` of the thread is 0. We resolved the second issue by giving the decrementing operation the value 1, and ignoring the actual value of the counter. Doing so does not violate the correctness of the operations, as explained in section 6.2A static tree structure for priority queues.

6.5.3 Evaluation

This section evaluates the performance of the stacks that with combined backoff and elimination schemes. We applied each of the schemes in two way. The first application, referred to as RS, reads the stack head before backing off. The second application, referred to as SR, backs off before reading the stack head. The tested code alternate between pushing and popping elements, as seen in listing 33. The test was chosen, because it gives optimal conditions for elimination, and we want to determine if elimination stacks are viable for the tested setup. As a blind test we also tested stacks without backoff, and stacks with backoff, but without elimination. The stacks with backoff were tested using both the RS and SR scheme.

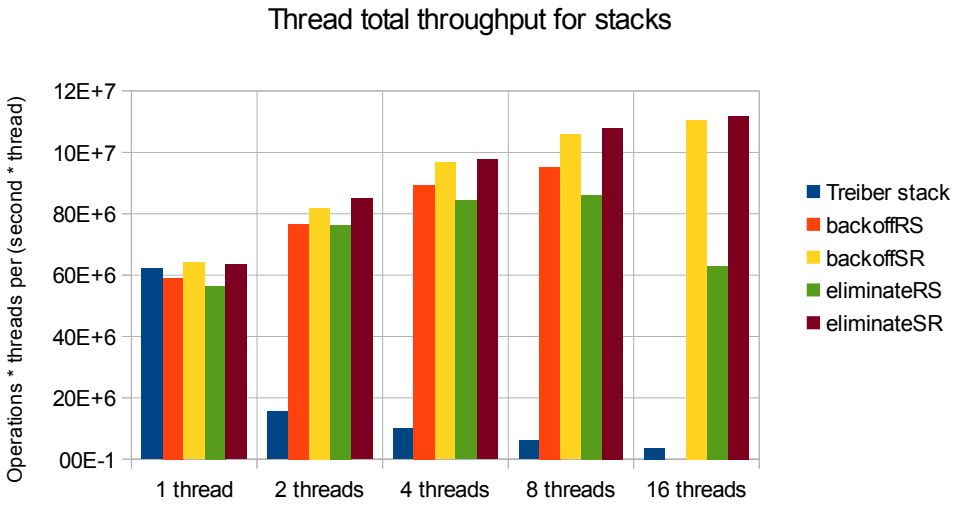
```
stack

testStack() {
  stack.push(val)
  val = stack.pop()
}
```

Listing 33: Test case for evaluating performance of stacks

Each test was performed using up to 16 threads, where each was bound to a specific CPU. Each thread measured the wall-clock time, ran the tested code 300.000 times, and measured the wall-clock time again. The tests started all the threads simultaneously. Each of the presented results are based on 160 measurements. To get 160 measurements, when testing with p threads, the tests are run 160 / p times.

System name	HP ProLiant SL165z G7 server
Ram	64 gb
Processors	2 x AMD Opteron 6168 (24 CPUs)
OS	Scientific Linux 6.1
Compiler	GCC (Ubuntu/Linaro 4.6.1-9ubuntu3)
Compiler flags	`-m64 -std=gnu++0x -fopenmp -DNDEBUG -fopenmp -Ofast -fwhole-program -static -flto -fno-align-functions -fno-align-labels -fno-align-loops -fno-align-jumps -s`

Table 15:A description of the platform for running the tests



Graph 8: Thread total throughput of stacks with elimination and/or backoff. RS refers to backoff schemes where the head of the stack is read before spinning, and SR to reading it after spinning.

Graph 8 shows the results of running the tests on the system described in table 15. The following 4 observations are immediately obvious from the results:

1. Applying the backoff schemes in an SR fashion is more scalable than the RS fashion.
2. Applying a backoff scheme with elimination, in an RS fashion gives very poor results.
3. Eliminating in the SR fashion provides less than a 4 % speedup, when compared to spinning.

4. Applying the backoff schemes in an SR fashion, is up to 36 % faster than the original stacks with elimination.

The first observation is somewhat surprising, given that we found RS schemes to scale better for counters. We are unsure exactly why RS is better for counters, while SR is better for stacks. It may be related to the fact that stack operations involve multiple memory locations, while a counter is a single field. We would also like to highlight that this test almost achieves twice as high thread total throughput for 16 threads, when compared to a single thread.

The poor results from the eliminateRS test, can be explained by the fact that it is not backing off properly. Whenever an operation is eliminated, the its thread will make an additional read of the stack header. The additional reads increases contention, leading to poor results.

The third observation indicates that for the given setup, elimination cannot provide significant improvements to throughput of stacks. The results do not state whether or not it makes sense to apply elimination on other hardware or data structures though. Additionally one might get a larger improvement, by improving the elimination scheme. One possibility that we have not worked with, is trying to bias elimination towards eliminating operations from CPUs that are physically close.

6.6 Conclusion

In this section we have looked at a fairly simple priority queue built from stacks and counter, described in the paper “Scalable Concurrent Priority Queue Algorithms”.

To support high levels of concurrency, we have investigated ways of reducing contention. We reduced contention through backoff schemes, and mechanisms for reducing the number of operations. We found that adding an elimination mechanism to the modified version of the truncated exponential backoff scheme gave the best results. Unfortunately we found that the performance gain from eliminating during truncated exponential backoff, is negligible on current hardware, for the data structures tested.

The poor results achieved when using elimination, may be explained by the fact that elimination is fairly slow, compared to the operations of the data structures tested. Future work in applying elimination mechanisms, could focus on applying it to more expensive operations. Another promising topic, would be optimizing the mechanisms to take take advantage of CPUs physical locality.

7 Investigation of wide search trees

7.1 Overview

This section covers the research into how one can create a lock-free B-tree, and how it can be applied as a priority queue.

Our solution is related to the data structures described in the papers “Non-blocking Binary Search Trees” [EFRB10], “Non-blocking k-ary Search Trees” [BH11] and “Locality-Conscious Lock-Free Linked Lists” [BP11]. In particular it uses a synchronization and helping scheme that is based on the schemes used in the two latter papers.

This chapter starts with a description of the non-blocking k-ary search tree data structure, because it is largely the foundation for our solution. The chapter then moves onto how we applied a similar data structure and synchronization scheme, to a B-tree like data structure. Since the data structure and synchronization is most similar to “Non-blocking k-ary Search Trees”, we will briefly explain how their solution works. Then we will describe how we applied a similar scheme to B-trees.

7.2 Non-blocking k-ary search tree

The non-blocking k-ary search tree, stores values in its leaf node. As a search tree it supports a dictionary interface, where key-value pairs can be inserted and removed. Keys are unique, so there can only be one key-value pair for a given key. The actual data stored, can be seen in listing 34.

```
class Node {
  Key key[k-1];
}
class InternalNode extends Node {
  Status* status;
  Node* nodes[k];
}
class LeafNode extends Node {
  int size
  Value* value[k];
}
```

Listing 34:Java-like pseudo code, defining the data structures layout

Internal nodes have k child pointers and k-1 separator keys satisfying:

$$largestKey(nodes_i) < key_i \wedge (key_i \leq key_{i+1} \vee i > k-1)$$

Leaf nodes may have up to k values and keys forming pairs as follows:

$$pair_i = < key_i, value_i > .$$

Illustration 5 shows an example of a 3-ary tree.

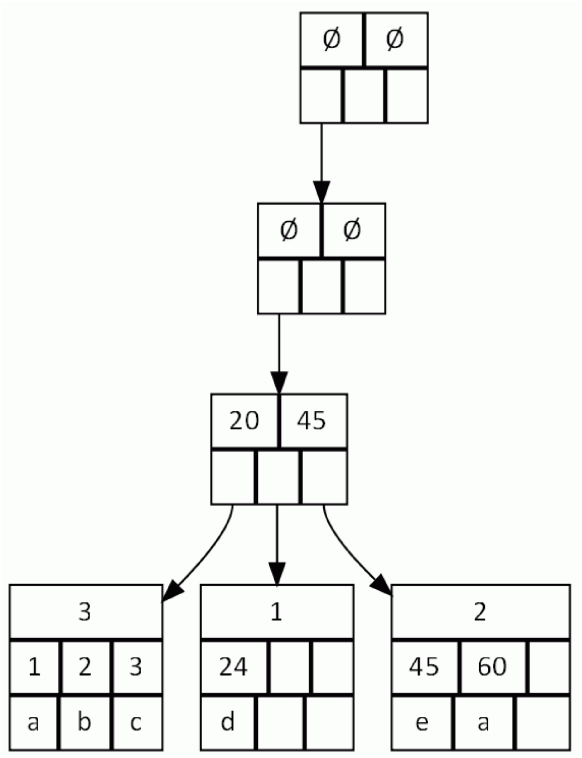


Illustration 5: A fairly balanced 3-ary tree
The tree contains the key-value pairs:
<1,a>, <2,b>, <3,c>,<24,d>,<45,e>, and <60,a>.
The status fields are omitted

7.2.1 Synchronization

Illustration 6 shows a flow graph of the operations. Any operation that changes the search tree, or any value stored within it, is done using help locking, as described in 4.2.3Providing non-blocking algorithms. The data structures assumes the presence of a garbage collector. This allows it to avoid the ABA problem, and problems related to memory reclamation, for data stored accessed as references.

The helping scheme works by setting the status fields of the parents of all involved nodes to point to an object describing the pending operation. The status field effectively works as a continuation for the operation. The order the status fields are set in, is specific to the operation being attempted.

Before setting the status field, it checks that the old status field points to a “no pending operations” status. The status field is set with a CAS operation, so that it fails if the node gets a new pending operation. If there is a pending operation on a node involved, it will instead cleanup any set status fields, help the pending operation, and retry the operation.

They use the following 4 status types:

1. `ReplaceFlag`: Used to specify that a parent is about to replace one of its children, with another child. Contains pointers to the child being replaced, the child its being replaced with, and the parent of the child.
2. `PruneFlag`: Used to specify that a grandparent is about to replace one of its children with a grandchild.
3. `Mark`: Used to specify that a parent is being replaced by one of its children.
4. `Clean`: Used to specify that there are no pending operations.

In the simplest cases, insertion and removal is handled by replacing the leaf node, with a new leaf node with one entry more or less. Before replacing the leaf the parent of the leaf node has its status field set to a `ReplaceFlag` pointing to the leaf node, and an updated leaf node.

When removing the last entry from a leaf node, whose parent has only 2 non-empty children, a “pruning” removal is performed. Pruning removals first set the grandparent's status field to a `PruneFlag`, then setting the parents status to a `Mark` status, and finally replacing the parent with its only nonempty child.

When inserting into a full node, they perform a “sprouting” insertion, where the leaf node is replaced by an internal node, with a leaf child for each entry in the old leaf node. Sprouting insertions set the status field of parent in the same manner as simple insertions and removals.

After finishing an operation, the status field of the new child’s parent is reset to a `Clean` state.

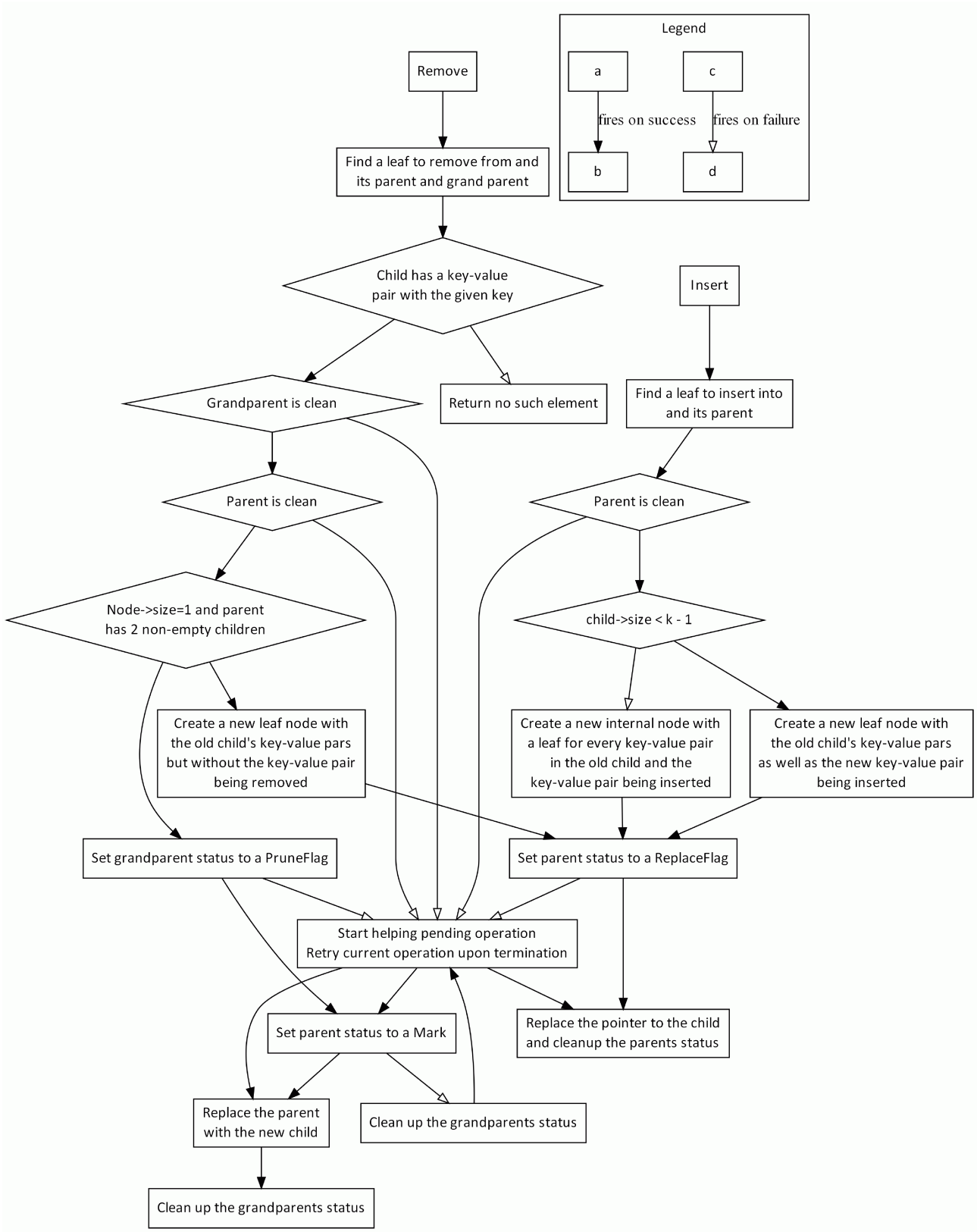


Illustration 6: Flow graph of insert and remove operations on the k-ary tree. All operations that help other operations are restarted after helping.

7.2.2 Issues with the k-ary search tree

The k-ary search tree provides no form of balance guarantees. The lack of balance means that the tree can degenerate into a having a height $h=\frac{n}{2}$, where n is the number of key-value pairs in the tree. An obvious way of solving this issue would be to use a B-tree instead, as the actual data structure is strikingly similar.

The synchronization scheme used in the k-ary tree handles operations in a simple and uniform way, and it might be possible to apply something similar to B-trees. The synchronization scheme does have 2 disadvantages however.

1. All operations require two CAS operations on the status field of the leaf nodes parent, as well as a updating the pointer to the child.
2. All operations require allocation of at least one new `LeafNode` and `Status` object.

The first issue means that at most one operation can make progress on a sub tree of height 2 at any given time. The second issue may cause significant performance penalties in the presence of slow memory allocation.

The first issue can be solved by having unordered key-value pairs in leaf nodes, and updating them directly with CAS operations. Any operation that depends on the contents of a leaf node not changing, must instead freeze the every key-value pairs, as in "Locality-Conscious Lock-Free Linked Lists"[BP11]. Having to freeze every key-value pair is obviously more expensive than updating a field in the parent, but hopefully it will not be necessary for common operations.

Allocating `Status` objects for the operation continuations, could be avoided by storing them by value, rather than reference. Doing so will require that the entire continuation can fit within a field that can be updated by a single CAS operation.

7.3 B-trees

7.3.1 General properties

B-trees are balanced search trees, storing key-value pairs, where values are stored in leaf nodes. All leaf nodes are stored at the same height, and in most implementations all nodes except the root are at least half full. The keys of any node are stored in increasing order, and every key stored by the tree must be unique.

The density of nodes and the balance of the tree, is typically ensured by balancing operations on the nodes. If a node becomes too full or to sparse, it rebalances either by splitting, merging with a sibling node, or stealing from a sibling node. Splits and merges change the number children that the parent has to manage, and may in turn require rebalancing of the parent. The root cannot merge, steal or split like the other nodes, since it does not have siblings. Whenever the root is too full, it creates a new root pointing to the old root, and then it splits the old root in two as usual. If the root only has a single child, it is replaced with its child.

The fact that the B-tree can only increase its height by changing the root, ensures that the height of every node is constant while it is in the tree. It also ensures that all nodes of the same height, also have the same depth.

If every internal node aside from the root has at least k/2 children, then the B-tree can at most have a height of $k = \log_{\frac{k}{2}} n$

7.3.2 Weakened properties

It has been pointed out, that achieving the properties of B-trees in highly concurrent implementations, can be problematic [BFGK05]. General schemes for creating lock-free data structures can be applied, specifically the scheme in “Locking without Blocking”, was applied to a B-tree. The authors of “Locality-Conscious Lock-Free Linked Lists” have also submitted a paper for a B-tree that follows the properties of traditional B-trees quite closely [Daniels11]. As far as we can tell, the paper has not yet been published.

We find that the main issues with the properties of B-trees, for our uses are:

1. Alternating insertions and removals, can lead alternating rebalancings, because the nodes are restricted to being no more than half empty.
2. The keys in leaf nodes are stored in an ordered fashion. Enforcing the ordering may require updating the entire node, when adding or removing.
3. Since B-trees are search trees, all keys must be unique. This property conflicts with using it as a priority queue.

The first issue can largely be avoided, by allowing nodes to be less than half full. This comes at the possible expense of internal memory fragmentation, and the height of the tree. The second issue can be avoided by storing the keys in the leaf nodes in an unordered fashion, to allow updating individual entries in leaf nodes. Synchronizing access to unordered entries, can be achieved similar to the solution found in “Locality-Conscious Lock-Free Linked Lists”. The third issue can be resolved by storing the value of a key-value pair in the least significant bits of the key. This change will obviously require longer keys, and it may even make the key size larger than the word size.

To summarize, we hypothesize that it should be possible to create a lock-free B-tree like data structure, with good performance, that is useable as a priority queue. This hypothesis is based on the structure of B-trees, and the synchronization schemes used in the papers “Non-blocking k-ary Search Trees” and “Locality-Conscious Lock-Free Linked Lists”. For operations on internal nodes, a helping scheme similar to “Non-blocking k-ary Search Tree” can be used. For operations on leaf nodes a scheme similar to “Locality-Conscious Lock-Free Linked Lists” can be used. By allowing nodes to be less then half full, it should be possible to reduce the frequency of rebalancing. We want to reduce rebalancing, since it affects multiple nodes, requiring more powerful synchronization.

7.4 Lock-free B-tree derivative

In this section we describe our lock-free B-tree based data structure. We start out by describing the layout and properties of the data structure, then the synchronization scheme used, present some pseudo-code, and finally explain how it handles memory reclamation.

7.4.1 Layout and Properties

The keys of the data structure are 31 bits, the values are 32 bits. The key 0 is reserved to represent key-value pairs that have not been set.

Key-value pairs must be unique, but keys do not need to be unique. The data structure supports the following 3 operations:

1. `void insert(key, value)` - inserts the key-value pair
2. `uint32_t remove(` key_{min} `,` key_{max} `)` - extracts a key-value pair with the key in $[key_{min}; key_{max}]$, with the smallest key
3. `uint32_t extractSmallest()` - extracts a key-value pair, with the smallest key

The remove operations can be used to partition the space of used keys, in order to store multiple priority queues inside the same data structure.

The data structure is structured as a search tree, with internal and leaf nodes. The root node is a special internal node. It uses rebalancing similar to B-trees, but with a few important differences:

1. Rebalancing never steals key-value pairs, since that would involve too many nodes, compared to how much rebalancing is performed.
2. 2 sibling nodes can be merged into 2 rebalanced siblings, to compensate for the lack of stealing.
3. Nodes are allowed to be more than 50 % empty, and a leaf node that is less than completely full might get split. How empty nodes are allowed to be is a compile-time constant, and it can be different for leaf and internal nodes.

Nodes that are removed from the tree, are deallocated using a hazard pointer scheme. The hazard pointers ensures that nodes are never deallocated or reused, when they are visible to any threads, or in the tree. This ensures that the ABA problem does not occur for pointer to the child nodes, as described in 4.1.3 The ABA problem. The details of how the nodes are reclaimed is described in section 7.7 Memory reclamation.

7.4.2 Layout of nodes

The general data structures used are described in listing 35. In the following sections we will describe what their fields are used for. The nodes of the tree are divided into `Leaf` nodes, that have a height of 0, and `Internal` nodes. Only `Internal` nodes store their height explicitly.

```
class Entry {
  uint32_t value;
  uint31_t key;
  uint1_t freeze;
} alignas(uint64_t);

class Leaf : Node {
  volatile Entry entries[KL];
}

class Mark {
  Entry key;
  Internal* parent;
  uint16_t cHeight;
  MarkType type;
}

class Internal : Node {
  uint16_t size;
  uint8_t height;
  Node* volatile c[K_internal];
  Entry s[K_internal - 1];
  volatile Mark m;
}
```

Listing 35:C++-like declaration for the types used in the lock-free B-tree

An `Entry` is a key-value pair. Each `Leaf` stores `KL` unordered key-value pairs, as per illustration 7. After a `Leaf` has been inserted into the tree, the `Entries` can be updated individually with read-modify-write operations. The entire key-value pair can be updated in a single CAS operation, since it resides within 64-bits. It is possible to update two adjacent `Entries` atomically with a `CMXCHNG16B` instruction, if the first `Entry` is aligned on a 128-bit address.

Operations that depend on multiple `Entries` use their `freeze` bit, to prevent them from changing. All `insert/remove/extract` operations, assume that the `freeze` bit is unset, and the operations will fail if it is set.

<3,8,0>	<9,6,0>	<0,0,0>	<2,7,0>

Illustration 7:A node with 4 Entries containing the key-value pairs <8,3>, <6,9>, and <7,2>, where no Entries are frozen

In `Internal` nodes `c` stores pointers to for its children, and `s[i]` specifies the largest allowed key-value pair of `c[i]`. After an `Internal` is added to the tree, its `height`, `s`, and `size` does not change. The the child pointers, and the `Mark` can change, and therefore need to be volatile.

`Marks` uniquely identify operations involving `Internal` nodes. The `key`, `cHeight` and `parent` fields are used to identify the nodes involved in the operation. Specifically the children involved in the operation have the height `cHeight`, they can be found by searching for `key`, and they have the parent `parent`.

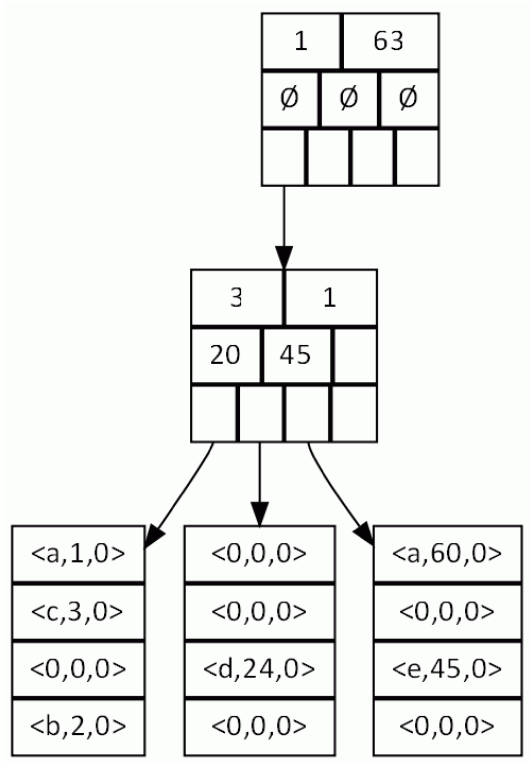


Illustration 8: B-tree representation of the same tree from illustration 5 Note that the B-tree is quaternary, because the internal nodes must not have fewer than 2 children

`MarkType` identifies the type of pending operation, and how far along it is, similar to the `Status` class used in the k-ary tree. The following types are used:

- `NOT_MARKED:` Used to specify that there are no pending operation.
- `REBALANCE_STEP2:` Used to specify that a grandchild of the marked node is about to be rebalanced.
- `REBALANCE_STEP3`: Used to specify that a child of the marked node is about to be rebalanced.
- `REBALANCE_STEP4`: Used to specify that this node is about to be rebalanced, possibly with one of its siblings.
- `REBALANCE_STEP5`: Used to specify that this node is about to be rebalanced with one of its siblings

In order to keep the `Marks` synchronized, they must fit into 128-bits, so they can set by a single CAS instruction. We solved this by storing `Internal` pointers in 32 bits, and `MarkType` in 16 bits, giving a total size of 128 bits. Only using 32 bits for `Internal` pointers reduces the usable address space for Internal objects, but inside an operating system it should be sufficient. If a 32 bit address space is too small, one could store `cHeight` and `MarkType` in 8 bits each, and store the `parent` reference as a 48 bit index.

The height of the child node could also be derived from the height of the marked node, and the type of mark, but storing the child height in marks simplifies some of the code.

To simplify management of the tree, it contains an additional "fake root", above the regular root. The fake root is an `Internal` node with the root as its only child. The fake root ensures that all `Leaf` nodes have grand parents.

7.5 Synchronization

The tree uses two significantly different forms of synchronization depending on the operation.
Most operations proceed by searching for a `Leaf` node, operating on it, and returning, without any rebalancing. Since such operations change a single entry, they are performed with a single CAS operation directly on the data. Before performing the CAS, it first checks to see if the entry is frozen. If an entry is frozen the `Leaf` must be rebalanced before the simple operation can proceed.

```
Entry eOld = entries[i];
if(eOld.freeze) {
  ... rebalance the leaf node
}
if(theOperationCanBePerformedOn(eOld)) {
  if(compareAndSwap(&entries[i], eOld, operate(eOld))) {
    operationSucceeded(eOld);
  }
}
```

Listing 36:Pseudo-code showing how to attempt operations on `Entries` in `Leafs`.

It may be not be possible to apply an operation directly on a `Leaf` node, if the `Leaf` node or its parent is being rebalanced. In such cases the thread will have to help finish up the rebalancing. Illustration 10 shows how rebalancing and simple operations are used to implement operations on the tree structure. One should be aware that rebalancing is significantly more complicated than performing an operation on a `Leaf` node.

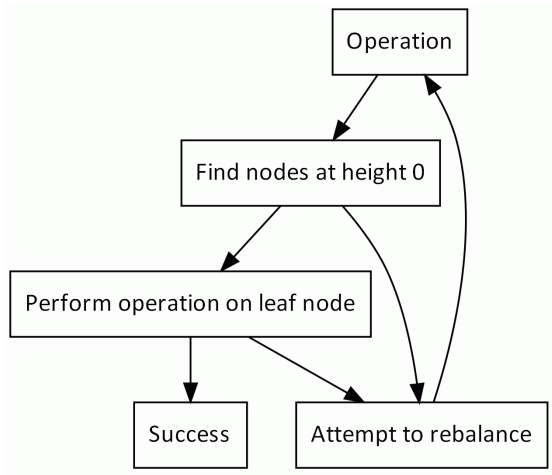


Illustration 9:A flow graph of the algorithms general structure. The edges going into "Attempt to rebalance" are taken if the action fails

7.6 Rebalancing

Rebalancing operations always end up replacing the parent of a node, with a parent where one or two of its children have changed. Rebalancing involves the node being rebalanced, its parent, and its grandparent. It may also include a sibling to the node being rebalanced, unless the child is being split.

In order to modify multiple nodes, a helping scheme is used, similar to the “Non-blocking K-Ary Search Tree”. The rebalancing is shown in illustration 10. First it locates the involved nodes. Then it sets the status field of the involved nodes, in the order grandparent, parent, node, sibling. The fields of the mark field is set as described in the section 7.4.2 Layout of nodes on page 75.

The `Mark` is effectively a continuation for the operation, that any thread can execute. When the `Mark` is set, it is only changed by threads that attempt to complete the operation. If a thread fails to change a `Mark,` because someone else changed it, the thread will instead help the new pending operation, before retrying its own operation.

Pending operations are described in a unambiguous way through `Marks`, such that when a thread executes a pending operation, one of two things can happen:

1. The thread fails to complete the operation.
2. The thread completes the operation and produces the exact same result as any other thread executing it would have.

The `Mark` is unambiguous in the sense that it provides unambiguous directions the operation, and all the nodes involved in the operation. The child being rebalanced and its parent can be found by searching for the key. If the found parent is different from the one in the `Mark,` then some other thread must have finished the operation. The sibling can be determined based on the index of the child. The grandparent can be verified, by checking that it has the `parent` as a child.

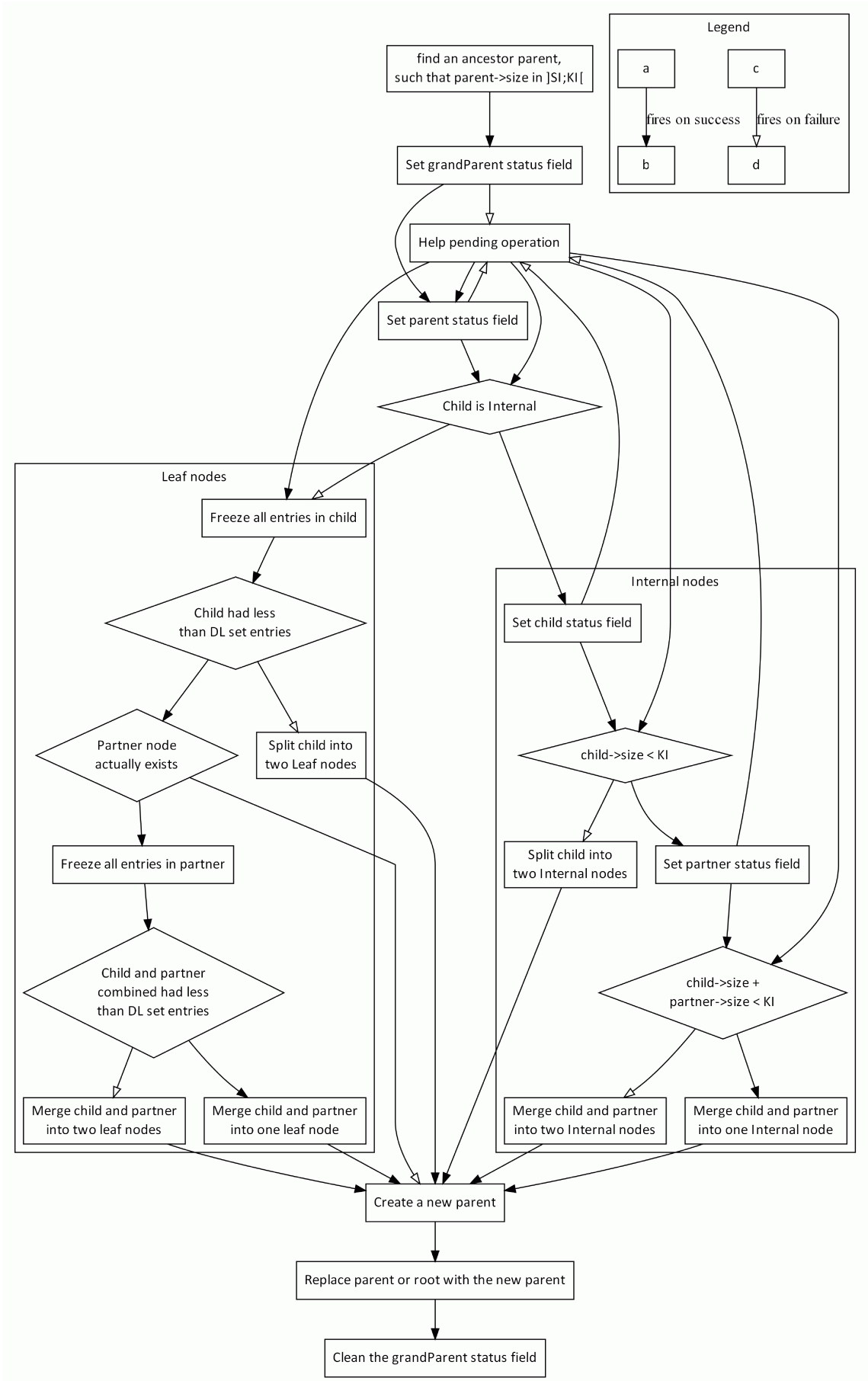


Illustration 10: Flow graph of the rebalancing

Since `Leaf` nodes have no status field, their parents store this field instead. This is sufficient, since rebalancing of a `Leaf` node, would also affect its parent, so the parent should already have its status field set. Rather than setting the status field of `Leaf` nodes, all their entries are frozen to prevent conflicting modifications. Freezing the `Entries` is retried until successful, so "setting the status field" of a `Leaf` node will always succeed. The rebalancing is effectively performed as follows:

1. The grandparents `Mark` is set to a `REBALANCE_STEP2` mark, with a CAS operation.
2. The parents `Mark` is set to a `REBALANCE_STEP3` mark, with a CAS operation
3.
    a) The childs status field is set to a `REBALANCE_STEP4` mark, with a CAS operation, if the node is an `Internal` node.
    b) Every `Entry` in the child is frozen, if the node is a `Leaf`
4.
    a) If the child is too full, goto step 5.
    b) The partners status field is set to `REBALANCE_STEP_5` mark, if the partner is an Internal node.
    c) Every `Entry` in the partner is frozen, if the partner is a `Leaf.`
5. Rebalance the children, into one or two nodes
6. Create a new parent with the new children
7. Replace the old parent with the new one
8. Allow for changes to the grandparent

Step 1 and 2 correspond to the first three states of illustration 9. Step 3, 4 and 5 correspond to the boxes Leaf nodes and Internal nodes in illustration 9. Step 6, 7, and 8 correspond to the last three states of illustration 9. The rebalancing can fail at step 1, 2, 3, 4, 7, and 8.

If it fails at 1, it must be because grandparent is involved in another rebalancing, so the current threads help finish that rebalancing before retrying its own child of the grandparent.

If it fails at 7, it must be because another thread finished the operation, so the current thread just executes step 8, to ensure that the operation no longer blocks other operations, and then it retries its original operation. If it fails at 8, another thread cleaned up after the operation, so the current thread should just retry its original operation.

If an operation fails at step 2, 3 or 4, then it must be because another rebalancing operation started on another descendant of the grandparent node. The current thread helps finish the other operation, and then retries its own original operation, rather than retrying the rebalancing. It is not necessary to clean up the after the failed rebalancing, because it can be finished by any thread at any time.

7.7 Memory reclamation

As previously mentioned the B-tree uses hazard pointers, also known as safe memory reclamation in order to reclaim memory. Hazard pointers are used for reclaiming `Leaf` and `Internal` nodes. Any other part of the data structure is either in local scope only, or assumed to have a well defined lifespan. For instance the tree itself is not reclaimed with hazard pointers, only the nodes in it. Using hazard pointers ensures that nodes cannot be deallocated while any thread is allowed to access data inside the node.

We use hazard pointers, because it is the most balanced approach, currently available. It is typically the second fastest reclamation scheme, with the second lowest bound on unreclaimed memory. Only QSBR is faster, and only reference counting has a lower bound on unreclaimed memory. The basic concepts of hazard pointers was covered in the section 4.2.2 Hazard pointers on page 14.

In order to apply hazard pointers you need:

1. To implement the algorithms required for hazard pointers [Michael04].
2. To find all the hazardous references and determine the number of objects any thread may access at a given time.
3. To implement efficient tests for the global visibility of hazardous references.

The last step can be problematic for arbitrary link based data structures, as described in the paper “Efficient and Reliable Lock-Free Memory reclamation based on reference counting” [GPST05], but it is possible for this tree based data structure.

7.7.1 Algorithms for hazard pointers

Hazard pointers require the presence of two serial data structures. The first data structure must support `insert(void*)` and `contains(void*)` operations, where the data being pointed to by the parameters must not be accessed. The second must support a way to store nodes that are eventually going to be reclaimed. Both data structures have an upper bound on the number of objects they will ever need to store.

For the second data structure we use a simple array. The paper suggests that the first data structure can be a hash set, if amortized average case running time is a concern, or a sorted list, for the sake of simplicity. We use a hash set, because an array backed hash set supporting the required operations is as simple as to implement, as a sorting algorithm.

Since the hash set must not access the data pointed to by the pointers inserted into it, we use a hash function that hashes the pointer itself. Alternatively one could just truncate the pointer, but that would likely cause collision problems, since the pointers cannot be assumed to be uniformly distributed. We use hash functions based on “Integer Hash Function”[Wang07]. Specifically we use the “Robert Jenkins 32 bit integer hash function” for 32 bit pointers, and “64 bit to 32 bit Hash function” for 64 bit pointers. The hash functions are optimized to the specific data size, unlike functions, such as CityHash [Google], MurmurHash [Appleby11], or SpookyHash [Jenkins].

7.7.2 Dealing with hazardous references

When searching in the tree for a given node, it will be necessary to access nodes that may be dynamically deallocated. Whenever a search proceeds from a parent node to its child, it must be sure that the parent and child are both still allocated (2 references).

Whenever performing an operation on a `Leaf`, the `Leaf` node is accessed (1 reference).

Whenever performing a rebalancing operation, up to 4 nodes may be accessed; two child nodes being rebalanced, the parent, and the grandparent of the nodes being rebalanced (4 references).

To simplify maintaining the needed hazard pointers, threads keep the acquired hazard pointers to the nodes at 3 levels while searching, rather than 2 levels. So once a search terminates, the thread will have hazard pointers to the desired node, its parent, and its grandparent. If the thread needs to rebalance the found node, then it already has the hazard pointers needed, with the possible exception of a neighbor to the desired node. If a rebalancing needs to access the neighbor of the node that has to be rebalanced, then it acquires a hazard pointer to the node when needed.

7.7.3 Testing for global accessibility

To implement an efficient test for the global visibility of a node, we take advantage of the synchronization scheme used. Since hazard pointers are practically only acquired during the search, we will start by covering how it is done there. During searches the code upholds the following invariants:

1. Whenever a node is no longer visible, the thread restarts the search.
2. Whenever a parent node may no longer be visible, due to ambiguity about how far along a rebalancing operation is, the thread tries to help the rebalancing operation.
3. No hazard pointer is acquired to the fake root node of the tree, as it is always present.
4. The child of the fake root is the actual root. Testing the global visibility of the root, is simply done by checking that the fake root still points to the root.
5. Testing the global visibility of children of the root, is done by checking that the fake root still points to the root, and that the root still points to the child.
6. Testing for the presence of child nodes further down the tree, is done by checking that the parent still points to the child, and that the status fields of the parent and grand parent are safe.
7. If the parent is marked with `REBALANCE_STEP4` or `REBALANCE_STEP5`, then the parent and grand parent may or may not be present in the tree anymore.
8. If the parent is marked with `REBALANCE_STEP3`, and the grandparent is not marked with an operation that is equivalent to the parents mark, then the parent is not present in the tree anymore.

Listing 37 shows how searches are performed, and how the invariants are upheld. Finding the child of an `Internal` node that might contain a key-value pair, is done with

the `findChild` function. The function finds the relevant descendant of an `Internal` node using binary search on `s`. This is possible because `s` contains a sorted list of upper bounds on the keys that the children can contain. The search is quite similar to the search used in “Lock-Free Multiway Search Trees” [SR10]. The memory barriers can be implemented with an `MFENCE` instruction, or by loading the hazard pointer after storing it, and executing `LFENCE`.

Outside of searching, it may be necessary to acquire a hazard pointer to a sibling of the node being rebalanced. This occurs after marking the grandparent, parent and the node being rebalanced. In that case, the only way the sibling may no longer be visible, is if the operation has finished already. This can be checked by seeing if the grand parent still points to the parent.

```
class SearchResult {
  union {
    Leaf* l;
    Internal* i;
    Node* n;
  }
  uint16_t index;
};

SearchResult findChild(uint32_t value, uint31_t key) {
  uint16_t index = binarySearch<Entry>(<value, key>, s, size - 1)
  return {c[index], index};
}

<SearchResult, SearchResult, Internal*> findNode(Entry key, uint8_t
height) {
retry:
  Internal* gParent = fakeRoot;
  SearchResult parent = <fakeRoot->c[0], 0>;
  hp[1] = parent.i;
  memoryBarrier();
  if(parent.n != fakeRoot->c[0]) {
    goto retry; // See rule 4
  }
  Node* child = parent->findChild(key);
  hp[2] = child.n;
  memoryBarrier();
  if(child.n != parent.i->c[child.index] || parent.n != fakeRoot-
>c[0]) {
    goto retry; // See rule 5
  }
  for(uint8_t cHeight = parent->height - 1; cHeight > height;
cHeight--) {
    gParent = parent.i;
    parent = child;
    hp[0] = hp[1];
    hp[1] = hp[2];
    child = parent->findChild(key);
    hp[3] = child.n;
    Mark m = parent.i->mark;
    if(child.n != parent.i->c[child.index]
       || !gParent->isGrandParentTo(m)) {
      goto retry; // See rule 6 and 8
    }
    if(m.isChild()) {
      … help finish the operation on parent instead. // See rule 7
    }
  }
  return <child, parent, gParent>;
}
```

Listing 37:Pseudo-code that searches for a node at a given height

7.8 Implementation

7.8.1 Implementation of helping

The section 7.6Rebalancing on page 79 describes how rebalancing operations work at a high level, and it represents the rebalancing operations with a flow diagram in illustration 9. In illustration 9, the state “Help pending operation” has quite a few ingoing and outgoing edges, and the control flow is not exactly easy to implement in a structured language. This section describes how we implemented helping at a pseudo-code level. The actual pseudo-code is in listing 38.

The helping is implemented in a function that is used for performing all operation steps that can be helped. In other words the function performs step 1 through 5 of the rebalancing. All of these steps are handled inside one function, because implementing helping will require being able to switch between different steps.

Switching between the different operations is handled with a switch statement, inside an infinite loop. When an operation successfully marks the involved node, it proceeds to the next case in the switch statement, corresponding to the next step in the rebalancing. When an operation fails to mark the involved node, it remembers the mark, breaks from the switch statement, and tries to help the operation described by the mark. It does so by finding the involved nodes, and reentering the switch statement.

The function returns, once it has witnessed some rebalancing operation completing. This can happen at the cases for step 3, 4 or 5, where a rebalancing operation can be completed. This can also happen at step 1, if someone rebalances descendants of the grandparent. Finally threads can also witness operations completing, while trying to load the parameters for helping the operation.

As an alternative to implementing helping by tying all of the helping operations into one function, one could have used proper continuations. Such continuations perform the operations based on a mark. We found that such continuations are more trouble than they are worth. For one the continuations fragments the code, making it hard to see what happens before and after each fragment. Another issue is that using continuations in this way had poor performance, unless the code for helping rebalancing and attempting rebalancing is separated. In this sense separating is practically equivalent to writing the code twice, and complicating the helping. Currently there is little incentive to use such continuations, since all helping steps are fairly related, and using continuations would likely be less clear and/or slower.

```
helpOperation(op, m) {
  while(1) {
    switch(op) {
    case MarkType::NOT_MARKED:
    {
      return;
    }
    case MarkType::REBALANCE_STEP1:
    {
      if(!markGrandParent()) {
        m = grandParentMark;
        break;
      }
      if(gParent->c[parentIndex] != parent) {
        gParent->cleanUp(oldMark);
        return;
      }
    }
    case MarkType::REBALANCE_STEP2:
    {
      if(!markParent()) {
        m = parentMark;
        break;
      }
    }
    case MarkType::REBALANCE_STEP3:
    {
      if(childHeight == LEAF_HEIGHT) {
        ... rebalance the leaf children
        return;
      }
      if(!markChild()) {
        m = childMark;
        break;
      }
    }
    case MarkType::REBALANCE_STEP4:
    {
      if(childSize >= DI) {
        ... split the child, and write back a new parent
        return;
      }
      if(!acquirePartnerHazardPointer()) {
        return;
      }
      if(!markPartner()) {
        m = partnerMark;
        break;
      }
    }
    case MarkType::REBALANCE_STEP5:
    {
      ... merge the children and write back a new parent
      return;
    }
    }
```

```
    // The operation was prevented by a another operation,
    // lets help the other operation then
    do {
      key = m.key;
    } while(!findNode(key, m.cHeight, gParent, parent, child));
    if(!grandParentMark->isEquivalent(m.toStep2())) {
      return; // Someone may have finished the operation
    }
    if (parent != m.parent) {
       // Someone may have finished the operation
       gParent->cleanUp(m.toStep2());
       return;
    }
    op = m.type;
    m = m.toStep2();
    if(op == MarkType::REBALANCE_STEP5 && !getPartnerHP()) {
      return;
    }
  }
}
```

Listing 38:Pseudo code for the first five steps of rebalancing and helping operations

7.8.2 Replacing the parent node

Step 6, 7, and 8 of the rebalancing are primarily concerned with replacing the parent of the nodes being rebalanced, with a new parent. The general control flow of rebalancing ensures that the new parents size is in $[SI; DI]$, unless the old parent is the root of the tree. If the old parent is the root of the tree, additional logic handles growing and shrinking the tree, to keep the roots size within $]1; KI[$, as shown in listing 39.

```
if(newParentSize == 1 && childHeight != 0) {
  … make the only child the new root
  … this reduces the height of the tree
} else {
  … create a new parent, containing the two children
  if(gParent == fakeRoot && newParentSize == KI) {
    … split the new parent in two
    … use the two pieces for a new parent/root
    … the new parent increases the height of the tree
  }
}
if(!CAS(gParent->c[parentIndex], oldParent, newParent)) {
  … free any allocated nodes
  cleanUpGParent()
} else {
  cleanUpGParent()
  … begin reclaiming the old nodes
}
```

Listing 39:Pseudo-code showing how to replace parent nodes, at the end of rebalancing, while maintaining the constraints for the B-tree

7.8.3 Optimizations

There are a number of opportunities for improving performance of the basic data structure, as described in this section. The impact of some of the optimizations is discussed in the section 7.9 Evaluation.

7.8.3.1 Memory system and Hazard pointers

As discussed in the section 7.7 Memory reclamation, the general use of Hazard Pointers may lead to a memory overhead of up to $O(k \cdot n^2)$ objects. It is possible to store $O(k \cdot n)$ freed objects on thread local stacks, without increasing the amortized memory overhead. Doing so improves the locality of the allocations, and may reduce contention in the memory allocator. This is basically the same concept that slab allocators [Bonwick94] use, but they do not have as well defined permitted memory overhead bounds. Based on these observations, we propose a fairly general extension to a hazard pointer framework, enable reuse of recently reclaimed objects. For the B-tree we only need to allocate `Leaf` and `Internal` nodes, so in this case a general scheme may seem like overkill, but the extension is fairly simple.

In order for a new object to be placed in an old objects memory location, the new object must not:

1. Be larger than the old object.
2. Require larger alignments than the old object.

Our solution to first requirement is to only allow objects of the same size to reuse the locations. Our solution to the second requirement, is to enforce a common alignment for all object types of the same size.

Every object size is identified with a unique id. The id is used to identify the correct stack to put recently reclaimed objects on. When reclaiming, retiring, and creating objects, both the object size id and the object size is passed along. For convenience the object size id can be determined from the object size at compile time, so the programmer only has to pass the type of object. Quite often the type, and hence size, of variables are known at compile-time. Specifically for the B-tree it is always known whether a node is a `Leaf` node or an `Internal` node when replacing the nodes.

The creation and retirement of objects depends is handled by the code shown in listing 40. The codes behavior depends on the number of elements on the threads recent stacks. In the code `retired` is a list of retired objects, `reclaimed` is the recent stacks, and `getType` is a function that finds the object size id for a given object size, at compile time. If the elements on a threads recent stacks exceeds an upper bound f, where $f \in O(k \cdot n)$, then the element is freed by the memory allocator. Otherwise it is pushed on a recent stack, for reuse. Upon allocation, if there are elements on a threads stack for that object size id, the memory location is popped from the stack, otherwise a new object is allocated.

```
template<class Type>
void retire(Type* object) {
  retireObject(object, getType<sizeof(Type)>());
}

template<class Type>
Type* create() {
  return (Type*) createObject(sizeof(Type), getType<sizeof(Type)>());
}

void* createObject(uint64_t objectSize, uint64_t objectType) {
  if(freed[objectType].getSize() == 0) {
    ... return a new object of size objectSize
  }
  return reclaimed[objectType].pop();
}

void retireObject(void* object, uint64_t objectType) {
  retired.push({object, objectType});
  if(retired.getSize() >= k * nP) {
    scan();
  }
}
```

Listing 40:Creation and retirement of objects, where recently reclaimed objects can be reused.

The changes require that the object type is known upon deallocation, so the list of retired objects must store both object size ids and object pointers. Our implementation stores it as actual tuples, but it is possible to be more space conserving. If the number of object sizes is no larger than the minimum object alignment guaranteed, then the object size id can be stored in the lower order bits of the object pointers. In current AMD64 code with 64 bit pointers, it is also possible to use the 16 most significant bits to store the object size id, since the address space is 48 bits.

7.8.3.2 Separating Entries into two Leaf nodes

When producing two `Leaf` nodes from a rebalancing, the algorithm must create two approximately evenly large nodes. In addition all the `Entries` in the first child, must be smaller than the `Entries` in the second child. This can be accomplished by dividing the `Entries` based on the median `Entry.`

Finding the median can be done in $\Theta(n)$ time with Hoare's find algorithm[Hoare61], where n is the total number of Entries in the children. Hoare's find algorithm is based on quicksort, it can be implemented in place, and has a side effect. All values that are lower than the median, are placed before the median, thereby doing the work of separating the `Entries` into `Entries` for the first and second child. Hoare's find algorithm does however have a very poor worst case running time of $O(n^2)$.

7.8.3.3 Ammortized running times

`Leaf` and `Internal` nodes have compile-time constant upper and lower bounds on their size. Different boundaries leads to different performance metrics.

It is possible to argue about the performance metrics from a theoretical point of view through their asymptotic running time. To simplify the reasoning about the asymptotic running time, the reasoning ignores the cost and probability of operations failing. This may seem like a large approximation, but it is fairly common practice.

Generally the operations can be described as: Search the tree, possibly rebalance, and search for an `Entry` a `Leaf` node. Searching in a `Leaf` node can require looking at every `Entry`, so it takes $O(kl)$. Rebalancing requires searching the tree, and performing the actual rebalancing of nodes. The actual rebalancing can be achieved in $\Theta(kl)$ for `Leaf` nodes, and $O(ki)$ for `Internal` nodes.

The running time of searching the tree is:

$\Theta(Tree_{SearchCost}) = \Theta(Tree_{Height} \cdot Search_{InternalNodeCost})$.

The height of the tree is:

$\Theta(Tree_{Height}) = \Theta\left(\log_{\frac{ki+si}{2}}\left(\frac{n}{2(dl+sl)}\right)\right) = \Theta\left(\frac{\log(n)-(dl+sl)}{\log(ki+si)}\right)$, assuming the average `Internal` node has a size of $\frac{ki+si}{2}$, and the average `Leaf` node has a size of $\frac{dl+sl}{2}$.

The the running time for searching in an internal node with binary search is:

$\Theta(Search_{InternalNodeCost}) = \Theta(\log(ki))$

Therefore the cost of searching in the tree is:

$\Theta(Tree_{SearchCost}) = \Theta(\Theta(Tree_{Height}) \cdot \Theta(Search_{InternalNodeCost})) = \Theta(\log(n)-(dl+sl))$

Giving all operations a running time of: $\Theta(\log(n)+ki) = \Theta(\log(n))$

The running time can be reduced to $\Theta(\log(n))$, because ki is a constant.

Theoretically, using larger `Leaf` nodes should lead to fewer rebalancings and faster search times, at the cost of slower insertions, removals, extractions and rebalancings. Using larger `Internals` should lead to fewer but slower rebalancing, while search times should not be significantly affected.

7.9 Evaluation

This section evaluates the performance priority queue performance of the lock-free B-tree. The test was performed as per linting 41. 300,000 elements are inserted and extracted from an initially empty priority queue. The ordering of insert and extract operations was decided by sampling from a Bernoulli(p) distribution, unless such an action would lead to more removals, than insertions.

The elements being inserted each have a unique value, to avoid collisions, and to test for correctness. The keys of the elements form a distribution where the following 4 constrains are met:

1. 20% are 1
2. 20% are $2^{31}-1$
3. 50% are 2^{30}
4. 10 % are discretely uniformly distributed from the set $[2;2^{31}-2]/2^{30}$

The distribution of keys is meant to represent the distribution of priorities used in tasks running on an operating system. In such a case, it is common for the vast majority of the tasks to use either the default, lowest, or highest priority.

For comparison we evaluated the performance of pairing heap based priority queues, using different locking mechanisms. The pairing heap implementation is from GCC's STL. Pairing heaps tend to be one of the fastest priority queues [TDK]. The locks used include MCS locks, and our improved MCS locks. Initially we also planned to use the OpenMP based locks, but the default locks on the tested system were simply too slow.

```
test() {
  … measure start time
  inserts = 0
  removes = 0
  while(inserts < 300000) {
    if(inserts == removes || rbern(p)) {
      … add an element to the queue
      inserts = inserts + 1
    } else {
      … remove an element from the queue
      removes = removes + 1
    }
  }
  while(removes < 300000) {
    … remove an element from the queue
    removes = removes + 1
  }
  … measure end time
}
```

Listing 41: The test case performed for priority queues. rbern(p) takes a sample from the Bernoulli(p) distribution

Each test was performed using up to 4 threads, where each was bound to a specific CPU. The tests started all the threads simultaneously. Each of the presented results are based on 160 measurements. To get 160 measurements, when testing with p threads, the tests are run 160 / p times.

System name	HP ProLiant SL165z G7 server
Ram	64 gb
Processors	2 x AMD Opteron 6168 (24 CPUs)
OS	Scientific Linux 6.1
Compiler	GCC (Ubuntu/Linaro 4.6.1-9ubuntu3)
Compiler flags	`-m64 -std=gnu++0x -fopenmp -DNDEBUG -fopenmp -Ofast -fwhole-program -static -flto -fno-align-functions -fno-align-labels -fno-align-loops -fno-align-jumps -s`

Table 16:A description of the platform for running the tests

Graph 9 shows the results of running the tests on the system described in table 16. The following 4 observations are immediately obvious from the results:

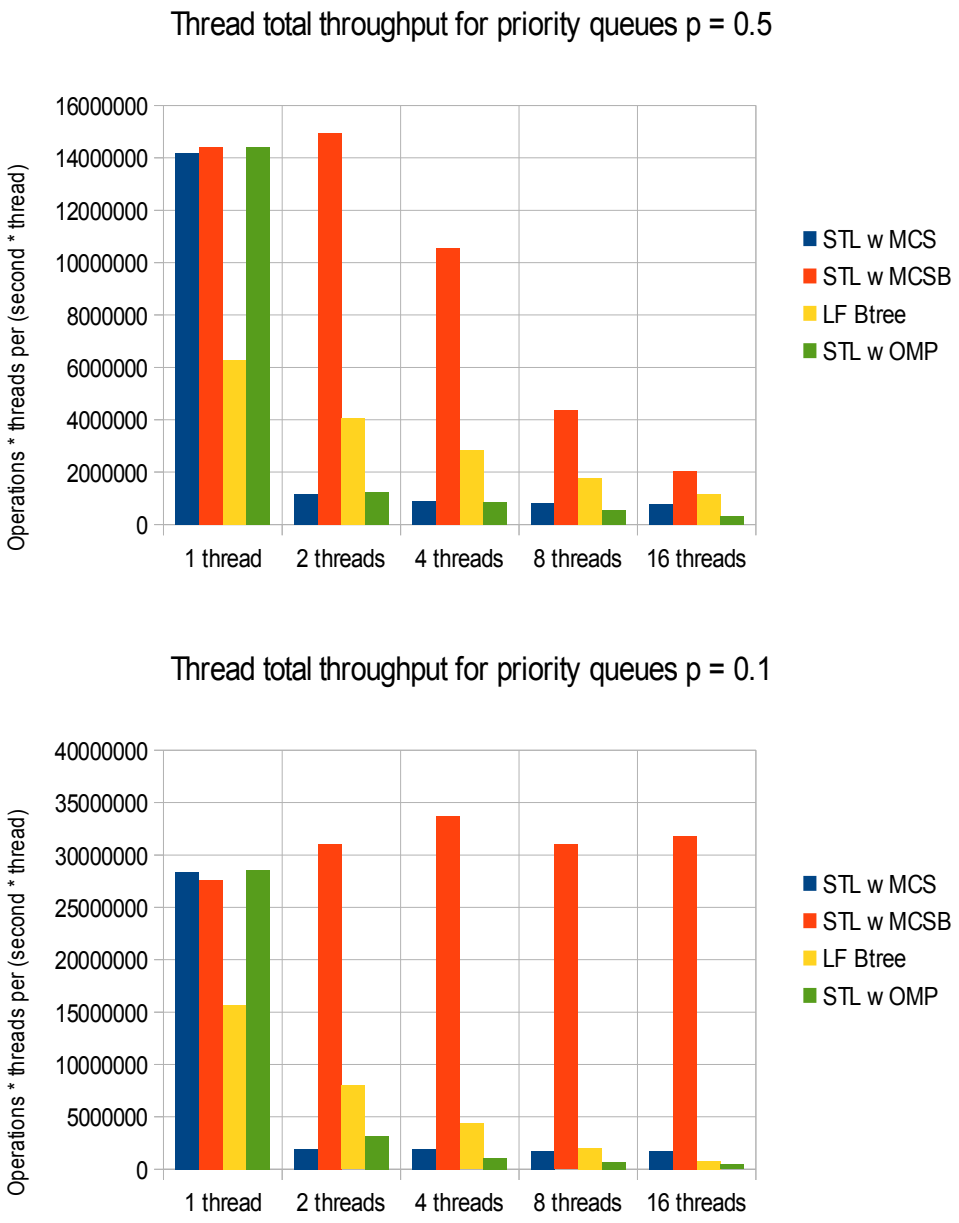
1. The STL w MCSB case shows that the our modified MCS locking mechanism also works quite well for longer locking periods.
2. The lock-free B-tree scales better in the Bernoulli(0.5) case, but in both cases performance is decreasing.
3. In the uncontended cases, all of the priority queues are approximately twice as fast in the Bernoulli(0.1), compared to the Bernoulli(0.5) case.
4. The STL w MCSB scales better in the Bernoulli(0.1) case.

The second observation indicates that the B-tree is contended. A backoff scheme, or randomization of the access patterns to `Leaf` nodes may lead to improved throughput. Applying a backoff scheme the a B-tree is somewhat more complex, than it is to apply it to a stack, due to operations having more steps.

The fourth observation indicates that factors, other than the locks synchronization, dominate the performance of the STL priority queue. If synchronization had been the bottleneck, then the Bernoulli(0.1) would scale worse, due to more lock acquisitions. One explanation for the poorer scalability in the Bernoulli(0.5) case, is the priority queue grows larger. Another possibility is the way memory allocations are handled in the STL priority queue.

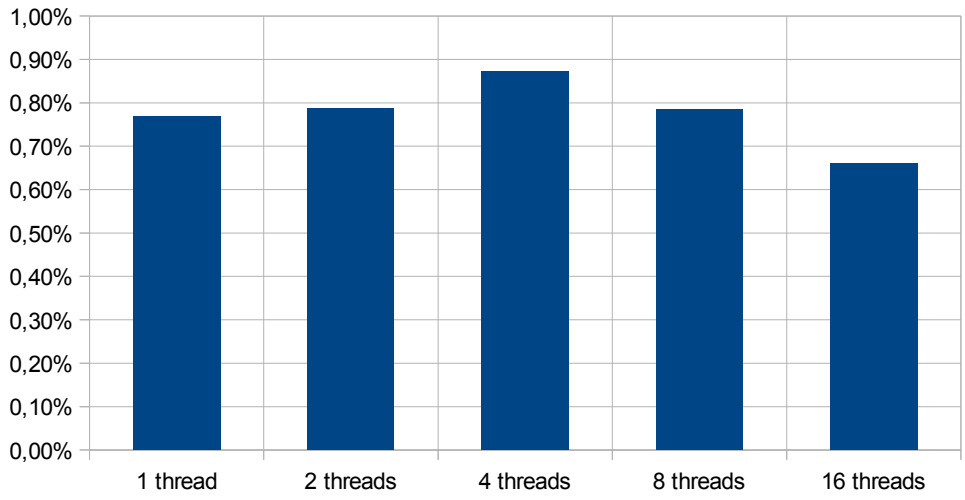
In general the evaluation shows that the throughput of locked priority queues depend highly upon the locking mechanism used, and the order of operations. The LF B-tree

was approximately 50 % faster than the STL based priority queue with regular MCS locks, at high contention levels in the Bernoulli(0.5) case.



Graph 9: Thread total throughput for priority queues. STL w MCS and STL w MCSB are the STL priority queues using MCS locks, with and without backoff. The LF Btree is the priority queue primarily covered in this chapter

We also kept track of the number of `malloc` calls made by hazard pointer framework, to evaluate the impact of reusing objects. Graph 10 shows the ratio of `malloc` calls to the number of objects created. The graph is based on the Bernoulli(0.5) distribution of operations.



Graph 10: Ratio of malloc calls to Node creations
The graph highlights 2 important properties:

1. Less than 1 % of the Nodes created are allocated using `malloc`
2. The fraction of allocations is highest in the case with 4 threads.

The first property shows that reusing a objects can dramatically reduce the the number of allocations. The second property can is likely due to the fraction of allocoations decreasing when the recent lists increases with the number of threads, together with the average queue size. A longer recent list leads to fewer allocations, improving the fraction at high concurrency levels. A smaller queue means less rebalancing, improving the fraction at low concurrency levels.

7.10 Conclusion

In this chapter we introduced a lock-free B-tree based on a help locking scheme. The B-tree is an ordered dictionary structure, that we adapted to a priority queue. The data structure uses hazard pointers to reclaim nodes, thus avoiding traditional garbage collection and reference counting. Inserting, removing, and extracting elements from the data structure has an amortized running time of $\Theta(\log n)$, in the uncontended case. The remove operations, can additionally provide upper and lower bounds on the keys being removed. This ability makes it possible to partition the space of used keys, to make it possible to store multiple priority queues inside the same tree structure.

We found that the data structure is approximately 50 % faster than a pairing heap based priority queues, protected by traditional MCS locks, at high contention levels. By comparison the the backoff scheme for with MCS locks we presented in 5.6 MCS locks, was appropriately 150 % faster for the locked priority queues.

8 Conclusions

This thesis has dealt with the design and implementation contention resistant priority queues. It covered a wide range of methods for providing contention resistant priority queues.

It introduced improved variations of truncated exponential backoff, and MCS locks. These schemes can be used to manage contention in lock-free and lock-based data structures. At the highest contention levels, the new MCS locks can provide 2700 % higher throughput for locked counters, and 150 % higher throughput for priority queues.

We have implemented and evaluated the performance of prior contention resistant data structures. These data structures allow threads to operate on them, without directly accessing the data structure. We have also designed and implemented similar data structures, by combining their concept of elimination with truncated exponential backoff. We found that our new data structures are up to 36 % faster, than similar eliminating data structures. Unfortunately the data structure is not significantly faster, than the same data structure without elimination.

We have also introduced a new lock-free priority queue. In our evaluation, the new priority queue does not have decreasing throughput at high contention. This is in spite of the priority queue not using any explicit measures against contention. We found that the priority queue approximately 50 % faster at high contention levels, than a lock-based priority queue, using traditional MCS locks.

The wide range of topics covered, leaves plenty of opportunities for future work. One possibility is investigating the impact of applying randomized access patterns to the lock-free priority queue. Another interesting topic, could be more extensive correctness proofs for the data structure. Another possibility is investigating the possibilities of applying combined backoff and elimination schemes to more complex data structures, with more expensive operations. One can also investigate ways of making elimination faster, to make it a more viable mechanism for reducing contention.

9 Project planning

This section shows how the project was planned, and how it progressed. The plan includes a risk analysis, and an actual plan for the project.

9.1 Risk analysis

This section covers the risks I estimated to be significant at the start of the project, and how I planned to reduce the risks.

9.1.1 Potential risks

These are the risks that can occur during the development, and that I should try to avoid by all means.

1. Finding minor bugs in the implementation will require absurd amounts of debugging.
2. Lost work due to accidents or regressions breaking existing work.
3. The priority queue cannot be directly applied to the scheduling in Fenix, because it does not uphold specific requirements.
4. Integrating the priority queue structure into Fenix will cause serious problems.
5. The project is not completed in time.
6. The theoretical background for the solution will not hold up.
7. I, or someone vital to the project, will get ill for a prolonged period.
8. After implementing and optimizing the solution we find that its performance is insufficient.

9.1.2 Reducing risks

9.1.2.1 Minor bugs

The risk of minor bugs slowing down work on the implementation is significant for concurrent code and kernel based code, but it can be reduced by doing test driven development. The work on the implementation should be done by alternating between developing a small feature, and subjecting the feature to tests, debugging tools and static analysis tools.

The tests would probably start out as minor tests in a main function, but eventually they should be moved to separate functions, so they can be used as unit tests. Whether or not the unit tests are run can be controlled by putting them in assert statements, so they are compiled out of performance builds, but run in debugging builds.

The debugging tool of choice is GDB, especially since Fenix is written GCC specific C++. In order to be able to debug properly, any statement that can possibly fail, or produce output that I want to debug, must not be written as a macro. Additionally it

might make sense to pass additional parameters to functions, as it may help when debugging. The extra parameters will be compiled away if the functions are inlined in the performance builds, so it might even make sense to keep redundant parameters when doing performance tests, or integrating the priority queue into Fenix.

Most static analysis tools do not work well with the gnu specific C++ dialect, so the most important static analysis tool will probably be GCC with additional warnings enabled. Some parts of the implementation are likely to contain no GCC specific code. Such code can be analyzed by other C++ static analysis tools that are available, such as cppcheck, Uno or Oracle Solaris Studio. The Fenix group is considering getting a license to PCLint.

9.1.2.2 Lost work due to accidents

To reduce the risk of losing work, I will try to save my work often, at least every 10 minutes, and I will keep any code or important documents stored on a remotely stored versioning system. New versions should be committed to the versioning system whenever things work, as well as at important steps between functional versions. To be able to the contents has developed, I will clearly mark non functional versions, and each commit will have a message briefly describing its changes, and the reasoning behind them. If I find that I have severely broken existing work, I will revert to a version where it is not broken and create a new fork, so that it will possible for me review parts the work done since the regression, in case it can be applied at a later time. I am writing this section after just having lost about an hour of work, due to my computer inexplicably shutting down, so I thought it would be a good idea to include this section.

9.1.2.3 Arriving at a inappropriate solution

To ensure that the priority queue is sufficient for use in scheduling in Fenix, the use cases and possible applications of it should be clear before choosing a design to implement. I can however make short and quick mock up implementations of parts of possible designs, to get a feel for the properties of such designs. To find possible use cases and applications of the priority queue, I should discuss it with other developers on Fenix, and look at the existing code. At the time of writing Sven is looking into various areas of Fenix where priority queues can be used. Doing so will serve to find different ways of applying them with different memory/processing/contention tradeoffs. Ideally the design should be sufficient to support any of those relevant tradeoffs, but it might be necessary to pick a design that is supports a subset of the tradeoffs, in order to simplify the requirements to the solution.

9.1.2.4 Problems with the integration into Fenix

Integrating the priority queue into Fenix is likely to cause minor headaches due to the projects current build process. We are currently in the updating the build process to support gcc's link time optimization, and this may cause changes that makes integrating new code easier. Another problem with integrating the new code into Fenix is the risk of breaking existing code. That risk can be reduced by integrating small parts of the solution at a time, honoring the existing API for the priority queues, and verifying the

functionality at each step. It is also important to initially verify that the priority queue is functional on its own merits outside of Fenix.

9.1.2.5 Running out of time

To avoid the risk of not completing the project in time, I should set up a loose timetable, that gives upper and lower bounds to different parts of the development, so that milestones and deadlines can be set up. This risk analysis provides some hint to the order in which development should proceed, and based on that I can create a somewhat realistic schedule. If the milestones are not met, I should either pick up the pace, chose a simpler solution, or drop parts of the project.

There are a number of ways the solution can be simplified:

1. I can support a reduced set of use cases for Fenix, for instance not supporting dynamic priorities.
2. I can worry less about providing a high performance priority queue in uncommon cases, for instance extremely high contention (simpler or no backoff scheme), extremely many elements on the queue, or reducing the range of supported priority levels.
3. I can implement a smaller foundation of the scheduling in Fenix, by not integrating scheduling into every aspect where it is relevant, but instead leaving that job to future projects.
4. I can reduce the project to only dealing with designing, verifying, implementing and testing a suitable priority queue in user space, and let another project deal with integrating it into Fenix.

9.1.2.6 Non-functional solution

To reduce the risk of having a priority queue that has a foundation that cannot be proven to work, it should whenever possible be based on existing work, and established conventions. After choosing a design and prior to integrating it into Fenix, important properties of the priority queue should be proven at least in sketched simplified proofs, and tests cases should be provided to back up the proofs.

9.1.2.7 Illness

To reduce the risk of illness interfering with the productivity of the project, I will be sure to dress appropriately when going outside (it is not exactly suitable weather for sandals and shorts anymore), and try to get lots of sleep if I feel sick. Additionally if illness does strike, I should try to continue working in some degree, possibly by focusing on less strenuous tasks, such as reading up on existing work.

9.1.2.8 Solution is too expensive

To reduce the cost of finding that the solution is too expensive, I should be able to argue about the amortized memory consumption and running time prior to implementing. After implementing operations I should set up somewhat realistic benchmarks for the

operations, and compare it to existing solutions. Comparing with existing solutions can be done by implementing simpler existing work, or by comparing to publicly available implementations of other solutions.

9.1.3 Evaluation

During this project I mainly followed the guidelines of the risk analysis, for reducing risks. However, 4 main problems were encountered:

1. Getting useable results from old test cases.

This was not predicted by the risk analysis, because it was largely a result of what had happened prior to writing the risk analysis. I made some early concepts that showed interesting results. Unfortunately at the time, they seemed interesting, but not promising, so it was abandoned in a fairly messy state, and I had not documented the findings. While writing the report, I believed it would be a good idea to present the results. Getting the results to a useful state, ended up taking a lot of time. A lesson to take from this issue, is that one should always document interesting findings, rather than assume that they are easy to reproduce.

2. Theoretical/implementation issues.

To reduce such issues, the risk analysis suggested using prior work, arguing early about correctness, using test driven development, and using debugging and static analysis tools. Debugging and static analysis tools were used extensively during development, and it helped resolve both minor and major issues. The main issues encountered, were related to the extensions made to prior work. Specifically the memory reclamation scheme, and details in the new synchronization scheme.

The largest issue, was that the original design of the solution, was too vague to tell if individual aspects were fully correct. Specifically I never wrote a full pseudo code for the data structure, until long after I was implemented. Instead I had descriptions of the properties that had to be maintained at every stage, and a description of the order things should happen in. This made it difficult to argue about correctness at a fine-grained level.
The weaker description, was chosen partly due B-trees being very complex beasts. The complexity would introduce a lot of noise in proper pseudo-code. Another contributing factor, was that describing complex help-locking functionality is very unintuitive in structured languages. The structure of the helping code I had initially imagined, would require handling a great many implementation specific details. That is why I thought it better to not use proper pseudo-code. When I was writing the documentation of the synchronization scheme used, I realized that it would be much easier to express it through a flow graph. This revelation would have helped immensely during development.

It turned out to be quite hard to apply test driven development to a data, since features of the data structure are connected. To reduce this issue, I initially skipped implementing memory reclamation, as most of the features can be tested even if memory is not reclaimed. Unfortunately this meant that the tests had to be limited in scope, in order to avoid running out of memory. This in turn meant that a lot of issues were caught fairly late.

3. The project did not finish at the planned time.

The two first problems meant that not all of the planned features were in the project. The risk analysis identified this risk, and suggested dropping or delaying less important features. The project was trimmed accordingly. The project did however drop features in a slightly different order, than what was proposed from the plan and risk analysis. Specifically the priority queue was never implemented into Fenix, but it was compared to other implementations, and optimized to some degree. The ordering was largely due to Fenix being in a state of flux, for the majority of the projects duration. As a result, the topic of the project has evolved from designing a priority queue for Fenix, towards designing priority queues in general.

4. Lost work due to accidents or regressions.

For the entire project I followed the guidelines for version control and data management for the code. The only issue I had, was that the restricted testing meant, that I was not always completely sure if revisions were completely correct or not. In my oppinion that is primarily due to the issues discussed under problem 2.

I did not follow the guidelines for version control, when writing the report, which lead to interesting issues. I probably lost less than one days worth of work in total. It is still a significant loss, which I do my best to avoid in the future. I failed to follow the guidelines, partially because I started writing the report fairly late, and at that point I had forgotten the guidelines.

9.2 Project process and time planning

This project informally started during the summer of 2011. In the beginning the project was very loosely organized, which makes it hard for me to specify exactly when it started. The project did not have a clear definition until August of 2011. The project was originally going to be handed in by December 24th 2011, but it was extended until January 20th 2012. Due to issues related to project registration, the official delivery was changed, to March 15th 2012 during week 11 of 2012. During week 11, from January 21st to week 11, I did not work on the project, but in week 11 I made several improvements to the report. The improvements were mainly fixing linguistic problems, and improving the presentation.

During the first months of the project, I investigated prior work in the field of concurrent priority queues. This investigation was primarily focused on bounded priority queues. The investigation resulted in the majority of the content presented in chapter 6. After experimenting with such data structures for about a month, without properly starting the project, I had a thorough meeting with Sven. In the meeting we planned the final direction of the project.

We decided to focus on the idea of creating a lock-free B-tree based priority queue. In the first weeks I read about B-trees, and similar wide search trees. During September I read up on lock-free search data structures, in order to get a basic understanding, of how such a tree could be made lock-free.

After getting an understanding of the problem, I made the time plan seen in graph 11. It contains mandatory and non-mandatory features, planned in order of importance. The

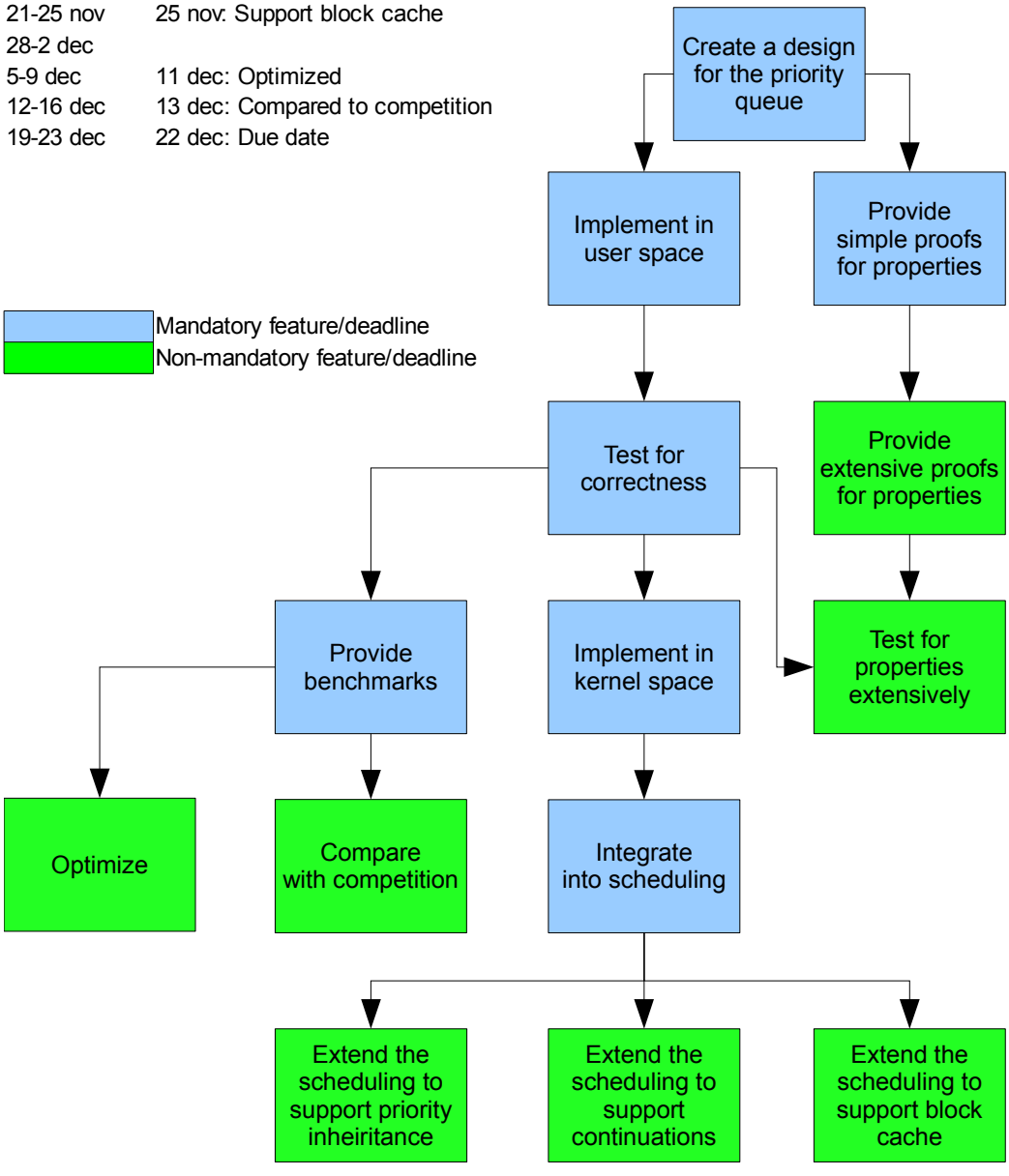
priorities are derived from the risk analysis, and the deadlines have been set in an attempt to make them possible, yet optimistic. The idea behind the mandatory and non-mandatory features, is that if a deadline is missed, it can be delayed by dropping non-mandatory features. That could provide a more flexible schedule, to ensure having a draft of the report a month before the due date, and that the due date was met.

Graph 12 shows what I actually worked on. Several factors meant that non-mandatory features were not implemented. In fact the only non mandatory features provided, are some optimizations, and a comparison to competing solutions.

The end result is that I did not have a working user space implementation before November, and at that point I did not have much in the way of technical report. This lead to the the draft being postponed to the middle of December. I had decided to describe my findings on bounded priority queues in the technical report, hoping that the I would be able to quickly reproduce my findings from last summer. Long story short, the code I had used to make my findings was extremely messy, and I had to spend a lot of time to make sure the findings were correct. This took me weeks, in contrast to the original time plan, which assumed that it was already done.

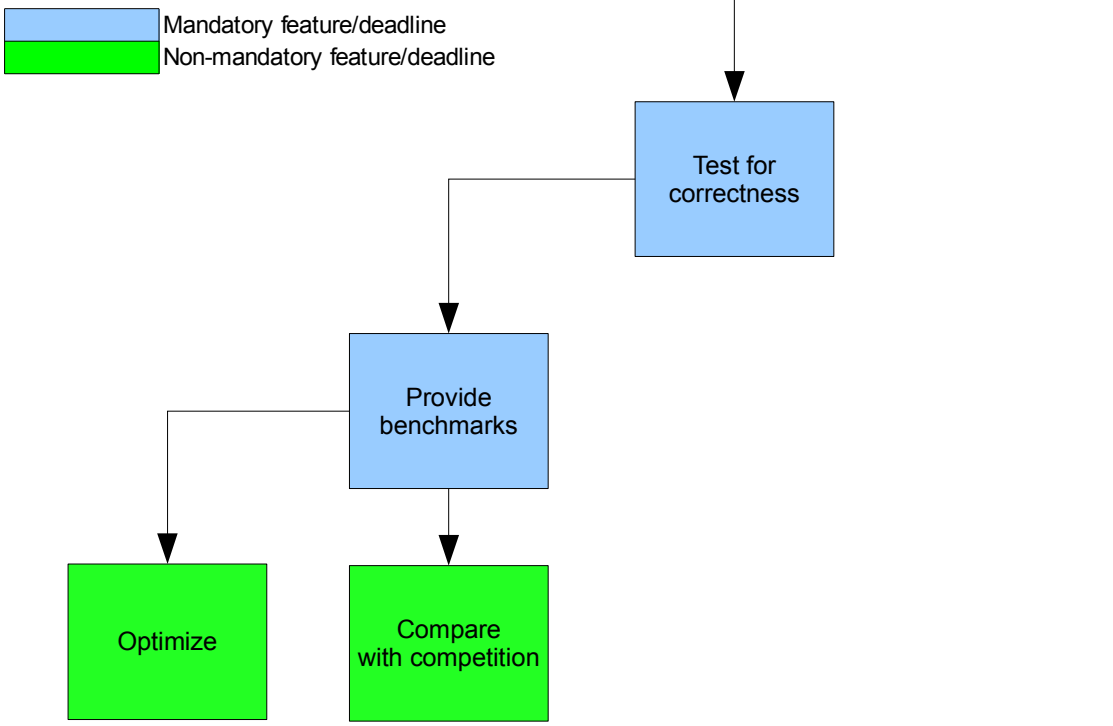
Cleaning up the findings, was one of the main reasons for this project needing an extension. Another reason was that the technical report was too unfinished, and that we would like compare the performance of the solutions with the competition. The comparison to competing priority queues is limited to GCC's STL implementation of the priority_queue class. I had applied for a free licence to LEDA (http://www.algorithmic-solutions.com/leda/index.htm) as well as various licence variants of NOBLE (http://www.non-blocking.com/), but I never received any feedback, so I was unable to evaluate their solutions.

Week	Deadlines
26-30 sept	30 sept: Finalize a design for priority queue
3-7 oct	9 oct: Provide simple proofs
10-14 oct	14 oct: User space implementation
17-21 oct	19 oct: Pass test for correctness. 21 oct: Provide benchmarks
24-28 oct	25 oct: Kernel space implementation
31-4 nov	31 oct: Integrate into scheduling. 4 nov: Support continuations
7-11 nov	8 nov: Support priority inheiritance. 10 nov: Extensive proofs
14-18 nov	16 nov: Extensive test of properties. 20 nov: Draft of technical report
21-25 nov	25 nov: Support block cache
28-2 dec	
5-9 dec	11 dec: Optimized
12-16 dec	13 dec: Compared to competition
19-23 dec	22 dec: Due date



Graph 11: Original time table

Week	Occurences
26-30 Sept	Had a concept for the priority queue, and arguments why it should work
3-7 Oct	Worked on user space implementation
10-14 Oct	Worked on user space implementation
17-21 Oct	Found a significant issue in the design, which was fixed the same week
24-28 Oct	User space implementation passed early simple test, but had no reclamation
31-4 Nov	Investigated ways to implement reclamation
7-11 Nov	Implemented working reclamation, extended tests, insert and extract worked
14-18 Nov	Worked on the report, improved data structure
21-25 Nov	Worked on the report, improved data structure
28-2 Dec	Worked on the report
5-9 Dec	Worked on the report, started verifying old findings
12-16 Dec	Worked on the report, verifying old findings
19-23 Dec	Worked on the report, verifying old findings
2-6 Jan	Worked on the old findings, fixed remove
9-13 Jan	Worked on the report, verifying old findings
16-20 Jan	Worked on the report
12-15 Mar	Worked on the report



Graph 12: Actual time table

10 Appendix

10.1 Read-modify-write update loops

This section contains the assembly code generated for the update loops described in section 5.4.1.3 Interfacing to read-modify-write instructions.

```
  Add
 Val
128-16-16-50
  402500:   49 89 c1              mov    %rax,%r9
  402503:   49 89 d2              mov    %rdx,%r10
  402506:   4d 89 cb              mov    %r9,%r11
  402509:   4d 89 d4              mov    %r10,%r12
  40250c:   4c 89 c8              mov    %r9,%rax
  40250f:   4d 01 eb              add    %r13,%r11
  402512:   4c 89 d2              mov    %r10,%rdx
  402515:   4d 11 f4              adc    %r14,%r12
  402518:   4c 89 db              mov    %r11,%rbx
  40251b:   4c 89 e1              mov    %r12,%rcx
  40251e:   f0 48 0f c7 4d 00     lock cmpxchg16b 0x0(%rbp)
  402524:   4c 89 d1              mov    %r10,%rcx
  402527:   49 31 c1              xor    %rax,%r9
  40252a:   48 31 d1              xor    %rdx,%rcx
  40252d:   4c 09 c9              or     %r9,%rcx
  402530:   75 ce                 jne    0x402500

64-7,6,25
     402090:    48 8b 16               mov    (%rsi),%rdx
  402093:   eb 03                 jmp    0x402098
  402095:   48 89 c2              mov    %rax,%rdx
  402098:   48 8d 2c 0a           lea    (%rdx,%rcx,1),%rbp
  40209c:   48 89 d0              mov    %rdx,%rax
  40209f:   f0 48 0f b1 2e        lock cmpxchg %rbp,(%rsi)
  4020a4:   48 39 c2              cmp    %rax,%rdx
  4020a7:   75 ec                 jne    0x402095
32-7,6,17
     401c40:    8b 16                  mov    (%rsi),%edx
  401c42:   eb 02                 jmp    0x401c46
  401c44:   89 c2                 mov    %eax,%edx
  401c46:   8d 2c 1a              lea    (%rdx,%rbx,1),%ebp
  401c49:   89 d0                 mov    %edx,%eax
  401c4b:   f0 0f b1 2e           lock cmpxchg %ebp,(%rsi)
  401c4f:   39 c2                 cmp    %eax,%edx
  401c51:   75 f1                 jne    0x401c44

16-6,5,21
  4017d3:   eb 02                 jmp    0x4017d7
  4017d5:   89 c2                 mov    %eax,%edx
  4017d7:   44 8d 1c 3a           lea    (%rdx,%rdi,1),%r11d
  4017db:   89 d0                 mov    %edx,%eax
  4017dd:   66 f0 44 0f b1 1e     lock cmpxchg %r11w,(%rsi)
  4017e3:   66 39 c2              cmp    %ax,%dx
  4017e6:   75 ed                 jne    0x4017d5
```

```
8-7,6,28(22)
      401635:     0f b6 16                movzbl (%rsi),%edx
  401638:   eb 08                   jmp    0x401642
  40163a:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
  401640:   89 c2                   mov    %eax,%edx
  401642:   44 8d 1c 0a             lea    (%rdx,%rcx,1),%r11d
  401646:   89 d0                   mov    %edx,%eax
  401648:   f0 44 0f b0 1e          lock cmpxchg %r11b,(%rsi)
  40164d:   38 c2                   cmp    %al,%dl
  40164f:   75 ef                   jne    0x401640

 bool
8-7(6),5,25(17)
  401635:   0f b6 06                movzbl (%rsi),%eax
  401638:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  40163f:   00
  401640:   8d 14 08                lea    (%rax,%rcx,1),%edx
  401643:   f0 0f b0 16             lock cmpxchg %dl,(%rsi)
  401647:   0f 94 c2                sete   %dl
  40164a:   84 d2                   test   %dl,%dl
  40164c:   74 f2                   je     0x401640

16-7(6),5,24(18)
  4017c5:   0f b7 06                movzwl (%rsi),%eax
  4017c8:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  4017cf:   00
  4017d0:   8d 14 38                lea    (%rax,%rdi,1),%edx
  4017d3:   66 f0 0f b1 16          lock cmpxchg %dx,(%rsi)
  4017d8:   0f 94 c2                sete   %dl
  4017db:   84 d2                   test   %dl,%dl
  4017dd:   74 f1                   je     0x4017d0

32-7(6),5,24(19)
  401c15:   8b 06                   mov    (%rsi),%eax
  401c17:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  401c1e:   00 00
  401c20:   8d 14 18                lea    (%rax,%rbx,1),%edx
  401c23:   f0 0f b1 16             lock cmpxchg %edx,(%rsi)
  401c27:   0f 94 c2                sete   %dl
  401c2a:   84 d2                   test   %dl,%dl
  401c2c:   74 f2                   je     0x401c20

64-6,5,19
  402060:   48 8b 06                mov    (%rsi),%rax
  402063:   48 8d 14 08             lea    (%rax,%rcx,1),%rdx
  402067:   f0 48 0f b1 16          lock cmpxchg %rdx,(%rsi)
  40206c:   0f 94 c2                sete   %dl
  40206f:   84 d2                   test   %dl,%dl
  402071:   74 f0                   je     0x402063

128-17(16),14,59(51)
  4024c0:   4c 8b 0e                mov    (%rsi),%r9
  4024c3:   4c 8b 56 08             mov    0x8(%rsi),%r10
  4024c7:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  4024ce:   00 00
```

```
  4024d0:   4d 89 cb                mov    %r9,%r11
  4024d3:   4d 89 d4                mov    %r10,%r12
  4024d6:   4c 89 c8                mov    %r9,%rax
  4024d9:   4d 01 eb                add    %r13,%r11
  4024dc:   4c 89 d2                mov    %r10,%rdx
  4024df:   4d 11 f4                adc    %r14,%r12
  4024e2:   4c 89 db                mov    %r11,%rbx
  4024e5:   4c 89 e1                mov    %r12,%rcx
  4024e8:   f0 48 0f c7 4d 00       lock cmpxchg16b 0x0(%rbp)
  4024ee:   0f 94 c1                sete   %cl
  4024f1:   84 c9                   test   %cl,%cl
  4024f3:   49 89 c1                mov    %rax,%r9
  4024f6:   49 89 d2                mov    %rdx,%r10
  4024f9:   74 d5                   je     0x4024d0

 goto 8-4,3,12
  401625:   0f b6 06                movzbl (%rsi),%eax
  401628:   8d 14 08                lea    (%rax,%rcx,1),%edx
  40162b:   f0 0f b0 16             lock cmpxchg %dl,(%rsi)
  40162f:   75 f7                   jne    0x401628

16-4,3,13
  4017a0:   0f b7 06                movzwl (%rsi),%eax
  4017a3:   8d 14 28                lea    (%rax,%rbp,1),%edx
  4017a6:   66 f0 0f b1 16          lock cmpxchg %dx,(%rsi)
  4017ab:   75 f6                   jne    0x4017a3

32-4,3,11
  401bd0:   8b 06                   mov    (%rsi),%eax
  401bd2:   8d 14 18                lea    (%rax,%rbx,1),%edx
  401bd5:   f0 0f b1 16             lock cmpxchg %edx,(%rsi)
  401bd9:   75 f7                   jne    0x401bd2

64-4,3,14
  401ff5:   48 8b 06                mov    (%rsi),%rax
  401ff8:   48 8d 14 08             lea    (%rax,%rcx,1),%rdx
  401ffc:   f0 48 0f b1 16          lock cmpxchg %rdx,(%rsi)
  402001:   75 f5                   jne    0x401ff8

128-12,13,53
  402440:   4c 8b 0e                mov    (%rsi),%r9
  402443:   4c 8b 56 08             mov    0x8(%rsi),%r10
  402447:   4d 89 cb                mov    %r9,%r11
  40244a:   4d 89 d4                mov    %r10,%r12
  40244d:   4c 89 c8                mov    %r9,%rax
  402450:   4d 01 eb                add    %r13,%r11
  402453:   4c 89 d2                mov    %r10,%rdx
  402456:   4d 11 f4                adc    %r14,%r12
  402459:   4c 89 db                mov    %r11,%rbx
  40245c:   4c 89 e1                mov    %r12,%rcx
  40245f:   f0 48 0f c7 0e          lock cmpxchg16b (%rsi)
  402464:   0f 85 c6 03 00 00       jne    0x402830
      ...
      402830:     49 89 c1                mov    %rax,%r9
  402833:   49 89 d2                mov    %rdx,%r10
  402836:   e9 0c fc ff ff          jmpq   0x402447
```

```
 BBool
8-6,6,17
  401600:   0f b6 13              movzbl (%rbx),%edx
  401603:   0f b6 c2              movzbl %dl,%eax
  401606:   01 ca                 add    %ecx,%edx
  401608:   0f b6 d2              movzbl %dl,%edx
  40160b:   f0 0f b0 13           lock cmpxchg %dl,(%rbx)
  40160f:   75 ef                 jne    0x401600

16-6,6,18
  401740:   0f b7 13              movzwl (%rbx),%edx
  401743:   0f b7 c2              movzwl %dx,%eax
  401746:   01 f2                 add    %esi,%edx
  401748:   0f b7 d2              movzwl %dx,%edx
  40174b:   66 f0 0f b1 13        lock cmpxchg %dx,(%rbx)
  401750:   75 ee                 jne    0x401740

32-4,4,13
  401b50:   8b 03                 mov    (%rbx),%eax
  401b52:   41 8d 54 05 00        lea    0x0(%r13,%rax,1),%edx
  401b57:   f0 0f b1 13           lock cmpxchg %edx,(%rbx)
  401b5b:   75 f3                 jne    0x401b50

64-4,4,14
  401f50:   48 8b 03              mov    (%rbx),%rax
  401f53:   48 8d 14 01           lea    (%rcx,%rax,1),%rdx
  401f57:   f0 48 0f b1 13        lock cmpxchg %rdx,(%rbx)
  401f5c:   75 f2                 jne    0x401f50

     BVal
8-8,7,27
  401600:   0f b6 13              movzbl (%rbx),%edx
  401603:   eb 02                 jmp    0x401607
  401605:   89 c2                 mov    %eax,%edx
  401607:   44 8d 04 0a           lea    (%rdx,%rcx,1),%r8d
  40160b:   0f b6 c2              movzbl %dl,%eax
  40160e:   45 0f b6 c0           movzbl %r8b,%r8d
  401612:   f0 44 0f b0 03        lock cmpxchg %r8b,(%rbx)
  401617:   38 c2                 cmp    %al,%dl
  401619:   75 ea                 jne    0x401605

16-8,7,39(29)
  401741:   0f b7 13              movzwl (%rbx),%edx
  401744:   eb 0c                 jmp    0x401752
  401746:   66 2e 0f 1f 84 00 00  nopw   %cs:0x0(%rax,%rax,1)
  40174d:   00 00 00
  401750:   89 c2                 mov    %eax,%edx
  401752:   44 8d 04 32           lea    (%rdx,%rsi,1),%r8d
  401756:   0f b7 c2              movzwl %dx,%eax
  401759:   45 0f b7 c0           movzwl %r8w,%r8d
  40175d:   66 f0 44 0f b1 03     lock cmpxchg %r8w,(%rbx)
  401763:   66 39 c2              cmp    %ax,%dx
  401766:   75 e8                 jne    0x401750
```

```
32-8,7,25
  401b60:   8b 13                   mov    (%rbx),%edx
  401b62:   eb 03                   jmp    0x401b67
  401b64:   44 89 c2                mov    %r8d,%edx
  401b67:   46 8d 04 2a             lea    (%rdx,%r13,1),%r8d
  401b6b:   89 d0                   mov    %edx,%eax
  401b6d:   f0 44 0f b1 03          lock cmpxchg %r8d,(%rbx)
  401b72:   39 c2                   cmp    %eax,%edx
  401b74:   41 89 c0                mov    %eax,%r8d
  401b77:   75 eb                   jne    0x401b64

64-8,7,28
  401f70:   48 8b 13                mov    (%rbx),%rdx
  401f73:   eb 03                   jmp    0x401f78
  401f75:   4c 89 c2                mov    %r8,%rdx
  401f78:   4c 8d 04 0a             lea    (%rdx,%rcx,1),%r8
  401f7c:   48 89 d0                mov    %rdx,%rax
  401f7f:   f0 4c 0f b1 03          lock cmpxchg %r8,(%rbx)
  401f84:   48 39 c2                cmp    %rax,%rdx
  401f87:   49 89 c0                mov    %rax,%r8
  401f8a:   75 e9                   jne    0x401f75

  Swap
 Val
8-6,5,17
  401610:   0f b6 1e                movzbl (%rsi),%ebx
  401613:   eb 02                   jmp    0x401617
  401615:   89 c3                   mov    %eax,%ebx
  401617:   89 d8                   mov    %ebx,%eax
  401619:   f0 0f b0 0e             lock cmpxchg %cl,(%rsi)
  40161d:   38 c3                   cmp    %al,%bl

  40161f:   75 f4                   jne    0x401615

16-6,5,19
  401760:   0f b7 16                movzwl (%rsi),%edx
  401763:   eb 02                   jmp    0x401767
  401765:   89 c2                   mov    %eax,%edx
  401767:   89 d0                   mov    %edx,%eax
  401769:   66 f0 0f b1 3e          lock cmpxchg %di,(%rsi)
  40176e:   66 39 c2                cmp    %ax,%dx
  401771:   75 f2                   jne    0x401765

32-6,5,16
  401b80:   8b 16                   mov    (%rsi),%edx
  401b82:   eb 02                   jmp    0x401b86
  401b84:   89 c2                   mov    %eax,%edx
  401b86:   89 d0                   mov    %edx,%eax
  401b88:   f0 0f b1 1e             lock cmpxchg %ebx,(%rsi)
  401b8c:   39 d0                   cmp    %edx,%eax
  401b8e:   75 f4                   jne    0x401b84

64-5,5,16
  401f90:   48 89 c2                mov    %rax,%rdx
  401f93:   48 89 d0                mov    %rdx,%rax
  401f96:   f0 48 0f b1 0e          lock cmpxchg %rcx,(%rsi)
```

```
  401f9b:   48 39 d0                cmp    %rdx,%rax
  401f9e:   75 f0                   jne    0x401f90

128-10,9,45(38)
  4023b0:   4c 8b 0e                mov    (%rsi),%r9
  4023b3:   4c 8b 46 08             mov    0x8(%rsi),%r8
  4023b7:   eb 0d                   jmp    0x4023c6
  4023b9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
  4023c0:   49 89 c1                mov    %rax,%r9
  4023c3:   49 89 d0                mov    %rdx,%r8
  4023c6:   4c 89 c8                mov    %r9,%rax
  4023c9:   4c 89 c2                mov    %r8,%rdx
  4023cc:   f0 48 0f c7 4d 00       lock cmpxchg16b 0x0(%rbp)
  4023d2:   49 31 d0                xor    %rdx,%r8
  4023d5:   49 31 c1                xor    %rax,%r9
  4023d8:   4d 09 c8                or     %r9,%r8
  4023db:   75 e3                   jne    0x4023c0

 Bool
8-5,4,16
  401610:   0f b6 06                movzbl (%rsi),%eax
  401613:   f0 0f b0 0e             lock cmpxchg %cl,(%rsi)
  401617:   41 0f 94 c2             sete   %r10b
  40161b:   45 84 d2                test   %r10b,%r10b
      40161e:     74 f3                   je     0x401613

16-5,4,15
  401760:   0f b7 06                movzwl (%rsi),%eax
  401763:   66 f0 0f b1 3e          lock cmpxchg %di,(%rsi)
  401768:   0f 94 c2                sete   %dl
  40176b:   84 d2                   test   %dl,%dl
  40176d:   74 f4                   je     0x401763

32-6(5),4,22(13)
  401b65:   8b 06                   mov    (%rsi),%eax
  401b67:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  401b6e:   00 00
  401b70:   f0 0f b1 1e             lock cmpxchg %ebx,(%rsi)
  401b74:   0f 94 c2                sete   %dl
  401b77:   84 d2                   test   %dl,%dl
  401b79:   74 f5                   je     0x401b70

64-6(5),4,23(15)
  401f75:   48 8b 06                mov    (%rsi),%rax
  401f78:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  401f7f:   00
  401f80:   f0 48 0f b1 0e          lock cmpxchg %rcx,(%rsi)
  401f85:   0f 94 c2                sete   %dl
  401f88:   84 d2                   test   %dl,%dl
  401f8a:   74 f4                   je     0x401f80

128-8(7),4,29
  4023a0:   48 8b 06                mov    (%rsi),%rax
  4023a3:   48 8b 56 08             mov    0x8(%rsi),%rdx
  4023a7:   45 89 ca                mov    %r9d,%r10d
  4023aa:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
```

```
  4023b0:   f0 48 0f c7 4d 00       lock cmpxchg16b 0x0(%rbp)
  4023b6:   41 0f 94 c1             sete   %r9b
  4023ba:   45 84 c9                test   %r9b,%r9b
  4023bd:   74 f1                   je     0x4023b0

 Goto
8-3,2,9
  401610:   0f b6 06                movzbl (%rsi),%eax
  401613:   f0 0f b0 0e             lock cmpxchg %cl,(%rsi)
  401617:   75 fa                   jne    0x401613

16-3,2,10
  401745:   0f b7 06                movzwl (%rsi),%eax
  401748:   66 f0 0f b1 3e          lock cmpxchg %di,(%rsi)
  40174d:   75 f9                   jne    0x401748

32-3,2,8
  401b45:   8b 06                   mov    (%rsi),%eax
  401b47:   f0 0f b1 1e             lock cmpxchg %ebx,(%rsi)
  401b4b:   75 fa                   jne    0x401b47

64-3,2,10
  401f35:   48 8b 06                mov    (%rsi),%rax
  401f38:   f0 48 0f b1 0e          lock cmpxchg %rcx,(%rsi)
  401f3d:   75 f9                   jne    0x401f38

128-4,2,14
  402340:   48 8b 06                mov    (%rsi),%rax
  402343:   48 8b 56 08             mov    0x8(%rsi),%rdx
  402347:   f0 48 0f c7 0e          lock cmpxchg16b (%rsi)
  40234c:   75 f9                   jne    0x402347

 BBool
8-3,3,9
  401600:   0f b6 03                movzbl (%rbx),%eax
  401603:   f0 0f b0 0b             lock cmpxchg %cl,(%rbx)
  401607:   75 f7                   jne    0x401600

16-3,3,10
  401730:   0f b7 03                movzwl (%rbx),%eax
  401733:   66 f0 44 0f b1 2b       lock cmpxchg %r13w,(%rbx)
  401739:   75 f5                   jne    0x401730

32-3,3,8
  401b31:   8b 03                   mov    (%rbx),%eax
  401b33:   f0 44 0f b1 2b          lock cmpxchg %r13d,(%rbx)
  401b38:   75 f7                   jne    0x401b31

64-3,3,10
  401f30:   48 8b 03                mov    (%rbx),%rax
  401f33:   f0 48 0f b1 0b          lock cmpxchg %rcx,(%rbx)
  401f38:   75 f6                   jne    0x401f30

 BVal
8-6,5,18
  401600:   0f b6 13                movzbl (%rbx),%edx
```

```
  401603:   eb 02                   jmp    0x401607
  401605:   89 c2                   mov    %eax,%edx
  401607:   0f b6 c2                movzbl %dl,%eax
  40160a:   f0 0f b0 0b             lock cmpxchg %cl,(%rbx)
  40160e:   38 c2                   cmp    %al,%dl
  401610:   75 f3                   jne    0x401605

16-6,5,21
  401740:   0f b7 13                movzwl (%rbx),%edx
  401743:   eb 02                   jmp    0x401747
  401745:   89 c2                   mov    %eax,%edx
  401747:   0f b7 c2                movzwl %dx,%eax
  40174a:   66 f0 44 0f b1 2b       lock cmpxchg %r13w,(%rbx)
  401750:   66 39 c2                cmp    %ax,%dx
  401753:   75 f0                   jne    0x401745

32-7,6,21
  401b50:   8b 13                   mov    (%rbx),%edx
  401b52:   eb 03                   jmp    0x401b57
  401b54:   44 89 c2                mov    %r8d,%edx
  401b57:   89 d0                   mov    %edx,%eax
  401b59:   f0 44 0f b1 2b          lock cmpxchg %r13d,(%rbx)
  401b5e:   39 c2                   cmp    %eax,%edx
  401b60:   41 89 c0                mov    %eax,%r8d
  401b63:   75 ef                   jne    0x401b54

64-7,6,24
  401f60:   48 8b 13                mov    (%rbx),%rdx
  401f63:   eb 03                   jmp    0x401f68
  401f65:   4c 89 c2                mov    %r8,%rdx
  401f68:   48 89 d0                mov    %rdx,%rax
  401f6b:   f0 48 0f b1 0b          lock cmpxchg %rcx,(%rbx)
  401f70:   48 39 c2                cmp    %rax,%rdx
  401f73:   49 89 c0                mov    %rax,%r8
  401f76:   75 ed                   jne    0x401f65

      xor
 Val
8-9,8,25
  401620:   0f b6 16                movzbl (%rsi),%edx
  401623:   eb 02                   jmp    0x401627
  401625:   89 c2                   mov    %eax,%edx
  401627:   89 d0                   mov    %edx,%eax
  401629:   31 c8                   xor    %ecx,%eax
  40162b:   41 89 c5                mov    %eax,%r13d
  40162e:   89 d0                   mov    %edx,%eax
  401630:   f0 44 0f b0 2e          lock cmpxchg %r13b,(%rsi)
  401635:   38 c2                   cmp    %al,%dl
  401637:   75 ec                   jne    0x401625

16-8,7,23
  401780:   0f b7 16                movzwl (%rsi),%edx
  401783:   eb 02                   jmp    0x401787
  401785:   89 c2                   mov    %eax,%edx
  401787:   89 d5                   mov    %edx,%ebp
  401789:   89 d0                   mov    %edx,%eax
```

```
  40178b:   31 fd                   xor    %edi,%ebp
  40178d:   66 f0 0f b1 2e          lock cmpxchg %bp,(%rsi)
  401792:   66 39 c2                cmp    %ax,%dx
  401795:   75 ee                   jne    0x401785

32-8,7,20
  401bb0:   8b 16                   mov    (%rsi),%edx
  401bb2:   eb 02                   jmp    0x401bb6
  401bb4:   89 c2                   mov    %eax,%edx
  401bb6:   89 d5                   mov    %edx,%ebp
  401bb8:   89 d0                   mov    %edx,%eax
  401bba:   31 dd                   xor    %ebx,%ebp
  401bbc:   f0 0f b1 2e             lock cmpxchg %ebp,(%rsi)
  401bc0:   39 c2                   cmp    %eax,%edx
  401bc2:   75 f0                   jne    0x401bb4

64-8,7,27
  401fd0:   48 8b 16                mov    (%rsi),%rdx
  401fd3:   eb 03                   jmp    0x401fd8
  401fd5:   48 89 c2                mov    %rax,%rdx
  401fd8:   48 89 d5                mov    %rdx,%rbp
  401fdb:   48 89 d0                mov    %rdx,%rax
  401fde:   48 31 cd                xor    %rcx,%rbp
  401fe1:   f0 48 0f b1 2e          lock cmpxchg %rbp,(%rsi)
  401fe6:   48 39 c2                cmp    %rax,%rdx
  401fe9:   75 ea                   jne    0x401fd5

128-13,12,54(47)
  402400:   4c 8b 06                mov    (%rsi),%r8
  402403:   4c 8b 4e 08             mov    0x8(%rsi),%r9
  402407:   eb 0d                   jmp    0x402416
  402409:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
  402410:   49 89 c0                mov    %rax,%r8
  402413:   49 89 d1                mov    %rdx,%r9
  402416:   4c 89 c3                mov    %r8,%rbx
  402419:   4c 89 c0                mov    %r8,%rax
  40241c:   4c 89 ca                mov    %r9,%rdx
  40241f:   4c 31 d3                xor    %r10,%rbx
  402422:   4c 89 c9                mov    %r9,%rcx
  402425:   f0 48 0f c7 4d 00       lock cmpxchg16b 0x0(%rbp)
  40242b:   49 31 d1                xor    %rdx,%r9
  40242e:   49 31 c0                xor    %rax,%r8
  402431:   4d 09 c1                or     %r8,%r9
  402434:   75 da                   jne    0x402410

 Bool
8-7,6,20
  401610:   0f b6 06                movzbl (%rsi),%eax
  401613:   89 c3                   mov    %eax,%ebx
  401615:   31 cb                   xor    %ecx,%ebx
  401617:   f0 0f b0 1e             lock cmpxchg %bl,(%rsi)
  40161b:   41 0f 94 c3             sete   %r11b
  40161f:   45 84 db                test   %r11b,%r11b
  401622:   74 ef                   je     0x401613

16-8(7),6,27(19)
```

```
  401765:   0f b7 06                movzwl (%rsi),%eax
  401768:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  40176f:   00
  401770:   89 c2                   mov    %eax,%edx
  401772:   31 fa                   xor    %edi,%edx
  401774:   66 f0 0f b1 16          lock cmpxchg %dx,(%rsi)
  401779:   0f 94 c2                sete   %dl
  40177c:   84 d2                   test   %dl,%dl
  40177e:   74
 f0                     je     0x401770

32-8(7),6,26(18)
  401b75:   8b 06                   mov    (%rsi),%eax
  401b77:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  401b7e:   00 00
  401b80:   89 c2                   mov    %eax,%edx
  401b82:   31 da                   xor    %ebx,%edx
  401b84:   f0 0f b1 16             lock cmpxchg %edx,(%rsi)
  401b88:   0f 94 c2                sete   %dl
  401b8b:   84 d2                   test   %dl,%dl
  401b8d:   74 f1                   je     0x401b80

64-8(7),6,29(21)
  401f85:   48 8b 06                mov    (%rsi),%rax
  401f88:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  401f8f:   00
  401f90:   48 89 c2                mov    %rax,%rdx
  401f93:   48 31 ca                xor    %rcx,%rdx
  401f96:   f0 48 0f b1 16          lock cmpxchg %rdx,(%rsi)
  401f9b:   0f 94 c2                sete   %dl
  401f9e:   84 d2                   test   %dl,%dl
  401fa0:   74 ee                   je     0x401f90

128-10(9),7,38(29)
  4023b0:   48 8b 06                mov    (%rsi),%rax
  4023b3:   48 8b 56 08             mov    0x8(%rsi),%rdx
  4023b7:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  4023be:   00 00
  4023c0:   48 89 c3                mov    %rax,%rbx
  4023c3:   48 89 d1                mov    %rdx,%rcx
  4023c6:   4c 31 c3                xor    %r8,%rbx
  4023c9:   f0 48 0f c7 4d 00       lock cmpxchg16b 0x0(%rbp)
  4023cf:   0f 94 c3                sete   %bl
  4023d2:   84 db                   test   %bl,%bl
  4023d4:   74 ea                   je     0x4023c0

 Goto
8-5,4,13
  401605:   0f b6 06                movzbl (%rsi),%eax
  401608:   89 c3                   mov    %eax,%ebx
  40160a:   31 cb                   xor    %ecx,%ebx
  40160c:   f0 0f b0 1e             lock cmpxchg %bl,(%rsi)
  401610:   75 f6                   jne    0x401608

16-5,4,14
  401750:   0f b7 06                movzwl (%rsi),%eax
```

```
  401753:   89 c2                   mov    %eax,%edx
  401755:   31 fa                   xor    %edi,%edx
  401757:   66 f0 0f b1 16          lock cmpxchg %dx,(%rsi)
  40175c:   75 f5                   jne    0x401753

32-5,4,12
  401b60:   8b 06                   mov    (%rsi),%eax
  401b62:   89 c2                   mov    %eax,%edx
  401b64:   31 da                   xor    %ebx,%edx
  401b66:   f0 0f b1 16             lock cmpxchg %edx,(%rsi)
  401b6a:   75 f6                   jne    0x401b62

64-5,4,16
  401f60:   48 8b 06                mov    (%rsi),%rax
  401f63:   48 89 c2                mov    %rax,%rdx
  401f66:   48 31 ca                xor    %rcx,%rdx
  401f69:   f0 48 0f b1 16          lock cmpxchg %rdx,(%rsi)
  401f6e:   75 f3                   jne    0x401f63

128-7,5,23
  402370:   48 8b 06                mov    (%rsi),%rax
  402373:   48 8b 56 08             mov    0x8(%rsi),%rdx
  402377:   48 89 c3                mov    %rax,%rbx
  40237a:   48 89 d1                mov    %rdx,%rcx
  40237d:   4c 31 d3                xor    %r10,%rbx
  402380:   f0 48 0f c7 0e          lock cmpxchg16b (%rsi)
  402385:   75 f0                   jne    0x402377

 BBool
8-6,6,17
  401600:   0f b6 13                movzbl (%rbx),%edx
  401603:   0f b6 c2                movzbl %dl,%eax
  401606:   31 ca                   xor    %ecx,%edx
  401608:   0f b6 d2                movzbl %dl,%edx
  40160b:   f0 0f b0 13             lock cmpxchg %dl,(%rbx)
  40160f:   75 ef                   jne    0x401600

16-6,6,18
  401740:   0f b7 13                movzwl (%rbx),%edx
  401743:   0f b7 c2                movzwl %dx,%eax
  401746:   31 f2                   xor    %esi,%edx
  401748:   0f b7 d2                movzwl %dx,%edx
  40174b:   66 f0 0f b1 13          lock cmpxchg %dx,(%rbx)
  401750:   75 ee                   jne    0x401740

32-5,5,13
  401b50:   8b 03                   mov    (%rbx),%eax
  401b52:   44 89 ea                mov    %r13d,%edx
  401b55:   31 c2                   xor    %eax,%edx
  401b57:   f0 0f b1 13             lock cmpxchg %edx,(%rbx)
  401b5b:   75 f3                   jne    0x401b50

64-5,5,16
  401f50:   48 8b 03                mov    (%rbx),%rax
  401f53:   48 89 ca                mov    %rcx,%rdx
  401f56:   48 31 c2                xor    %rax,%rdx
```

```
  401f59:   f0 48 0f b1 13          lock cmpxchg %rdx,(%rbx)
  401f5e:   75 f0                   jne    0x401f50

 BVal
8-9,8,27
  401600:   0f b6 13                movzbl (%rbx),%edx
  401603:   eb 02                   jmp    0x401607
  401605:   89 c2                   mov    %eax,%edx
  401607:   89 d0                   mov    %edx,%eax
  401609:   31 c8                   xor    %ecx,%eax
  40160b:   44 0f b6 c0             movzbl %al,%r8d
  40160f:   0f b6 c2                movzbl %dl,%eax
  401612:   f0 44 0f b0 03          lock cmpxchg %r8b,(%rbx)
  401617:   38 c2                   cmp    %al,%dl
  401619:   75 ea                   jne    0x401605

16-9,8,41
  401741:   0f b7 13                movzwl (%rbx),%edx
  401744:   eb 0c                   jmp    0x401752
  401746:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  40174d:   00 00 00
  401750:   89 c2                   mov    %eax,%edx
  401752:   41 89 d0                mov    %edx,%r8d
  401755:   0f b7 c2                movzwl %dx,%eax
  401758:   41 31 f0                xor    %esi,%r8d
  40175b:   45 0f b7 c0             movzwl %r8w,%r8d
  40175f:   66 f0 44 0f b1 03       lock cmpxchg %r8w,(%rbx)
  401765:   66 39 c2                cmp    %ax,%dx
  401768:   75 e6                   jne    0x401750

32-9,8,36(26)
  401b62:   8b 13                   mov    (%rbx),%edx
  401b64:   eb 0d                   jmp    0x401b73
  401b66:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  401b6d:   00 00 00
  401b70:   44 89 c2                mov    %r8d,%edx
  401b73:   41 89 d0                mov    %edx,%r8d
  401b76:   89 d0                   mov    %edx,%eax
  401b78:   45 31 e8                xor    %r13d,%r8d
  401b7b:   f0 44 0f b1 03          lock cmpxchg %r8d,(%rbx)
  401b80:   39 c2                   cmp    %eax,%edx
  401b82:   41 89 c0                mov    %eax,%r8d
  401b85:   75 e9                   jne    0x401b70

64-9,8,30
  401f80:   48 8b 13                mov    (%rbx),%rdx
  401f83:   eb 03                   jmp    0x401f88
  401f85:   4c 89 c2                mov    %r8,%rdx
  401f88:   49 89 d0                mov    %rdx,%r8
  401f8b:   48 89 d0                mov    %rdx,%rax
  401f8e:   49 31 c8                xor    %rcx,%r8
  401f91:   f0 4c 0f b1 03          lock cmpxchg %r8,(%rbx)
  401f96:   48 39 c2                cmp    %rax,%rdx
  401f99:   49 89 c0                mov    %rax,%r8
  401f9c:   75 e7                   jne    0x401f85
```

References

AMD09: AMD, AMD64 Architecture Programmer's Manual Volume 3:General-Purpose and System Instructions, 2009,
AMD10: AMD, AMD64 Architecture Programmer's Manual Volume 2: System Programming, 2010,
Anderson09: Sean Eron Anderson, Bit Twiddling Hacks, 2009, http://graphics.stanford.edu/~seander/bithacks.html
Anderson90: Thomas E. Anderson, The performance of spin lock alternatives for shared-money multiprocessors, 1990,
Andrews00: Gregory R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, 2000,
Appleby11: Austin Appleby, MurmurHash, 2011, https://sites.google.com/site/murmurhash/
BC07: Hans-J. Boehm, Lawrence Crowl, C++ Atomic Types and Operations, 2007, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html
BER07: Daniel Bauer, Luis Garcés-Erice, Sean Rooney, Tempo - A Simple Time-Sensitive Messaging System, 2008,
BFGK05: Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, Bradley C. Kuszmaul, Concurrent Cache-Oblivious B-Trees, 2005,
BH11: Trevor Brown, Joanna Helga, Non-blocking k-ary Search Trees, 2011,
Bonwick94: Jeff Bonwick, The Slab Allocator: An Object-Cacheing Kernel Memory Allocator, 1994,
BP11: Anastasia Braginsky, Erez Petrank, Locality-Conscious Lock-Free Linked Lists, 2011,
CLRS09: Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford Stein, Introduction to Algorithms, 2009,
Craig93: Travis S. Craig, Queuing spin lock algorithms to support timing predictability, 1993,
Daniels11: Adam Daniels, Progress with Progress Guarantees, 2011, http://www.slideserve.com/adamdaniel/progress-with-progress-guarantees
DB08: Kristijan Dragicevic, Daniel Bauer, Survey of Concurrent Priority Queue Algorithms, 2008,
DBRD91: Richard P. Draves, Brian N. Bershad, Richard F. Rashid, Randall W. Dean, Using Continuations to Implement Thread Management and Communication in Operating Systems, 1991,
Dean93: Randall Dean, Using Continuations to Build a User-Level Threads Library, 1993,
Draves94: Richard P. Draves, Control Transfer in Operating System Kernels, 1994,
EFRB10: Faith Ellen, Panagiota Fatourou, Eric Ruppert, Franck van Breugel, Non-blocking binary search trees, 2010,
Fomitchev03: Mikhail Fomitchev, Lock-Free Linked Lists and Skip Lists, 2004,
Fraser04: Keir Fraser, Practical lock-freedom, 2004,
GNU11: GNU, Atomic Builtins, 2011, http://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc/Atomic-Builtins.html
Google: Google, CityHash, 2011, http://code.google.com/p/cityhash/
GPST05: Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, Philippas Tsigas, Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting,

2005,
GPT05: Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas, Allocating memory in a lock-free manner, 2005,
Herlihy91: Maurice Herlihy, Wait-free synchronization, 1991,
HMBW07: Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, Jonathan Walpole, Performance of memory reclamation for lockless synchronization, 2007,
HMPS96: Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, Michael L. Scott, An efficient algorithm for concurrent priority queue heaps, 1996,
Hoare61: C.A.R. Hoare, Algorithm 65: Find, 1961,
HSAH06: Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, Benjamin C. Hertzberg, McRT-Malloc: a scalable transactional memory allocator, 2006,
HSY10: Danny Hendler, Nir Shavit, Lena Yerushalmi, A scalable lock-free stack algorithm, 2009,
IBM83: IBM, System/370 Extended Architecture, Principles of Operation, 1983,
IR93: Amos Israeli, Lihu Rappoport, Efficient Wait-Free Implementation of a Concurrent Priority Queue, 1993,
Jenkins: Bob Jenkins, Bob Jenkins' Web Site, , http://burtleburtle.net/bob/
Knuth97: Donald E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 1997,
KP11: Alex Kogan, Erez Petrank, Wait-free queue with multiple enqueuers and dequeuers, 2011,
KPS09: Gabriel Kliot, Erez Petrank, Bjarne Steensgaard, A Lock-Free, Concurrent, and Incremental Stack Scanning Mechanism for Garbage Collectors, 2009,
MCS91: John M. Mellor-Crummey and Michael L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, 1991,
Michael04: Maged M. Michael, Hazard Pointers: Reclamation for Lock-Free Objects, 2004,
ML84: Udi Manbar and Richard E. Ladner, Concurrency control in a dynamic search structure, 1984, http://doi.acm.org/10.1145/1270.318576
MLH94: Peter Magnusson, Anders Landin, and Erik Hagersten, Queue Locks on Cache Coherent Multiprocessors, 1994,
MS98: Paul E. McKenney, J.D. Slingwine, Read-copy update: using execution history to solve concurrency problems, ,
PMS09: Erez Petrank, Madanlal Musuvathi, Bjarne Steensgaard, Progress guarantee for parallel programs via bounded lock-freedom, 2009,
PPL11: Charles Torre, Asynchronous Programming for C++ Developers: PPL Tasks and Windows 8, 2011, http://channel9.msdn.com/Blogs/Charles/Asynchronous-Programming-for-C-Developers-PPL-Tasks-and-Windows-8
RK79: David P. Reed and Rajendra K. Kanodia, Synchronization with eventcounts and sequencers, 1979,
SL00: Nir Shavit, Itay Lotan, Skiplist-Based Concurrent Priority Queues, 2000,
SR10: Michael Spiegel, Paul F. Reynolds, Jr., Lock-Free Multiway Search Trees, 2010,
ST05: Håkan Sundell, Philippas Tsigas, Fast and lock-free concurrent priority queues for multi-thread systems, 2005,
Sundell04: Håkan Sundell, Efficient and Practical Non-Blocking Data Structures, 2004,
SZ00: Nir Shavit, Asaph Zemach, Combining funnels: a dynamic approach to software combining, 2000,
SZ99: Nir Shavit, Asaph Zemach, Scalable concurrent priority queue algorithms, 1999,

TDK: Ami Tavory and Vladimir Dreizin, IBM Haifa Research Laboratories, and Benjamin Kosnik, Red Hat, Policy-Based Data Structures, ,
http://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/
TSP92: John Turek, Dennis Sasha, Sundeep Prakash, Locking without blocking: making lock based concurrent data structure algorithms nonblocking, 1992,
Valois95: J.D. Valois, Lock-free linked lists using compare-and-swap, 1995,
Wang07: Thomas Wang, Integer Hash Function, 2007,
http://www.cris.com/~Ttwang/tech/inthash.htm
Warton05: Matthew Warton, Single Kernel Stack L4, 2005,